POLITECNICO DI TORINO
Repository ISTITUZIONALE

Machine Learning for Hardware Security: Classifier-based Identification of Trojans in Pipelined Microprocessors

*Original*
Machine Learning for Hardware Security: Classifier-based Identification of Trojans in Pipelined Microprocessors / Damljanovic, Aleksa; Ruospo, Annachiara; Sanchez, Ernesto; Squillero, Giovanni. - In: APPLIED SOFT COMPUTING. - ISSN 1568-4946. - ELETTRONICO. - 116:(2022), pp. 1-16. [10.1016/j.asoc.2021.108068]

*Availability:*
This version is available at: 11583/2941732 since: 2021-12-02T08:49:14Z

*Publisher:*
Elsevier

*Published*
DOI:10.1016/j.asoc.2021.108068

*Terms of use:*

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

*Publisher copyright*
Elsevier postprint/Author's Accepted Manuscript

© 2022. This manuscript version is made available under the CC-BY-NC-ND 4.0 license http://creativecommons.org/licenses/by-nc-nd/4.0/.The final authenticated version is available online at: http://dx.doi.org/10.1016/j.asoc.2021.108068

(Article begins on next page)

29 April 2026

# Machine Learning for Hardware Security: Classifier-based Identification of Trojans in Pipelined Microprocessors

Aleksa Damljanovic, Annachiara Ruospo, Ernesto Sanchez, Giovanni Squillero*

*Politecnico di Torino, Italy*

**Abstract**

During the last decade, the Integrated Circuit industry has paid special attention to the security of products. Hardware-based vulnerabilities, in particular Hardware Trojans, are becoming a serious threat, pushing the research community to provide highly sophisticated techniques to detect them. Despite the considerable effort that has been invested in this area, the growing complexity of modern devices always calls for sharper detection methodologies. This paper illustrates a pre-silicon simulation-based technique to detect hardware trojans. The technique exploits well-established machine learning algorithms. The paper introduces all the background concepts and presents the methodology. The validity of the approach has been demonstrated on the *AutoSoC* CPU, an industrial-grade, safety-oriented, automotive benchmark suite. Experimental results demonstrate the applicability and effectiveness of the approach: the proposed technique is highly accurate in pinpointing suspicious code sections. None of the hardware trojans from the set has been left undetected.

*Keywords:* Hardware Security, Machine Learning, Hardware Trojans, AutoSoC, Microprocessor Cores

*Corresponding author

*Email addresses:* `aleksa.damljanovic@polito.it` (Aleksa Damljanovic), `annachiara.ruospo@polito.it` (Annachiara Ruospo), `ernesto.sanchez@polito.it` (Ernesto Sanchez), `giovanni.squillero@polito.it` (Giovanni Squillero)

Authors are listed in alphabetical order.

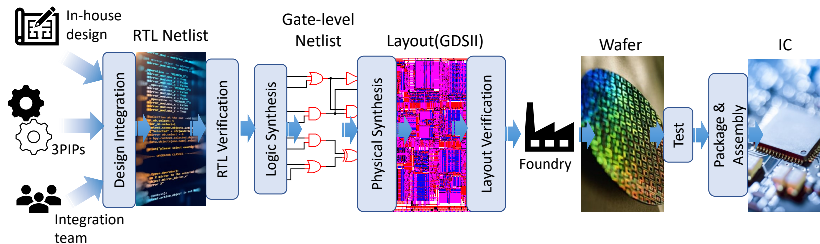



Figure 1: IC production flow

## 1. Introduction

The evergrowing complexity of modern devices and the fabrication costs led the Integrated Circuit (IC) industry to pursue a new global business model. In that regard, more companies around the world are deeply involved in all phases of the IC supply chain (Fig. 1). The outsourcing of part of the process to untrusted third-party entities raises increasing concerns about the hardware security of the products. The situation is becoming both critical and challenging and requires a careful regard.

Specific measures need to be taken for detecting, avoiding, and mitigating potential threats based on a component's importance. Since the security needs are driven by the evolving types of attacks, i.e., new adversary models, types, and intended use of the device, there is no solution that is able to provide complete protection. A common stance both in academia and industry is that the best approach is a set of flexible, technologically-driven solutions that are to be applied during the whole life-cycle of the device: development, deployment and operation.

Apart from detecting and localizing accidental bugs as a part of the design and production flow, it is necessary to identify intentionally placed malicious circuits. Different reports warn about such threats from both malicious and negligent actors and vulnerabilities they are able to exploit. Particularly, the so-called Hardware Trojans (HTs) continue to gain worldwide attention not only from industry (military) and academia, but also from government bodies [1].

A Hardware Trojan is defined as a malicious and intended alteration of a circuit, that endangers the trustworthiness and the security of the hardware, leading to unexpected behaviour. For instance, it may leak secret information, change the circuit functionality, or degrade the performance. A typical HT (*trigger activated*) is composed of a trigger and a payload circuit (Fig. 2).

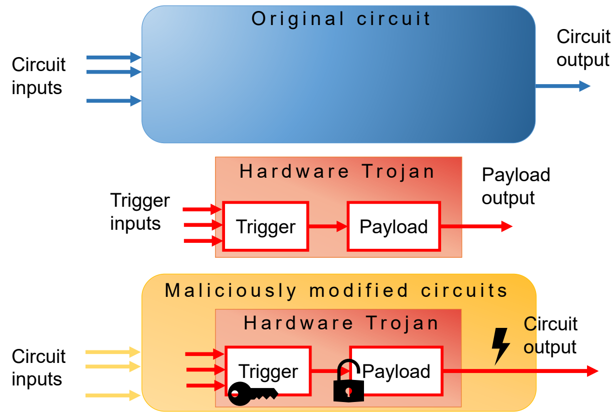



Figure 2: Hardware Trojan structure

The trigger usually monitors specific signals or series of events under some internal or external conditions. When the trigger condition is met, it informs the payload circuit, which executes the malicious function. The trigger is usually hidden under rare conditions, so the HT is dormant for most of the time and the payload inactive. In that case, the circuit acts as a Trojan-free circuit. If the activation does not depend on the trigger circuit, the Trojan belongs to another category, denoted as *always-on*. Such Trojan gets activated as soon as its host design is powered-on. Techniques to deal with the latter exist on different levels of abstraction in IC design: from logic-level search for sequentially-deep states to unexpected patterns of power consumption.

A malicious alteration can be performed during any phase of the production cycle. One category of Hardware Trojans are those inserted at the manufacturing stage. In such a scenario, an adversary could access the mask and modify it to add malicious logic. It may be supposed that such logic is inserted in an intelligent manner, difficult to activate with manufacturing tests given the combination of rare internal signal values that are used to trigger it. However, more interesting are HTs inserted earlier in the design cycle, at the Register Transfer Level (RTL) or into the gate-level netlist. Apart from superfluous complex reverse engineering from the attacker's point of view, HTs inserted early in the design process can potentially remain implanted even in the next generations of the device [2].

Researchers have faced the hardware-based security problems from several angles. The state-of-the-art detection techniques can be classified according

to different factors: the Trojan typology, insertion time, abstraction level, location, activation mechanism, effects, physical characteristics, the need for a golden model, etc.

As a result, providing a review of the work done on the subject is a rather challenging task. An interesting overview is provided in [1], where the authors summarize what has been covered and suggest a road-map for future research in this field. Interestingly, many methodologies utilizing Machine Learning (ML) for HT defence have emerged [3]. Among the existing ML-based techniques, Artificial Neural Networks (ANN) are commonly used for predictive analysis. Another commonly used ML approach for the problem of binary classification is Support Vector Machine (SVM). The success and popularity of ML methodologies in various research domains has motivated both industrial and academic communities to explore the potential of applying it to the hardware security field. In particular, the main progress in ML-based techniques has been achieved in three widely used detection methodologies: reverse engineering [4, 5], circuit feature analysis [6, 7], and side-channel analysis [8, 9].

We selected two different, yet well-known and widely adopted ML techniques for HT detection. The market size for Artificial Neural Networks is booming worldwide, with reports and successful stories from different domains; any approach claiming to exploit modern artificial intelligence needs to consider it. On the other hand, Support Vector Machine is a rock-solid supervised, learning algorithm — it has been defined the best "off-the-shelf" supervised learning algorithm [10] by Andrew Ng in his Stanford Lectures on Machine Learning. The former is best suited when a big amount of data is available, while the latter could be a better choice to cope with limited training sets.

Our proposed methodology is applied during the pre-silicon phase of the supply chain and is based on a deep learning analysis of the dynamic and static properties extracted from the design RTL model. The former properties are gathered by exciting the model, i.e., executing software code; the latter uniquely depends on the structure of the model and on the code/data dependency. As it will be explained in depth in Section 4, these two properties are jointly used to feed the ML model which then performs classification, i.e., calculates the probability of input samples belonging to the malicious insertion.

In brief, the main new contributions of our work are as follows:

- We propose a novel technique for detecting HTs in a pipelined microprocessor design at the RTL.
- Unlike the common approaches, this one combines both static and dynamic properties for building a comprehensive detection methodology at the pre-silicon stage, resorting to robust ML algorithms.
- We run experiments on a new set of benchmarks that prove the technique's efficacy and showing that this technique could be used with complex industrial designs, in an automatized manner, reducing both effort and time.
- The whole flow has been built and is adjustable for other commercial tools that can simulate the design and generate a code coverage report. Additional features can be introduced, while the format of inputs for the classification stage can also be modified.

The rest of the paper is structured in the following order. Section (Section 2) provides an insight into the field of hardware security. Furthermore, it offers an overview of the state-of-the-art regarding HT design and detection techniques. The third section 3 introduces HT benchmarks and ML methodologies that are used for the classification. Section 4 presents the technique for detecting RTL HTs relying on a machine learning-based approach. As stated previously, the approach is based on a set of features extracted from the design in both static and dynamic analysis. Section 5 reports on experimental results and demonstrate the efficacy of the technique. Finally, Section 6 draws some conclusions and highlights the future work directions.

## 2. Related Works

Understanding the SoC supply chain (Fig. 1) is the first necessary step for delineating the possible attack scenarios. In [1], the authors provide an interesting overview of the SoC development flow and all the entities involved. They identify three main phases: the *Intellectual Property (IP) Development*, the *SoC Integration* and the *Foundry*. The first one involves all the IPs providers. An SoC is typically comprised of more than one IP unit. To reduce research and development costs, some of them are built in-house, others bought from third-party IP vendors. Once all the IPs are available, they are joined to build an SoC. This phase is known as *SoC*

*Integration.* Both SoC designers and IP providers rely on EDA tools for facilitating the design process. At this point, all the side structures are already integrated into the SoC, for example, Design-For-Testability modules, Debug Units, and Built-In Self-Test blocks are typically entrusted to third-party specialized vendors. Once the SoC post-layout phase is done, it is sent to the foundry for the IC fabrication. The fabrication process is usually the most costly stage of the flow, thus is usually granted to external foundries. A malicious actor present in any stage can insert a HT at various levels of abstraction. The key issue lies in understanding which of these entities are trusted and which are not. Once it has been established, a threat model can be drawn. In [11] and [1], the authors provide a comprehensive list of adversarial models showing exactly when, where, and how a Trojan can be placed into an IC. The field of hardware security given the practically unlimited number and type of the attacks is quite vast. [12] systematizes the classification of threat models, state-of-the-art defenses, and evaluation metrics for important hardware-based attacks.

