POLITECNICO DI TORINO

Department of Control and Computer Engineering (DAUIN)

Master's Degree Thesis

A FOSS-Based Toolchain for Automated Hardware Trojan Injection in RISC-V Architectures

Supervisor
Alessandro Savino

Co-Supervisors
Riccardo Cantoro
Samuele Yves Cerini

Candidate
Davide Giuffrida

Academic Year 2023/2024

Abstract

Digital electronics is one of the biggest revolutions in the history of mankind. Over the last fifty years, digital devices have found a wealth of possible applications, ranging from supercomputers to the Internet of Things (IoT). Such a complexity requires appropriate hardware infrastructures, including application specific circuits together with "general purpose accelerators" (like GPUs) that can aid high-level computation in implementing the more and more specific functionalities required in each context. In the majority of domains there is a tendency to include as many specific digital (and even analog) circuits as possible in the same die, resulting in the so-called System On Chips.

This trend is made possible by advancements in both digital technology, illustrated by Moore's law, and the overall design process, which counts many steps and tools involved. In particular, semiconductor companies may rely on libraries of Intellectual Properties (IPs) not produced in-house to speed up the whole design phase. Third-party IPs are untrusted by definition, due to them potentially including vulnerabilities, bugs or even openly malicious components. Such modifications of the circuit functionality, either malicious or not, are named Hardware Trojans, and they could be leveraged by ill-intentioned actors to wreak havoc during system operation.

Due to the increasing complexity of the design flow as a whole, Trojan insertion is made possible at many steps and at different abstraction levels (RTL, gate-level, layout etc.). Many techniques have been developed to identify Trojan affected circuitry, based on different approaches. On the other side, attackers managed to elude most of them through sophisticated injection strategies and stealthy Trojan architectures. Producing a tool capable to automatically inject Trojans is a recurring topic in recent literature, both from the attacker and the defender points of view. Through this technique it would be possible to engineer vulnerable circuits on a large scale, feeding the demanding detection tools with meaningful samples.

This thesis work aims to implement such a toolchain for Trojan insertion on processor cores, adopting a scalable architecture-agnostic approach which focuses on the RISC-V ISA. The final product is customizable in all its parts, allowing for new cores or new injection strategies to be included through targeted modifications. In order to make the results accessible to the widest possible audience, the whole flow includes only Free and Open Source Software (FOSS). The decision to rely exclusively on these tools addresses the general shortage of open source based solutions, offering hobbyists and students a free alternative to proprietary EDA software. Some of the latest detection techniques have been used to test the Trojans produced, assessing the quality of the toolchain in the process.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Over the last years, integrated circuits complexity raised considerably. The necessity to include many components, ranging from digital computing units to analog converters, made it necessary for engineers to refine the design process. The typical flow includes multiple third party EDA tools, accounting for each abstraction level (from RTL to GDSII), and third party subcircuits, which allow for more complex circuits to be produced. Both the former [6] and the latter [7] are untrusted, because of them coming from third parties. In particular, malicious circuits may hide inside these IP components or they can be stealthily injected through an EDA tool. The part of the design which implements the unwanted functionality is identified as an *Hardware Trojan* (*HT*). There may be some cases in which these Trojans are injected on purpose by the designer, for example to introduce a backdoor to be used for debugging.
Identifying unwanted components in an IC is still a very demanding task, expecially because the injection could be performed at any step in the design flow. Many examples of Trojans at different abstraction layers have been recorded in literature, including RTL [8], gate level and finalized layouts [9]. Moving to a lower abstraction level increases the degrees of freedom, because it makes it possible to edit more fine-grained design parameters (like parasitic capacitances [10] and wires placement [11]).
Multiple strategies have been proposed over the years to act as countermeasures against these Trojans, either through detection techniques [12] or through the addition of hardware components to compensate for malicious behavior during operation [13]. Among the first class of countemeasures it is possible to identify *dynamic* and *static* techniques, where the former are simulation-based while the latter involve analyzing the netlist topology. Historically, *VeriTrust* [14] and *FANCI* [15] are the two main examples for these categories, since they acted as stepping stones for the implementation of more complex solutions.
From the attacker side, new injection methodologies have been conceived to elude the aforementioned detection strategies. Among these, *deTrust* [3] is the one which will play a crucial role in this thesis. The availability of meaningful state of the art Trojan samples is vital for the "defense" team as well, because it helps in identifying new malicious features to refine current detection techniques. All these samples are usually collected in hubs like *TrustHUB*, which are used as training pools for novel AI or ML-based detection frameworks [16]. For this reason, the necessity to design a toolchain able to produce hundreds of Trojan affected circuits has become a popular topic over the last years [9]. This thesis aims to address this point, focusing on the implementation of a fully automated flow for designing HTs.
Since the objective presented is the one to generate as many trojans as possible, creating a flow which may be run in as many instances as possible should be preferred. Proprietary tools usually impose strict constraints in terms of execution instances, so they would make

it more difficult to parallelize the executions. For this reason, building the toolchain by relying exclusively on free tools could be beneficial. Such a solution would represent one of the few entirely open environments in the Hardware Trojan injection scene, providing a good opportunity for beginners and hobbyists to learn all the intricacies of the Trojan design problem. This thesis addresses this aspect as well, proposing a scalable and automated solution built on top of tools from the *OpenLane* [17] flow.
As anticipated, popular detection strategies from the state of the art will be considered when implementing stealthy Trojans. Many solutions built starting from FANCI and VeriTrust will be thoroughly explained and analyzed, such as *FASTrust* [4], *ANGEL* [18] and *UCI* [19]. Some algorithms to bypass each one of these detection tools will be proposed, expanding some solutions already existing in literature and combining them together. A novel strategy will be proposed to elude the majority of static techniques, partially sacrificing trojan stealthiness against dynamic ones.

The thesis is structured as follows.
Chapter 2 details all the characteristics of Hardware Trojans, providing multiple taxonomies based on: the abstraction level targeted, the Trojan structure, the harmful/harmless nature etc.
Chapter 3 introduces all the tools that are needed to understand the toolchain, providing some meaningful examples of how these tools work singularly, how they could be integrated together and the syntax and commands to be used to make them work.
Chapter 4 takes a picture of the state of the art for both injection and detection techniques, highlighting how the vulnerabilities of the latter are covered by the former. A formal description of each technique will be provided, which will be used as reference for implementing the actual RTL code of Trojans designed in the following parts of the thesis.
Chapter 5 consists in a detailed explanation of the toolchain, detailing all the parts and how tools coordinate to implement the desired functionalities. A comprehensive list of each script involved is provided in this chapter.
Chapter 6 shows all the experimental results obtained by comparing the Trojans with some of the aforementioned detection techniques, plus an analysis of area occupation and time needed by the toolchain to produce working solutions.
Chapter 7 proposes some possible ways to improve the quality of the toolchain, refining the quality of the Trojans designed and reducing the time needed to produce them.

Chapter 2

Hardware Trojans

2.1 A definition

Hardware trojans (*HT*s) are modifications to a circuit functionality that have been inserted during the design or manufacturing flow, aiming to force malfunctioning, place *backdoors* or to silently leak data. Despite the majority of them being designed by ill-intentioned actors, there are also trojans consisting in benign backdoors intentionally placed by engineers during development. Such tools may become vulnerabilities in case they are not removed before manufacturing, because thrid actors could exploit them for their means. In modern design processes, many functionalities of each design are implemented by relying on *third party IP*s (*3PIP*), which are assumed to be untrusted due to having been designed by an external actor. These *IP*s are involved almost in every phase of the process, since they could be either simple components included during *specification* or *synthesis* (like a library multiplier), or they could be created and injected by untrusted CAD tools in a toolchain [6]. This problem became even more critical with the rise of *System on chip* (*SoC*) technology, due to the sensible increase in complexity of each single chip. A single SoC, like the one in a mobile phone, could include a processor, a GPU and a modem each one developed by a different actor. As a further complication, the whole design phase consists in many steps, going from *specification* to actual manufacturing, and a trojan could potentially be injected in any of them. In order to shield against these threats, it is necessary to understand in details what are the main characteristics of these trojans, which are their properties and some circuit parameters that can be observed to detect them.

2.2 Hardware trojans taxonomy

A proper classification of trojans requires first a primer about how they work, together with the description of their main components. In the most general take on the problem, a trojan consists in two parts:

- a *trigger*, corresponding to a circuit which is in charge of recognizing when a particular condition is verified: when this happens, the trojan functionalities are unleashed;
- a *payload*, the actual malicious addition to the circuit, which may look very different depending on the functionality being added. It is usually integrated in the original circuit, to better mask it.

As anticipated, trojan injection can be done in each step during the design process. Moreover, a huge variety in terms of both triggers and payloads is possible. In the

remaining paragraphs a proper taxonomy and classification will be given, by looking at the problem from multiple angles. The classifications proposed in [7] and [20] will be taken as references, focusing on the following criteria:

- the abstraction level targeted by the injection process, which is indirectly related to the trigger and payload structures and how different design descriptions are developed during the design phase. The analysis will be detailed in Section 2.2.1;
- in which part of the design flow the injection is performed, showcasing how this could affect the trojan concealment property in Section 2.2.4;
- trigger types and payload types, in Section 2.2.2 and Section 2.2.3 respectively.

Additional criteria could be specified, such as the location of the trojan in a complex system such as a System on Chip, but they can be considered of secondary importance in our analysis, mainly due to the fact that only trojans targeting a core will be considered in this thesis.

