Golden Model-Free Hardware Trojan Detection by Classification of Netlist Module Graphs

Alexander Hepp[*], Johanna Baehr[*], Georg Sigl[*†],
[*]Department of Electrical and Computer Engineering
Technical University of Munich, Munich, Germany
Email: {alex.hepp,johanna.baehr,sigl}@tum.de
[†]Fraunhofer Institute for Applied and Integrated Security (AISEC), Munich, Germany

***Abstract*—In a world where increasingly complex integrated circuits are manufactured in supply chains across the globe, hardware Trojans are an omnipresent threat. State-of-the-art methods for Trojan detection often require a golden model of the device under test. Other methods that operate on the netlist without a golden model cannot handle complex designs and operate on Trojan-specific sets of netlist graph features.**

In this work, we propose a novel machine-learning-based method for hardware Trojan detection. Our method first uses a library of known malicious and benign modules in hierarchical designs to train an eXtreme Gradient Boosted Tree Classifier (XGBClassifier). For training, we generate netlist graphs of each hierarchical module and calculate feature vectors comprising structural characteristics of these graphs. After the training phase, we can analyze the synthesized hierarchical modules of an unknown design under test. The method calculates a feature vector for each module. With this feature vector, each module can be classified into either benign or malicious by the previously trained XGBClassifier. After classifying all modules, we derive a classification for all standard cells in the design under test. This technique allows the identification of hardware Trojan cells in a design and highlights regions of interest to direct further reverse engineering efforts.

Experiments show that this approach performs with >97% Sensitivity and Specificity across available and newly generated hardware Trojan benchmarks and can be applied to more complex designs than previous netlist-based methods while maintaining similar computational complexity.

I. Introduction

With the ever-increasing use of Integrated Circuits (ICs) in both home appliances and industrial production, there is a growing risk of malicious functionality added to an IC by a third party. At the same time, the internationalization of production chains and accompanying trends mean that the entire production chain cannot remain under close surveillance. This lack of control allows for the addition of Hardware Trojans (HTs) that endanger all security principles. A HT is a change of functionality created so that it stealthily blends into an electronic device such as an IC and hides its malicious capabilities [1]. It typically consists of a trigger and a payload circuit. The trigger activates the payload circuit depending on specific conditions, while the payload performs malicious activity after activation.

In answer to this threat, multiple techniques for HT detection have been proposed. Usually a distinction is made between pre- and post-silicon methods [2], i.e. between methods that can be applied before and after manufacturing. The first group of pre-silicon methods relies on heuristics to find weak spots in designs that are prone to HT insertion [e.g. 3] or on heuristics to find seldomly used signals or inputs [e.g. 4]. Another group of methods performs logic code analysis in order to prove that the design performs as specified [e.g. 5] or that the design has not been tampered with [e.g. 6]. Post-silicon methods either use a side-channel analysis [e.g. 7] or a functional, behavioral analysis [e.g. 8] to discover changes in the design.

To summarize, most hardware Trojan detection techniques use a variant of fingerprinting. These detection techniques become ineffective, when HTs are designed that evade the detection heuristics and a golden model of the design is not available. Golden model selection is a expensive and difficult task and might be impossible if the HT is already inserted in the design phase. Many HT detection techniques can only be applied using some form of golden model, for example by simulation [9] or assuming that only part of the fabricated ICs carry Trojans [10]. Furthermore, even sophisticated heuristics detecting Trojan characteristics can be broken by designing appropriate HTs [e.g. 11].

Besides these methods, hardware Trojans can be detected based on netlist analysis. A suspicious design netlist is inspected pre-silicon, to find HT inserted by the system designer or in third party IP-cores. Previous Reverse Engineering(RE)- and machine learning-based techniques for HT detection in netlists focus on small design sizes and detect HTs by few HT-specific features, which are not generally found in every HT [12]–[14]. Yu, Gu, Liu, *et al.* [15] proposed a netlist detection method based on a feature vector for each standard cell using local structural features of cells fan-in and fan-out. This still can not utilize structures beyond a threshold logic level. To summarize, previous HT detection methods for netlists search for local fingerprints or signatures of HTs.