Apart from malicious insertions, side channel analysis and IC counterfeiting are another concern. In this context, a growing interest is given to Physical Unclonable Functions (PUFs) [13, 14]. PUF is a physical function that maps manufacturing variations of the circuit that occur in almost all physical systems to digital outputs. Though such variations can cause many different issues that can affect the function of the circuit, its lifetime, etc., they can also be used for security purposes. This is possible due to unique 'fingerprint' that cannot be intentionally reproduced in another circuit. In [15], authors propose a novel robust and low-overhead physical unclonable function (PUF) authentication and key exchange protocols that are resilient against reverse-engineering attacks. PUFs can be used for hardware obfuscation, prevention of reverse engineering and HT embedding, HW and SW IP protection [16]. As an example, in [17] authors propose a PUF construction method, named delay chains array PUF (DAPUF), to extract the unique power-up state for each chip which is corresponding to a unique key sequence. By introducing confusions between delay chains, they prevent the adversary from understanding the real functionalities of the circuit, thus making it difficult for the adversary to insert hard-to-detect Trojans.

As for the HT detection, related methodologies are developed both for pre-silicon and post-silicon phases and this can be considered as a principle classification. Post-silicon methods mainly rely on a side-channel analysis to measure circuit parameters such as current, operating frequency, power,

temperature, and radiation. However, if the modified, i.e., inserted circuit is small, the effects on the side-channel parameters could be negligible and the HT could escape detection. In [18], authors suggest generation of test patterns and combination of logic tests with side-channel analysis. It is worth adding that ML has been successfully applied for this purpose. In [8, 9], authors use ML as a side-channel detection methodology. Another non-destructive or noninvasive type of detection approaches is logic testing. Logic testing methods look for test vectors to activate the HT and then propagate triggered Trojans effect to some observable nodes [19]. On the other hand, most of the reverse engineering approaches are destructive (irreversibly change the physical properties of the device) and include scanning of exposed layers with scanning electron microscope (SEM) [4, 20, 21].

As for the pre-silicon detection approaches, two main classes can be identified here. Techniques in the first one exploit formal methods to prove the existence of malicious hardware, such as in [22]. The second class adopts simulation, structural analysis and functional tests generation to excite suspicious parts of the circuit where HTs can be hidden, e.g., [23, 24] and circuit feature analysis [6].

The proposed work describes a ML-based, pre-silicon methodology to detect HTs applicable at the RTL. Note that exploiting ML-based techniques is likely to imply the creation of models from the set of historical data that will later on be used for prediction [3]. Two different learning tasks can be identified: supervised learning and unsupervised learning. The former exploits labeled data to perform model training, while the latter focuses on relations between data when labels are unavailable. To better position this work, the following subsections introduce most relevant state-of-the-art HT design methodologies and detection techniques at the RTL.

### *2.1. HT Design*

Some works have focused on design possibilities and proposed certain methodologies to create new types of HTs. In [25], authors discuss design and implementation of RTL HTs to be hard to trigger and able to evade hardware trust verification based on unused circuit identification (UCI). They rely on a specific coding style and trigger input selection. Additionally, signal controllability is examined from the attacker's perspective. In [26], different implementations of HTs were explored with different combinations of triggers, payloads, as well as unique sections of the architecture that each HT

attacks. They were all designed with a varying level of sophistication, allowing the attacker to trade-off design time, ability to evade detection, and payload. The authors concluded that RTL designs can be quite vulnerable to hardware attacks given the vast insertion space and functional testing can often be useless in detecting them. Apart from introducing a metric for quantifying HT activation and effect, [27] introduces a vulnerability analysis flow for determining hard-to-detect areas and provides public trust benchmarks. Some works proposed automatic techniques (malicious CAD tool) for HT insertion. To generate HTs using a highly configurable generation platform, authors in [28] use transition probability to identify rarely activated internal nodes to target for HT insertion, rather than functional simulation as in existing platforms. The platform has been tested to generate HT-infected circuits and then evaluated by the ML detection technique [29] — the Controllability and Observability for HT Detection (COTD).

### *2.2. Detection Techniques*

Methodologies for detecting triggered-type HTs at the RTL can be broadly classified as *dynamic* and *static*. The former considers the adoption of verification test patterns and dynamic type of analysis based on, for instance, code coverage metrics. On the other hand, static techniques rely exclusively on static proprieties of the target RTL model, without applying any stimuli. As regards the first class (dynamic), one of the first approaches dates to 2010, when Hicks et al. [30] presented *BlueChip*, a hybrid design time/runtime system for detecting and neutralizing malicious circuits at RTL. BlueChip is based on the assumption that a part of the circuit is dormant during design verification and could therefore hide a HT. The UCI technique is able to flag a part of the circuit as suspicious and deactivate it by raising an exception when it becomes active. Its weakness has been demonstrated in [31], showing a class of HTs that evade detection. Although UCI technique may be able to discover many of the HTs shown in literature, it is sensitive to the actual coding style. From another perspective, authors in [32] describe a framework for generating directed test cases to activate HTs. It mixes concrete simulation and symbolic execution. The results are compared with EBMC[1], a state-of-the-art formal model checker, and demonstrate good scalability on large designs. A similar approach is presented in [33], where authors pro-

[1]http://www.cprover.org/ebmc/

posed an automated test generation technique for activating multiple targets in RTL models by the means of concolic testing.

Concerning the static approaches, in [34], authors exploited a subgraph isomorphism algorithm for HT detection inside an RTL model. Resorting to a static predefined library of known HTs, the algorithm searches for the occurrence of similar structures inside the Control Flow Graph (CFG) of the device under verification. A CFG is a representation in the form of a graph of all the paths in the RTL model that might be traversed during the execution. However, this approach produces a considerable number of false positives. To overcome this limitation, in [35] the authors combined it with a classifier based on a Probabilistic Neural Network, i.e., a feed-forward neural network usually used for classification tasks [36]. Even though the number of false positives has been greatly reduced, the main drawback is still the difficulty of finding the HTs that were not included in the pre-defined library.

## 3. Preliminaries

The following section introduces four important concepts that underlie the proposed work. First, the HT benchmarks that have been used for validating the approach are described in Section 3.1. Then, fundamental characteristics of digital design verification are introduced (Section 3.2); this background knowledge is useful for describing the dynamic analysis used in our detection methodology. However, it may be superfluous for the readers that possess some basic knowledge on this topic. Next, an overview of Artificial Neural Networks is provided in Section 3.3 and finally, the Support Vector Machine technique is presented in Section 3.4. Readers that already have a basic understanding of ML may safely skip the aforementioned subsections.

### *3.1. HT Benchmarks*

The set of benchmarks that is used for validating the proposed approach was first introduced in [37]. A total of 28 benchmarks are available[2], also referred to as RTL *trojan models*.

The proposed benchmarks are HTs conceived for a pipelined Central Processing Unit (CPU). Such Trojans are implanted into an individual Intellectual Property (IP) core of the System-on-Chip (SoC) and can affect only the

[2]https://github.com/ale-dam/HT-uP

specific IP in which they are embedded [3]. The benchmarks comply with the taxonomy and the classification scheme used by the research community [1, 27, 11]. Furthermore, the following attributes are outlined for each benchmark: abstraction level, insertion phase, location, activation mechanism, trigger, payload, and effect. For the sake of completeness, the insertion phase of the HTs is the design phase, while the abstraction level is the RTL for all introduced benchmarks. Concerning the effects, the benchmarks might prove to be disastrous or introduce minor damage. Three different payload categories can be identified:

1. **Degrade Performance (DP)**: The availability of the system under attack might not be affected, remaining fully operational. However, the HT might damage the performance of an IC and, in a worst-case, cause it to fail.
2. **Denial Of Service (DoS)**: The HT when activated stops all the activities of the system.
3. **Change the Functionality (CF)**: The HT alters the functionalities of the system, causing it to perform malicious, unauthorized operations. The CF might also lead to a DP or DoS.

The proposed RTL Hardware Trojans are implemented in the mor1kx CPU, whose architecture and HTs' respective faulty location are depicted in Fig. 15. The *mor1kx*[3] is an open-source core provided by the OpenRISC community; it is a configurable 32 or 64-bit load and store RISC architecture, written in Verilog Hardware Description Language (HDL). Due to the high design flexibility, it is possible to customize the core by choosing the best trade-off between area and performance. The version selected in this work (*Cappuccino*) has a pipeline with 4 stages, supports delay slot and is tightly coupled with the caches. It also integrates a Programmable Interrupt Controller (PIC), a Tick Timer (TT) and Debug units. In this work, HTs are injected in the original HDL design, one at a time, by directly modifying the RTL code.
On top of 8 primary HT designs, detailed in Table 1, we performed modifications concerning the complexity of trigger conditions and coding style to expand our benchmark library and to obtain an additional set of 20 HT designs. Table 1 reports all the essential details related to the benchmarks: their name, location, trigger description, payload effect and payload category.

[3]https://github.com/openrisc/mor1kx

Table 1: Trojan Benchmarks Description

| Name | Location | Trigger description | Payload effect | Payload Category |
|---|---|---|---|---|
| OR1K-**T1**00 | Decode Unit | Sequence of instructions | Periodically forcing signal values | DP |
| OR1K-**T2**00 | Control Unit | Counters monitoring read accesses to SPRs | Entering the supervisor mode | DoS |
| OR1K-**T3**00 | PIC Unit | Counters for mask and status reg. write access | Disabling external interrupts | CF |
| OR1K-**T4**00 | Control Unit | 3 counters for monitoring instructions | Disabling control flag bit | CF |
| OR1K-**T5**00 | Decode Unit | A specific sequence of instructions | Introducing "bubbles" to stall the pipeline | DP |
| OR1K-**T6**00 | Data Cache | Counters monitoring Data Cache Final State Machine (FSM) transitions | Invalidating dcache content | DP |
| OR1K-**T7**00 | Load & Store Unit | Instruction type, order and number | Exception on the data bus | DoS |
| OR1K-**T8**00 | Instr. Cache | Counters monitoring Instr. Cache FSM transitions | Invalidating icache content | DoS |