2.2.1 Abstraction level-based classification

The IC design process involves working on different abstraction levels depending on the problem being tackled. For example, handling the interfacing between high level computational units (such as multipliers, adders ...) is usually done at the *behavioral/RT level*, while a precise estimation of power consumption and critical paths delays require a *gate level* or even *physical level* description. As a rule of thumb, injecting trojan at lower abstraction level makes them invisible to checks that are performed on higher level descriptions, plus it widens the design space due to the increased number of design parameters. This analysis is strongly related to the one covering the design phases, because when advancing in the design flow the developers tend to move to lower abstraction level descriptions. Let us make the classification clearer by showing some examples at different levels:

- *system-level*: at this level, interactions between different instantiated modules are defined (in the form of *protocols*). As hinted in [20], the main modification that can be introduced at this level may consist in a redefinition of these protocols. This would involve modifying multiple components, creating a trojan which affects the system as a whole and that is potentially distributed on a wide area. Protocols are usually defined by the company which designs the IC, so the attack surface is the smallest possible.
- *behavioral/RT-level*: devoted to describing high-level functionalities for each module being developed, by relying on combinational units, signals and registers. At this abstraction level, a trojan could tamper with any kind of local protocol, adding extra logic functions and/or registers with respect to the specifications. Trojans functionalities are implemented through concurrent assignments or processes, which should be as short as possible to minimize detection probability by *RTL* netlist parsing tools [21]. As will be presented in Section 4.2.1, malicious designers should also disguise the trojan circuit as a useful one to bypass unused circuit detection techniques such as *UCI* [22]. At this abstraction level the attacker doesn't have full control on how the additional circuit will be synthesized and mapped, so an injection could result in sub-optimal synthesis outputs. Additionally, formal verification checks are performed by synthesis tools, potentially recognizing redundant hardware: for this reason, the trojan description should be properly disguised inside the original

RTL code. Since the majority of designs are too complex to be implemented completely *in-house*, library IPs will be leveraged to offer micro-functionalities needed by the each RTL-described component. This widens the attack surface, because untrusted IPs could host trojan circuits.

- *gate-level*: consisting in a synthesized (and potentially mapped, hence technology-dependent) netlist, where the behavioral functionalities are not evident anymore. The majority of existing tools for pre-layout trojan injection operate at this level, producing additional gates to be attached to the existing ones to implement trigger and payload logic functions. This is usually the deepest functional level at which the designer operates, at least in *standard cells based* designs, since layout manipulation is usually done only to fix design rules (or to optimize over design metrics) and not to implement boolean logic functionalities. Working at this level ensures full controllability over the primitives used to implement the desired logic function. Exactly as in the previous case, library IPs act as the main cause behind circuit vulnerability to *gate-level* trojans.

- *layout-level*: the lowest-level description of the circuit, produced as a result of both the placement of standard cells on the silicon substrate (*floorplanning*) and the *routing* of interconnections on different metal layers. Trojans injected at this level could involve the modification of metal layers, together with cell replacement in the floorplan. An interesting proposal for *layout-level* injection leverages *Engineering Change Order* (*ECO*) [23][1] flows to repurpose filler or spare cells (or gate-arrays) as primitives for a trojan circuit, relying on the flexibility of ECOs and on how they act minimizing modifications introduced in the finalized layouts (especially for *post-mask ECO*s) [24] [9]. Since ECOs are sometimes executed directly by the foundry, this enlarges the attack surface considerably.

2.2.2 Trigger-based classification

The trigger circuitry is yet another parameter that could be considered when classifying HTs. According to the presence/absence of memory in the trigger mechanism, together with the analog/digital nature, the taxonomy can be expanded as follows:

- *always-on trojans*: these are the trojans with no trigger. They can affect the circuit functionality at any moment, resulting in potentially high observability. For this reason they are rarely employed in real-life applications.

- *analog triggers*: activated when conditions involving analog values are verified in the circuit. One of these conditions could be the temperature exceeding a threshold due to increased activity in the circuit, monitored through a malicious temperature sensor [7] [25]. In general, it is possible to leverage analog sensors to build rare conditions, which may be related to the environment where the device operates. Analog values are somewhat more difficult to control since they may vary in ways that cannot be exhaustively reproduced during testing phases, thus making these trojans more difficult to detect during functional testing. Another approach to this kind of triggers is based on exploiting the intrinsic analog behavior of digital logic, evident when transitions are at play. For instance, triggers can be built by forcing charge to accumulate in a large capacitor by switching a signal multiple times [7], or

[1]Tools used to fix minor logic defects by starting from the finalized layout, without resynthesizing the whole design and without redesigning every mask layer. They require support from EDA tools, which should place some spare cells on the layout for implementing logic fixes.

by keeping it high for many consecutive cycles to activate a clock glitching circuitry repeatedly [10]. Exploiting *cross-talk* is also a valid way to build an analog trigger. This has been in done in [11] and summed up in [26], where metal lines placement and gate parasitics have been altered to manufacture a big capacitance on a wire and to move it closer to another signal (the proper *trigger*). As a result, repeated transitions on the trigger wire may force charge to leak to the altered wire (through coupling effects) and to accumulate in the parasitic capacitance. As formulated by Jain *et al.* [26], these kinds of trojans are formally equivalent to sequential ones, due to their sensitivity to events spanning multiple clock cycles. As a solid advantage, they don't show the big area occupations that are needed to account for flip-flops (FFs) allocation in proper digital logic based sequential trojans. The (in)visibility of these triggers is affected by the *design rule checks* (DRCs) tests performed during the last phases of the design process, since they might be able to detect close wires and to reallocate the interconnections.

- *digital triggers*: the ones that have been studied the most, since they rely on rare conditions occurring on digital values. This category can be further divided in two, depending on the type of condition:
  - a condition that involves checking values over a single clock cycle demands a *combinational* trigger. These circuits are usually very simple, accounting only for some `AND` and possibly `NOT` gates. These hardware structures can be responsible for the creation of *rare wires* in the topological sense, which are wires resulting from a combinational logic function with a strongly unbalanced truth table (the *N-inputs* `AND` is a typical example, with only a single 1 value on 2^N). These wires are sensitive to netlist inspection based techniques, which can locate them and mark them as suspicious [3].
  - if the condition is splitted on multiple clock cycles then the trigger is said to be a *sequential* one. Triggers belonging to this categories are the stealthiest ones, because introducing memory can be used to manufacture a trigger which activates after many occurrences of a simple combinational condition. However, adding memory cells can affect both area and power, making the trojan more visible to side-channel analysis (for power) and layout inspection through *Scanning Electrons Microscopes* (*SEM*s) [27]. As will be shown in details later, rare conditions can be vulnerable to simulation-based techniques like *UCI* [6] and *VeriTrust* [3], which locate rarely excited signals during the application of test vectors or benchmarks. Among these triggers it is possible to count multiple subcategories, including: *time-bombs*, which simply activate after a fixed time window expires, properly set-up to minimize activation probability during functional testing; *counter-based*, which are triggered when a combinational condition is verified multiple times; the particular kind of analog triggered trojans which activate with repeated transitions and/or signals kept high for multiple cycles (these don't belong to the digital realm, however).

Another important criterion to be considered when classifying trojans is related to the way in which triggers are activated, which could require direct intervention from the malicious actor. This leads us to introduce a further distinction between:

- *internally triggered trojans*: which are activated as a consequence of an internal condition occurring. In this way no intervention from the attacker is needed, making it superfluous to have direct access to the victim device. The need for a condition which activate on its own is one of the main vulnerability of this kind of

HTs, because this condition must happen during normal circuit operation. This is responsible for making the trojan more vulnerable against functional tests, which are supposed to test almost exhaustively every operating condition. However due to lack of intervention from the attacker, this kind of trojans is the most diffused one, especially for large scale attacks.

- *externally triggered trojans*: the ones which require a direct action from the attacker in order to be triggered. Even if it may look like a very restrictive constraint, it is important to keep into account that the physical presence of the attacker is not needed in the majority of cases. Some attacks leverages a simple internet connection and they use a particular packet format as a trigger condition, making the attacker able to activate the trigger by issuing a remote command. The important thing to note is that the condition forced by the attacker may not happen spontaneously during normal operation, so it may not be covered by functional tests.

2.2.3 Payload-based classification

The payload is the other half of the HT, and designing it can be as complex as designing the trigger. Again, each payload will be classified depending on the analog/digital nature of the circuit and on the functionalities:

- *analog payloads*: the ones which act on the analog behavior of the circuit. Trojans of this kind may aim to affect values coherence, introducing bridging faults in the final layout, and/or to tamper with delays, by increasing parasitics [7]. These kind of modifications are usually introduced by rogue elements in the foundry during the manufacturing phase, since they are related to physical properties of the gates. Still, it would be possible to introduce them at the *gate-level*, for example by using multiple dummy gates connected in parallel to increase the parasitic capacitance. However, dummy gates can potentially be detected by formal techniques and by *Static Timing Analysis* (*STA*), in case the path affected is among the critical ones. According to Chakraborty's classification [7] there is yet another subcategory of analog payloads, which includes trojans whose purpose is to accelerate circuit wearing process. These trojans either consist in defects injected during manufacturing [28] or they can be actual digital circuits which force spikes in temperature, that are responsible for an exponential decrease in the *Mean Time To Failure* (*MTTF*) for each potential cause of failure [28].
  Analog payloads can also be used for information leakage, in particular through power-based side-channels. This can be done by introducing circuit elements whose power consumption can be regulated through a *switch*, defining a protocol which maps a certain power consumption to the logic value to be leaked (for example normal power consumption corresponding to '0', increased power consumption corresponding to '1'). The added power must be inferred from the overall power trace of the circuit, paying attention not to make it too visible to shield against *Side-Channel Analysis* (*SCA*). Ring oscillators (*RO*s) can act as candidate circuits for implementing configurable power sinks [29].

- *digital payloads*: which maliciously modify digital values of signals inside the circuit. Among them, some common classes can be identified [9]:

  - *leak payload*(n,M,m), which forces a n-bits shift register to be overwritten with the payload value, leaking $1/2^c$ bits for each clock cycle. The protocol M to be used to leak bits can be chosen among *Frequency-Shift Keying* (*FSK*),

Differential Binary Phase-Shift Keying (*DBPSK*) and so on. These payloads can be used to leak information to the outside, either by coupling them with analog based side-channels or peripherals activity. An interesting example of the latter is provided in [30], where an *RS-232* channel is used at a non-functional frequency to mask confidential data transmission. Techniques like these are almost guaranteed to produce a trojan invisible to functional testing, but parameters should be chosen carefully to avoid affecting power consumption too much.

