Is your FPGA bitstream Hardware Trojan-free? Machine learning can provide an answer

Alessandro Palumbo, Luca Cassano, Bruno Luzzi, José Alberto Hernández, Pedro Reviriego, Giuseppe Bianchi, Marco Ottavi

Is Your FPGA Bitstream Hardware Trojan-free? Machine Learning Can Provide an Answer

Alessandro Palumbo[1], Luca Cassano[2], Bruno Luzzi[1], José Alberto Hernández[3], Pedro Reviriego[3], Giuseppe Bianchi[1], Marco Ottavi[1,4]

Abstract

Software exploitable Hardware Trojan Horses (HTHs) inserted into commercial CPUs allow the attacker to run his/her own software or to gain unauthorized privileges. Recently a novel menace raised: HTHs inserted by CAD tools. A consequence of such scenario is that HTHs must be considered a serious threat not only by academy but also by industry. In this paper we try to answer to the following question: can Machine Learning (ML) help designers of microprocessor softcores implemented onto SRAM-based FPGAs at detecting HTHs introduced by the employed CAD tool during the generation of the bitstream? We present a comparative analysis of the ability of several ML models at detecting the presence of HTHs in the bitstream by exploiting a previously performed characterization of the microprocessor softcore and an associated ML training. An experimental analysis has been carried out targeting the IBEX RISC-V microprocessor running a set of benchmark programs. A detailed comparison of multiple ML models is conducted, showing that many of them achieve accuracy above 98%, and κ values above 0.97. By identifying the most effective ML models and the best features to be employed, this paper lays the foundation for the integration of a ML-based bitstream verification flow.

Keywords: CAD, Hardware Security, Hardware Trojans, Machine learning, Microprocessors, RISC-V, SRAM-based FPGA

1. Introduction and Related Work

In past years, *Hardware Trojan Horses (HTHs)* were exclusively considered an academic issue because of the difficulty of insertion in real-world circuits and the limited advantages that the attacker could count on. Nevertheless, it has recently been demonstrated that complex and dangerous *software-exploitable HTHs* can be inserted in real-world commercial microprocessors. This class of more powerful HTHs, allows the attacker to execute his/her own malicious software, to modify the running software or to acquire root privileges [1, 2, 3]. Finally, in 2018, a HTH, called the *Rosenbridge backdoor*, has been found in a commercial Via Technologies C3 processor [4, 5]. The Rosenbridge backdoor could be activated via software and allowed the attacker to enter in supervisor mode. Given these premises, HTHs must nowadays be considered an issue not only by academy but also by industry.

Recently, a new security-related menace raised: HTHs introduced in the designed circuit by the employed CAD tool [6, 7]. In [8, 9] the don't cares of the design are exploited to insert HTHs both in the RTL code or gate-level netlist. In [10] a black-hat high-level synthesis tool has been presented: starting from a high-level specification, i.e., a C/C++/SystemC, of the desired functionality the tool produces an HTH-infested hardware implementation of the corresponding IP core. The authors also demonstrated that several types of HTHs could be introduced in the produced IP core: HTHs downgrading performance, changing the implemented functionality and draining the battery of the system. Finally, in [11] the authors demonstrate that all electronic CAD tools, i.e., high-level synthesis, logic synthesis, physical design, verification, test, and post-silicon validation, are potential threat vectors to different degrees. Similar considerations can also be made when looking at the FPGA scenario instead of the ASIC one. It has indeed been demonstrated that CAD tools may represent a serious threat for the security and trust of FPGA-based systems [12, 13, 14]. In particular, it has been demonstrated that malicious CAD tools may tamper the produced bitstream before FPGA configuration to introduce HTHs in the system [15, 16]. Given this discussion, it is crucial to provide designers with effective tools to detect malicious modifications introduced in the system by the employed CAD tool before sending the design to the foundry (in the case of an ASIC design) or before integrating it in the final system (in the case of an FPGA-based design).

Several techniques for HTHs detection before system deployment have been proposed in the last two decades. These are generally *circuit-level* techniques that aim at detecting HTHs at design time via logic testing [17], formal property verification [18], side-channel analysis [19], structural and behavioral analysis [20, 21] and machine learning [22, 23]. On the other hand, since HTHs are extremely stealthy by nature and since modern integrated circuits own a huge amount of resources among which a HTH can be hidden, it is extremely hard to detect HTHs before the system has been deployed. Therefore, a new paradigm raised, the so called *Design for Trust*, where the interest is on *system-level* techniques that allow to obtain a trusted

[1]University of Rome Tor Vergata, Italy, email: {name.surname}@uniroma2.it

[2]Politecnico di Milano, Italy, email: luca.cassano@polimi.it

[3]Universidad Carlos III de Madrid, Spain, email: jahgutie@it.uc3m.es, revirieg@it.uc3m.es

[4]University of Twente, The Netherlands, email: m.ottavi@utwente.nl

system built with untrusted components [24, 25, 26]. A similar paradigm has been proposed in [27, 28] where the focus is on microprocessor-based systems and the goal is to enable a trusted software execution on an untrusted CPU. All these works address the problem of detecting HTHs introduced either by the vendor of the purchased IP cores or by the silicon foundry. On the other hand, the problem of detecting HTHs introduced by an untrusted CAD tool in a trusted IP core, i.e., designed in-house or purchased from a trusted IP provider, has not been thoroughly explored yet. Most of the existing wok focused on the analysis of the vulnerability of finite state machines either through automatic test pattern generation, as in [29], of formal property verification, as in [30]. Recently, a new paradigm, dubbed *Security Rule Checking*, has been proposed to drive the implementation and the adoption of CAD tools specifically meant to support the verification of the trustworthiness and security of systems throughout the whole design process [31]. In the very same paper the authors claimed the urgent need for tools able to implement such new paradigm. The work most similar to ours is the one proposed in [32] where reverse engineering of the generated configuration bitstream is exploited to extract the layout actually implemented onto the FPGA device. Such "real" (possibly infected) layout is then compared with the "ideal" layout coming from the design phases to find differences, and thus, to signal the presence of a HTH. The big difference between our approach and the one in [32] is that we exploit machine learning to identify these differences while the one in [32] relies on a deep knowledge of the FPGA architecture and bitstream details.

In this paper we pose the following question: is Machine Learning (ML) effective and efficient in detecting HTHs introduced in the generated bitstream by the employed CAD tool when designing microprocessor softcores meant to be implemented onto SRAM-based FPGAs? To answer this question we performed a comparative analysis of the ability of several ML models in detecting the presence of HTHs in the bitstream by exploiting a previously performed characterization of the target microprocessor softcore and associated ML training. Five ML models have been considered in our analysis, namely Logistic Regression, Decision Tree, Support Vector Machines, Random Forest and Gradient Boosting Machine. Looking at the attack, we considered four HTH models: a HTH that alters the clock signal arriving in the infected microprocessor, a HTH that modifies the critical path of the infected microprocessor and two HTHs that interfere with the fetching activity by modifying the fetched instruction and the accessed instruction memory address, respectively. In particular, the ML models are here trained and then employed to observe and analyse not only generic circuit-level features, e.g., number of logic gates, working frequency, as generally done by ML-based HTH detection methodologies, but also microprocessor-specific microarchitectural features, e.g., the fetching activity, the content of the performance counters. The experimental analysis has been carried considering an IBEX microprocessor, which is a low-power version of the well-known RISC-V architecture, running a set of benchmark programs. A detailed comparison of multiple ML models is conducted, showing that many of them achieve accuracy values above 98% in the test set, and κ values above 0.97 both in the case where the injected HTH has been triggered during simulation and in the case where it remains untriggered.