To give an example, let us consider the HT *T200*. It is located in the control unit of the processor. Counters are used to monitor read and write access of special purpose registers (Fig. 3). With each access, a corresponding counter is incremented. When all of the counters reach pre-defined values, a trigger is activated (Fig. 4). The payload in this case is integrated into the existing code by adding a single OR condition to go from user to supervisor mode (Fig. 5). Such behaviour is typical when an exception occurs. As a result, interrupts and timer exceptions are disabled, as well as Data and Instruction MMU. Additionally, once the device is in supervisor mode, access to some sensitive registers is enabled, which in some safety-critical

```
always @(posedge clk `OR_ASYNC_RST)
  if (rst) begin
    counter1 = 0;
    counter2 = 0;
    counter3 = 0;
    counter4 = 0;
    counter5 = 0;
    counter6 = 0;
    counter7 = 0;
    counter8 = 0;
    counter9 = 0;
    counter10 = 0;
    counter11 = 0;
  end
  else if (spr_we | spr_read) begin
     if (spr_access[`OR1K_SPR_SYS_BASE]) begin
     if (`SPR_OFFSET(spr_addr)==`SPR_OFFSET(`OR1K_SPR_CPUCFGR_ADDR))
            counter1 = counter1 + 1;
         else if (`SPR_OFFSET(spr_addr)==`SPR_OFFSET(`OR1K_SPR_EPCR0_ADDR))
            counter2 = counter2 + 1;
         else if (`SPR_OFFSET(spr_addr)==`SPR_OFFSET(`OR1K_SPR_SR_ADDR))
            counter3 = counter3 + 1;
     end
     else if (spr_access[`OR1K_SPR_DC_BASE]) begin
    if (`SPR_OFFSET(spr_addr)==`SPR_OFFSET(`OR1K_SPR_DCCR_ADDR))
           counter4 = counter4 + 1;
     end
     else if (spr_access[`OR1K_SPR_PC_BASE]) begin
    if (`SPR_OFFSET(spr_addr)==`SPR_OFFSET(`OR1K_SPR_PCCR0_ADDR))
           counter5 = counter5 + 1;
        else if (`SPR_OFFSET(spr_addr)==`SPR_OFFSET(`OR1K_SPR_PCMR0_ADDR))
           counter6 = counter6 + 1;
     end
     else if (spr_access[`OR1K_SPR_PM_BASE]) begin
        if (`SPR_OFFSET(spr_addr)==`SPR_OFFSET(`OR1K_SPR_PMR_ADDR))
           counter7 = counter7 + 1;
     end
     else if (spr_access[`OR1K_SPR_PIC_BASE]) begin
    if (`SPR_OFFSET(spr_addr)==`SPR_OFFSET(`OR1K_SPR_PICMR_ADDR))
```

Figure 3: Trigger T200 counters

```
assign trojan_en = (counter1 == 10665 ) && (counter2 == 12122) && (counter3 == 4323) && (counter4 == 123) && (counter5 == 543) &&
                   (counter6 == 2112) && (counter7 == 5533) && (counter8 == 10101) && (counter9 == 9888) && (counter10 == 3424) &&
                   (counter11 == 7321)? 1 : 0;


always @(posedge clk `OR_ASYNC_RST)
  if (rst)
    trojan_en_r <= 0;
  else if (trojan_en & !trojan_en_r)
    trojan_en_r <= 1;

assign trojan_edge = trojan_en & !trojan_en_r;
```

Figure 4: Trigger T200 condition

applications may have serious consequences.

### *3.2. Digital Design Verification*

Digital systems are created by following a series of steps that comprise several intermediate design phases. Clearly, the lower the abstraction level, the higher the complexity of the resulting model. Identifying and removing

```
//==========================================================================
//  Enter the supervisor mode and disable interrupts and timer exceptions,
//  disable Data and Instruction MMU (as if an exception occurred)
//======================TROJAN_PAYLOAD=====================================
// Supervision register
always @(posedge clk `OR_ASYNC_RST)
  if (rst)
    spr_sr <= SPR_SR_RESET_VALUE;
  else if (exception_re | trojan_edge)
    begin
   // Go into supervisor mode, disable interrupts, MMUs
   spr_sr[`OR1K_SPR_SR_SM  ] <= 1'b1;
   if (FEATURE_TIMER!="NONE")
     spr_sr[`OR1K_SPR_SR_TEE ] <= 1'b0;
   if (FEATURE_PIC!="NONE")
     spr_sr[`OR1K_SPR_SR_IEE ] <= 1'b0;
   if (FEATURE_DMMU!="NONE")
     spr_sr[`OR1K_SPR_SR_DME ] <= 1'b0;
   if (FEATURE_IMMU!="NONE")
     spr_sr[`OR1K_SPR_SR_IME ] <= 1'b0;
       if (FEATURE_DSX!="NONE")
     spr_sr[`OR1K_SPR_SR_DSX ] <= ctrl_delay_slot;
   if (FEATURE_OVERFLOW!="NONE")
     spr_sr[`OR1K_SPR_SR_OVE ] <= 1'b0;
    end
  else if ((spr_we & spr_access[`OR1K_SPR_SYS_BASE] &
           (spr_sr[`OR1K_SPR_SR_SM] & padv_ctrl | du_access)) &&
          `SPR_OFFSET(spr_addr)==`SPR_OFFSET(`OR1K_SPR_SR_ADDR))
    begin
   spr_sr[`OR1K_SPR_SR_SM  ] <= spr_write_dat[`OR1K_SPR_SR_SM  ];

   spr_sr[`OR1K_SPR_SR_F   ] <= spr_write_dat[`OR1K_SPR_SR_F   ];

   if (FEATURE_TIMER!="NONE")
     spr_sr[`OR1K_SPR_SR_TEE ] <= spr_write_dat[`OR1K_SPR_SR_TEE ];
```

Figure 5: T200 payload

logic errors in a design is not a trivial task; in fact, nowadays, the development resources devoted to these tasks amount about 50%-60% of the total cost in the design process [38]. Actually, a series of verification processes are required intending to guarantee that the design model meets the expected specifications [39]. Very different methodologies have been developed to generate verification stimuli. The main possibilities range from manual verification techniques to formal verification techniques, including random and semi-random approaches. In particular, simulation-based methodologies try to completely exercise the current model of the device to uncover design errors. Briefly, a simulation-based verification process is composed of three basic elements: input data (also called a set of stimuli), the model of the device under evaluation (also called design or device under verification or DUT); and the response checker, which generates the pass/fail information regarding the current process by performing a comparison of the obtained

results against the expected ones. To qualify a set of stimuli, one of the most used methodologies is based on collecting a series of measurements obtained by computing the code coverage metrics during the simulation of the device while running the stimuli set. These metrics identify which code structures belonging to the circuit description are exercised by the set of stimuli, and whether the control flow graph corresponding to the code description has been thoroughly traversed. The structures exploited by code coverage metrics range from a single line of code to if-then-else constructs. Today CAD tools are able to measure, among others, the statement coverage, branch coverage, condition coverage, expression coverage, toggle coverage, and metrics based on Finite State Machine models [40].

*3.3. Artificial Neural Networks*

The origin of Neural Networks dates to 1950s, when the basic building block of modern neural networks, the perceptron, was first proposed [41]. Over the years, key theoretical discoveries and technological advances allowed this concept to evolve into a brand new field. The original perceptron contains a single input layer and an output node, as shown in Figure 6, and may implement a linear binary classifier. The input layer does not perform any computation and thus it is not included in the count of the number of layers in a neural network. Therefore, in modern terminology, the perceptron would be considered a single-layer network. It is worth underlying that modern neural networks are certainly built with more than one computational layer.

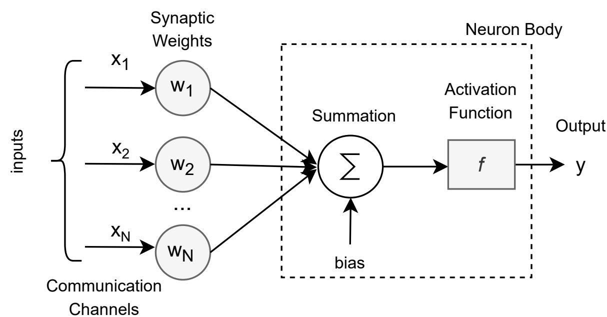



Figure 6: The basic architecture of the perceptron.

In more detail, the input layer of a perceptron contains $N$ nodes that transmit the $N$ features $\mathrm{X} = [\mathrm{x}_1 \ldots \mathrm{x}_\mathrm{N}]$ with edges of weight $\mathrm{W} = [\mathrm{w}_1 \ldots \mathrm{w}_\mathrm{N}]$ to an output node $y$. The prediction of the perceptron is computed as follows:

$$y = f(\sum_{j=1}^{N} w_j x_j + b) \quad (1)$$

In Eq. (1), $x_j$ are the inputs, $w_j$ the weights, b is the bias. The activation function ($f$) defines how the weighted sum of the input is transferred to the output node. The choice of $f$ is considered a critical part of neural network design since it has a large impact on the capability and performance of the neural network. The interpretation of the perceptron as a computational unit is useful, and it allows us to combine multiple units (i.e., multi-layer perceptron) to develop far more efficient models [42]. Broadly speaking, Artificial Neural Networks are computing models composed of computing nodes, connected to one another through communication links (Fig. 7). Nodes are arranged in layers, at least one input layer, one intermediate (or hidden), and one output layer.

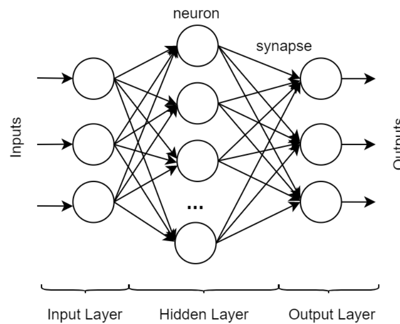



Figure 7: Artificial Neural Network: a basic representation.

Nowadays, the conventional machine learning applications used for example to identify objects in images or transcribe speech into text, make use of techniques stemming from NN and labeled as "deep learning" [43]. Thanks to the multiple levels of representation, quite complex functions can be learned; nevertheless, in the building blocks of such structures it is still recognizable the old idea of perceptron.

Over the years, many different neural network architectures have been created depending on the layers and their organization, the activation functions, and many other exploited features. Among the most common and widespread types are: convolutional neural networks (CNNs) and residual neural networks (ResNet) for image classification and object detection tasks;

recurrent neural networks (RNNs) for tasks that involve sequential inputs such as speech and language. Recent studies have demonstrated that state-of-the-art neural networks can surpass human-level performance: for example, in [44] the authors achieved 4.94% top-5 test error on the ImageNet classification dataset, and the human-level performance was 5.1%, according to Russakovsky et al. [45]. The potential of this kind of deep and complex neural networks (e.g., ResNet [46]) reflects the fact that biological neural networks gain much of their power from depth.

### *3.4. Support Vector Machine*

An SVM, close to its current form, was described in [47] as a training algorithm that maximizes the margin between the training patterns and the decision boundary. It has been developed from the Statistical Learning Theory in the 1960's [48]. The goal of the SVM algorithm is to define an optimal separating hyperplane for a two-class dataset. SVM tries to maximize the width of margin between the so-called support vectors, that is, training samples that lie closest to the separating hyperplane (Figure 8).