- *Shift'n'burn payload*, to burn energy by shifting values in a shift register at a rate of n transitions per clock cycle.
- *modify payload*, which modifies a signal by forcing it to assume a new value. This is the payload that is used the most, since it allows the attacker to modify virtually any part of the design. Possible applications for this kind of payloads vary from simple *denial of service*, when continuously setting an active high reset signal, to actual *privilege escalation*, for example by forcefully setting the *machine/user* mode flag [3].
- *fault payload*, which simulates a random fault injection on a signal bits. This may be done by resorting to an *Linear Feedback Shift Register* (*LFSR*), which helps in defining the randomicity. As will be shown later, *LFSR*s can be detected by reverse engineering techniques implemented in tools like *HAL* [18], so they should be used with caution.

2.2.4 Design step based classification

The last step in defining a proper taxonomy involves identifying the manufacturing steps when HTs injections are possible. A brief list of these steps will follow, together with some examples:

- *specification phase*: the one where golden behavior of the circuit is defined. Specifications consist in a formal description of all the properties and characteristics of the final product. In case the design is commissioned by an external customer, engineers build a description based on specific requests coming from the customer. Altering this description can prove to be a very effective way of implementing a trojan, because it compromises what is supposed to be the reference (*golden model*) which defines the intended behavior for each implementation produced during the steps that follow. This would potentially invalidate all testing techniques which rely on this model, such as formal verification methods [31]. In this phase, however, security perimeter is very tight due to this golden model being produced in-house. As a consequence, tampering with specifications would require the presence of rogue elements inside the company which manufactures the IC. This is assumed to be very unlikely, so these kinds of trojans have not been analyzed in details.

- *design phase*: the phase during which the actual functionalities are implemented. As anticipated in Section 2.1, *3PIP* are massively used in modern designs, so the output of this phase can't fully trusted to be trojan free. Every trojan which affects the circuit logic behavior has to be injected in this step, therefore this kind of HTs is well documented in literature. This phase includes many descriptions of the circuit, going from the very high level behavioral description (the first one produced from the specification) to the very low level ones, such as the gate-level netlist or even the actual *GDSII* layout. This exposes many windows for injecting trojans, with some possibilities already discussed in Section 2.2.1. The following parts of the

thesis will be devoted mainly to trojans injected in this stage, since the other ones require very complex toolchains (and/or access to foundry tools).

- *manufacturing phase*: involving the actual IC fabrication in a foundry, starting from the *GDSII* description. It is the phase when the security perimeter widens considerably, due to direct intervention on the circuit by external actors. The trojans that can be injected during manufacturing are usually the ones that involve injecting faults (by modifying physical characteristics, for example through improper baking [12]), but it could be possible to alter logic functionalities too by employing ECOs (explained in Section 2.2.1). Another way to reconstruct high-level functionalities is offered by *reverse-engineering* tools [9]. Package manufacturing is also a part of this process, so it should be considered as a possible way to introduce (or at least mask) an HT [12].

2.3 Hardware trojans detection approaches

Stealthiness is surely of paramount importance when discussing about hardware trojans, since a trojan that is not stealth enough could be detected in one of the checks that are usually performed as a part of the design flow. In order to devise an HT which is stealthy but effective, there is the need to learn about the ways in which an HT in a circuit could be spotted. A comprehensive list of the most used trojan detection techniques will follow, highlighting some of the properties that could allow the *Device Under Test* (DUT) to evade them.

2.3.1 Testing tecnhiques

Testing is one of the possible ways in which a trojan can be identified. A common denominator between all testing techniques is the need to actually stimulate the DUT with some values, then evaluating the result produced through simulation. In order to tackle this problem more effectively, let's show some kinds of testing techniques that could be employed:

- *functional testing*: which consists in stimulating the circuit with values that could actually be produced during normal operation. The purpose of functional testing is the one to assess how the circuit will react in the environment where it is supposed to operate, without necessarily covering different operating conditions (useful in reducing *overtesting*[2]). This technique fails mainly against *externally-triggered trojans* (recalling the definition in Section 2.2.2), which can rely on non-functional conditions to excite the trigger. *Software Based Self Tests* ($SBST$s) are a typical example of these techniques.

- *scan chains based testing*: consisting in the application of test vectors for circuits whose flip flops (FFs) are organized in scan chains. This technique offers a way to compensate for the complexity of sequential *Automatic Test Pattern Generation* ($ATPG$), which proves to be unfeasible when run on circuits with traditional FFs. As a consequence, hardware support for scan chains has been integrated in almost every sequential circuit. The powerful feature about this technique resides in the possibility to transform an N-stage long sequential circuit in a simple *1*-stage FSM, making the problem of generating test patterns solvable through simple combinational ATPG.

[2]The procedure of excessively testing an IC, marking it as defective in correspondence of test vectors that cannot be generated during normal operation.

A potential side effect of this technique resides in the vulnerability to *overtesting*, which is detrimental for our trojan too since some non-functional conditions could be exploited as explicit triggers. Patterns generated by ATPGs can be applied to the circuit either through *Automated Test Equipments* (*ATE*s) or *Built-In Self Test* (*BIST*) hardware.
Due to scan chains limiting the complexity of the circuit to the sole combinational network, replacing all the FFs that are part of the FSM of a sequential trigger trojan with scan FFs should be avoided. If this rule is not observed, there is the risk of reducing a complex sequential trigger to a simple combinational one. Unfortunately, non-scan FFs can be located through SEM imaging [26], so their number should be kept to a minimum.

2.3.2 Static analysis techniques

Another way to locate malicious functionalities is offered by netlist inspection tools, which may operate on a description of the design at any level. Differently from testing-based detection, the inspection process doesn't require simulation. In the following, *behavioral-level* and *gate-level* trojans will be analyzed in details, covering the corresponding detection techniques as well. The main focus of this discussion will be on introducing the distinction between:

- *behavioral-level netlist parsing*: it mainly consists in evaluating each statement depending on precise observability and controllability metrics, marking as suspicious the lines which fall under/over a threshold. Salmani *et al.* [8] described a possible implementation of such techniques, assigning a score to each assignment depending on how nested it is inside `IF` conditions or `FOR` loops. The algorithm aims to identify lines showing low controllability and low observability, which are potential sites where a trojan could be placed. The majority of these techniques employ graphs, emulating dataflow in the circuit and transforming the problem of evaluating netlist properties into the application of graphs algorithms. Since the gate-level structure of the circuit is not known yet, at this level nodes are usually mapped to signals or behavioral statements in the circuit. Besides, due to the limited amount of signals at the behavioral level, this kind of analysis is usually much faster than post-synthesis inspection.

- *gate-level netlist parsing*: it follows the same principle based on computing controllability and observability, but working directly on the post-synthesis or post-mapped netlist. Due to the regularity of these kinds of netlists, many more algorithms are available, spanning from *supervised* [32] or *unsupervised* [16] machine learning techniques to properties computation through exact algorithms, such as *FANCI* [3] or *ANGEL* [18]. Some tools, *HAL* in particular, allow the user to reconstruct high-level structures which could be part of a trojans, including *FSM*s or *LFSR*s [18].
Working on gate-level netlists makes the analysis much longer, due to the increase in complexity and in the number of signals. This helps the attacker, who is able to locate many additional injection point which were not visible at higher abstraction levels. For this reason, further attention will be devoted to these techniques throughout the thesis.

- *layout-level netlist parsing*: a group of techniques which is mainly based on gate-level functionalities reconstruction through reverse engineering. Provided that the user is able to retrieve a reverse-engineering tool from *GDSII* to *gate-level*, the detection

problem can be solved by resorting to the tools described before. Hepp *et al.* [9] showed how such tool could be included in the workflow.

2.3.3 Dynamic analysis techniques

Dynamic analysis is somewhat closer to testing, since both techniques rely on circuit simulation. The main difference between the two is related to how the detection process is performed. On the one hand, testing involves a simulation of the circuit to find out if there are discrepancies between the expected behavior and the actual one, while dynamic analysis uses simulation to compute metrics which help in classifying a signal as malicious or harmless. As seen with static techniques, the values that should be estimated are controllability and observability, which are usually compared to threshold values. The simplest way to do so involves evaluating *toggle* and *level coverage* for each signal, which quantify the amount of value changes and the cycles spent at value V (`'0'`/`'1'`) respectively. This approach is mainly used to locate rare trigger conditions, helping both the attacker and the defender. For this reason, making trojans with a single signal acting as trigger doesn't guarantee enough stealthiness, resulting in more and more attackers designing complex triggering conditions involving multiple rare signals. Unfortunately for the attacker, this is not enough, since the output of a complex `AND` network (a classic trigger) is still a single wire, showing the same vulnerabilities as the single trigger approach (and probably even worse coverage values). Chapter 4 will present a more detailed description of some dynamic analysis algorithms, primarily focusing on *VeriTrust* [14] [3].

2.3.4 Formal verification

Formal verification (FV) is yet another discipline from which some techniques for trojan detection can be borrowed. As known, *verification* is the process of proving the coherence between implementation and specification, where the proof becomes a mathematical one when talking about *formal* verification. Various approaches can be followed to detect trojans through these means, where the main ones are summarized here [31]:

- *reachability analysis*: which focuses on determining rare states in the circuit, where each state corresponds to a N-tuple of binary values (N is the number of FFs). This technique usually starts from a well-known state and it moves through the circuit state diagram. Due to the exponential dependency between N and the number of states (2^N), the computational complexity of this techniques explodes in correspondence of linear increments in topological complexity.