Extensive work has been conducted on the use of ML in identifying software malware, like studies in [33, 34, 35] where different ML models are used to identify Android Malware and also to separate truly dangerous malware (i.e. Trojans, etc) from soft adware which only seek to show ads to the users. At the same time, there also is a vast literature about the application of ML models to detect HTHs, as it is demonstrated by a number of recently published surveys on the topic [36, 37, 38]. All these works agree in highlighting the following research directions that have not been taken into account yet:

- Exploring the application of ML to detect HTHs at higher abstraction level than only the chip-level;
- Facing the threat represented by HTHs introduced in all the design phases by the employed CAD tool; indeed, most existing works focus on HTHs introduced either by IP providers or silicon foundries;
- Analysing the applicability of ML models for HTH detection on real-world complex microprocessors running a given program.

To the best of our knowledge, the current paper is the first comparative study of the ability of several ML models specifically designed, implemented, trained and evaluated to detect HTHs introduced in the generated bitstream by the employed CAD tool when designing microprocessor softcores meant to be then implemented onto SRAM-based FPGAs.

The remainder of this paper is organized as follows: Section 2 first presents the considered threat model and then briefly overviews the five ML models considered in our comparison; Section 3 depicts the envisioned verification flow in which ML could help in detecting HTHs introduced in bitstreams while Section 4 presents and discusses the results from the proposed experimental comparison; Section 5 concludes the paper.

2. Background

We here present some background on Hardware Trojan Horses (HTHs) and the considered threat models and we then overview the five ML models considered in our comparison.

2.1. Hardware Trojan Horses

As it has been previously discussed, a *Hardware Trojan Horse (HTH)* is a very hard-to-detect modification of a system i) that keeps silent most of the time, and becomes active under specific rare conditions, altering the nominal behavior of the system, o ii) that is always active and covertly steals sensitive information processed by the system. According to the taxonomy presented in [39], HTHs may be classified based on their *triggering mechanism*, *payload* and *insertion phase*.

A HTH may be triggered:

- *internally* by logical signals (or sequences of logical signals, in case of sequential HTHs) or by physical quantities, e.g., internal temperature or voltage, or by a hidden ad-hoc configured counter (the so-called time bombs);
- *externally* by either received messages or commands, or by physical interactions, e.g., again the external temperature or voltage; and
- *always-on*, i.e., HTHs that become active as soon as the system is turned on.

Under the point of view of the payload, i.e., their effect on the infested system, HTHs may be classified in:

- *Change functionality HTHs* that alter the nominal functionality of the infected system, e.g., make the system execute a malicious code;
- *Information stealing HTHs* that steal secret information from the system either through the available communication interfaces, e.g., by sending unauthorized messages to the attacker, or through covert side-channels, e.g., temperature or magnetic field; and
- *Denial-of-service HTHs* that stop the functioning of the system, e.g., by introducing *nop* instructions, by draining the system's batteries, by jamming the communication interfaces, by altering the timing behaviour.

Finally, looking at the insertion, HTHs may be inserted by IP providers in the purchased 3PIPs, by rogue designers and by the employed CAD tools possibly in every stage of the design flow and by the foundry during chip fabrication.

2.1.1. The Considered Threat Model

For the comparative analysis proposed in this paper we considered HTHs injected by the employed CAD tool when translating the final design files of microprocessor softcores meant to be implemented onto SRAM-based FPGA devices into the configuration bitstream. Therefore, the here considered source of attack is the CAD tool generating the bitstream, while all other tools are considered to be trusted or verifiable. We believe that this is a reasonable assumption: indeed, the generated bitstream is the most hard-to-verify output among all the intermediate outputs of the FPGA-based design flow and thus the most suitable for the insertion of a HTH. Indeed, all the intermediate outputs of the FPGA-based design flow can be verified via formal equivalence checking and formal property verification, logic testing, design-rule checking and simulation. On the other hand, the same verification activities are much harder when dealing with the final configuration bitstream, since they would require strong reverse engineering capabilities, which are not always available in a design team. Indeed, as it has been discussed in [40], although reverse engineering the structure of the logic netlist, i.e., retrieving the number of LUTs and FFs employed in the design, is "quite simple", understanding the content of the LUTs and the structure of the routing, i.e., the actual functionality implemented in the device, is a much harder task. Therefore, we argue that applying equivalence checking at the bitstream may be unfeasible.

When looking at the specific HTH models, we considered the following ones (that have been originally presented in the Trust-Hub repository [41]):

- `Clk_Mod`, that is a HTH able to slow down the microprocessor's clock thus impacting on programs' throughput. By following the previous classification, this HTH is denial-of-service and it may be either triggered or always-on, depending on the configuration.
- `Critical`, that is a HTH that inserts additional combinational logic in a path of the circuit. If inserted on the critical path of the circuit, this HTH has an impact on its timing behaviour, and thus, possibly, on performance. This HTH is denial-of-service and always-on.
- `MitM`, that is a HTH that emulates a malicious man in the middle between the microprocessor and the instruction memory. Indeed, this HTH interferes with the fetching activity of the microprocessor by modifying the fetched instructions, thus forcing the execution of a malicious program. This HTH is triggered and changes functionality.
- `Fetch`, that is a HTH that interferes with the fetching activity of the microprocessor by altering the instruction memory address from which instructions are loaded, thus again, forcing the execution of a malicious program. Also this HTH is triggered and changes functionality.

2.2. The Considered Machine Learning models

There exists a large number of supervised ML classification algorithms in the literature, both linear and non-linear, that can be directly applied to the HTH detection problem. Following the no-free lunch (NFL) theorem, there is no single algorithm suitable for all problems, so a wide number of them must be tested and adjusted using cross-validation techniques with grid-search for fine hyper-parameter tuning, as explained in [42, 43]. For our experimental comparison we considered the following well-know machine learning models[5]:

- **Logistic Regression**, which is a fast, simple, linear regression model. Although these advantages, logistic regression often shows underfitting since it cannot capture non-linear data patterns. Still, logistic regression provides a performance baseline for comparison and a hint about which features in the considered dataset are most relevant in the classification task.
- **Decision Tree**, that classifies instances of a problem by sorting them based on the values of a set of features. Nodes in a DT represent features, while branches represent values that the associated features may assume.

[5] A detailed presentation of all the considered machine learning models can be found in Kotsiantis et al. [44].