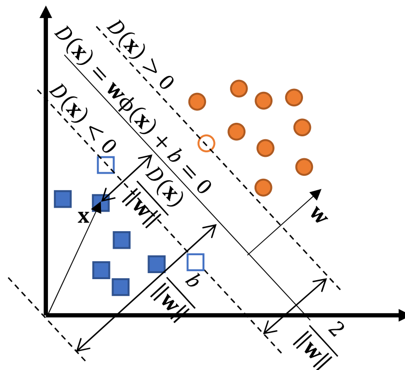



Figure 8: Defining a border between classes using an SVM (support vectors are marked with □ and ∘)

Training input for the system can be represented as a set of $r$ elements: $\{(\mathbf{x_1}, y_1), (\mathbf{x_2}, y_2), (\mathbf{x_3}, y_3), \ldots, (\mathbf{x_r}, y_r)\}$, where $\mathbf{x_i}, i = 1, 2, \ldots, r$ represents a $n$-dimensional input sample vector with the corresponding response value $y_i, i = 1, 2, \ldots, r$ (2).

$$y_i = \begin{cases} 1, & \text{if } \mathbf{x} \in A \\ -1, & \text{if } \mathbf{x} \in B \end{cases} \tag{2}$$

After the training process, the class of the new input data vector $\mathbf{x}$ will depend on the decision function value, $D(\mathbf{x})$, in such a way that it represents a position below or under the hyperplane separating the two classes. This function can be expressed as a linear combination of parameters (3),

$$D(\mathbf{x}) = \sum_{j=1}^{n} w_j x_j + b = \mathbf{w}\mathbf{x} + b \tag{3}$$

where $w_j$ are coefficients, $\mathbf{x}$ is input vector and $b$ is a bias coefficient. Conditions for discrimination between input sample $\mathbf{x_i}$ being on one (4) or the other side (5) of the hyperplane, can be unified into a single condition (6).

$$\mathbf{w}\mathbf{x_i} + b \geq 1, y_i = 1 \tag{4}$$

$$\mathbf{w}\mathbf{x_i} + b \leq -1, y_i = -1 \tag{5}$$

$$y_i(\mathbf{w}\mathbf{x_i} + b) \geq 1 \tag{6}$$

$$\mathbf{w}\mathbf{x}_+ + b = 1 \tag{7}$$

$$\mathbf{w}\mathbf{x}_- + b = -1 \tag{8}$$

$$\mathbf{w}(\mathbf{x}_+ - \mathbf{x}_-) = 2 \tag{9}$$

$$M = \frac{\mathbf{w}}{\|\mathbf{w}\|}(\mathbf{x}_+ - \mathbf{x}_-) = \frac{2}{\|\mathbf{w}\|} \tag{10}$$

Margin M is defined using a difference (9) of two samples $\mathbf{x}_+$ (7) and $\mathbf{x}_-$ (8) lying on two boundaries. The objective of this algorithm is finding the coefficient vector $\mathbf{w}$ in order to maximize the margin $M$ (10). To summarize, the goal is minimizing $\frac{\|\mathbf{w}\|^2}{2}$ with the condition of correctly classifying all the points $y_i(\mathbf{w}\mathbf{x_i} + b) \geq 1$.

In many applications, constructing a hyperplane is not possible and will not result in successful classification of the input data. By applying the technique called "kernel trick", input space gets mapped into a higher dimensional, linearly separable feature space Fig. 9. Most commonly used complex kernels are polynomial, gaussian and sigmoid. Linear kernel is the simplest

one, corresponding to the dot-product between two vectors and is used for linearly separable data to construct a hyperplane. A linear operation in the feature space is equivalent to a nonlinear operation in the input space.

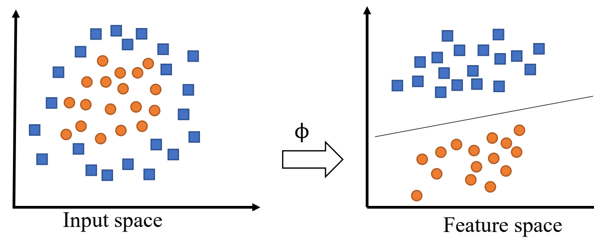



Figure 9: Using non-linear kernel functions to map input space

$$K(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y}$$
$$K_p(\mathbf{x}, \mathbf{y}) = (1 + \mathbf{x} \cdot \mathbf{y})^p$$
$$K_{rbf}(\mathbf{x}, \mathbf{y}) = \exp \frac{-\|\mathbf{x}^2 - \mathbf{y}^2\|}{2\sigma^2}$$
$$K_s(\mathbf{x}, \mathbf{y}) = \tanh(1 + \mathbf{x} \cdot \mathbf{y})^p$$

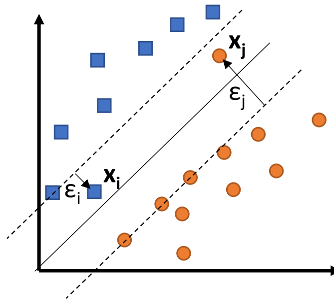



Figure 10: To prevent overfitting, avoiding narrow margin is recommended. This can sometimes be achieved by introducing a margin and allowing a certain degree of misclassification.

Having a non-linearly separable classification problem, a deviation can be introduced, using a parameter to adjust the desired error/margin. Such

parameter is usually referred to as probability threshold. Although we allow for a certain misclassification, the boundary margin is still retained (Fig. 10).

When SVM algorithm is used for classification the decision which kernel fits the best in practice is brought empirically; their performance is analyzed observing ROC curves, confusion matrix results, and the performance metrics associated with the confusion matrix [49]. This activity is commonly referred to as hyper-parameter tuning, and several machine-learning toolkits support it (e.g., the grid-search functionality in scikit-learn [50]).

Since the SVM is intrinsically a linear separator, when the classes are not linearly separable the data can be transformed into a high dimensionality space and with a high probability find a linear separation. This is stated by the Cover's theorem [51] and the RBF Kernel does exactly that: it projects the data into infinite dimensions and then finds a linear separation. Furthermore, the linear kernel can be used when there are a lot of features because it is likely that data are already linearly separable and an SVM will find the best separating hyperplane. Linear kernels also work very well with sparse data like text. On the other hand, when data are not linearly separable the rule of thumb is trying an RBF kernel first. The practical way to decide which kernel to use is by cross-validation.

## 4. Proposed Approach

The proposed methodology relies on a supervised learning scheme. It is necessary to underline that, apart from [35], most ML-based techniques are applied at the gate-level. However, more and more examples of HTs inserted at RTL are available, due to the flexibility for implementing various malicious functions. Hence, there is a pressing need for more RTL HT detection techniques. To fill the above-mentioned gaps, this paper presents a ML-based methodology for detecting triggered-type Hardware Trojans. It combines a *dynamic* approach with a *static* analysis of the RTL model. Indeed, if a static approach analyzes the structure of the model looking for a similarity with the structure of a Trojan, a dynamic method considers the true activity of the circuit. For this reason, the proposed work picks up the best of the two methods to cover a greater set of HTs and thus, generalize the detection approach.

The proposed flow is shown in Fig. 11. The input of the framework is the design that is about to be processed; it is the behavioural RTL model description. The output is a report indicating suspicious parts in the design,

i.e., the code fragments that should be checked more thoroughly for malicious HTs insertion. The RTL design is processed in order to extract both dynamic and static information. While the dynamic is derived from observing the model behavior under different stimuli, the static is obtained without any code execution and is related to the structure and control/data dependency in the code. The data extracted from the RTL model are embedded in CFGs. Static/Dynamic data are used as *attributes* to create input samples out of node sets for the classification task. At the end, a ML-based binary classification is used for distinguishing between input samples originating from the CFGs. The proposed approach is based on the following steps:

- Control Flow Graph Extraction:
  1. Static Attributes
  2. Assign DataFlow Map
  3. Dynamic Attributes
- Data Formatting
- Classification

In the following subsections, each of the steps is described in a more detailed manner.

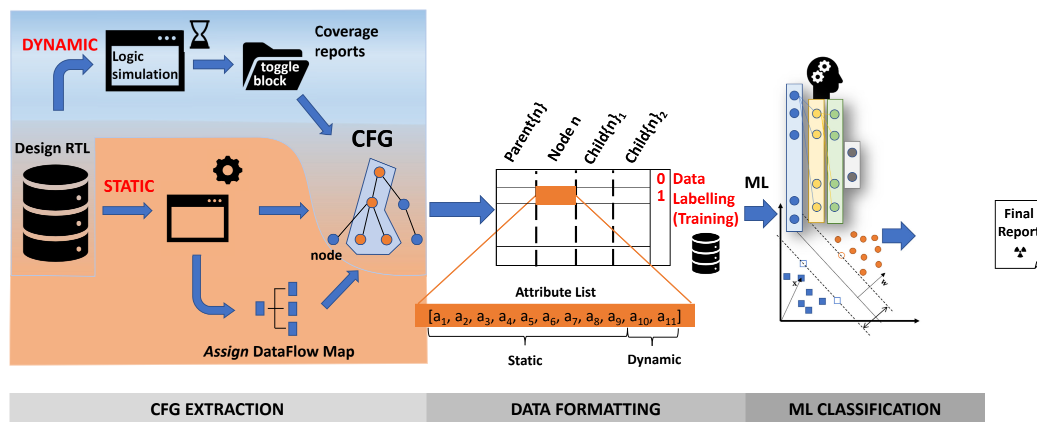



Figure 11: Detailed framework flow including three main steps: CFG extraction, Data formatting and ML classification

### 4.1. *Control Flow Graphs Extraction*

The RTL model of the design is described as a set of concurrent "processes". Two main hardware description languages are VHDL [52] and Verilog [53]. At the initial stage, the RTL design is represented in the form of a CFG, which incorporates key properties of the design: the static, dynamic, and dataflow map. These are essential for the training of the NN which is responsible for identifying malicious insertion in the code.

A CFG is a directed graph $G = (V, E, in, out)$, where $V$ is a set of vertices (nodes) and $E$ set of edges. For each process $P$ in the RTL design $D$, a CFG $G$ can be extracted. A node $v \in V$ of the graph $G$ can be:

- a single non-blocking statement – allow scheduling assignments without blocking the procedural flow;
- a conditional statement/loop (IF-ELSE, CASE, FOR, WHILE).

$E$ is a finite subset of $V \times V$; $e$ is an edge between the nodes $v1$, $v2$ if and only if $v2$ can be executed after $v1$ in the process $P$. $in$ and $out$ are the first and the last node in a CFG, respectively, used to mark entering the process and leaving the process. An example of the structure and its corresponding CFG are shown in Fig. 12. Then, each node in the CFG holds an attribute list, which will be created as described in the following.