- *equivalence checking*: generally used to demonstrate equivalence between RTL specification and a gate-level implementation, allowing the design to identify potential threats or misbehaviors introduced by malicious synthesis tools and/or library IPs. Two approaches are possible, where the first one is based on *Reduced Ordered Boolean Decision Diagrams* ($ROBDD$)[3] while the other one reduces the equivalence problem to a boolean satisfiability one, solving it with a *SAT solver*.
  ROBDD-based technique requires building two ROBDDs, one from the specification and one from the implementation, comparing them to check topological equivalence (*isomorphism*). This final check can be done in linear time, but the tree complexity strongly varies depending on the ordering chosen. As a consequence, choosing the best ordering becomes the capital problem of this approach when it comes to large

[3]A ROBDD is a tree representation of a boolean expression. It is built in three steps: first an order for the variables is chosen (all nodes at the same depth will evaluate the same variable), then the complete tree is built and in the end a simplification is performed to reduce the number of nodes.

circuits, where heuristics or variable sifting [33] must be used.
The heart of the other technique consists in SAT-solving a certain number of problems to determine if specification and implementations show the same behavior. A SAT problem over a boolean expression $f(\mathbf{x})$ can be defined as "*determining an assignment for the* $\boldsymbol{x}$ *inputs such that* $f(\boldsymbol{x}) = 1$". In the context of the comparison between two logic circuits, this problem can be obtained by building a *miter*. Let us assume to excite both designs with the same $\mathbf{x} \in \{0,1\}^M$ inputs, in correspondence of which $\mathbf{f}(\mathbf{x}) \in \{0,1\}^N$ are the outputs from the specification and $\mathbf{g}(\mathbf{x}) \in \{0,1\}^N$ from the implementation. A miter $\mathbf{M} \in \{0,1\}^N$ performs the following operation:

$$M_i(f_i(\mathbf{x}), g_i(\mathbf{x})) = f_i(\mathbf{x}) \oplus g_i(\mathbf{x}), \quad \forall i \in \{0, 1, ..., N-1\} \tag{2.1}$$

which consists in a bitwise `XOR` between corresponding outputs. In case at least one of the components of $\mathbf{M}$ evaluates to 1, then the two designs behaviors diverge. The whole purpose of SAT-solving for equivalence checking is the one to find a condition under which this behavior occurs, proving a discrepancy between the two DUTs.

- *model checking*: it targets some properties that both the design and the implementation should show. These properties are usually written according to a *temporal logic*, and the whole implementation is translated to a formal language recognizable by the model checker (*SMV* or similar). After receiving these inputs, the checker builds a ROBDD starting from the implementation and tries to find exceptions to the rules (it is possible to resort to SAT-solving too).

As an important note, applying the previous methods on a circuit as-is could lead to unfeasible results due to the complexity of sequential parts. A possible solution requires *partitioning* each design in simpler parts, exploiting *fanin* and *fanout* cones, together with supports from *Design for Testability* (*DfT*) techniques like scan chains, which can reduce sequential SAT to combinational SAT by allowing scan FFs outputs to be considered as *Pseudo Primary Inputs* (*PPI*s) [34].

2.3.5 Imaging

Imaging is a good way to spot trojans after manufacturing, since it works directly on the finalized product without any need for simulations. From now on, the circuit being analyzed will be referred to as the *Implementation Under Analysis* (*IUA*). A wealth of imaging applications are available, which make use of different particles or require photographing the die from different orientations: frontside or backside. Frontside imaging consists in hitting the top-side of the IUA with particles. If the die-to-package connection is assumed to be done with *flip chip*, the top-side is the one at the opposite side of the silicon substrate. This means that in order to reach the silicon *active region*[4], the particles must pass through all the metal and silicon dioxide layers. Since this is unfeasible due to the rays nature, this kind of approach requires removing all these layers through chemical processes (and it is dubbed *destructive*). The backside approach is more conservative, but it requires different preparations depending on the particles used. The particles that are most commonly used are: electrons, Helium ions (*He-ion*s) or infrared (*IR*) rays. Electrons-based imaging (*SEM*) is surely the most used one in literature [35] [36] [37], even if it offers noisier images with respect to *He-ion*-based approaches [38]. Using electrons for backside imaging requires *thinning* the silicon layer first, due to their limited penetration power. *IR*-based imaging has been applied to backside imaging, because it shows more

[4]The layer where doping is performed and gates are implemented, placed at the surface of the silicon substrate.

penetration power with respect to electron-based one. However, infrared rays resolution is more limited, making it more difficult to reveal very small and condensed trojans.

2.3.6 Side-Channel Analysis

Side-Channel Analysis (*SCA*) is primarily used to locate the so-called *side-channels*, information channels that are not part of functional specifications and that are born due to how the circuit physically works. Some examples of these side channels could be: power consumption, cache latencies, electromagnetic (EM) emissions etc. In Section 2.2.3 the idea of having a configurable power sink as a payload has been introduced, where ring oscillators are candidates for such a role. This strategy aims to build a power side-channel, where different power consumption values could be used to leak sensitive information. Such attacks require the attacker to have physical access to the platform, due to the need to perform the measurement. Oscilloscopes and similar tools are not needed in modern environments, where cheap and powerful platforms like *ChipWhisperer*™ are used. Even if no channel is purposefully altered by the HT, normal trojan activity can lead to power consumption traces different from the golden ones (mainly if the trojan is big in size). Because of that, trojan designers should make the design compact and small, in order to minimize switching power consumption due to interconnections and gate inputs/outputs.

2.3.7 Runtime techniques

Literature shows another realm of detection techniques, which try to locate and promptly disable malicious outputs during runtime. Doing so requires additional area (and possibly power) overhead, due to the need to perform analysis while legit computation is ongoing. Some of these techniques simply require additional circuits without layout constraints (as *DEFENSE* [13]), others exploit new technologies to offer additional detection capabilities. A shining example of the latter is offered by Huffmire *et al.* [39], where 3D die technology is used to separate computation and security checks on different silicon planes.

2.4 A brief primer on the injection problem

Until now the discussion has focused on about every possible detection technique, highlighting some examples that will be examined in details when presenting the implementation. The main focus of this thesis, however, should be the problem of *injecting* a trojan in a netlist. As briefly hinted while presenting the taxonomy, the two problems are actually specular, since tools used to detect possible trojan sites can also be used to find injection spots. Naturally, further additions are needed to disguise the trojan, otherwise there would no competition between detection and injection tools. Let us summarize which are the possible limitations of each detection technique that can be leveraged by malicious actors, trying to find intersections:

- functional testing offers the opportunity to inject arbitrarily complex explicit triggers, because they could leverage non-functionally reachable states. Besides, exhaustive functional testing is very difficult to implement, leaving space for the attacker to operate on conditions that are difficult to test.

- scan chains based testing is very powerful, since on *full-scan* designs it reduces sequential ATPG to combinational ATPG. However, the attacker can simply avoid including trojan FFs in scan chains to turn the design into a *partial-scan* one. This would increase the ATPG problem complexity, but many trojan conditions are very

specific and thus it is still simple to generate patterns to excite them (since they mainly consist in `AND` of different wires).

- static analysis techniques are able to recognize rare topological conditions, making them able to spot simple triggers as `AND` trees. In origin, only techniques working on combinational networks were available, which were unable to identify trojans spread on multiple sequential stages. In the last years, graph FF cutting has been used to implement more sophisticated solutions [18]. As a consequence, attackers must devise solutions which exploit dynamic behavior to mask the static one. Later chapters in this thesis will show a possible approach to this problem.

- dynamic analysis focuses on identifying wires with low toggle/value coverage, which could potentially belong to trojan circuitry. The powerful feature about this kind of analysis is the possibility to spot rare conditions wherever they happen, with no distinctions between sequential or combinational behavior. The key to defeat this solution is to properly merge trigger and payload circuitry, preventing the creation of rare wires. Zhang *et al.* [3] described a possible way to do so through an algorithm that has been generalized and applied in this thesis as well.

- formal verification can be bypassed almost in the same way as an attacker would bypass scan chains based testing, because both of them are very complex if made sequential.

- bypassing imaging requires different strategies depending on the kind of imaging at play, but it is possible to find some common denominators. As a rule-of-thumb, keeping the trojan size small (a thousand of cells in modern designs) and condensed may help in eluding inspection. Conversely, scattering the trojan circuitry on the whole die area could help too, because it would make it more difficult to reverse engineer malicious functionalities and it would force the detection tool to flag as suspicious many different spots. This forced increase in complexity is sometimes the only way to fool injection tools, forcing them to flag thousand of regions and making it difficult for any engineer to manually inspect them.

- SCA can be bypassed in different ways, depending on if the objective of the attacker is to make use of the side channel or not. In case the payload relies on a side channel, information should be properly hidden inside the power trace to shield against trace inspection techniques. On the other hand, reducing the trojan complexity is the way to follow if side-channels are an unwanted side effect.

- due to their limited applications, runtime techniques will not be covered in the following discussions.

Chapter 3

Introduction to the tools

3.1 Main topics and chapter organization

The present chapter is devoted to introducing the toolchain, listing the basic functionalities and giving a brief technical outline about how all the functionalities are implemented by each component. A more detailed description of the parts will be offered in the following chapters, where the actual workflow and the toolchain configuration will be shown. The chapter is structured as follows:

- a first section to show the expected functionalities, showing all the tools that have been used at each step;
- the following sections will introduce one tool each, without going in technical details about how they have been integrated in the actual toolchain.