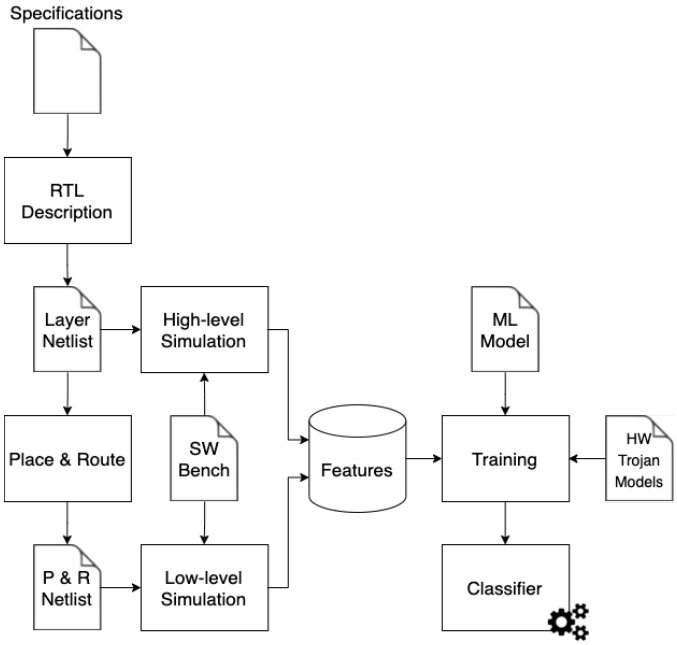


Figure 1: The flow for the definition and training of the classifier.

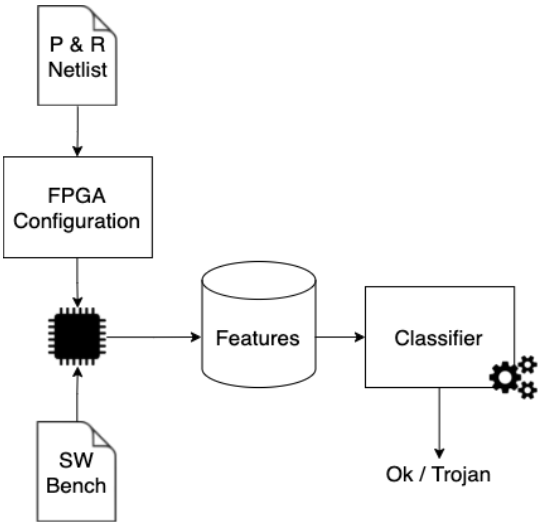


Figure 2: The flow for the bitstream verification.

Leaves represent the result of the classification. Instances of the problem are classified starting from the root and traversing it based on the actual values of the features. Examples of tree algorithms include ID3, C4.5 or C5.0 algorithms.

- **Random Forest**, which combine a number of decision trees to build an ensemble classifier that outperforms individual decision trees by always selecting the class identified by the majority of the trees in the considered forest.
- **Support Vector Machines**, which represent the instances of a dataset as points of an n-dimensional space. An SVM is an ML technique based on the identification of one or more hyperplanes (based on the number of classes of the problem) used to isolate the classes the elements of the available training set belong to. In particular, an SVM identifies the hyperplanes having the largest margin, i.e., the distance between the hyperplane and the closest instance of every class, to minimize the classification error.
- **Gradient Boosting Machine**, which, like random forest, exploit an ensemble of prediction models, typically decision tree, to produce a majority-based classification. GBM are specially suitable for classification problems suffering class imbalance, as it is the case of the HTH dataset of this work.

3. Proposal: a Hardware Trojan Horse Detection Flow

In this section we briefly depict the envisioned flow for HTH-aware bitstream verification that would benefit from the availability of effective ad hoc designed and trained ML-based models. Nevertheless, we point out that the focus of the current paper is on demonstrating the feasibility of detecting HTHs introduced in the bitstream of microprocessor softcores by observing a set of circuit-level and runtime features through a ML model. Moreover, we are interested in a thorough experimental comparison among the considered ML models in order to identify the most suitable for the targeted problem. On the other hand, the implementation of the verification flow itself falls outside the scope of this paper.

The envisioned verification flow is meant to be a design-time flow, to be executed before deploying the microprocessor softcore under analysis in the final system. The idea is that, since, as discussed before, the bitstream generation tool is considered untrsuted, the generated bitstream needs to be verified against the output of the previous steps of the design flow (which are considered either trusted or verifiable). To do so we exploit trusted simulations at the logic netlist-level and at the post-place&route-level to extract a number of features, e.g., the number of executed Load instruction or the average power consumption. These features are in turn used to train a ML-based classifier. This ML-based classifier is finally used to verify the behaviour of the real system when the (possibly infected) bitstream is downloaded onto the considered FPGA device. Indeed, in this final verification phase the ML-based classifier is fed with features coming from the execution of the real system configured onto the target FPGA. The flow would be divided in two parts to be executed one after each other: the *classifier definition flow* and the *bitstream verification flow*. The first part of the flow would allow the designer to extract circuit-level and runtime features and to effectively train the ML-based classifier. The second part of the flow would be the one actually in charge of verifying whether the produced bitstream (and thus the obtained microprocessor) is HTH-free and can be deployed in the final system, or not. In the following we present some details about the two sub-flows.

3.1. The classifier definition flow

Figure 1 depicts the flow that would be required to define and train the ML-based classifier that would then be used to

verify the bitstream. The left-side of the flow is the standard FPGA-based design flow. The two intermediate artifacts of the logic synthesis and place&route phases, namely the logic netlist and the post place&route netlist, would be exploited in two simulation processes: a *high-level simulation* and a *low-level simulation* where the microprocessor logic and post place&route netlists are simulated together with the considered software benchmark(s). The high-level simulation allows to extract runtime features like statistics on the executed instructions and the number of clock cycles required to complete a given instruction as well as details related to the logic netlist, such as the number of LUTs and FFs. We employed the Verilator simulation environment [45] to extract the runtime features while the synthesis tool integrated into the Vivado design suite [46] has been used to extract the netlist-level details. On the other hand, the low-level simulation allows to extract fine-grained circuit-level features like details about the power consumption and the timing of the circuit. In particular, we exploited the Switching Activity Interchange Format provided by the Vivado suite to extract detailed and accurate power and temperature analysis with high level of confidence; the timing-related information are extracted by Vivado when synthesizing and after the place&route phase; All these extracted features as well as the adopted machine learning model and the considered HTH models are then fed into the training algorithm to actually train the ML-based classifier.

3.2. The bitstream verification flow

Figure 2 depicts the flow that would be required to actually verify whether the produced bitstream is HTH-free or not. By following the standard FPGA design flow, the post place&route netlist is translated into a bitstream that is finally loaded onto the configuration memory of the target SRAM-based FPGA device. At this point, the very same software benchmark(s) previously used in the high- and low-level simulations can be actually executed on the configured microprocessor. This step allows to collect circuit-level and runtime features related to the real configured microprocessor softcore.

More in details, it is worh mentioning that all the high-level features that we consider fall in the category of the *hardware performance counters*; therefore, they are already made available by the microprocessor either through generic interfaces, e.g., AXI, UART, I2C, or through dedicated debugging interfaces. Indeed, every modern microprocessor provides a given number of registers dedicated to the storage of a set of previously defined and configured hardware performance counters. As a consequence, no additional hardware shall be included in the design.

The timing information of the real prototype implemented onto the FPGA can be retrieved by exploiting an external clock generator. We provide the circuit with a clock signal with an increasing frequency: as soon as we observe the first failure of the system, we can infer that the timing of the critical path has been violated.