This paper follows a novel and orthogonal approach in *large design* HT detection, by employing a recently introduced form of automated netlist reverse engineering [16]. The authors show that fuzzy netlist submodule identification is possible with an

The authors would also like to thank the anonymous reviewers for their valuable comments and helpful suggestions. This work was partly funded by the German Ministry of Education and Research in the project RESEC under Grant No.: 16KIS1009

Authorized licensed use limited to: ETRI. Downloaded on July 03,2025 at 02:01:53 UTC from IEEE Xplore. Restrictions apply.

approach based on graph partitioning and machine learning. They build upon the notion that in Application Specific Integrated Circuit (ASIC) netlist graphs, form follows function, i.e. the design functionality can be identified by analyzing the netlist graph structure. In this work, we automatically identify and classify submodules in gate-level netlists containing a HT and do not depend on a golden model being available, but operate on a set of known designs and primitives that are identified within the netlist of a design. As such a netlist of the design is available for black-box IP-cores, this approach may also be used for detecting HTs in protected Intellectual Property (IP) or even Field-Programmable-Gate-Array (FPGA) netlists. Without loss of generality, this paper focuses on ASIC-HTs.

The important difference to pre-silicon fingerprinting techniques such as FANCI [4] or netlist based fingerprinting [12] is that the features we collect from the flat netlist are not Trojan-specific, but are generally able to identify building blocks used in the design under test.

The presented approach also does not create a whole-design fingerprint, unlike methods such as [7], which create one side-channel fingerprint per design. We believe that sufficiently complex hardware designs can not be analyzed without splitting it into smaller elements.

Many HT detection methods require a golden chip that is HT-free. The method proposed in this work can also be used to select such a golden model, so that these, typically simpler but also less complex detection methods can be applied.

The contribution of this paper consists of four parts:

1) We define a novel process and algorithm for fuzzy hardware Trojan detection in large netlists.
2) We show that the computational complexity of the novel process is on par with the complexity of the previous method.
3) We show by experiment that detecting hardware Trojans using this process is successful.
4) We analyze the existing machine learning-based HT-detection technique on large netlists and show that the local HT-specific features are not useful for complex netlists nor generalize well towards unknown designs.

The rest of the paper is structured as follows: In section II we describe the RE-based HT detection process and the individual steps necessary to perform our analysis. In section III we empirically evaluate the performance of our approach and compare it to the existing HT detection method based on machine learning.

II. Methodology

A. Definitions

In this work, a design netlist is perceived as hypergraph $\mathcal{D} = (\mathcal{M}, \mathcal{N})$, with the set of modules (i.e. standard cells) $\mu \in \mathcal{M}$ and the nets $\nu \in \mathcal{N}$ connecting the modules. All modules connected to the net ν are in the set $\mathcal{M}_\nu$. A hierarchical design can be described by a tree $\mathcal{H}$ with nodes $\mathcal{H}_i^l$ on a hierarchy level l starting with $l = 0$ describing the complete design. Each hierarchical node in the tree instantiates at least two subnodes from higher levels, i.e. $H_i^l = \{H_i^{l'} | l_{max} \geqslant l' > l \geqslant 0\}$ and $|H_i^l| > 0$. Leaf nodes $H_i^{l_{max}}$ are in the highest level l_{max} of the tree and are identical to the modules μ, i.e. $\forall i : H_i^{l_{max}} \in \mathcal{M}$.

$\bar{H}_i^l = \bigcup_{H_i^{l'} \in H_i^l} \bar{H}_i^{l'}$, with $\bar{H}_i^{l_{max}} = H_i^{l_{max}}$, is defined as the flattened hierarchy, i.e. the set of μ belonging to the hierarchy node itself or to one of the subnodes from higher levels.

For each $\bar{H}_i^l$, we generate a directed graph $\bar{G}_i^l = (\bar{V}_i^l, \bar{E}_i^l)$ such that each node $\bar{v}$ is a module $\mu \in \bar{H}_i^l$ and $\bar{V}_i^l$ is thus a set of standard cells in $\bar{H}_i^l$. An edge $\bar{e} \in \bar{E}$ is generated between two modules $\mu_1, \mu_2 \in \bar{H}_i^l$ if there is a net $\nu \in \mathcal{N}$ in the design netlist with $\mu_1 \in \mathcal{M}_\nu \wedge \mu_2 \in \mathcal{M}_\nu$. As net names and cell names are arbitrary and should not influence the classification, only the type of μ is kept as an attribute of $\bar{v}$. Placement information is not utilized, thus the edges $\bar{e}$ have unit length. Another graph $G_i^l = (V_i^l, E_i^l)$ is generated, such that each node v is a module $\mu \in \{\mu | \mu \in H_i^l\}$, i.e. this graph comprises only the nodes describing the immediate function of this hierarchy node. The graph $\bar{G}_0^0$, in which no hierarchy is given, is the flat netlist produced at the end of logic synthesis. In this work, we refer to the set of all G_i^l and $\bar{G}_i^l$ in one design as L. When distinguishing between the members of L is not necessary, the symbol G is used. The union of all L in a given library of designs is denoted as $\mathcal{L}$. $\mathcal{L}$ is thus a set of subgraphs, defined by the respective design hierarchy, of all flat netlists in a library of designs. In such a library of HT-infected circuits, a *base-design* is the benign large circuit into which HTs can be inserted. Each insertion forms a new HT specimen, that is a base-design with inserted HT.