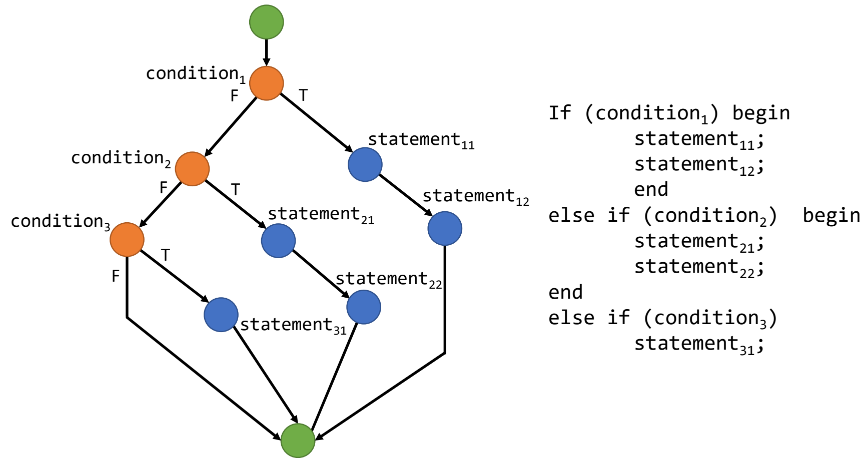



Figure 12: CFG with the corresponding code structure

#### *4.1.1.* ***Static Attributes***

The static attributes have been extracted from the RTL design by parsing the source code files. Given the complexity of modern designs, such task requires an automated tool. Usually, such tools provide as an output an abstract syntax tree (AST). AST is a convenient hierarchical tree-like representation of the abstract syntactic structure of source code. Then, syntax trees generated by the parser are traversed to perform the extraction of the CFGs in accordance with the definition that was introduced previously. It is worth noting that each of the source files may contain more than one process, which are all elaborated sequentially. The algorithm extracts the list of input signals, registers, wires, output signals, and parameters. A CFG node is identified by its unique name and a unique line number that get assigned inside the processes while creating nodes and attaching them to the corresponding graph. Since one node can represent either a conditional statement, i.e., a loop, or a non-blocking statement, it is possible to extract static properties from such constructs. These include the number of input signals, the number of output signals, the number of logic operators, relational and equality operators, arithmetic operators, and numbers (constants). Additionally, each node has its depth in the CFG (level - the number of edges in the path from the root to the node).

**Algorithm 1** Generating Assign DataFlow Map

**function** GENERATEASSIGNDATAFLOWMAP($gen$, $len$, $i$, $d$)
    $assignMap \leftarrow [\ ]$
    **while** $statement$is $assign$ **do**
        $L, R \leftarrow statement$
        **if** $L$ in $assignMap_{keys}$ **then**
            $assignMap[L][0] \leftarrow assignMap[L][0] + R_{attributes}$
            $assignMap[L][1] \leftarrow assignMap[L][1] \cup R_{signals}$
        **else**
            $assignMap[L] \leftarrow [\ [attributes(R), set(signals(R))]\ ]$

#### *4.1.2.* ***Assign DataFlow Map***

To deal with the combinational logic (e.g., the `assign` statements in Verilog), the proposed flow introduces an auxiliary structure. Creating an Assign DataFlow Map allows the information outside of the (sequential) processes to be captured and incorporated later into the CFGs. Left part of the assign

```
assign signal1 = input11 & !(input12 | input13) &
(counter1 == 121346) ? 1 : 0;
assign signal2 = (input21>output21) &
(counter2 == 314431) ? 1 : 0;
assign signal3 = (counter3 == 122214) ? 1 : 0;
assign signal12 = signal1 & signal2;
assign signal123 = signal12 & signal3;
```

Figure 13: Assign statements

```
signal1: [3, 0, 0, 1, 4, 0, 0, 3], {counter1, input11, input12, input13}
{counter1, input11, input12, input13}
[3, 0, 4, 1, 0, 0, 0, 3]

signal2: [1, 1, 1, 2, 0, 0, 0, 3],{counter2, input21, output21}
{counter2, input21, output21}
[1, 1, 1, 2, 0, 0, 0, 3]

signal3: [0, 0, 0, 1, 0, 0, 0, 3],{counter3}
{counter3}
[0, 0, 0, 1, 0, 0, 0, 3]

signal12: [0, 0, 1, 0, 0, 0, 0, 0], {signal2, signal1}
{counter1, counter2, input11, input12, input13, input21,
output21}
[4, 1, 5, 3, 0, 0, 0, 6]

signal123: [0, 0, 1, 0, 0, 0, 0, 0],{signal12, signal3}
{counter1, counter2, counter3, input11, input12, input13,
input21, output21}
[4, 1, 5, 4, 0, 0, 0 9]
```

Figure 14: Assign DataFlow Map

statement is used as a key to identify an item in such structure, while the corresponding value is in a form of a list. Its first element is an array of properties that coincide with the ones for the statements inside the process (*static attributes*). The second one is a list of used signals, either inputs (*input*), registers of integers (*reg*, *integer*) or other wires (*wire*). The map is searched recursively for all of its key elements, summing up the attributes for a corresponding signal list. It stops when there are no more wire signals, i.e., if the remaining ones are a register, integer, input or output. For example, in Fig. 14, for `signal123`, it adds the attributes of `signal12` and `signal3`, then it does the same for `signal12`, taking the attributes of `signal1` and `signal2`. On the other hand, `signal1`, `signal2`, and `signal3` do not con-

tain in their signal set any keys from the map entries. While creating the CFGs and extracting their nodes' static attributes, the influence that a signal present in Assign DataFlow Map has on a statement inside the process is taken into account by adding its attributes from the corresponding value in the map entry.

#### *4.1.3.* ***Dynamic Attributes***

Logic simulations of the design under assessment are performed to collect code coverage reports, based on standard metrics such as statement and toggle coverage. The idea is to gather information from a set of programs that thoroughly exercise the design under analysis. It is essential to outline that such set of programs may have been written either as a part of pre-silicon or post-silicon verification, validation, or even manufacturing tests etc., targeting different parts and different features of the system. For every instance in the design, the uncovered sequential statements belonging to a process are listed with their line number, source code, and type (*if* and *case* conditional structures, *for* and *while* loops together with non-blocking assign statements). The second type of reports focuses on toggle activity of the signals that are being used outside of sequential processes as inputs/outputs, to model combinational logic in assign statements. For each and every program in the library, a statement-coverage report is generated, while only one merged report for all runs regarding the signal toggling.

Hence, two additional fields have been created in the attribute list for such purpose: one for execution probability and one for signal toggling activity.

Regarding the former, a category is decided for each node (statement) based on the number of executions, i.e., how many times it was covered. This technique is an important tool for preparing numerical data for ML and is referred to as unsupervised discretization [54]. It consists of transforming data from continuous to discrete, using e.g., equally wide intervals. Typical use case is having many unique values to model effectively. In Eq. (11) that shows the range for deciding a category, $n_{exec}$ is a number of times a statement has been covered out of $M$ runs. $N$ is the number of intermediate categories, set to 5. Consequently, apart from the two extreme categories *never* (N) and *always* (A), there are other five: *almost never* (XS), *rarely* (S), *sometimes* (M), *often* (L), and *almost always* (XL).

$$i\frac{M}{N} \leq cat(n_{exec}) < (i+1)\frac{M}{N}, i \in \{0, 1, 2, \cdots, N\} \tag{11}$$

As for the latter, toggle reports from all of the runs are merged into a unique report, showing if a wire signal has toggled in at least one run, fully or partially (rise and fall). The algorithm embeds such information into a node belonging to a process statement in the following manner: wire signals are listed in such statement, if any, otherwise score 0 is set; based on their total number $t$ and the number of those that toggled $d$ a ratio $R = \frac{d}{t}$ is calculated; $R$ falls into one of the ranges, 0, $(0, \frac{1}{4}]$, $(\frac{1}{4}, \frac{2}{4}]$, $(\frac{2}{4}, \frac{3}{4}]$, $(\frac{3}{4}, 1)$, 1, and gets assigned a value from 6 to 1.

### *4.2.* ***Input Data Formatting***

By capturing the structural and functional dependency between the nodes in a CFG, the context and neighbourhood information is brought into the predictions. To do so, a node with its closest neighbours is selected to form a set, i.e., to obtain an input sample. Clearly, such sets may vary in size, given the bound that is chosen for grouping the nodes. It is desirable not to be too generic neither too specific, since this action will have an impact on the learning capabilities. For this reason, we have considered a set of 4 nodes. Therefore, each node that has at least one parent and at least one child is processed. For nodes with more than one parent $P$ and more than two children $C$, all the possible combinations are extracted $P \cdot \binom{C}{2}$. A child having no siblings is included in the selection two times. For all the CFGs, the algorithm implementing a set of above-mentioned rules extracts a set $S$ of node selections $t_i = (p_i, n_i, c_{1i}, c_{2i})$. Subsequently, by expanding its nodes with their incorporated attributes gets transformed into $t_a i = (a(p_i)[\,], a(n_i)[\,], a(c_{1i})[\,], a(c_{2i})[\,])$. For the training such input data have to be labeled relying on the set of Trojan Benchmarks introduced in [37]. If a central node $n_i$ for which we select its environment belongs to the malicious insertion then, the set of 4 nodes is marked as positive. Otherwise, it is marked as negative, i.e., non-suspicious.

### *4.3.* ***Classification***

Once the data have been extracted, the problem may be tackled as a pure Machine Learning classification problem. The learning phase, i.e., the training process, relies on the features obtained from the data formatting. Here, we apply different paradigms to perform the classification and confront their performance in the following subsections. The first one is using SVM algorithm, while the second is based on a fully connected feed-forward neural network.

*4.3.1. Classification with Support Vector Machine*

SVM algorithm is used with different kernels to choose the one that fits best for the problem in question. Often the differences in the scales across input variables may affect the training process and therefore the final result. A model might become unstable, meaning that it would suffer from poor performance in both learning and validation/test phases as a result of high sensitivity to input data and higher generalization error. Therefore, using pre-processing techniques such as scaling or normalizing input data is preferred when working with many ML algorithms. Normalization is a scaling of the data from the original range so that all values are within the new range between 0 and 1. It can be performed on each individual data sample (row-wise) or across data features (column-wise). Standardization on the other hand, includes transforming data to change its distribution of values: the mean of the observed values becomes 0 and the standard deviation 1. For this particular purpose, we perform scaling across the features: $\overline{X} = \frac{X - \mu}{\sigma}$.

*4.3.2. Classification with an Artificial Neural Network*

Given the number of attributes, the number of inputs for a fully connected feed-forward neural network is set to 60, after expanding some of the features with one-hot-encoding. Following the common experience of machine learning experts, having too many layers when dealing with a limited number of training data (an order of magnitude of 1000 samples) may result in underfitting. Furthermore, the number of NN inputs is a limiting factor when defining the number of nodes in layers. Given the previous consideration as well as an empirical analysis, the following topology has been adopted: (64, *tanh*), (32, *tanh*), (32, *relu*), (2, *sigmoid*). For the sake of clarity, the first number indicates the number of neurons that constitutes the fully-connected layer, while the second parameter specifies the activation function, e.g., hyperbolic tangent, rectified linear unit, sigmoid.

For a fixed topology, tuning training parameters may significantly enhance the NN learning capabilities. Hence, K-fold cross-validation method is employed to find the best optimizer and select optimal parameters such as batch and number of epochs. One of the challenges faced in ML is memorizing the input samples, especially when having a small training dataset. However, the NNs have shown to be more resilient to such problem. In any case, to reduce the generalization error, i.e., to prevent overfitting, a Gaussian noise is added to the input. In this way, the training process is made

more robust.