In order to measure the amount of logic resources employed by the circuit under analysis after having been configured onto the FPGA reverse engineering methodologies like the one proposed in [40] may be employed. Indeed, such methodologies allow for a rough estimation of the number of LUTs and FFs although not being able to provide all the details related to the functionality implemented by the analysed bitstream.

Finally, The bitstream verification flow exploits the Xilinx XADC Wizard v3.1 LogiCORE monitor to measure the temperature of the circuit under analysis after having configured the desired bitstream onto the target FPGA device. Essentially temperature estimations are collected as follows:

1. Turn-on the board,
2. Wait for 30 minutes,
3. Configure the bitstream of the circuit under analysis,
4. Leave the system reset low (reset active low) for one minutes and then start the execution of the desired program,
5. Collect temperature data samples,
6. Turn-off the board and wait 30 minutes before restarting the process for another program.

The 30 minutes waiting time are needed because we noticed that there was some noise on the temperature measurements due the power supply fluctuations just after powering on the board. On the other hand, after after about 30 minutes we saw the temperature increase due to power supply fluctuations becomes negligible.

Finally, it is important to note that the features used in the proposed methodology can be increased adding new features or reduced if extracting the information for some of the features is too costly but the overall approach would still be applicable.

It is worth noting that, of course, the features collected during this verification phase are the same as the ones collected during the training phase. Finally, based on these collected features and on the previously performed training, the previously trained ML-based classifier is asked to decide whether these features belong to a HTH-free bitstream, or not. Therefore, by implementing and exploiting such verification flow it would be possible to prevent the deployment of possibly malicious bitstreams.

4. Experimental Comparison

We here describe the considered experimental setup for the presented experimental comparison in terms of considered target microprocessor, FPGA device and prototyping of the previously depicted verification flow, considered features and adopted quality metrics and we then report and discuss the obtained results.

4.1. Experimental Setup

The processor selected for our experimental campaign is the lightweight IBEX core [47] which is a 32-bit, low-power, in-order, two stage pipelined RISC-V core often employed in

Table 1: The considered benchmarks.

Benchmark	Description
Coremark	CPU performance benchmark
Median	Median image filter
Multiply	Numbers multiplication
Rsort	Sorting algorithm
Towers	Solver for the tower puzzle

Table 2: The set of considered features.

Feature	Description
Benchmark	Program under execution
Cycles	# clock cycles to execute the program
InstrRet	# instructions retired in the program
LSUs	# waiting cycles to access data memory
FetchWait	# waiting cycles before instruction fetch
Load	# load instructions
Store	# store instructions
Jump	# jump instructions
CondBr	# conditional branches
TakCBran	# taken conditional branches
CompIns	# compressed
MulWait	# cycles for mul. completion
DivdWait	# cycles for div. completion
LUTs	# Look Up Tables
FFs	# Flip Flops
AvgDynPow	Avg. dyn. power consumption
AvgPower	Avg. total power consumption
Timing	Worst negative slack
Temperature	Temperature trend

IoT systems. The synthesis and the implementation of the considered microprocessor have been performed on Vivado targeting a Xilinx XC7Z020 1CLG484C Zynq-7000 FPGA. The obtained core counted 4591 LUTs, 3619 FFs, 256 BRAM cells and worked at 125MHz. Moreover, since the presented analysis has to be conducted while the microprocessor is executing a program, we considered the benchmarks described in Table 1.

The previously depicted bitstream verification flow has been implemented using the open-source `R` programming language and then connected with the standard FPGA design flow implemented by the Vivado environment. In particular, we exploited the ML-specific libraries which are available in the `R`, namely `randomForest` for the Random Forest, `kernlab` for Support Vector Machine, `C50` for Decision Tree, `glmnet` for Logistic Regression, *gbm* for Gradient Boosting Machine and `caret` and `mlbench` for models cross-validation and benchmarking.

The features considered in the proposed experimental comparison are the ones reported in Table 2. As it has previously been discussed, it can be noticed that a number of features are related to the high-level simulation (the top part of the table) and other features are related to the low-level simulation (the ones reported in the bottom of the table). All the considered features are numerical except for the benchmark name, which is a string. We believe that this large set of features is actually able to capture all the circuit- and program-level information that may allow to distinguish between a genuine core running a given program and an infected one.

The dataset used for training and testing the considered machine learning models has been built as follows. We synthesized the considered microprocessor (without any HTH injection) by following four different optimization strategies, i.e., by changing the optimization parameters offered by the synthesis tool. Then, we ran the previously mentioned high- and low-level simulations of these four genuine microprocessor versions while running the five considered benchmark programs. This first set of simulations allowed us to extract the previously discussed features for the HTH-free versions of the microprocessor, thus obtaining data for 20 *good* samples. Then, we injected the previously presented HTH models[6] into the four versions of the considered microprocessor; in particular, for each of the four considered HTH model we derived four different implementations, thus obtaining a total number of 16 infected microprocessors. Then again, we extracted the considered features by running the high- and low-level simulations of these microprocessors while executing the five considered benchmark programs in two distinct conditions: one where the injected HTH was not triggered and one where it was triggered, thus obtaining data for 160 *infected* samples (80 samples for each condition). It is worth mentioning here that all the low-level features have been extracted through a number of experiments on a real prototype of the microprocessor downloaded onto a Xilinx XC7Z020 1CLG484C Zynq-7000 FPGA board. Finally, in order to obtain more balanced datasets, we employed the `DMwR` library implementation of the synthetic minority over-sampling technique (SMOTE) [48] to oversample by 3x the *good* class to count on more non-trojaned circuits. Therefore, overall the proposed experimental comparison has been conducted on two almost balanced sets of 140 (60 `good` plus 80 `infected`) data samples each, namely the *triggered* and the *untriggered* datasets.

Finally, the full dataset has been split into a *train* and a *test* set, by following the classical 65/35% and 80/20% partition rule. The training set is further split into proper *train* and *validation* sets to optimize each model's hyper-parameters following a Leave-One-Out Cross-Validation (LOOCV) strategy. In LOOCV, N data-samples that belong to the train set (i.e. 65 or 80 samples), are split into $N-1$ for training and only 1 for testing, repeating the N samples and taking the performance average for the N cases. This procedure, although computationally expensive, is repeated for a combination of 20 hyper-parameters, just to select the best ML model for each type (LR, DT, RF, SVM, GBM); this is often referred to as grid-search cross-validation, as specified in [42]. The optimal model is then applied to the test dataset for the final evaluation.

[6] As its has been previously discussed, we took into account all the available microprocessor-specific HTH models coming from the Trust-Hub repository [41]: these HTHs are all "change-functionality". On the other hand, our methodology does not rely on the observation of the implemented functionality, but on a number of low- and high-level features. Therefore, we argue that our methodology would allow to detect also denial-of-service and information-stealing HTHs as long as they have an impact on the employed high-level and/or low-level features.