3.2 Some remarks on the open source nature of the toolchain

A perfect injection environment should allow us to design a trojan which eludes all the techniques described in Chapter 2, at each level of the IC design flow. Since tackling such a problem is very complex, this thesis work will focus on gate-level injection, without analyzing lower abstraction level descriptions. Some hints about how choices can have an influence on layout and routing detection techniques will still be presented, being aware of the possibility to extend the present work in the future.
Among the features of this toolchain it is important to count the *free and open source* (*FOSS*) nature, one of the major standpoints which distinguish it from existing solutions. Currently, the hardware design domain presents little to none open source flows, making it impenetrable to amateurs or hobbyists that don't have access to industry-grade tools licenses. For this reason, it has been decided not to rely on any proprietary tool in the workflow.
Another positive element resides in the configurability and extensibility of the toolchain. Some proposals for future extensions will be shown in Chapter 7, but for now it is enough to know that the flow can be easily extended to lower abstraction layers through *OpenLane* tools [17]. OpenLane offers a complete infrastructure to design a circuit, moving from very high-level descriptions to *GDSII*. Some tools included in the OpenLane flow has been used as part of the thesis, such as Verilator and Yosys. Working on huge open-source toolchains can be difficult, mainly due to the need to interface all the tools, but it allows many actors to reproduce the results. Additional difficulties rise from the limited support for existing *Hardware Description Languages* (*HDL*s), with the majority of available tools supporting Verilog only.

The solution shown in this thesis can be applied for virtually any processor, provided that the user integrates the simulation environment with the existing flow. A simple *RISC-V* open source processor has been chosen, named *SRV32* [40], which natively supports a pre-synthesis Verilator-based simulation environment and can boot and run *FreeRTOS* [41]. In order to assess if it was possible to generalize the toolchain, some parts of the flow have been tested against another processor too. This additional core, named *AFTAB*, was originally written in VHDL, so a preliminary *VHDL-to-Verilog* conversion is needed (as detailed in Section 3.5). The following discussion will implicitly refer to SRV32, but some explicit references to AFTAB will be included as well.

3.3 The workflow

The approach to the injection problem first requires identifying rare wires in the post-synthesis netlist, selecting N intermediate trigger candidates among them. The whole flow is *simulation driven*, so rareness evaluation is done by simulating some benchmarks on the core and then retrieving toggle/value coverage metrics. After building a group of a few thousand of signals, an evolutionary algorithm is used to select M[1] of these triggers, which will be combined together to create the final condition. The second half of the toolchain is devoted to payload injection, which is done by inspecting the netlist to find suitable points. Existing logic and payload logic are merged together, so it is necessary to use graph-building tools to easily manipulate and modify the netlist gate-level structure. Design choices will be detailed in Chapter 5, where a step-by-step explanation will be presented.

3.4 The tools

Let us now expand a little bit on the description in the previous section, listing all the tools needed at each step:

- the first step consists in synthesizing the design, since all the simulations will be done on the synthesized or mapped netlist. The chosen tool for this task is Yosys [42] (see Section 3.5), which supports scripts to customize every step in the synthesis flow.
- then a simulation over the result of the previous step should be performed, in order to find out if there are rare wires. This approach borrows from dynamic analysis techniques introduced in Section 2.3.3, but it will try to overcome some of the weaknesses. As anticipated, the simulation environment is built on Verilator (Section 3.6). The flow is then customized to retrieve value/toggle coverage, populating coverage reports.
- benchmarks to be used for simulations can be written manually or automatically. The first category includes benches which involve FreeRTOS or mathematical algorithms, while automatically-written programs simulate the application of random test vectors. For this reason, a code generator named *RISCV-DV* [43] (Section 3.7) has been used, which should be customized to generate the subset of instructions from the ISA supported by the victim architecture.

[1] In our discussion $N = 4000$, while M will range between 16 and 32. Both parameters can be chosen arbitrarily, with the obvious constraint $N \geq M$.

- the actual trigger injection is performed by an *evolutionary algorithm* (*EA*), which selects M triggers among the previously identified signals. The objective of this tool is to optimize over a certain fitness function, taking into account how many times each candidate trigger is raised, how the activations are distributed on the selected benchmarks and so on. The aforementioned functionalities have been implemented through Byron [44] (Section 3.8), a *microGP* [45][2] based source code *fuzzer*.

- payload injection must be done by parsing the netlist, so it requires a *white-box* approach. Among the available inspection tools, *HAL* [47] (3.9) has been chosen due to the wide number of reverse-engineering and netlist deconstruction plugins.

- as part of the last step, it should be evaluated whether the trojan affects program execution or not. Section 5.6 will show why this problem is very complex, since the effect on a program is always *user-perceived* and it escapes formal definitions. A possible way to provide an evaluation is the one to resimulate the program for every payload (with the same Verilator flow) and to compare blindly all the values in the traces, assigning additional weight to the output values. Another solution, which could potentially be much faster, relies on using a SAT solver to generate input patterns which propagate the payload to the output. The tools to be used in this case is *SymbiYosys* [48] (Section 3.10), a Yosys front-end which enables easy access to SAT engines. As will be shown, this approach should be coupled with a simulation-based one to facilitate the engine job.

3.5 Yosys

Yosys is an open source digital circuits synthesizer and mapper. Yosys main strength is given by the customizability, which rivals the one of commercial tools. The support for TCL scripts allow the user to setup every step in the synthesis process, specifying which kind of optimizations should be performed and in which way. An example script is reported in Listing 1, where comments highlight some of the main functionalities. For the present discussion, only the following commands will be considered: `flatten`, `techmap`, `dfflibmap` and `abc`. `flatten` is used to remove any hierarchy that could be present in the design, making it possible for the optimization engine to optimize across different modules. Hierarchy flattening is performed by replacing port map statements with the RTL code of the actual components, producing a single entity as output. `techmap` is almost equivalent to *Design Compiler*™ `elaborate` command, which maps RTL operators (adders, multipliers, bitwise logic) to *GTECH* cells functionalities. As hinted in Yosys documentation [49], `flatten` and `techmap` are very similar in concept, since the first one replaces port maps with actual modules implementations, while the second one replaces high-level operations with low-level cell definitions. `dfflibmap` and `abc` are used to implement the technology mapping for FFs and combinational logic respectively, providing the library *liberty file*[3] as argument (lines 14 and 16).

```
1 # read design
2 read_verilog rtl/top.v
```

[2] microGP is an evolutionary algorithm used to solve optimization problem over large design spaces. Possible applications of microGP range from automatically generating assembly programs to creating Core War warriors, assembly programs which compete for gaining control over a computer in a gaming environment. [46].

[3] The file collecting data built through cells characterization. Information in liberty files are used when mapping, during STA and during time/area/power optimization, when cell parasitics are needed.

```
hierarchy -check -top top
setattr -set keep 1 w:\*

# high-level refactoring
proc; opt; fsm; opt; memory; opt
flatten; opt;

# mapping to internal cell library (elaborate)
techmap; opt -purge

# mapping to cell library
dfflibmap -liberty pdk/gscl45nm.lib
opt
abc -liberty pdk/gscl45nm.lib
opt_clean -purge

# report
tee -o report/check.rpt check
tee -o report/stats.rpt stat -liberty pdk/gscl45nm.lib

# write .dot file
show -prefix output/synth

# write synthesized design
write_verilog output/top.v
```

Listing 1: An example Yosys script.

Yosys offers a way to partially circumvent the HDL limitations introduced in Section 3.2, allowing an automated VHDL to Verilog translation through the following couple of commands:

```
ghdl -a -g --std=08 --ieee=synopsys aftab_core.vhd
yosys -m ghdl -p 'ghdl -fsynopsys aftab_core.vhd -e aftab_core;
↪  write_verilog aftab_core.v'
```

which requires the *GHDL* [50] tool to be installed on the machine, plus additional support for the *ghdl-yosys-plugin* [51]. GHDL is a VHDL simulator that could replace Verilator in our flow, assuming a VHDL synthesized description of the design is available. Unfortunately no open-source VHDL synthesizer exists, so it is necessary to rely on Verilator. The tool can still be used to compile VHDL files and provide them to Yosys, which will parse them to build an RT-level Verilog netlist. As anticipated, such flow has been followed to test the toolchain on the AFTAB.

3.6 Verilator

Verilator is a Verilog simulator, which will be relied on for both the initial simulation and the estimation of each payload effect. It qualifies as a *cycle-based*[4] and *compilation-based*

[4]Cycle-based simulators evaluate signals only on one clock edge (once per cc), potentially missing intra-cc glitches.

simulator [52], which sacrifices a little bit of *event-based*[5] simulation accuracy to achieve a considerable speedup. As evident from Section 3.4, speed improvements are desirable due to the need to perform many simulations inside the flow, especially for payloads that need to be simulated one by one. These properties are the main reasons why Verilator has been chosen over adversaries open source event-based simulators, such as *Icarus Verilog*.
The workflow involves first a *Verilation* of the design, done by translating Verilog code into cycle-accurate `C++` or SystemC code. The Verilated code is then compiled to create object files implementing both the design and additional libraries (for coverage values collection and so on), which have to be linked together and executed. Simulation may involve both a Verilog testbench and a `C++`/SystemC one, the second one being absolutely necessary to drive the Verilated top-module inputs. This can be done through a set of APIs described in [53].
For what concerns this thesis work, the feature that is needed the most is logging signal values at each clock cycles. Luckily, Verilator has *Value Change Dump* (*VCD*) support, so they can be logged in this format. However, many benchmarks may last many clock cycles, making the VCD file very long and the parsing more time consuming. This is the reason why *Fast Signal Trace* (*FST*) has been used instead, since it shows an higher degree of compression with respect to VCD. Both an FST [54] and a VCD [55] parser have been included in the toolchain, since AFTAB testbenches limited complexity doesn't require FST support. The Verilator flow used in the thesis will be shown in the following chapters, together with discussions about the design choices.