## 5. Experimental Evaluation

### *5.1. Experimental Setup*

The selected platform is AutoSoC [55], an open-source SoC benchmark suite, conceived to serve the needs for standardization and benchmarking in the automotive area.

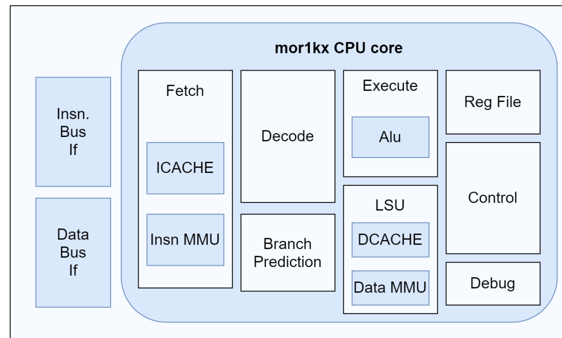



Figure 15: mor1kx CPU core in cappuccino configuration

For each one of the 28 benchmarks described previously, the following experimental procedure was used:

1. Parsing of the design model using a set of Python tools and an in-house developed tool to generate CFGs;
2. Performing the logic simulation and report generation using state-of-the-art commercial tools; then, adding the information originating from the coverage and toggle reports to the CFGs;
3. Node extraction: a selection of nodes with their neighborhood is made (parents and children) to create textual files whose rows contain the attributes for each of the 4 nodes. For the training process such data have to be labelled manually; repeating items, if any, are eliminated.

The whole setup has been developed to perform logic simulation and generate reports in Linux environment on a server equipped with a dual Intel Xeon CPU E5-2680 v3 and 256 GB of RAM. The process itself is managed by a set of bash scripts taking care of design elaboration, design simulation, calculating the coverage and merging the reports. Given the fact that for the

training process designs with different type and implementation of HTs have to be simulated multiple times, the time required for obtaining the reports can become significant. To speed-up the execution time, a multi-process environment has been developed. For this purpose, a library of test programs for *mor1kx* CPU has been simulated on all the 28 RTL *trojan models*. The test program library includes 46 programs for a total of 64 KB. Launching a set of 46 program simulations on one design in this configuration requires 22 minutes on average. By merging the contribution of each single program, the entire test program library achieves 85% of statement and toggle coverage on the golden design model (Fig. 16). It is worth underlying that the test program library is not able to activate the Hardware Trojans, being coherent with the assumption that HTs hide under rare trigger conditions.

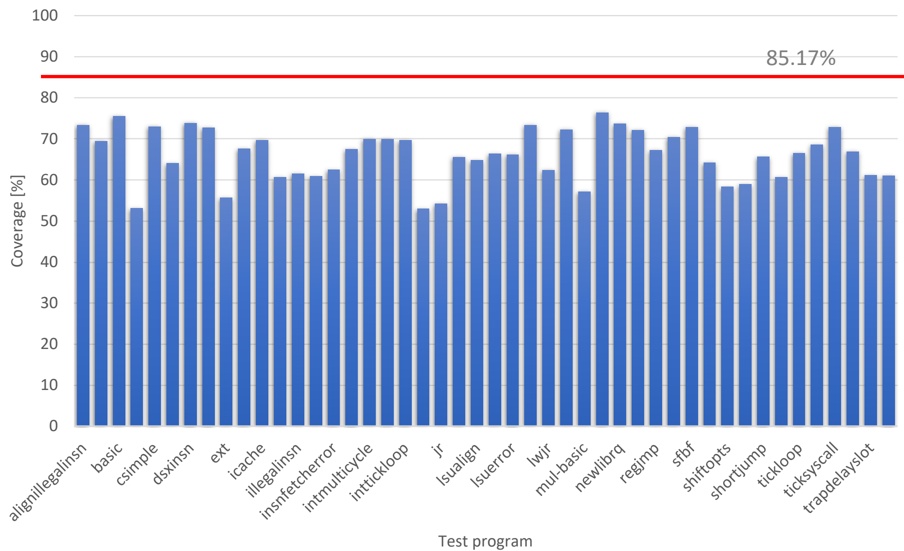



Figure 16: Individual coverage of each program on *mor1kx* core

In our approach, the tool for performing the task of parsing is Pyverilog [56]. It is a Python-based hardware design processing toolkit for Verilog HDL. The tool relies on Icarus,an open-source tool for performing the preprocessing. It flattens the hierarchy by implementing the `include` and `define` directives, producing the equivalent output related to such directives. Successively, Pyverilog reads the source code and generates Abstract Syntax Tree (AST) in the form of Python nested class objects. The parser is built

upon PLY[4] which is used as a parser generator (compiler-compiler). PLY is a Python implementation of the Lex-Yacc lexical analyzer.

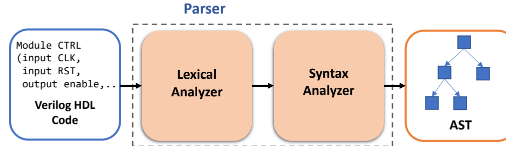



Figure 17: Pyverilog parser

### *5.2. Experimental results with Support Vector Machine*

The first set of experiments is intended to utilize SVM as a model to perform classification of code sections given in the form of attributes belonging to the family of nodes. Common practice when working with supervised learning and data classification is to split the data set into three exclusive sets: training set, validation set, and test set. However, by partitioning the available data into three sets, we drastically reduce the number of samples used for the learning phase. Consequently, such action might have a negative impact on the model's performance. Furthermore, the results can depend on a particular random choice when choosing/creating training and validation sets. A solution to this issue is using k-fold cross-validation. It consists in splitting the training set into $k$ smaller sets. The following procedure is followed for each of the k "folds": a model is trained using $k-1$ folds as input data for the training; the resulting model is validated on the remaining part of the data (i.e., it is used as a test set to compute a performance measure). Training/validation data and test data contain respectively, 80% and 20% of the complete data set.

The average recall, precision, accuracy and F1-score [57] were calculated on cross-validation sets with 10 folds for each of the four classifiers and reported in the first four columns of the Table 2. Subsequently, the model was trained on the whole training data set (80%), with a particular model configuration. Next, we examined the models' strength by applying test data that had not been used previously, i.e., the remaining 20% of the initial complete set.

Receiver Operating Characteristic (ROC) curve is a graphical plot showing the influence of the threshold margin on the performance of the binary

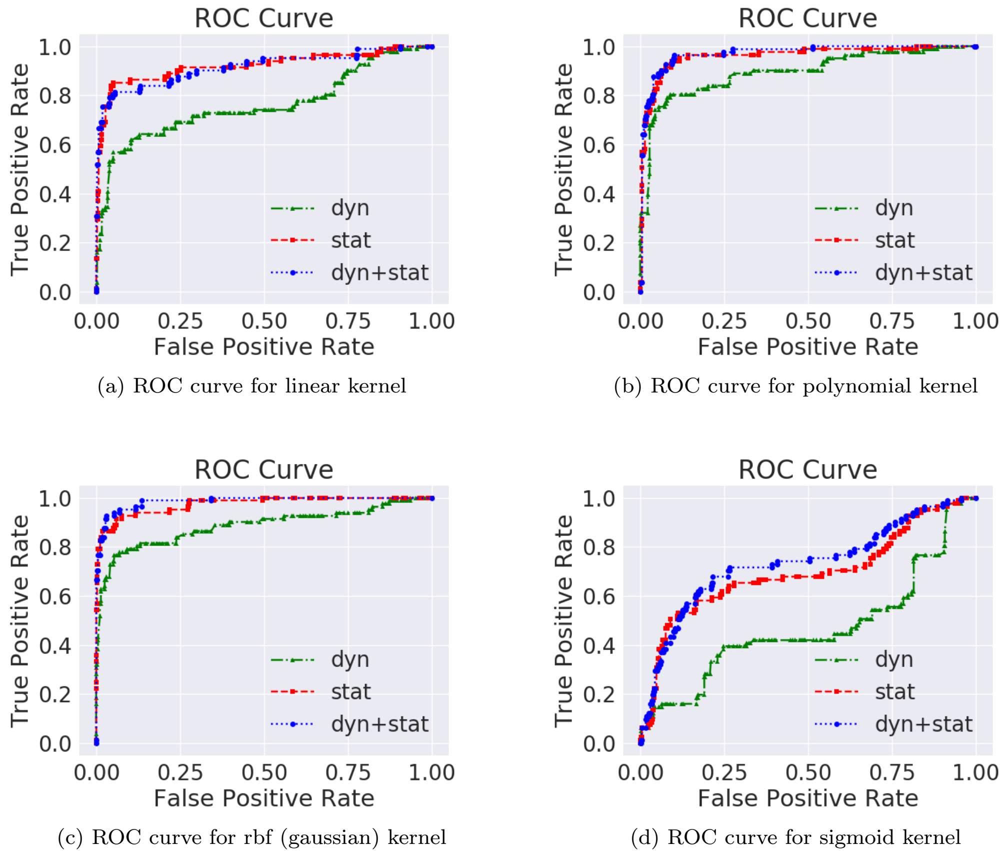
[4] `http://www.dabeaz.com/ply/`

Table 2: Experimental results of the four SVM classifiers

.

| | Cross-Validation $10 - fold$ | | | | Training [80%] | | Test [20%] | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Kernel | Rec. | Prec. | Acc. | F1-sc. | Rec. | Prec. | Rec. | Prec. | Acc. | F1-sc. | TN FP<br>FN TP |
| Linear | 0.79 | 0.60 | 0.87 | 0.69 | 0.80 | 0.64 | 0.81 | 0.61 | 0.87 | 0.70 | 314 42<br>15 66 |
| Polynomial | 0.49 | 0.90 | 0.90 | 0.63 | 0.57 | 0.97 | 0.64 | 0.95 | 0.93 | 0.76 | 353 3<br>29 52 |
| **RBF** | **0.82** | **0.81** | **0.93** | **0.82** | **0.88** | **0.90** | **0.91** | **0.87** | **0.96** | **0.87** | **345 11**<br>**7 74** |
| Sigmoid | 0.67 | 0.42 | 0.77 | 0.52 | 0.68 | 0.4 | 0.64 | 0.41 | 0.76 | 0.5 | 280 76<br>28 53 |

classifier system; it gives a trade-off between *sensitivity* (true positive rate) and *specificity* (1 - false positive rate). Classifiers with corresponding ROC curves closer to the top-left corner indicate a better performance. On the other hand, the closer the curve comes to the 45-degree diagonal of the ROC space, which is used as a baseline for the random classifier, the less powerful the classifier becomes. Four Receiver Operating Characteristic (ROC) curves for linear, polynomial, rbf and sigmoid kernels are given in Fig. 18. They provide enough information to analyze the predictive power of a classifier and find the optimal threshold. Based on the aforementioned analysis, the threshold was set to 0.19. Moreover, the RBF kernel was chosen as the best one in terms of performance when compared to the other 3. This claim can be supported by observing the numerical values in Table 2, where we report recall and precision on the training set, and successively, recall, precision, accuracy and F1-score on the test set, together with the corresponding confusion matrix. Additionally, here we decided to split the attributes extracted from the set of nodes and examine their partial influence on the performance of the classifiers. As shown in the Fig. 18, we performed training using exclusively static attributes (*stat*), then dynamic attributes (*dyn*) and finally, latter and former combined (*stat+dyn*). All of the classifiers clearly underperformed when relying only on the dynamic attributes. In case of the classifier with the RBF kernel, using the complete set of attributes instead of static attributes only resulted in improved classification power; in particular, 0.91 instead of 0.88 for recall, 0.87 instead of 0.8 for precision, 0.96 instead




(a) ROC curve for linear kernel

(b) ROC curve for polynomial kernel

(c) ROC curve for rbf (gaussian) kernel

(d) ROC curve for sigmoid kernel

Figure 18: ROC curves for 4 different kernels including different set of extracted attributes (farther from the 45-diagonal, i.e., closer to the upper-left corner, the better)

of 0.94 for accuracy and 0.87 instead of 0.84 for f1-score.