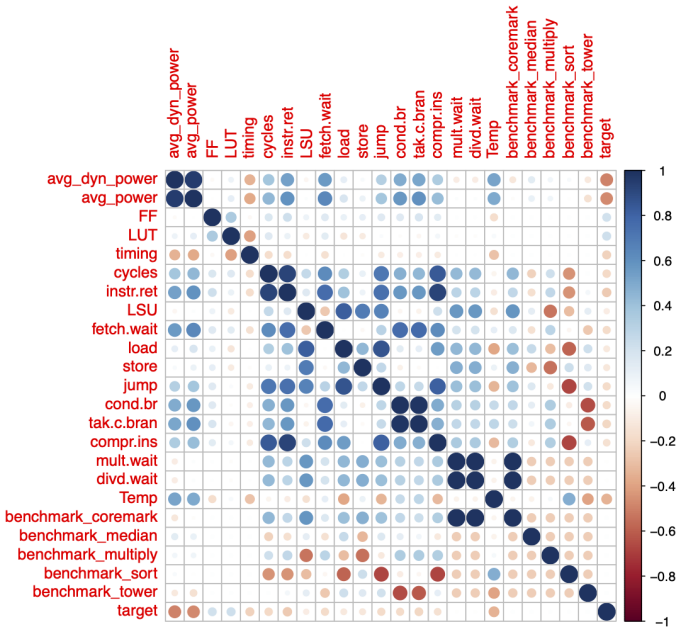


Figure 3: Correlation between features

Concerning the performed evaluation, Accuracy, Sensitivity (also called Recall or True Positive Rate), Specificity (also called True Negative Rate) and Precision have been considered as quality metrics. For the reader's convenience, we here report how the considered quality metrics can be calculated:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \quad (1)$$

$$Sensitivity = \frac{TP}{TP + FN} \quad (2)$$

$$Specificity = \frac{TN}{FP + TN} \quad (3)$$

$$Precision = \frac{TP}{TP + FP} \quad (4)$$

where TP, TN, FP, FN represent the True Positives, True Negatives, False Positives and False Negatives, respectively, being HTH-infested circuits the positive class. Furthermore, we considered the Cohen's Kappa coefficient (κ), that can be calculated as follows:

$$\kappa = \frac{O - E}{1 - E} \quad (5)$$

where O represents the Observed accuracy and E the Expected accuracy, i.e. the accuracy provided by a dummy classifier that always selects the majority class. The difference between the Observed and Expected Accuracy is reflected by the Kappa coefficient showing the benefits of the ML model with respect to a random classifier. Typically, κ values above 0.6 reflect substantial agreement between observed and expected accuracy, while values above 0.8 represent nearly perfect agreement [49].

4.2. Experimental Results

As a first analysis, we assessed the correlation among the considered features, along with the correlation with the `target` one that specifies whether the considered circuit is trojan-free or trojaned, using the R function `corrplot`. The results of this preliminary analysis are reported in Figure 3. As shown, some features are highly correlated with the `target` label, e.g., `avg_dyn_power`, `avg_power` and `temp`, showing somehow that these features may be relevant at the identification of hardware Trojans. Moreover, the features that specify the benchmark under execution also correlate (either positively or negatively) with relevant features (mainly high-level features) in the identification of the Trojans. For example, `benchmark_coremark` positively correlates with the number of clock cycles required to complete the program and with the number of executed store instructions or `benchmark_tower` negatively correlates with the number of conditional branches and the number of taken conditional branches. Finally, several features correlate one to each other: low-level with low-level features, e.g., `avg_dyn_power` with `timing`, can be observed, as well as high-level with high-level features, e.g., `load` with `jump`, and cross-correlation belonging to features from both groups, e.g., `temp` with `load`.

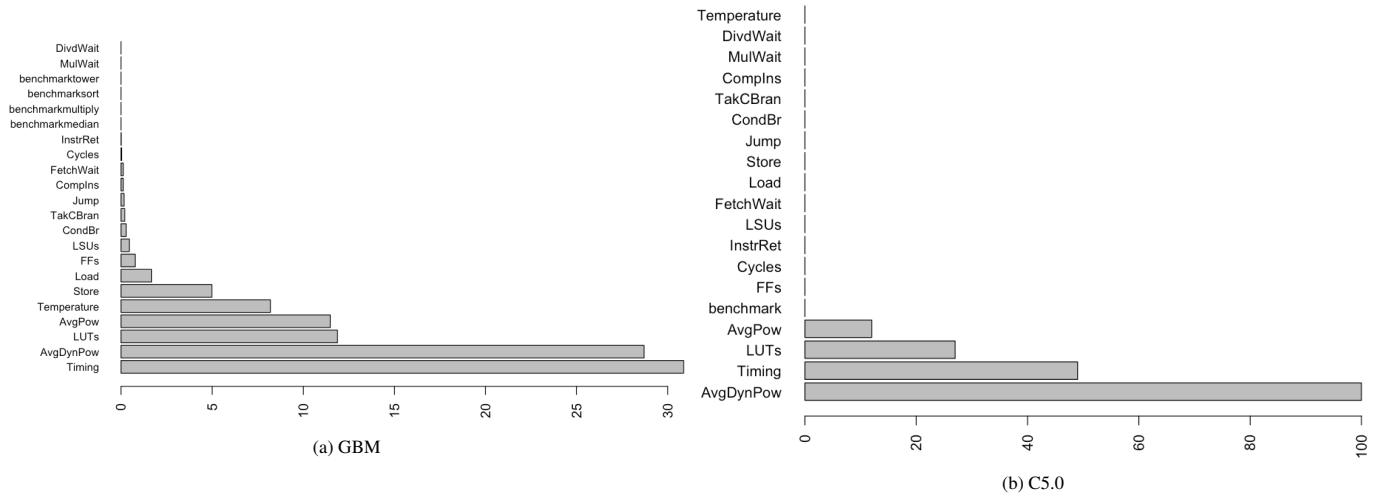
We then conducted the actual experimental campaign. As a first step of our comparative analysis we trained and tested all the previously presented ML models with the considered datasets. Tables 3 and 4 show the accuracy and Kappa values achieved during testing, for both 65/35% (top part of the table) and 80/20% partitions (bottom part of the table) for the *triggered* and *untriggered* dataset, respectively. It can be noticed

Table 3: ML performance results for splits: 65/35% (top) and 80/20% (bottom) when HTHs are triggered.

ML Model (65/35%)	Acc.	κ
Dummy Classifier	0.5700	0
LR with Regularization	0.9503	0.8971
SVM (Radial Basis Function)	0.9473	0.8898
DT (algorithm C5.0)	0.9834	0.9661
Random Forest	0.9894	0.9779
Gradient Boosting Machine	0.9894	0.9784
ML Model (80/20%)	**Acc.**	**κ**
Dummy Classifier	0.5700	0
LR with Regularization	0.9578	0.9124
SVM (Radial Basis Function)	0.9526	0.9016
DT (algorithm C5.0)	0.9907	0.9810
Random Forest	0.9947	0.9890
Gradient Boosting Machine	0.9934	0.9864

Table 4: ML performance results for splits: 65/35% (top) and 80/20% (bottom) when HTHs are not triggered.