3.7 RISCV-DV

RISCV-DV is a random assembly program generator that can potentially be used as a support tool for SBST writing. The real RISCV-DV tool requires UVM support to generate code, since it specifies API-based constraint in the form of UVM statements. Unfortunately no FOSS simulation tool supports UVM, so a Python port implemented by the same authors will be used instead. This port, dubbed *pygen-experimental*, is enough for this application, even if it lacks some of the features from the original. In this tool, ISA and code structure based constraints definition is done through *python-constraint* [56], a Python library which makes it possible to solve *Constraint Satisfaction Problems* (*CSP*s). *pygen-experimental* toolchain divides the problem of generating a random program in multiple parts, each one corresponding to an *instruction stream*. Multiple types of streams are available to test different parts of the DUT, where some instruction streams are built for a specific task (*load and store instruction streams*) while some of them are fully customizable. Each instruction and each stream are formalized as *problems*, which can be solved after imposing proper constraints. The typical flow is the following one:

```
# problem definition
self.problem = constraint.Problem(constraint.MinConflictsSolver())
# variables definition
self.problem.addVariable(self.instr_name, utils.riscv_instr_name_t)
self.problem.addVariable(self.instr_category,
↪  utils.riscv_instr_category_t)
self.problem.addVariable(self.instr_imm_t, utils.imm_t)
self.problem.addVariables([self.instr_src2, self.instr_src1,
↪  self.instr_rd], utils.riscv_reg_t)
```

[5] Event-based simulators perform a simulation cycle at every signal variation, capturing intra-cc behavior.

```
8
9  # constraints definition
10 if(not_system == 1):
11       self.problem.addConstraint(non_system, [self.instr_category])
12
13 # call to the solver and post-randomization constraints checks
14 self.solution = self.problem.getSolution()
15 self.post_randomize()
```

This snippet shows some of the variables and constraints that could be defined when generating an instruction. Variables definition can be performed through `addVariable()`, which receives as input the destination parameter together with a list of possible values it can assume. The next steps involves defining constraints over previously defined variables, which depend on the kind of instruction being generated. For example, at line 11 the instruction is constrained not to be a `SYSTEM` one. This check is enforced by `non_system()` during the problem solving phase (line 14), when each candidate solution gets checked against every constraint. After generating a valid solution, some post-randomization adjustments are performed in `post_randomize()`.
RISCV-DV main strength resides in the extensive support for customization, since every program (and DATA section) can be built by putting together streams whose characteristics are used definable. The only constraints that have been added are related to privileged instructions support, since it may be necessary to limit machine mode CSR usage to emulate real world applications. When tweaking stream parameters it is important to avoid configuring the tool to produce code that exceeds ELF section sizes, which proved particularly critical in the AFTAB toolchain (due to tight constraints in the linker script). Since Verilator is much faster than free Modelsim version (the tool used in the original AFTAB toolchain), it has been decided to switch environment and to extend section sizes in the script.

3.8 Evolutionary tool: Byron

Evolutionary algorithms are heuristic-based optimization algorithms, aiming to reproduce the mechanisms of biological evolution to improve a fitness value generation by generation. The majority of programs start from a pool of random individuals (candidate solutions), evaluating each fitness value **f**. After the evaluations, new individuals are generated by reproducing the ones which show the highest fitness values (through a *selection* operator). The reproduction process involves some additional operators to favor variability, mainly *recombination* and *mutation*. The first one swaps existing "*genetic material*" to create a new individual, while the latter introduces new characteristics by randomly changing some features.
For this thesis work, *Byron* has been chosen among possible environments. Byron's evolutionary algorithm can be customized with parameters like *entropy* and *temperature*. An high entropy forces the *selection* operator to keep a certain degree of separation between genomes of individuals in the pool, favoring a wide design space exploration, while temperature defines how likely it is for the algorithm to accept worse solutions hoping to escape from a non-global optimum (*hill-climbing*). The introduction of temperature makes the process of selecting fitting solutions a probabilistic one, exactly like in *simulated annealing*, where an high temperature means an high probability of selecting worse solutions. In this last case, temperature is decreased over time, assuming that the algorithm gets closer and closer to the global optimum as in Figure 3.1 (so it shouldn't

flee from that point).

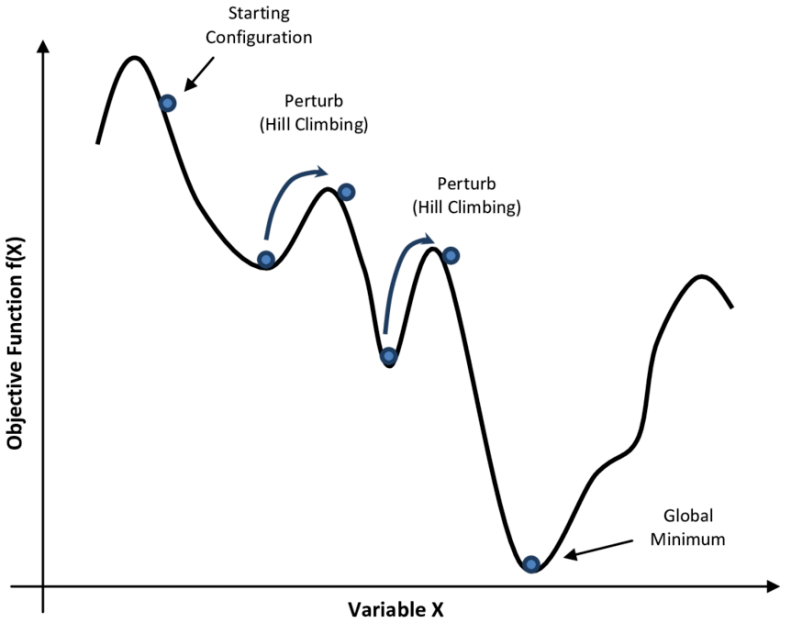

Figure 3.1: The *hill-climbing* effect regulated by temperature [1].

Byron is highly customizable, since the whole flow can be parallelized and many possible evaluators can be used. Let us show the two main components of a typical application: the *main script* and the *evaluator*.

3.8.1 Main script and library of macros

The first step in every application is the one to ensemble the structure of an individual, which is done by grouping together different `macros`. A macro could be anything, ranging from a single number to an instruction depending on the individual to be generated. The definition looks like this for a single number:

```
trigger_id = byron.f.choice_parameter([f"{n}" for n in range(0, MAX_NUM +
↪  1)])
macro = byron.f.macro('{v}', v = trigger_id)
```

where the macro can assume a integer value between 0 and `MAX_NUM`. Macros are then put together in the following way:

```
trigger_bunch = byron.framework.bunch(
        [macro],
        size=(NUM_TRIGGERS, NUM_TRIGGERS + 1),
        weights=[trigger_id.NUM_ALTERNATIVES],
    )
```

defining a `NUM_TRIGGERS`-sized `bunch` $\mathbf{n}$ of integers such that $n_i \in [0, \texttt{MAX\_NUM}]$. This array of numbers will be used as individual for the definition of the final triggers, where each n_i represents one of the N signals mentioned in Section 3.3. Bunches of macros have to be further combined together, creating a `sequence` of bunches:

```
frame = byron.f.sequence((main_prologue, trigger_bunch, main_epilogue),
↪  max_instances=1)
```

where `main_prologue` and `main_epilogue` are additional macros or bunches which may have different functionalities depending on the specific case.
Since macros and bunches definition are usually much more complex than the ones showed here, they are usually collected in a Python library that is then imported in the main script. This organization is used as a template for many Byron applications, especially the ones where individuals are actual code snippets (as for the *Go* one-max example in the repo [44]). After declaring the individual composition, the main script should define the evaluator with the following command:

```
evaluator =
↪  byron.evaluator.ParallelScriptEvaluator('./shell_evaluator.sh',
↪  "individual.s", other_required_files = additional_files, timeout =
↪  None, default_result = DEFAULT_RESULT)
```

Then the actual evolutionary algorithm is started, receiving as parameters the individual structure, the evaluator and additional *hyperparameters*:

```
final_population = byron.ea.adaptive_ea(
        top_frame,
        evaluator,
        max_generation=200,
        lambda_=20,
        max_fitness= [1.0,1.0,1.0],
        temperature=0.7,
    )
```

3.8.2 Evaluator

`ParallelScriptEvaluator` is not the only evaluator available. A list of some of the evaluators that have been used in *alpha* versions of the toolchain (or in other applications in the repo) is shown here:

- `MakefileEvaluator`, which allows the user to perform the evaluation as part of a *make* toolchain;
- `ScriptEvaluator`, which supports a generic bash script as evaluator. Each generation of individuals will be passed as argument to the script, which will output on *stdout* the fitness value of each program;
- `ParallelScriptEvaluator`, exactly as `ScriptEvaluator`, but it parallelizes evaluation on different individuals in the same generation. This flow will be presented in more details in Section 5.5, because it intersects with some design choices;
- `PythonEvaluator`, which performs evaluation based on a Python function.

3.9 HAL: the Hardware AnaLyzer

HAL is a reverse engineering tool to be used to gate-level mapped netlists, aimed at reconstructing high-level features, applying static detection algorithms or simply editing

the netlist structure. Netlists are represented as graphs, where gates are nodes and wires are edges. Thanks to this abstraction, it will be possible to parse the netlist through recursive graph search algorithms, enabling metrics evaluation for each wire. HAL will not only be used to merge new payload circuitry with the original one, but also to demonstrate mutual *topological independence* between selected triggers. Topological independence is a desirable property, because it ensures that a transition on a trigger doesn't immediately cause a transition on another one (it would make the whole trigger condition weaker). This kind of analysis should be repeated on the payload, in order to prevent the creation of a combinational loop trigger-payload-trigger.
Netlist exploration is done through APIs, which allow the user to retrieve the wanted net by name or by ID and then to move from there through the following set of commands (for backward movement, forward movement relies on specular APIs):

```
1 netlist = NetlistFactory.load_netlist("top.v", "cmos_cells.lib")
2 first_net = netlist.get_net_by_id(net_id)
3 source_endpoint = first_net.sources()[ENDPOINT_INDEX] # explore only in
  ↪ the direction of a particular endpoint
4 for net in source_endpoint.get_gate().get_fan_in_nets():
5     recursive_fanin(netlist, net.get_id(), net_marker, level + 1)
```

Listing 2: A conceptual netlist exploration with HAL APIs.