B. Input Data and Attacker Model

We specify the attacker model using the HT-taxonomy in [1]. In the realm of this paper, an attacker may be capable of inserting a HT at the *specification* and *design* phases. The inserted HTs may operate at any abstraction level down to the Gate level, may use any activation mechanism, may have any effects, and a location in the processor, the I/O and the clock tree. (see fig. 1) This means that the presented method cannot, for example, identify a HT that does not have a trigger and only modifies a metal wire width for side channel leakage or increased power consumption. This limitation can be compensated with orthogonal side-channel and testing based HT-detection methods. If the HT evades such methods with a complicated trigger that hides the change on the physical level, the trigger circuit will be visible on the gate-level and can be detected using our approach described in this paper.

Our methodology requires netlists and known hierarchy information for a collection of benign and infected *library designs* to train the machine learning, in which the HTs are implemented in one or multiple H_i^l, separate from the rest of the design. Such a database could be maintained by an entity collecting Trojan samples from the whole silicon industry and providing pre-trained machine learning models for Trojan detection. If the library is large enough, it allows the machine learning to generalize on the features that constitute a

Authorized licensed use limited to: ETRI. Downloaded on July 03,2025 at 02:01:53 UTC from IEEE Xplore. Restrictions apply.

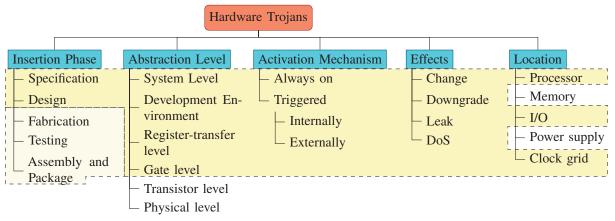

Figure 1. Hardware Trojan taxonomy [1]. The highlighted area in yellow shows the assumed attacker model.

hardware Trojan and thus also detect Trojans that are previously unknown.

The trained machine learning hardware Trojan detector as described in this paper is purposed for pre-silicon detection. The netlist and hierarchy information for the design under test is retrieved as a synthesis output, in order to detect HTs inserted by the hardware designer, the synthesis tool or in IP-cores. In this work, we assume that the HTs are implemented in one or multiple H_i^l, separate from the rest of the design.

Additionally, a flat netlist could be retrieved from silicon reverse engineering post-silicon, in order to detect HTs inserted at any step in the supply chain. For a flat netlist, it is necessary to generate detected hierarchies $\hat{\mathcal{H}}_i$ with partitioning algorithms as described in [16]. This has the potential to increase the attacker capabilities to an insertion at any *insertion phase*, as detection happens for the final end product.

C. Machine Learning-Assisted Reverse Engineering

The process of detecting HTs with in netlist graphs comprises three steps for all library designs:

Graph generation Convert the netlist to a graph representation and generate the set L.

Graph labelling Label all graphs in $\mathcal{L}$ as malicious (1), if they belong to the Trojan circuit, or as benign (0) otherwise.

Feature vector generation For each graph in L, generate a graph embedding feature vector.

Afterwards, we train a eXtreme Gradient Boosted Tree Classifier (XGBClassifier) with the graphs in L, represented by their feature vectors and test our approach with a separate testing set, that was prepared identical to the training set. When training is completed, we can identify HTs in a design under test as follows:

Graph generation Convert the netlist to a graph representation and generate the set L for the netlist.

Feature vector generation For each G_i^l and $\bar{G}_i^l$, generate a graph embedding feature vector.

Graph classification Use the trained XGBClassifier to create a label for each G_i^l and $\bar{G}_i^l$.

Entire design assessment Join the resulting subgraph labels to a label for each μ in the entire design.

The rest of this section describes these steps in detail.

1) Graph generation: As larger hardware designs under test contain a multitude of different functionalities and Trojans may be spread across, it is not sufficient to identify and classify the design functionality for the design as a whole. Generating a feature vector for a large design may hide the small changes introduced by a HT.