ML Model (65/35%)	Acc.	κ
Dummy Classifier	0.5700	0
LR with Regularization	0.9368	0.8671
SVM (Radial Basis Function)	0.9451	0.8850
DT (algorithm C5.0)	0.9864	0.9718
Random Forest	0.9872	0.9729
Gradient Boosting Machine	0.9887	0.9763
ML Model (80/20%)	**Acc.**	**κ**
Dummy Classifier	0.5700	0
LR with Regularization	0.9578	0.9119
SVM (Radial Basis Function)	0.9578	0.9122
DT (algorithm C5.0)	0.9894	0.9777
Random Forest	0.9921	0.9834
Gradient Boosting Machine	0.9960	0.9917



(a) GBM

(b) C5.0

Figure 4: Variable importance

Table 5: Performance metrics of the classifiers on the 80/20 train/test split when the HTHs are triggered.

Metric	LR	SVM	C5.0	RF	GBM
κ	0.9124	0.9016	0.9810	0.9890	0.9864
Sensitivity	0.9671	0.9523	0.9921	0.9945	0.9984
Specificity	0.9760	0.9687	0.9807	0.9932	0.9869
AUC-ROC	0.9919	0.9921	0.9956	0.9998	0.9986

Table 6: Performance metrics of the classifiers on the 80/20 train/test split when the HTHs are not triggered.

Metric	LR	SVM	C5.0	RF	GBM
κ	0.9119	0.9122	0.9777	0.9834	0.9917
Sensitivity	0.9320	0.9406	0.9906	0.9898	0.9929
Specificity	0.9723	0.9651	0.9947	0.9963	0.9984
AUC-ROC	0.9904	0.9936	0.9975	0.9999	1.0000

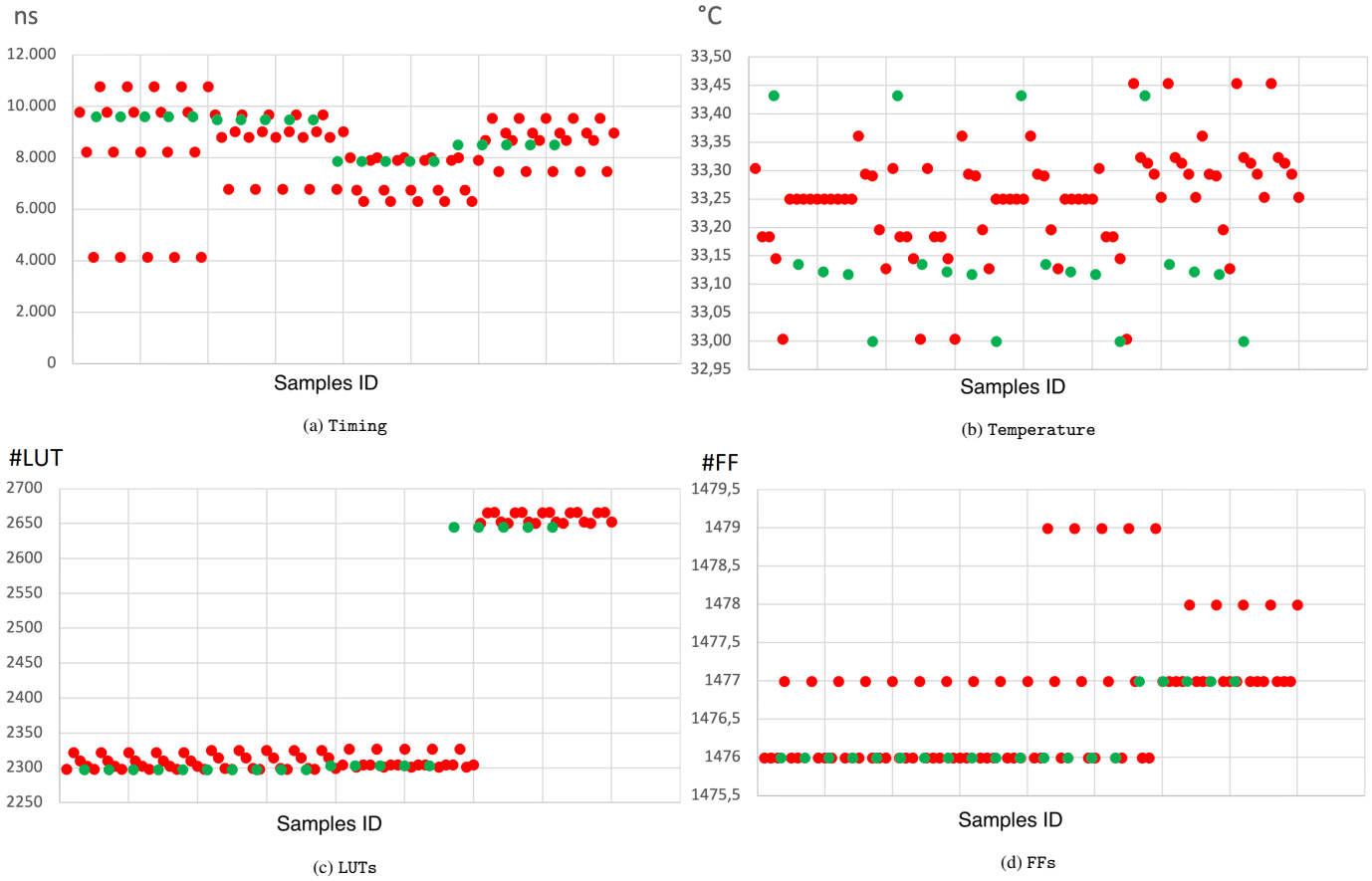
that the expected accuracy E is 57%, since this is the result that a dummy classifier would score by just assigning the majority class to all data points (remark that the HTH class represents 57% of the dataset).

From the table it can be noticed that the best models are C5.0, RF and GBM, which achieve accuracy values above 98% in both *triggered* and *untriggered* scenarios. However, such results have to be compared against the dummy classifier which is 57% accurate. Such comparison is provided by the κ parameter, which takes into account both the Observed and the Expected accuracy values. In this case, the κ coefficient for the three winning models is always above 0.97, which is a very high value representing the actual effectiveness of the three ML models identified as the best ones (the baseline κ provided by the dummy classifier is zero).

The tables also reveal several interesting observations: first, we can see that the considered ML models achieve very high and comparable accuracy both in the case of *triggered* and *untriggered* HTHs. This is because many of the considered features are only affected by the presence of the HTH and not by its activation. Moreover, the three winning models are quite robust in the sense that, after trained with only 65% datasamples, they are still capable of extracting the patterns and producing good performance results (κ values above 0.97) in all cases. When trained with some more data samples (80/20% case), the accuracy and κ results are even improved, reaching nearly perfection at HTH classification (i.e. accuracy values of up to 99% and κ about 0.98). We further deepened our analysis by looking more in detail at the results achieved by the identified best ML models. Tables 5 and 6 summarize the most widely used ML performance metrics for the ML models in the *triggered* and *untriggered* scenarios, respectively. As observed in the table, the GBM model should be considered as the final winner as it outperforms all others in most of the ML performance metrics.