The first important command is `load_netlist()`, which builds the graph starting from the netlist Verilog description and the gate library liberty file. `get_net_by_id()` is used to retrieve a Net object from the netlist which matches the `net_id`, that should be known in advance (in alternative it is possible to match by name). `sources` identifies the gate which drives this net, making it possible to move backward. In circuits with no high-Z buffers each net has a single driver, so `ENDPOINT_INDEX` is usually 0. It will be assumed that the processor under analysis doesn't rely on the high-Z state, because otherwise a *two state* simulator like Verilator wouldn't be able to simulate it. The last for loop (lines 4 and 5) iterates on each input of the gate driving the initial net, calling `recursive_fanin` once per input. `recursive_fanin` follows the same flow shown here, but it requires additional terminating conditions. These may depend on how far the exploration goes, but in the general case it possible to go up to the primary inputs.
By including checks in this flow, it is possible to implement whichever static analysis algorithm among the ones described in 2.3.2. It is also possible to cut away some particular cells (like FFs), adapting the circuit for the application of techniques like *ANGEL* [18]. Case by case explanations will follow in the detailed description of the toolchain.

3.10 SymbiYosys

The last tool that will be discussed is *SymbiYosys*, a Yosys frontend which generates TCL scripts starting from an higher level description: the *sby* script. As anticipated, this new level of scripting makes access to SAT solving features easier, for example by finding the best SAT engine automatically (through *autotune*) and parallelizing multiple SAT engines runs by mapping them to separate cores. SymbiYosys is actually a suite of multiple tools. One of them is the versatile `sby`, that can be used for every formal verification task. Another tool is `eqy`, which has been conceived for equivalence testing. In the following analysis only `sby` will be used, but `eqy` is another viable alternative when it comes to find a pattern that propagates a malicious payload values to the primary outputs. In order to use `sby` for equivalence-checking, however, it is necessary to create

a *miter* between the golden circuit and the infected one. Here is a typical SymbiYosys script, showing all the parts that are needed in the current discussion:

```
1  [tasks]
2  prove : default
3
4  [options]
5  prove:
6  mode prove
7  depth 5
8
9  [engines]
10 smtbmc boolector
11
12 [script]
13 read -sv top_no_map.v
14 read -sv top_no_map_infected.v
15 read -sv miter.sv
16 prep -top miter
17 setundef -zero -undriven -init
18
19 [files]
20 top_no_map.v
21 top_no_map_infected.v
22 miter.sv
```

Each `sby` script is divided in multiple sections. The mode (from the *options*, line 6) defines how the engine should operate, in this case it should prove or disprove the `assert` statements in the code. In the same section, the depth parameter is used to indicate the sequential depth of the exploration. Since sequential SAT solving is a very complex problem, this value should be kept low and additional measures should be enforced to help the engine. The actual engines to be used are listed in the homonymous field, plus it is possible to instruct `sby` to perform attempts with multiple engines and select the one which performs best. The way to do this is through the following command:

```
sby --autotune -f srv32_miter.sby
```

which can potentially make this method competitive with the simulation based alternative (refer to the last point in Section 3.4), since it parallelizes the flow by evaluating each engine on a separate core. The remaining part of the script consists in the actual commands to be executed, which are in charge of parsing the verilog files (`read`), building the hierarchy (`prep`) and setting an initial value for wires which are not initialized in the netlist (`setundef`). This last command is the key in optimizing the engine work, because imposing some wires values limits the degrees of freedom in the generation of input patterns. Section 5.6.4 will make it clear how to use this principle to help generating payload-propagating patterns through simulation and `initial` statements.

Chapter 4

State of the art gate-level detection and injection techniques

4.1 Introduction

Developing a meaningful injection framework requires both breaking some of the up-to-date detection techniques and building on top of the existing injection tools. Since the analysis in this thesis is focused on a very specific problem in the HT domain, which is the gate-level injection, all metrics and countermeasures that are used at that level to spot suspicious circuitry will be listed. The majority of the concepts that are introduced here have been tackled in the toolchain too, while implementing the remaining ones will be suggested as future work. The two main categories of detection techniques, *static* and *dynamic* ones, will be analyzed together with testing techniques, detailing their evolution over time. Some strategies to elude them that have been developed in previous works will be presented too.

4.2 Dynamic techniques

As anticipated in section 2.3.3, this class of techniques relies on simulation to compute parameters which may hint about the presence of an HT. Toggle and value coverage are the main indicators, since trojan circuitry should be activated very rarely during execution (ideally never during testing). An analysis which targets these values could potentially spot both candidate triggers and trojan internal wires. Dynamic and static techniques can coexist, since *switching activity* and *0/1-probability* can be computed also by dynamically evaluating 0/1-probabilities on the inputs and then propagating them through the netlist via static algorithms [57]. The key limitation for simulation-based techniques is related to the need for an exhaustive testbench, capable of exciting all wires in a way that is compatible with what will happen during normal operation. In the following paragraphs *UCI* (Section 4.2.1) and *VeriTrust* (Section 4.2.2) will be presented. These are two dynamic algorithms that still find space in nowadays papers, despite having been developed in 2010 and 2013 respectively.

4.2.1 Unused Circuit Identification (UCI)

UCI is built on a very simple but effective idea, which consists in the identification of *unused* circuits during simulation. Zhang *et al.* [22] formulated a definition of (un)usability (based on Hicks original work [19]), which prescribes that "if any pair of related signals are equal throughout all test cases, ... [then] the circuit ... [in-between] can be regarded as

unused circuitry". Circuits which fall under this definition are marked as suspicious, and they will be inspected manually by a verification engineer. Such condition can identify every trojan which rely on a simple `AND` or a `MUX` to separate between normal execution and malicious payload behavior. When UCI verification tools have been developed originally, the majority of trojans on *TrustHUB*[1] belonged to this category. A first take on trojan analysis through UCI was done on RTL netlists, where a vulnerable HT looks like this:

```
always @ (posedge clk)
    if (pattern == PATTERN|counter == COUNTER)
        t_en <= 1'b1;
    else t_en <= 1'b0;
always @ (posedge clk)
    if (t_en == 1'b1) f <= f_m;
    else f <= f_n;
```

This trojan relies on a single bit (t_{en}) as a multiplexer selection signal. If this trigger is never asserted during simulation, the multiplexer will always output f_n and will be marked as *unused* circuit due to UCI recognizing (f,f_n) as equivalent. The same applies for the gate which perform t_{en} driving, so t_{en} is flagged as *unused* too. Zhang's paper describes a possible way to bypass UCI checks, which requires dividing the *pattern* in k sub-patterns and the original circuit output f_n in two subfunctions f_{n_1} and f_{n_2} weighted by two coefficients c_1 and c_2. The main purpose of these steps is to make it impossible for the algorithm to find couples of nodes which constantly assume the same value. In particular, the single t_{en} has been replaced by k t_{en_k} intermediate signals, which are then `AND`ed together to find the final trigger (not explicitly defined as a signal in the netlist). The f_n definition is rewritten as $c_1\&f_{n_1}|c_2\&f_{n_2}$ since in this way f is driven by both f_{n_1} and f_{n_2}, making it impossible to identify the couple (f,f_n) from the original example. A proof of concept of the code is reported here:

```
always @ (posedge clk)
    if (pattern_1 == PATTERN_1|counter_1 == COUNTER_1)
        t_en_1 <= 1'b1;
    else t_en_1 <= 1'b0;
    ...
    if (pattern_k == PATTERN_k|counter_k == COUNTER_k)
        t_en_k <= 1'b1;
    else t_en_k <= 1'b0;
end
always @ (posedge clk)
    f <= ((t_en_1&...&t_en_k)&f_m)|(~t_en_1|...|~t_en_k)&(c_1&f_n1|c_2&f_n2);
end
```

As an important note, this redefinition is actually invulnerable to UCI only if:

- each t_{en_k} is justified to 1 at least once, so the pattern partitioning is balanced;
- each f_{n_i} is able to drive the f value at least once, and at that time $f_{n_i} \neq f_{n_j}$ (for $i \neq j$). This imposes a constraint on c_1 and c_2, which should be balanced and allow for this condition to happen.

[1] https://trust-hub.org/

The additions to be introduced play on RTL netlist limited signal scope, avoiding to introduce new signals for f_m and $f_{n_{1/2}}$ related parts. In case a new signal for f_n is introduced, replacing line 11 with:

f_n <= $c_1 \& f_{n_1} | c_2 \& f_{n_2}$;
f <= $(t_{en_1} \& ... \& t_{en_k}) \& f_m | (\sim t_{en_1} | ... | \sim t_{en_k}) \& f_n$;

then f and f_n would be equivalent according to UCI, thus invalidating the approach. Since the toolchain aims to work at a lower level, not-explicitly defined signals shouldn't be considered for disguising the trojan (gate-level has visibility on every wire outside the cells). For these reasons, the strategy to be followed is the one proposed by Sturton *et al.* [2], who managed to find some gate-level structures that don't show UCI-equivalent couples. One of them is reported in Figure 4.1, and it consists of two chained `MUX`es. As the truth table in Figure 4.2 clearly reports, no signal is equivalent to the output when $(\sim t_1 | \sim t_2)$, which corresponds to the condition when the global trigger $t_1 \& t_2$ is not active. It is important to note that the two logic functions in trigger and non-trigger conditions differ ($i_0 \& i_1$ and $i_0 | \sim i_1$ respectively), so the circuit implements a malicious payload.
Both the previous examples suggest that the key in defeating UCI resides in reformulating the logic function to avoid making the real functionality explicit, since the algorithm simply tries to "short-circuit" the output f with an intermediate equivalent signal f_n bypassing potentially malicious and redundant logic in the middle. In other words, the main vulnerability of this method resides in its inability to modify the logic functionality of the circuit. Another critical point, which plagues almost every method, is related to the complexity of implementing this technique in a real circuit. According to the algorithm, every couple of signals among the n ones in the netlist should be considered to attempt to demonstrate equivalence. This check is usually done by adding testing hardware, which performs an `XOR` between the two signals in each couple and it is responsible for an increase in the simulation duration. When the simulation is done, a check over each result should be performed, to assess if the `XOR` evaluated as true at least once: this accounts for a complexity of $\mathcal{O}(n^2)$ if a FF is used to store the `XOR` result, or $\mathcal{O}(tn^2)$ in case the result has to be examined at every clock cycle among the t ones.