Table I
OVERVIEW OF FEATURES, BOTH ORIGINAL AND SINGLE. COMPLEXITY IS BASED ON ASSUMING A SPARSE NETLIST GRAPH, I.E. $|V| \sim |E|$

Feature Type	Feature	Complexity
Original	degree	$\mathcal{O}(\|V\|^2)$
	node in-degree	$\mathcal{O}(\|V\|^2)$
	(in/out-degree, in/out-edge katz, eigenvector, betweenness)-centrality	$\mathcal{O}(\|V\|)$
	closeness centrality	$\mathcal{O}(\|V\|^2)$
	pagerank	$\mathcal{O}(\|V\|)$
Single	Nr. of *TYPE* nodes	$\mathcal{O}(\|V\|)$
	Node Type Ratio for *TYPE*	$\mathcal{O}(\|V\|)$
	number of nodes and edges	$\mathcal{O}(1)$
	density	$\mathcal{O}(\|V\|)$
	degree assortativity coefficient	$\mathcal{O}(\|V\|)$
	condensation factor	$\mathcal{O}(\|V\|)$
	length of dominating set	$\mathcal{O}(\|V\|)$
	flow hierarchy	$\mathcal{O}(\|V\|^2)$

Graph generation is performed for the *library designs* and the design under test. We use the available hierarchical description and generate the G_i^l and $\bar{G}_i^l$ for all nodes in the hierarchy tree, to receive the set $\mathcal{L}$. This step is performed during training and during testing.

2) Graph labelling: Any H_i^l that *directly implements* Trojan functionality is called infected, else it is called benign. We define *directly implements* such that if a H_i^l is called infected, there is no submodule $H_i^{l'}$, $l' > l$ which is not inserted by the HT designer. Thus, a $\bar{H}_i^l$ is called infected, if H_i^l is called infected and a G_i^l or $\bar{G}_i^l$ is called infected, if the corresponding H_i^l or $\bar{H}_i^l$ is called infected. This naming can be converted to a discrete-valued labeling function $m_\mathrm{L}(G) : \mathcal{L} \mapsto \{0, 1\}$ for each subgraph. This function can also be formulated on the cell-level as $m_\mathrm{c}(\mu) : \mathcal{M} \mapsto \{0, 1\}$ for each μ. $m_\mathrm{c}(\mu)$ is either 1, if the *malicious* μ is part of any infected H_i^l or 0 otherwise.

3) Feature Vector Generation: In this work, we assume that the functionality of a netlist can be inferred by inspection and comparison of its graph structure. Consequently, the input feature vectors for machine learning predominantly contain values that express the structure of the netlist graph.

Each previously generated G_i^l and $\bar{G}_i^l$ is transformed into a graph embedding feature vector. Some feature values are generated by statistically evaluating information for each v or e in the graph, referred to as original features. These original features include node degree and various centrality measures. Table I summarizes the used original features. After original feature generation, the values for each v or e in the design are statistically agglomerated with average, min, max, mode, median, entropy and an empirically parameterized histogram. The result is a varying number of derived features per original feature. These derived features are part of the feature list.

Other features are single agglomerate values for the whole graph, such as the number of input and output dummy nodes, the number of v of a specific gate type (such as AND or OR gates) or graph connectivity and modularity measures. Table I summarizes these single features, which are directly part of

Authorized licensed use limited to: ETRI. Downloaded on July 03,2025 at 02:01:53 UTC from IEEE Xplore. Restrictions apply.

the feature vector.

In total, this results in 230 numeric feature values saved in a feature vector, along with the identifier, for each G_i^l and $\bar{G}_i^l$. This step is performed during training and during testing.

Using the generated feature vectors, a XGBClassifier is trained without scaling and preprocessing. The XGBClassifier is a time and space efficient implementation of machine learning using gradient tree boosting, especially fit for the large number of features and learning samples used in this work. The training set is the list of feature vectors for all graphs in the library $\mathcal{L}$, which includes benign designs, the benign parts of infected designs, as well as the *infected* G_i^l and $\bar{G}_i^l$ from the infected designs, which directly implement Trojan functionality. The truth vector consists of the previously defined ground truth values for each graph in $\mathcal{L}$.

4) Graph classification: HT detection is performed based on the feature vectors of all graphs in L of a design under test using the previously trained XGBClassifier. The output of the XGBClassifier is a discrete-valued function $\hat{m}_{\mathrm{L}}(G) \in \{0,1\}$ mapping each G to a maliciousness value.