As an additional analysis, we can further rank the considered features based on their importance for the employed ML model. Indeed, by analyzing different metrics provided by the ML model itself it is possible to identify which of the considered features contributed the most to the classification task and model building. Thus, by selecting the most relevant features the designer can optimize the employed ML model both in terms of accuracy and efficiency. Both GBM and C5.0 models offer a methodology to rank the most relevant features, computed as the amount that each attribute contributes to improving some classification performance metric. The identified feature ranking for GBM and C5.0 is shown in Figure 4. As shown, `Timing` and `AvgDynPow` are the most relevant features, while features like `DvdWait` and `MulWait` contribute very little to model building. Concerning the C5.0 model, the model only uses `Timing`, `AvgDynPow`, `LUTs` and `AvgPow` as critical features, the other ones are not relevant at all (zero contribution).

Finally, in order to further motivate the need for ML models for the task of distinguishing among trojan-free and trojaned circuits, in Figure 5 we show the dispersion diagrams of some of the considered features, namely: `Timing`, `Temperature`, `LUTs`, `FFs`. In the diagrams the green points represent the feature values for trojan-free circuits, while the red points are associated with the feature values for trojaned circuits. By looking at all these diagrams it clearly appears how there is no single feature where it is trivial to separate green points and red ones. This appears even clearer when looking at the boxplots



(a) `Timing` (b) `Temperature` (c) `LUTs` (d) `FFs`

Figure 5: Dispersion diagrams for some of the considered features. Green: golden samples; Red: trojaned samples

reported in Figure 6 where the predictive power of the `Timing`, `Temperature`, `LUTs`, `FFs` is graphically represented. Again, it clearly appears how difficult would be to define a threshold for perfect separation between classes. Indeed, in some cases like `LUT`, the two classes seem to overlap, while in others like `AvgPower`, there is a better separation between classes. These considerations motivate both the need for finding and optimizing ML models which, after combining all the features, provide a high-dimensional non-linear separation hyperplane for maximum classification accuracy between trojan-free and trojaned circuits.

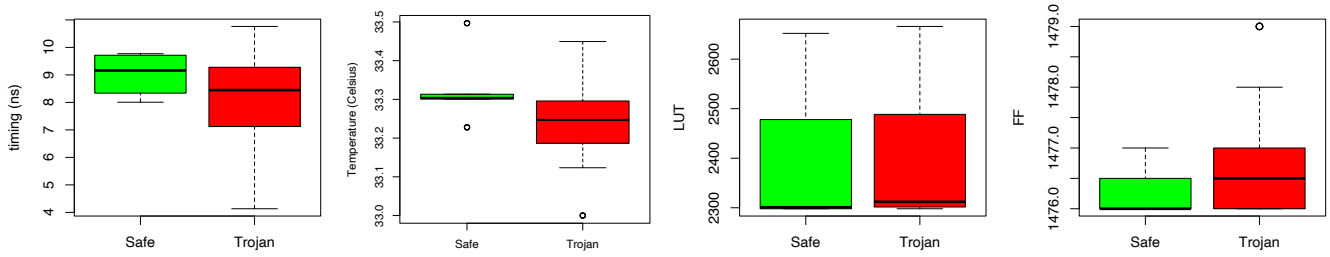
5. Conclusions and Future Work

We presented the first comparative analysis of the effectiveness of several ML models in detecting the presence of HTHs in the bitstream of microprocessor softcores meant to be implemented onto SRAM-based FPGAs. A detailed feature importance ranking has been discussed. The considered multi-parametric approach led to promising results and confirms literature trend to prefer utilization of several observable features. The exploitation of ML models confirms the generalisation potential of the proposed approach. Indeed, many of the considered ML models achieve accuracy values above 98% in the test set, and κ values above 0.97 both in the case where the injected HTH has been triggered during simulation and in the case where it remains untriggered. In the end, by identifying the most effective ML models and the best features to be employed, this paper lays the foundation for the integration of a ML-based bitstream verification flow.

Future work will be devoted to integrating process variation models into the proposed methodology in order to take into account fluctuations of the low-level features, e.g., temperature and average power consumption, between simulation (when training the detection methodology) and the real FPGA device (when verifying the downloaded bitstraem). Moreover, we will analyse how the effect of faults would affect the extracted features and, as a consequence, the detection capability of the proposed methodology.

6. Acknowledgements

J. A. Hernández and P. Reviriego acknowledge the ACHILLES PID2019-104207RB-I00 and 6G-INTEGRATION-3 TSI-063000-2021-127 projects and the Go2Edge RED2018-102585-T network funded by the Spanish Agencia Estatal de Investigación (AEI) 10.13039/501100011033 and the Madrid Community research project TAPIR-CM grant no. P2018/TCS-4496.



(a) `Timing` (b) `Temperature` (c) `#LUTs` (d) `#FFs`

Figure 6: Boxplots for some of the considered features

References

[1] Y. Jin, M. Maniatakos, Y. Makris, Exposing vulnerabilities of untrusted computing platforms, in: Proc. Int. Conf. Computer Design, 2012, pp. 131–134.

[2] N. G. Tsoutsos, M. Maniatakos, Fabrication attacks: Zero-overhead malicious modifications enabling modern microprocessor privilege escalation, IEEE Trans. Emerging Topics in Computing 2 (2014) 81–93.

[3] X. Wang, T. Mal-Sarkar, A. Krishna, S. Narasimhan, S. Bhunia, Software exploitable hardware trojans in embedded processor, in: 2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), IEEE, 2012, pp. 55–58.

[4] C. Domas, Hardware backdoors in x86 cpus, https://i.blackhat.com/us-18/Thu-August-9/us-18-Domas-God-Mode-Unlocked-Hardware-Backdoors-In-x86-CPUs-wp.pdf, 2018.

[5] project:rosenbridge, last access Feb. 2022. URL: `https://github.com/xoreaxeaxeax/rosenbridge`.

[6] J. A. Roy, F. Koushanfar, I. L. Markov, Extended abstract: Circuit cad tools as a security threat, in: 2008 IEEE International Workshop on Hardware-Oriented Security and Trust, 2008.

[7] M. Potkonjak, Synthesis of trustable ics using untrusted cad tools, in: Proceedings of the 47th Design Automation Conference, 2010, pp. 633–634.

[8] N. Fern, S. Kulkarni, K.-T. T. Cheng, Hardware trojans hidden in rtl don't cares — automated insertion and prevention methodologies, in: 2015 IEEE International Test Conference (ITC), 2015.

[9] W. Hu, L. Zhang, A. Ardeshiricham, J. Blackstone, B. Hou, Y. Tai, R. Kastner, Why you should care about don't cares: Exploiting internal don't care conditions for hardware trojans, in: 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2017.

[10] C. Pilato, K. Basu, F. Regazzoni, R. Karri, Black-hat high-level synthesis: Myth or reality?, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 27 (2018) 913–926.

[11] K. Basu, S. M. Saeed, C. Pilato, M. Ashraf, M. T. Nabeel, K. Chakrabarty, R. Karri, Cad-base: An attack vector into the electronics supply chain, ACM Transactions on Design Automation of Electronic Systems (TODAES) 24 (2019) 1–30.

[12] S. Sunkavilli, Z. Zhang, Q. Yu, New security threats on fpgas: From fpga design tools perspective, in: 2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2021, pp. 278–283.