### *5.3. Experimental Results with Artificial Neural Networks*

The second set of experiments is related to training the NN and evaluating its performance. For selecting the parameters of the NN during the training process, exhaustive experiments were run using LazyGrid[5], an open-source package that eases hyper-parameters tuning and comparing different machine-learning models.

To evaluate the effectiveness of the proposed NN approach, eight different

[5] `https://github.com/glubbdubdrib/lazygrid`

Table 3: Experimental results of the NN

| Training Dataset $\cup T\backslash$ | Training performance TP | TN | FN | FP | Test Dataset ($n_{CFG}$) | Test performance TP | TN | FN | FP | FP rate [%] | Det. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $T_1*$ | 260 | 1781 | 42 | 8 | $T_1$(183) | **23** | 1493 | 7 | 1 | 1.1 | ✓ |
| | | | | | $T_{11}$(183) | **18** | 1493 | 6 | 1 | 1.1 | ✓ |
| | | | | | $T_{12}$(183) | **27** | 1493 | 9 | 1 | 1.1 | ✓ |
| | | | | | $T_{13}$(184) | **24** | 1493 | 7 | 1 | 1.1 | ✓ |
| | | | | | $T_{14}$(185) | **23** | 1493 | 8 | 1 | 1.1 | ✓ |
| $T_2*$ | 308 | 1764 | 13 | 14 | $T_2$(182) | **19** | 1490 | 15 | 5 | 0.1 | ✓ |
| | | | | | $T_{21}$(183) | **24** | 1491 | 16 | 6 | 0.1 | ✓ |
| | | | | | $T_{22}$(182) | **19** | 1493 | 11 | 5 | 0.1 | ✓ |
| $T_3*$ | 358 | 1769 | 22 | 11 | $T_3$(182) | **8** | 1487 | 1 | 10 | 0.3 | ✓ |
| | | | | | $T_{31}$(183) | **6** | 1489 | 2 | 10 | 0.3 | ✓ |
| | | | | | $T_{32}$(184) | **5** | 1490 | 7 | 12 | 0.3 | ✓ |
| $T_4*$ | 346 | 1770 | 25 | 16 | $T_4$(182) | **5** | 1485 | 9 | 10 | 0.3 | ✓ |
| | | | | | $T_{41}$(182) | **4** | 1487 | 10 | 10 | 0.3 | ✓ |
| | | | | | $T_{42}$(182) | **40** | 1494 | 11 | 10 | 0.3 | ✓ |
| | | | | | $T_{43}$(183) | **3** | 1487 | 11 | 10 | 0.3 | ✓ |
| $T_5*$ | 305 | 1770 | 16 | 13 | $T_5$(184) | **35** | 1487 | 7 | 8 | 1.4 | ✓ |
| | | | | | $T_{51}$(184) | **28** | 1489 | 6 | 8 | 1.8 | ✓ |
| | | | | | $T_{52}$(184) | **35** | 1491 | 8 | 13 | 1.8 | ✓ |
| | | | | | $T_{53}$(185) | **38** | 1489 | 7 | 8 | 1.8 | ✓ |
| $T_6*$ | 343 | 1641 | 30 | 9 | $T_6$(181) | **7** | 1489 | 9 | 5 | 1.2 | ✓ |
| | | | | | $T_{61}$(181) | **9** | 1492 | 5 | 5 | 1.1 | ✓ |
| | | | | | $T_{62}$(181) | **9** | 1486 | 15 | 5 | 1.3 | ✓ |
| $T_7*$ | 358 | 1781 | 32 | 7 | $T_7$(183) | **23** | 1494 | 2 | 3 | 1.0 | ✓ |
| | | | | | $T_{71}$(183) | **21** | 1496 | 2 | 1 | 1.0 | ✓ |
| | | | | | $T_{72}$(184) | **29** | 1496 | 4 | 1 | 1.1 | ✓ |
| $T_8*$ | 340 | 1656 | 34 | 5 | $T_8$(181) | **8** | 1490 | 13 | 2 | 0.4 | ✓ |
| | | | | | $T_{81}$(181) | **10** | 1493 | 8 | 2 | 0.4 | ✓ |
| | | | | | $T_{82}$(181) | **10** | 1493 | 8 | 7 | 0.5 | ✓ |

experiments have been conducted, one for each group of HTs. To determine how the NN will generalize for an independent data set, we used cross validation technique. In other words, the NN has been trained on a set completely independent from the test one. The results show that even though the NN

learns only on a category of HTs, it is able to discover different types as well.

In Table 3, we report for each training data set, the results obtained by evaluating the learning capabilities of the NN on the corresponding test sets. For a subset of benchmarks $T_{k*}$ that is used later for test, we first train the NN on the whole set of all benchmarks ($\bigcup T$) *excluding* that one particular subset $T_k$ and benchmarks derived from modifying it ($T_{k*}$). Confusion matrix terminology is used to present training and test performance given the predicted and expected classes for binary classification. The number of CFGs in a design (a set) is equal to the number of processes it contains. Finally, a false positive rate ($\frac{FP}{FP+TN}$) is given in the penultimate column of the table.

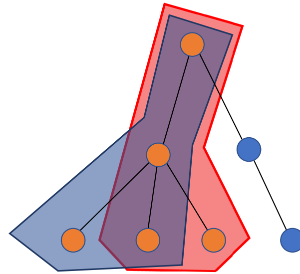


Figure 19: Set of nodes belonging to HT as TP and FN

| Classification | Explanation |
|---|---|
| True positive (TP) | Trojan code correctly recognized as malicious |
| True negative (TN) | Circuit code correctly considered safe |
| False positive (FP) | Safe circuit code believed to be malicious (i.e., a false alarm) |
| False negative (FN) | Malicious code that escaped detection (i.e., a major error) |

Table 4: Meaning of the confusion matrix in context of HT detection

Elements of the confusion matrix in context of HT detection are given in Table 4, with their corresponding explanation and the effect from the user's point of view. The number of FPs (a non-trojan detected as trojan) should ideally be 0, i.e., in practice it should be kept as low as possible, together with the FNs. However, the obtained numbers (FP and FN) are still significantly low, given the total number of samples that have been evaluated ($\sim$1.5k).

It is essential to outline that, first of all, the number of FPs remains significantly lower than the number of TNs, while being comparable to TPs. Therefore, checking all samples marked as positive (TPs + FPs), does not

represent a huge effort. Secondly, even though there are FNs, it does not mean some parts of malicious code escape the final analysis and remain undetected. As it can be seen from Fig. 19, a set of nodes marked in orange belongs to the HT (inserted malicious code), while those in blue are not. Those nodes covered in red polygon are detected as malicious, therefore enter in TP category, while those in blue polygon are left undetected, belonging to the FN. By revealing one, others can be examined and by tracing back all TPs, a verification engineer is able to completely discover all of maliciously inserted code. Thus, we can confirm that all of the Trojans in the test set have been discovered.

## 6. Conclusions

It has been drawn to the attention of the reader that approaches to detect HTs at the RTL are essential because the contamination of the design cycle is prevented. However, the literature review discovered only a few approaches that incidentally, work only for a specific type of HTs. In this paper, we have addressed the problem of detecting RTL HTs resorting to ML-based techniques in a pipelined CPU. A mixed approach consisting of static and dynamic model analysis is presented where robust machine learning algorithms are used to perform classification. Experimental results prove the technique's efficacy: no HT was left undetected, showing that this technique could be used with similar complex industrial designs, in an automatized manner, reducing both effort and time. The in-house tool was built and integrated into the whole flow to provide a fast and efficient analysis. It is adjustable for other commercial tools that can simulate the design and generate a code coverage report. Additional items and rules can be introduced for feature extraction, as well as different CFG node environments to create the classification input. The final flow processing of the input, given as an RTL behavioral model, includes logic simulation, CFG extraction and annotation, and input formatting. The final result of the evaluation is the list of suspicious locations in the code. By "out-of-sample" testing we showed that the NN method is able to identify all HTs embedded in a complex design aggravating the detection process. Additionally, we evaluated the performance of four different SVM classifiers. The one using RBF kernel was shown to generalize very well. Comparing two models in terms of performance, SVM RBF kernel is more successful in discovering the set of nodes that is marked as malicious and also takes less time to train. Nevertheless, both approaches

in the end detect each HT as an entity, following the discussion that a set of nodes might represent only one section of a HT. The relatively small amount of training data might be responsible for a poorer performance of the NN.

It should be acknowledged that there are certain limitations to this approach. Since it is both static and dynamic, it requires input data for the simulation, either high-level software code, e.g., C code, or directly on the hardware side, e.g., an RTL testbench. Nowadays, in the industrial practice of chip design and development, writing such (verification) programs can start early in the life-cycle, which makes it suitable for this application. Trojans evolve in structure and their location is unpredictable. A lot of effort is being invested into their classification and development. Since the supervised type of learning is used to train both SVM and ANN, it is uncertain how their classification performance will change with new types of HTs. However, such new malicious insertions may be included into the training set. Another limitation is the number of false positives. As it has been explained in the discussion of experimental results, false positives at the output of the classifier mean that a part of HT has not been identified as malicious. This does not mean that the whole Trojan escaped detection: it implies that some additional manual effort is required to decide. The efficiency of the approach nevertheless remains high, given the size and complexity of modern designs.

Future work will be focused on examining and extending the set of properties used for the analysis and validating the approach further with some other HTs.

## Acknowledgements

The work has been partially supported by the European Commission through the Horizon 2020 RESCUE-ITN project under the agreement No. 722325.

## References

[1] K. Xiao, et al., Hardware trojans: Lessons learned after one decade of research, ACM Trans. Des. Autom. Electron. Syst. 22 (2016).

[2] S. V. Pham, J. Dworak, An analysis of differences between trojans inserted at rtl and at manufacturing with implications for their detectability, 2012.