5) Entire Design Assessment: Based on the previous results, it is possible to assess and evaluate a complete design under test. For this, we calculate the cell-level maliciousness function $\hat{m}_c(\mu) = \max_{G|\mu \in G} \hat{m}_{\mathrm{L}}(G)$ that finds a maliciousness value for each standard cell μ by finding if μ is part of any G that was detected to be infected.

As a result, a design under test is labeled into a benign and a malicious part (i.e. those μ where $\hat{m}_c(\mu) = 0$ and $\hat{m}_c(\mu) = 1$, respectively).

III. Experimental Results

To evaluate the approach presented in this work, a high number of HT samples are required to train the classifier. The number of available public HT-benchmarks is far too small to allow the necessary training.

In order to generate the required number of HT samples, a simple HT-generator was used, that created synthetic variants of existing HTs from the benchmark database Trust-Hub [17]. To this end, the available samples for three base-designs were analyzed. It was found that the available samples only differ in the combination of the used trigger and payload circuits, while the concrete implementation of the triggers and payloads, including constant values such as the trigger condition, stay the same across the samples. This provided an opportunity to create more synthetic Trojans by varying the combination of trigger and payload and the used constant values. The HT generator outputs verilog RTL and is designed to put the HT implementation into separate verilog modules, one for the trigger and one for the payload.

The three base-designs are: An RSA implementation, a fully pipelined AES implementation (due to the used synthesis tool, the AES implementation was reduced to one round) and a UART implementation. Each sample contains a Trojan-free design (one of the original designs) and a design that is infected.

With the Trojan generator, a total of $2\,999$ HT samples were generated, ca. 1000 per base-design. The samples were synthesized for ASIC using the open source synthesis tool-chain `qflow` in version 1.3, that employs the `yosys` verilog synthesis tool, version 0.8. Using the synthesis logs, for each output netlist, the hierarchical modules each standard cell belongs to were determined. This provides the necessary hierarchy information and the ground truth $m_{\mathrm{L}}(G)$ for each HT sample netlist. In addition, 361 non-Trojan designs were synthesized using the same process. For synthesis, the `osu035` library from the Oklahoma State University with a $0{,}35\,\mu\mathrm{m}$ technology was used.

In total, this results in $115\,269$ G_i^ls and $\bar{G}_i^l$s, of which $10\,844$ stem from non-infected library designs, $110\,377$ are non-infected parts of the HT samples and $4\,892$ are the infected HT modules.

Six additional netlists are used to asses the generalization capabilities. Four of them are additional HT samples from Trust-Hub and two are additional benign hardware designs. These additional netlists and their modules are not used during training of the XGBClassifier, but only used for testing the learned model.

After synthesis, the netlists were parsed into a verilog AST using a custom parser based on pyverilog [18]. As in previous work on netlist RE, buffer structures were removed from the netlists [19], [20]. From the AST, a graph representation was generated in python using networkx [21]. In order to improve feature generation speed, the calculations for feature vector generation were performed partially with graph-tool [22] and igraph [23], which use a C++ and C-based backend, respectively. The statistical evaluation was performed using the algorithms from scikit-learn [24], while the XGBClassifier from the xgboost library [25] was used.

In order to receive an optimal set of hyperparameters, a random subset of 591 of the HT samples, as well as the 361 non-Trojan designs and the additional netlists were used to perform a grid search on $34\,560$ parameter options. The `GroupShuffleSplit` cross-validation iterator from scikit-learn was used on the subset to provide training, validation and testing sets for the grid search. Grouping input data was necessary to make sure that all G_i^ls and $\bar{G}_i^l$s belonging to one HT sample are put in the same set, either training, validation or testing. The number of folds was set to 5. Matthews correlation coefficient was used to evaluate a hyperparameter combination, because this metric is robust to an imbalanced classification task. The optimum hyperparameters are shown in table II. Note that the trained best classifier resulting from hyperparameter grid search is discarded. This means that the additional netlists are accounted for in the hyperparameters, but do not belong to the trained knowledge of the final XGBClassifier.

For training and evaluating the final classifier, a training set and a test set must be defined. From the available synthetic HT samples, $90\,\%$ are used for training, the remaining are used for testing. Again, it is ensured that all graphs of one sample are put in the same set. This ensures that training is balanced and that during testing, the complete HT sample must be scored by

Authorized licensed use limited to: ETRI. Downloaded on July 03,2025 at 02:01:53 UTC from IEEE Xplore. Restrictions apply.