[13] J. Zhang, G. Qu, Recent attacks and defenses on fpga-based systems 12 (2019).

[14] A. Duncan, F. Rahman, A. Lukefahr, F. Farahmandi, M. Tehranipoor, Fpga bitstream security: A day in the life, in: 2019 IEEE International Test Conference (ITC), 2019, pp. 1–10.

[15] M. Ender, P. Swierczynski, S. Wallat, M. Wilhelm, P. M. Knopp, C. Paar, Insights into the mind of a trojan designer: the challenge to integrate a trojan into the bitstream, in: Proceedings of the 24th Asia and South Pacific Design Automation Conference, 2019, pp. 112–119.

[16] S. Sunkavilli, Z. Zhang, Q. Yu, Analysis of attack surfaces and practical attack examples in open source fpga cad tools, in: 2021 22nd International Symposium on Quality Electronic Design (ISQED), 2021, pp. 504–509.

[17] X. Chuan, Y. Yan, Y. Zhang, An efficient triggering method of hardware Trojan in AES cryptographic circuit, in: Proc. Int. Conf. Integrated Circuits and Microsystems, 2017, pp. 91–95.

[18] J. Zhang, F. Yuan, L. Wei, Y. Liu, Q. Xu, Veritrust: Verification for hardware trust, IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems 34 (2015) 1148–1161.

[19] Y. Liu, Y. Zhao, J. He, A. Liu, R. Xin, Scca: Side-channel correlation analysis for detecting hardware trojan, in: Proc. Int. Conf. Anti-counterfeiting, Security, and Identification, 2017, pp. 196–200.

[20] H. Salmani, M. Tehranipoor, Analyzing circuit vulnerability to hardware trojan insertion at the behavioral level, in: Proc. Int. Symp. Defect and Fault Tolerance in VLSI and Nanotechnology Systems, 2013, pp. 190–195.

[21] H. Salmani, M. Tehranipoor, Layout-aware switching activity localization to enhance hardware trojan detection, IEEE Trans. Information Forensics and Security 7 (2012) 76–87.

[22] C. Dong, J. Chen, W. Guo, J. Zou, A machine-learning-based hardware-trojan detection approach for chips in the internet of things, International Journal of Distributed Sensor Networks 15 (2019) 1550147719888098.

[23] Z. Huang, Q. Wang, Y. Chen, X. Jiang, A survey on machine learning against hardware trojan attacks: Recent advances and challenges, IEEE Access 8 (2020) 10796–10826.

[24] D. Šišejković, F. Merchant, R. Leupers, G. Ascheid, S. Kegreiss, Controllock: Securing processor cores against software-controlled hardware trojans, in: Proceedings of the 2019 on Great Lakes Symposium on VLSI, GLSVLSI '19, 2019, pp. 27–32.

[25] D. M. Shila, V. Venugopalan, C. D. Patterson, Fides: Enhancing trust in reconfigurable based hardware systems, in: 2015 IEEE High Performance Extreme Computing Conference (HPEC), 2015, pp. 1–7.

[26] A. Basak, S. Bhunia, T. Tkacik, S. Ray, Security assurance for system-on-chip designs with untrusted ips, IEEE Transactions on Information Forensics and Security 12 (2017) 1515–1528.

[27] J. Dubeuf, D. Hély, R. Karri, Run-time detection of hardware trojans: The processor protection unit, in: 2013 18th IEEE European Test Symposium (ETS), 2013, pp. 1–6.

[28] G. Bloom, B. Narahari, R. Simha, Os support for detecting trojan circuit attacks, in: 2009 IEEE International Workshop on Hardware-Oriented Security and Trust, 2009, pp. 100–103.

[29] A. Nahiyan, K. Xiao, K. Yang, Y. Jin, D. Forte, M. Tehranipoor, Avfsm: A framework for identifying and mitigating vulnerabilities in fsms, in: 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), 2016.

[30] F. Farahmandi, P. Mishra, Fsm anomaly detection using formal analysis, in: 2017 IEEE International Conference on Computer Design (ICCD), 2017.

[31] K. Xiao, A. Nahiyan, M. Tehranipoor, Security rule checking in ic design, Computer 49 (2016) 54–61.

[32] W. Danesh, J. Banago, M. Rahman, Turning the table: Using bitstream reverse engineering to detect fpga trojans, Journal of Hardware and Systems Security 5 (2021) 237–246.

[33] I. Martin, J. A. Hernandez, S. de los Santos, Machine-learning based analysis and classification of android malware signatures, Future Generation Computer Systems 97 (2019) 295 – 305.

[34] I. Martín, J. A. Hernández, S. De Los Santos, A. Guzman, Analysis and evaluation of antivirus engines in detecting android malware: A data analytics approach, in: 2018 European Intelligence and Security Informatics Conference (EISIC), 2018, pp. 7–14.

[35] I. Martin, J. A. Herandez, A. Muñoz, A. Guzman, Android malware characterization using metadata and machine learning techniques, Security and Communication Networks 2018 (2018) 11.

[36] Z. Huang, Q. Wang, Y. Chen, X. Jiang, A survey on machine learning against hardware trojan attacks: Recent advances and challenges, IEEE Access 8 (2020) 10796–10826.
[37] K. Hasegawa, Y. Shi, N. Togawa, Hardware trojan detection utilizing machine learning approaches, in: 2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), IEEE, 2018, pp. 1891–1896.
[38] S. Kundu, X. Meng, K. Basu, Application of machine learning in hardware trojan detection, in: 2021 22nd International Symposium on Quality Electronic Design (ISQED), IEEE, 2021, pp. 414–419.
[39] M. Tehranipoor, F. Koushanfar, A survey of hardware trojan taxonomy and detection, IEEE Design & Test of Computers 27 (2010).
[40] F. Benz, A. Seffrin, S. A. Huss, Bil: A tool-chain for bitstream reverse-engineering, in: 22nd International Conference on Field Programmable Logic and Applications (FPL), 2012, pp. 735–738.
[41] B. Shakya, T. He, H. Salmani, D. Forte, S. Bhunia, M. Tehranipoor, Benchmarking of hardware trojans and maliciously affected circuits, Journal of Hardware and Systems Security 1 (2017) 85–102.
[42] M. Kuhn, K. Johnson, Applied predictive modeling, Springer, 2013.
[43] T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning, Springer Series in Statistics, Springer New York Inc., New York, NY, USA, 2001.
[44] S. B. Kotsiantis, I. Zaharakis, P. Pintelas, et al., Supervised machine learning: A review of classification techniques, Emerging artificial intelligence applications in computer engineering 160 (2007) 3–24.
[45] W. Snyder, Verilator and systemperl, in: North American SystemC Users' Group, Design Automation Conference, 2004.
[46] T. Feist, Vivado design suite, White Paper 5 (2012) 30.
[47] Ibex RISC-V Core, last access Feb. 2022. URL: `https://github.com/lowRISC/ibex/`.
[48] N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer, Smote: synthetic minority over-sampling technique, Journal of artificial intelligence research 16 (2002) 321–357.
[49] J. Cohen, A coefficient of agreement for nominal scales, Educational and Psychological Measurement 20 (1960) 37–46.