[3] R. Elnaggar, K. Chakrabarty, Machine learning for hardware security: Opportunities and risks, Journal of Electronic Testing 34 (2018) 1–19.

[4] C. Bao, D. Forte, A. Srivastava, On application of one-class svm to reverse engineering-based hardware trojan detection, in: Fifteenth International Symposium on Quality Electronic Design, 2014, pp. 47–54.

[5] W. Li, Z. Wasson, S. A. Seshia, Reverse engineering circuits using behavioral pattern mining, in: 2012 IEEE International Symposium on Hardware-Oriented Security and Trust, 2012, pp. 83–88.

[6] K. Hasegawa, M. Yanagisawa, N. Togawa, Trojan-feature extraction at gate-level netlists and its application to hardware-trojan detection using random forest classifier, in: 2017 IEEE International Symposium on Circuits and Systems (ISCAS), 2017, pp. 1–4.

[7] E. Zhou, et al., A novel detection method for hardware trojan in third party ip cores, in: 2016 International Conference on Information System and Artificial Intelligence (ISAI), 2016, pp. 528–532.

[8] S. Wang, et al., Hardware trojan detection based on elm neural network, in: 2016 First IEEE International Conference on Computer Communication and the Internet (ICCCI), 2016, pp. 400–403.

[9] Y. Liu, Y. Jin, A. Nosratinia, Y. Makris, Silicon demonstration of hardware trojan design and detection in wireless cryptographic ics, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 25 (2017) 1506–1519.

[10] Support Vector Machines, Andrew Ng, 2019. URL: `https://zkf85.github.io/public/cs229/cs229-notes3.pdf`.

[11] 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). doi:10.1007/s41635-017-0001-6.

[12] M. Rostami, F. Koushanfar, R. Karri, A primer on hardware security: Models, methods, and metrics, Proceedings of the IEEE 102 (2014) 1283–1295. doi:10.1109/JPROC.2014.2335155.

[13] Y. Gao, S. F. Al-Sarawi, D. Abbott, Physical unclonable functions, in: Nature Electronics, volume 3, 2020. doi:10.1038/s41928-020-0372-5.

[14] M. Rostami, J. B. Wendt, M. Potkonjak, F. Koushanfar, Quo vadis, puf?: Trends and challenges of emerging physical-disorder based security, in: 2014 Design, Automation Test in Europe Conference Exhibition (DATE), 2014, pp. 1–6. doi:10.7873/DATE.2014.365.

[15] M. Rostami, M. Majzoobi, F. Koushanfar, D. S. Wallach, S. Devadas, Robust and reverse-engineering resilient puf authentication and key-exchange by substring matching, IEEE Transactions on Emerging Topics in Computing 2 (2014) 37–49. doi:10.1109/TETC.2014.2300635.

[16] K. F. Rührmair U., Devadas S., Security based on physical unclonability and disorde, in: Introduction to Hardware Security and Trust, Springer, New York, 2012. doi:https://doi.org/10.1007/978-1-4419-8080-9_4.

[17] M. Xue, J. Wang, Y. Wang, A. Hu, Security against hardware trojan attacks through a novel chaos fsm and delay chains array puf based design obfuscation scheme, in: Z. Huang, X. Sun, J. Luo, J. Wang (Eds.), Cloud Computing and Security, Springer International Publishing, Cham, 2015, pp. 14–24.

[18] S. Bhunia, M. Hsiao, M. Banga, S. Narasimhan, Hardware trojan attacks: Threat analysis and countermeasures, 2014, pp. 1229–1247.

[19] M. Nourian, M. Fazeli, D. Hely, Hardware trojan detection using an advised genetic algorithm based logic testing, Journal of Electronic Testing 34 (2018). doi:10.1007/s10836-018-5739-4.

[20] C. Bao, D. Forte, A. Srivastava, On reverse engineering-based hardware trojan detection, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 35 (2016) 49–57. doi:10.1109/TCAD.2015.2488495.

[21] F. Courbon, P. Loubet-Moundi, J. J. Fournier, A. Tria, A high efficiency hardware trojan detection technique based on fast sem imaging, in: 2015 Design, Automation Test in Europe Conference Exhibition (DATE), 2015, pp. 788–793. doi:10.7873/DATE.2015.1104.

[22] M. Rathmair, F. Schupfer, C. Krieg, Applied formal methods for hardware trojan detection, in: 2014 IEEE International Symposium on Circuits and Systems (ISCAS), 2014, pp. 169–172.

[23] A. Waksman, M. Suozzo, S. Sethumadhavan, Fanci: Identification of stealthy malicious logic using boolean functional analysis, 2013, pp. 697–708. doi:10.1145/2508859.2516654.

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

[25] J. Zhang, Q. Xu, On hardware trojan design and implementation at register-transfer level, 2013, pp. 107–112. doi:10.1109/HST.2013.6581574.

[26] Y. Jin, N. Kupp, Y. Makris, Experiences in hardware trojan design and implementation, in: 2009 IEEE International Workshop on Hardware-Oriented Security and Trust, 2009, pp. 50–57. doi:10.1109/HST.2009.5224971.

[27] H. Salmani, M. Tehranipoor, R. Karri, On design vulnerability analysis and trust benchmarks development, in: 2013 IEEE 31st International Conference on Computer Design (ICCD), 2013, pp. 471–474.

[28] S. Yu, W. Liu, M. O'Neill, An improved automatic hardware trojan generation platform, in: 2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2019, pp. 302–307. doi:10.1109/ISVLSI.2019.00062.

[29] H. Salmani, Cotd: Reference-free hardware trojan detection and recovery based on controllability and observability in gate-level netlist, IEEE Transactions on Information Forensics and Security 12 (2017) 338–350. doi:10.1109/TIFS.2016.2613842.

[30] M. Hicks, et al., Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically, in: 2010 IEEE Symposium on Security and Privacy, 2010, pp. 159–172.

[31] C. Sturton, M. Hicks, D. Wagner, S. T. King, Defeating uci: Building stealthy and malicious hardware, in: 2011 IEEE Symposium on Security and Privacy, 2011, pp. 64–77.

[32] A. Ahmed, F. Farahmandi, Y. Iskander, P. Mishra, Scalable hardware trojan activation by interleaving concrete simulation and symbolic execution, in: 2018 IEEE International Test Conference (ITC), 2018.

[33] Y. Lyu, A. Ahmed, P. Mishra, Automated activation of multiple targets in rtl models using concolic testing, in: 2019 Design, Automation Test in Europe Conference Exhibition (DATE), 2019, pp. 354–359.

[34] L. Piccolboni, A. Menon, G. Pravadelli, Efficient control-flow subgraph matching for detecting hardware trojans in rtl models, ACM Trans. Embed. Comput. Syst. 16 (2017).

[35] F. Demrozi, R. Zucchelli, G. Pravadelli, Exploiting sub-graph isomorphism and probabilistic neural networks for the detection of hardware trojans at rtl, in: 2017 IEEE International High Level Design Validation and Test Workshop (HLDVT), 2017, pp. 67–73.

[36] D. F. Specht, Probabilistic neural networks, Neural Netw. 3 (1990) 109–118.

[37] A. Damljanovic, A. Ruospo, E. Sanchez, G. Squillero, A Benchmark Suite of RT-level Hardware Trojans for Pipelined Microprocessor Cores, 24th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS) (to be published) (2021).

[38] D. K. Pradhan, I. G. Harris, Practical Design Verification, Cambridge University Press, 2009. doi:10.1017/CBO9780511626913.

[39] A. Piziali, Functional Verification Coverage Measurement and Analysis, 1st ed., Springer Publishing Company, Incorporated, 2007.

[40] S. Tasiran, K. Keutzer, Coverage metrics for functional validation of hardware designs, IEEE Des. Test 18 (2001) 36–45. URL: https://doi.org/10.1109/54.936247. doi:10.1109/54.936247.

[41] F. Rosenblatt, The perceptron: A probabilistic model for information storage and organization in the brain., Psychological Review 65 (1958).

[42] C. C. Aggarwal, Neural Networks and Deep Learning, Springer, 2018. doi:10.1007/978-3-319-94463-0.

[43] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (2015) 436–444. URL: https://doi.org/10.1038/nature14539. doi:10.1038/nature14539.

[44] K. He, X. Zhang, S. Ren, J. Sun, Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, in: Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV), ICCV '15, IEEE Computer Society, USA, 2015, p. 1026–1034. URL: https://doi.org/10.1109/ICCV.2015.123. doi:10.1109/ICCV.2015.123.

[45] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, L. Fei-Fei, Imagenet large scale visual recognition challenge, International Journal of Computer Vision (IJCV) 115 (2015) 211–252. URL: https://doi.org/10.1007/s11263-015-0816-y. doi:10.1007/s11263-015-0816-y.

[46] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), IEEE, Las Vegas, NV, USA, 2016, pp. 770–778. URL: https://doi.org/10.1109/CVPR.2016.90. doi:10.1109/CVPR.2016.90.

[47] B. E. Boser, I. M. Guyon, V. N. Vapnik, A training algorithm for optimal margin classifiers, in: Proceedings of the Fifth Annual Workshop on Computational Learning Theory, COLT '92, Association for Computing Machinery, New York, NY, USA, 1992, p. 144–152. URL: https://doi.org/10.1145/130385.130401. doi:10.1145/130385.130401.

[48] V. N. Vapnik, The Nature of Statistical Learning Theory, Springer-Verlag, Berlin, Heidelberg, 1995.

[49] C. Savas, F. Dovis, The impact of different kernel functions on the performance of scintillation detection based on support vector machines, Sensors 19 (2019). URL: https://www.mdpi.com/1424-8220/19/23/5219.

[50] python, tuning the hyper-parameters of an estimator, scikit-learn (0.24.2). URL: https://scikit-learn.org/stable/modules/grid_search.html.

[51] T. M. Cover, Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition, IEEE Transactions on Electronic Computers EC-14 (1965) 326–334. doi:10.1109/PGEC.1965.264137.

[52] Ieee standard for vhdl language reference manual, IEEE Std 1076-2019 (2019) 1–673. doi:10.1109/IEEESTD.2019.8938196.

[53] Ieee standard for verilog hardware description language, IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001) (2006) 1–590. doi:10.1109/IEEESTD.2006.99495.

[54] J. Dougherty, R. Kohavi, M. Sahami, Supervised and unsupervised discretization of continuous features, in: A. Prieditis, S. Russell (Eds.), Machine Learning Proceedings 1995, Morgan Kaufmann, San Francisco (CA), 1995, pp. 194–202. URL: https://www.sciencedirect.com/science/article/pii/B9781558603776500323. doi:https://doi.org/10.1016/B978-1-55860-377-6.50032-3.

[55] F. Silva, et al., Special session: Autosoc - a suite of open-source automotive soc benchmarks, 2020, pp. 1–9. doi:10.1109/VTS48691.2020.9107599.

[56] S. Takamaeda-Yamazaki, Pyverilog: A python-based hardware design processing toolkit for verilog hdl, in: Applied Reconfigurable Computing, 2015.

[57] A. Géron, Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems, O'Reilly Media, 2019.