Table II
HYPERPARAMETERS FOR THE XGBCLASSIFIER USED IN GRID SEARCH AND THEIR OPTIMUM VALUES.

Hyperparameter	Grid Search Value Set	Optimum
`gamma`	{0, 0.5, 1, 10}	0
`learning_rate`	{0.1, 0.3, 0.8, 1}	0.1
`max_delta_step`	{1, 3, 10}	10
`max_depth`	{2, 4, 6, 12}	4
`min_child_weight`	{0.5, 1, 3, 10}	0.5
`n_estimators`	{10, 20, 50, 200, 400}	50
`scale_pos_weight`	{1.0, 10, 27}	1.0
`subsample`	{0.5, 0.8, 1.0}	1.0

the classifier. The benign designs are added to training and test set with the same split ratio. Finally, all additional HT samples are added to the test set for the generalization assessment.

Preparation of the library and the testing set was performed on two *AMD EPYC 7601 32-Core Processor*s clocked at 2,6 GHz, running Ubuntu 18.04 with a maximum of 20 GB RAM at 60 threads during the feature vector generation. The total runtime of this approach is ca. 30 h, of which ca. 0,5 h were spent on specimen generation and RTL synthesis, ca. 0,5 h on verilog parsing and graph generation, ca. 15 h on feature generation, ca. 14 h on hyperparameter optimization and only a few minutes to train the XGBClassifier. Note that this preparation of the HT-detector is a one-time task. The application of the trained model to evaluate a design for HTs requires only to generate the feature vector for the design under test and apply the classification and entire design evaluation. For example, for the additional test netlist `wb_conmax-T200`, this takes ca. 10 min.

In the following, we discuss the effectiveness of using machine learning classification to correctly define a design, or $G \in L$ of a design, as malicious or benign.

A. Machine Learning Performance

In this section, the machine learning algorithms are evaluated for their ability to correctly classify the graphs and to perform an entire design assessment that strongly correlates with the ground truth maliciousness of the design.

1) Partition and standard-cell classification: The performance of the classification can be assessed by the Sensitivity (Se) and Specificity (Sp), i.e. the True-Positive Rate (TPR) and the True-Negative Rate (TNR). A true positive is a true-infected $\bar{G}_i^l$ or G_i^l that is classified as "malicious", a true negative is a true-benign $\bar{G}_i^l$ or G_i^l that is classified as "benign". For the synthetic HT samples in the test set, the classifier performs perfectly, with $Se = Sp = 100{,}0\,\%$.

To ensure that the XGBClassifier-results generalize well across additional base-designs that are unknown to the classifier, we assess the classifier performance for the additional HT samples (see table III). For the majority of the samples, the classifier performs perfectly. For `wb_conmax-T200`, 193 of 250 G in the design are false-positive. As these modules are very small, the cell-based Specificity is not as severely harmed. The HT itself is detected with high Sensitivity.

The classifier also generalizes well for both unknown benign designs. `risc16f84` is a microcontroller design and the benign version of the `PIC*` HTs. It is marked as benign with perfect Specificity. `versatile_mem_ctrl` is a memory controller for SDRAM. This design is marked as benign with good Specificity.

Table III
GENERALIZATION RESULTS FOR SIX ADDITIONAL NETLISTS. THE LAST TWO COLUMNS SHOW RESULTS FOR THE METHOD IN [12].

		$\hat{m}_L(G)$		$\hat{m}_c(\mu)$		$\hat{m}_c$ of [12]	
Design name	$\|\mathcal{M}\|$	Se	Sp	Se	Sp	Se	Sp
PIC16F84-T200	2510	1.0	1.0	1.0	1.0	0.2	0.99
PIC16F84-T300	2523	1.0	1.0	1.0	1.0	0.06	1.0
PIC16F84-T400	2632	1.0	1.0	1.0	1.0	0.06	0.99
wb_conmax-T200	32547	1.0	0.16	1.0	0.63	0.04	0.63
risc16f84	2446	N/A	1.0	N/A	1.0	N/A	0.99
versatile_mem_ctrl	9537	N/A	0.69	N/A	0.87	N/A	0.84

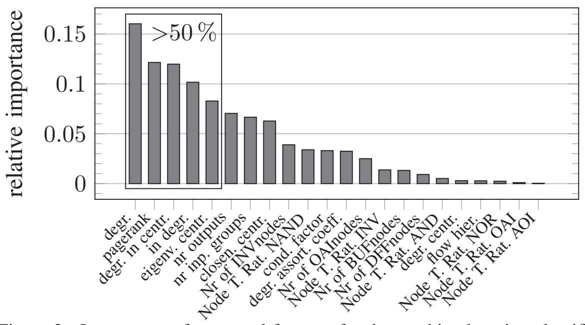

Figure 2. Importance of structural features for the machine learning classifier. For the importance of the original features, the importance values of derived features were summed up per original feature.

2) Importance of Structural Features in the HT Context: It is important to understand which structural features correlate to the maliciousness of a G, in order to asses the capabilities of the presented classification approach. As typical for tree-based machine learning models, a trained XGBClassifier reports the feature-importance. Figure 2 shows the reported relative importance after training of the classifier. The five most important features are graph structural features, achieving an importance of 58,6 %. The 22 shown original features allow to achieve the perfect results of section III-A1, the remaining 26 original features are not used by the classifier. Among these features are primarily original features, while the single features are predominantly in the set of important features. No degradation of performance is expected when removing the unused features. However, the importance of structural features strongly depends on the training samples. As further designs are added to the training sample set, the feature importance values might change. It is therefore not sensible to propose the removal of features at this point in time. Also, the feature calculation routines are optimized for speed, so it was not necessary to reduce the amount of features from the proposed set of features in [16].

B. Comparison to Hardware Trojan Specific Features

1) Machine learning performance: We implemented the method proposed in [12] using an efficient matrix-multiplication based approach and applied it to our dataset. We

Authorized licensed use limited to: ETRI. Downloaded on July 03,2025 at 02:01:53 UTC from IEEE Xplore. Restrictions apply.

employed the same train-test-split as for our experiments and used the XGBClassifier in order to achieve comparable results. The experiment reveals an average Specificity of $99{,}99\,\%$, but an average Sensitivity of only $78\,\%$. In particular, for designs which were not tested in the original papers (such as the HT-specimen in the RSA circuit), the sensitivity decreases significantly. Even worse results were found for HT detection in designs that are not in the training set. In the generalization test set (see table III), sensitivities less than 0.2 were found, leading to the assumption that an approach using local HT specific features does not generalize well to unknown designs.

2) Complexity Analysis: Under the assumption that G is sparse (i.e. $(|E| = \mathcal{O}(|V|))$), matrix multiplication requires $\mathcal{O}(|V|)$, a breath-first-search (BFS) of the graph requires $\mathcal{O}(|V|)$, and a BFS for a limited depth is constant time $\mathcal{O}(1)$.

The complexity of the calculation of HT specific features as proposed in [12] can be grouped into three groups: those requiring a constant number of sparse matrix multiplications for each standard cell (resulting in $\mathcal{O}(|V|^2)$), those requiring a BFS for each standard cell (resulting in $\mathcal{O}(|V|^2)$) and those requiring a BFS of limited depth for each standard cell (resulting in $\mathcal{O}(|V|)$). Thus, the complexity of the calculation of HT specific features can be approximated with $\mathcal{O}(|V|^2)$.

For our method, the complexity of feature calculation can also be approximated with $\mathcal{O}(|V|^2)$ (see Table I), and thus is comparable in complexity to other feature based approaches.

This theoretical result is proven empirically, as the feature calculation for the method of [12] required ca. $13\,\mathrm{h}$ on our machine, which is comparable to the required time for the approach of this work.

IV. Future Work

In order to apply this method for real world HT detection, the training library must be substantially improved and shared across users. For this, a larger set of HT triggers and payloads should be developed. An increased library also results in a larger run-time, but significant effects are only expected for the feature vector generation step. Fortunately, this effort need only be performed once for each library design. Additionally, the used features should be repeatedly evaluated for importance to remove any features that impair runtime, but do not provide substantially to detection quality. The machine learning and design evaluation performance is satisfactory, nevertheless an increase and diversification of the library, as new designs (both benign and malicious) are included, will allow to improve the results for the generalization case, as well. To protect IP, it is also possible to share only feature vectors or pre-trained machine learning models.

V. Conclusion

This works presents a novel method for golden-model free hardware Trojan detection in large netlist graphs. By partitioning a design under test into subgraphs using pre-silicon hierarchy information, we retrieve functional blocks. Using a feature vector comprised of a broad set of structural graph features, the functional blocks are classified into benign or malicious. This allows us to assess unprecedentedly large and complex designs. By experiment, the process was shown to be almost perfectly sensitive and specific for the available Trojan specimens, and outperformed algorithms using Trojan specific features. The dataset will be published at `10.5281/zenodo.5776362`.

References

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

[2] K. Xiao *et al.*, "Hardware Trojans: Lessons Learned After One Decade of Research," *ACM TODAES*, vol. 22, no. 1, May 2016.

[3] H. Salmani, "COTD: Reference-Free Hardware Trojan Detection and Recovery Based on Controllability and Observability in Gate-Level Netlist," *IEEE TIFS*, 2017.

[4] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: Identification of Stealthy Malicious Logic Using Boolean Functional Analysis," in *Proc. CCS '13*, 2013.

[5] J. Rajendran, A. M. Dhandayuthapany, V. Vedula, and R. Karri, "Formal Security Verification of Third Party Intellectual Property Cores for Information Leakage," in *Proc. VLSID '16*, 2016.

[6] W. Hu, B. Mao, J. Oberg, and R. Kastner, "Detecting Hardware Trojans with Gate-Level Information-Flow Tracking," *Computer*, 2016.

[7] J. Balasch, B. Gierlichs, and I. Verbauwhede, "Electromagnetic circuit fingerprints for Hardware Trojan detection," in *Proc. EMC*, 2015.

[8] M. Banga and M. S. Hsiao, "A Novel Sustained Vector Technique for the Detection of Hardware Trojans," in *Proc. VLSID*, 2009.

[9] J. He, Y. Zhao, X. Guo, and Y. Jin, "Hardware Trojan Detection Through Chip-Free Electromagnetic Side-Channel Statistical Analysis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017.

[10] R. Bian, M. Xue, and J. Wang, "A Novel Golden Models-Free Hardware Trojan Detection Technique Using Unsupervised Clustering Analysis," in *Proc. ICCCS*, 2018.

[11] J. Zhang, F. Yuan, and Q. Xu, "DeTrust: Defeating Hardware Trust Verification with Stealthy Implicitly-Triggered Hardware Trojans," in *Proc. CCS*, 2014.

[12] T. Kurihara, K. Hasegawa, and N. Togawa, "Evaluation on Hardware-Trojan Detection at Gate-Level IP Cores Utilizing Machine Learning Methods," in *2020 IEEE 26th Int. Symp. on On-Line Testing and Robust System Design (IOLTS)*, 2020.

[13] S. Li *et al.*, "A XGBoost based Hybrid Detection Scheme for Gate-Level Hardware Trojan," in *2020 IEEE 9th Joint Int. Information Technology and Artificial Intelligence Conf. (ITAIC)*, 2020.

[14] K. G. Liakos *et al.*, "Conventional and machine learning approaches as countermeasures against hardware trojan attacks," *Microprocessors and Microsystems*, 2020.

[15] S. Yu, C. Gu, W. Liu, and M. O'Neill, "A Novel Feature Extraction Strategy for Hardware Trojan Detection," in *2020 IEEE Int. Symp. on Circuits and Systems (ISCAS)*, 2020.

[16] J. Baehr, A. Bernardini, G. Sigl, and U. Schlichtmann, "Machine Learning and Structural Characteristics for Reverse Engineering," in *Proc. ASPDAC '19*, 2019.

[17] B. Shakya *et al.*, "Benchmarking of Hardware Trojans and Maliciously Affected Circuits," *HASS*, 2017.

[18] S. Takamaeda-Yamazaki, "Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL," in *Appl. Reconfigurable Comput.*, 2015.

[19] B. Cakir and S. Malik, "Revealing Cluster Hierarchy in Gate-level ICs Using Block Diagrams and Cluster Estimates of Circuit Embeddings," *ACM TODAES*, 2019.

[20] ——, "Reverse Engineering Digital ICs Through Geometric Embedding of Circuit Graphs," 2018.

[21] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring Network Structure, Dynamics, and Function using NetworkX," in *Proc. SciPy '08*, 2008.

[22] T. P. Peixoto, "The graph-tool python library," 2014.

[23] G. Csárdi and T. Nepusz, "The igraph software package for complex network research," *Int J Complex Syst*, 2006.

[24] F. Pedregosa *et al.*, "Scikit-learn: Machine Learning in Python," *J. Mach. Learn. Res.*, 2011.

[25] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proc. ACM KDD '16 Conf.*, 2016.

Authorized licensed use limited to: ETRI. Downloaded on July 03,2025 at 02:01:53 UTC from IEEE Xplore. Restrictions apply.