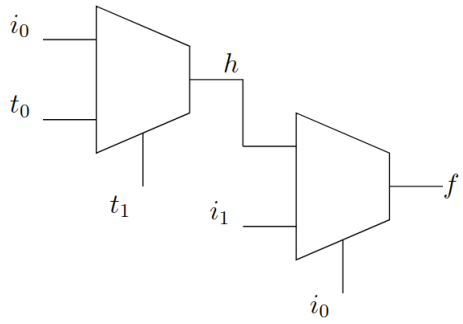

Figure 4.1: A `MUX` based circuit with no UCI-equivalent couples of signals [2].

t_1	t_0	i_1	i_0	h	f	Comments
0	0	0	0	0	0	
0	0	0	1	1	0	$h \neq t_1$, $h \neq t_0$, $h \neq i_1$
0	0	1	0	0	0	
0	0	1	1	1	1	
0	1	0	0	0	0	
0	1	0	1	1	0	$f \neq t_0$, $f \neq i_0$, $f \neq h$
0	1	1	0	0	0	
0	1	1	1	1	1	
1	0	0	0	0	0	
1	0	0	1	0	0	$h \neq i_0$
1	0	1	0	0	0	$f \neq t_1$, $f \neq i_1$
1	0	1	1	0	1	
1	1	0	0	1	1	Trigger condition is true.
1	1	0	1	1	0	Trigger condition is true.
1	1	1	0	1	1	Trigger condition is true.
1	1	1	1	1	1	Trigger condition is true.

Figure 4.2: The truth table of the circuit on the left [2].

4.2.2 VeriTrust

In order to compensate for UCI sensibility to how the logic function is implemented, Zhang *et al.* [14] designed a new algorithm to identify redundant inputs in the truth table or Karnaugh map (*K-Map*). Such an approach is independent on how gates are structured to implement the function, because it considers only the inputs and the outputs. The novel technique, named *VeriTrust*, proves to be able to identify circuits which are invisible to UCI as the *MUX*-based one. Let us spend some additional words on this last circuit, whose K-Map is reported in Figure 4.3 on the left. It is possible to notice immediately how this kind of trojan extends the dimensionality of the trojan-free map (which was 4x1, consisting only in i_0 and i_1), replicating the (0,0,1,0) on every column except for (t_1, t_0) = (1,1). Such class of trojans is defined as *parasite-based* in [14], where another example of an UCI-invulnerable trojan belonging to this category is reported. As Zhang *et al.* showed, gate-level trojans which aim to be stealthy are supposed to belong to this category. For this reason, VeriTrust will be built to detect them. The algorithm flow consists in the following steps:

1. defining a proper verification test suite, which sensitizes as many rare conditions as possible. Ideally the objective is to obtain full coverage over every genuine circuit functionality, while trojan-related circuitry is assumed not to be activated during tests;

2. identifying truth map lines or K-Map entries that are not covered, replacing them with *don't cares* (X);

3. verifying if some inputs have become redundant as a result of don't cares insertion.

According to *Lemma 3*[2] in [14], by following this technique it is guaranteed to catch trojan related inputs. However, since full-coverage on genuine functionalities is not achievable for realistic designs, there is the risk for some benign inputs to be marked as trojan-related (producing *false positives*). As a consequence, manual inspection by designers is required to make sure that X inputs indicate malicious functionalities.
What makes VeriTrust shine with respect to UCI is the possibility to actually modify the logic function to exclude potential trojans. Identifying redundant inputs is only the first step in fixing the circuit, since it is also possible to re-implement the circuit from scratch excluding these signals. In particular, if the new K-Map for circuit 4.1 is examined after VeriTrust application (Figure 4.3), it can be noticed how the new red implicant makes the 4-variables function equivalent to $f = i_0 \& i_1$. Due to that, the designer could simply decide to redesign the circuit excluding t_0 and t_1. This algorithm shows an even higher complexity than UCI, since it requires verifying that each potentially redundant input doesn't cause a variation in the value of the function when switched (after defining the unsensitized products as don't cares). Given n inputs, this task is $\mathcal{O}(2^n)$. In order to make this procedure faster, Zhang *et al.* [14] proposed some additional heuristic-based *checkers* which try to find redundant inputs in less expensive ways. Another key vulnerability in VeriTrust is the possibility to apply it only to combinational logic, which makes it unable to spot trigger values hidden in multiple sequential stages. A technique to leverage this behavior has been proposed as part of the *DeTrust* [3] framework, and it will be implemented in the toolchain in a generalized form. The results which follows are partially present in the paper. Let us suppose to have this generic logic function which is altered

[2] "*For a signal affected by a parasite-based HT, if all entries of the malicious functionalities of the HT are set as don't-cares, the dedicated trigger inputs used to activate HTs become redundant.*"

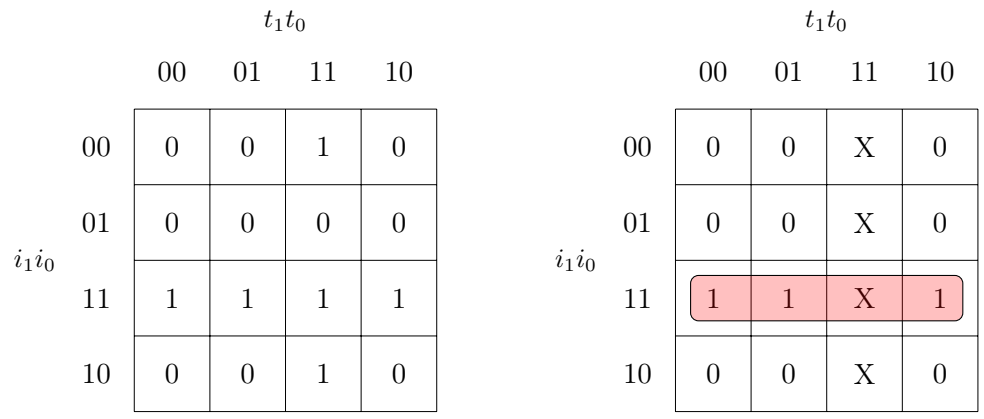

Figure 4.3: K-Maps before (left) and after (right) the application of VeriTrust.

by a trojan with a single on-set term[3]:

$$f = f'_n + (c_{n_0}p_{n_0} + c_{m_0}p_{m_0}) \tag{4.1}$$

where $c_{m_0}p_{m_0}$ is the malicious on-set term and $c_{n_0}p_{n_0}$ is a benign term which shares at least a literal with the malicious term. It is possible to collect the common literal c^cp^c to obtain:

$$f = f'_n + c^cp^c(c^r_{n_0}p^r_{n_0} + c^r_{m_0}p^r_{m_0}) \tag{4.2}$$

where $c_{n_0} = c^cc^r_{n_0}$, $p_{n_0} = p^cp^r_{n_0}$, $c_{m_0} = c^cc^r_{m_0}$, $p_{m_0} = c^cp^r_{m_0}$. Equation 4.2 can be further rewritten as:

$$f = h_1h_2 + h_3 \tag{4.3}$$

defining $h_1 \coloneqq c^cp^c$, $h_2 \coloneqq c^r_{n_0}p^r_{n_0} + c^r_{m_0}p^r_{m_0}$, $h_3 \coloneqq f'_n$. An example of this procedure is reported in Figure 4.4. The key point in ensuring stealthiness is allocating h_1,h_2 and h_3 computations in different sequential levels and choosing properly both the $c_{n_0}p_{n_0}$ and the c^cp^c. A formal proof about how the new structure doesn't show redundant inputs is reported in [3]. Just for the sake of simplicity, let us assume that:

$$c_{m_0}p_{m_0} = xyz$$

where x,y and z are three intermediate triggers. c^cp^c will then be a literal standing for one of these triggers. Intuitively it can be observed that:

- h_3 is not redundant, since f'_n is a genuine function from the original circuit.
- provided that c^cp^c is justified to 1 at least once during execution, then h_1 will not be redundant.
- h_2 is a critical one, because a proper value of c^cp^c has to be chosen such that neither $c^r_{m_0}p^r_{m_0}$ nor $c^r_{n_0}p^r_{n_0}$ are redundant (both the terms should evaluate at one at least once during execution).

[3]An on-set term is one of the products in the *Sum of Products* (*SOP*) form.

- h_1h_2 product in f computation in the second sequential stage should not be redundant too. This imposes a strict condition too, but it can be managed again by properly selecting both c^cp^c, which should be the most probable trigger, and $c_{n_0}p_{n_0}$, which should contain a $c^r_{n_0}p^r_{n_0}$ term that is sensitized frequently.

This method makes it possible to build a two level sequential circuit, where the first literal c^cp^c of the trigger string is collected as h_1 and the remaining ones are part of h_2. In case the residual trigger string is too long or still difficult to sensitize, another iteration of the algorithm can be performed inside h_2. Indeed h_2 could be renamed as h_{20} and rewritten as:

$$h_{20} = f'_{n_1} + (c_{n_1}p_{n_1} + c_{m_1}p_{m_1})$$

and then $c^c_1p^c_1$ should be collected to obtain:

$$h_{20} = f'_{n_1} + c^c_1p^c_1(c^r_{n_1}p^r_{n_1} + c^r_{m_1}p^r_{m_1})$$

where $c_{n_1}p_{n_1}$ and $c_{m_1}p_{m_1}$ are defined in the same way as in Equation 4.2. The generic h_{2i} looks like this:

$$h_{2i} = f'_{n_{i+1}} + c^c_{i+1}p^c_{i+1}(c^r_{n_{i+1}}p^r_{n_{i+1}} + c^r_{m_{i+1}}p^r_{m_{i+1}}) \tag{4.4}$$

This could ideally go on until a single trigger is left. However, there is a practical limitation which usually makes it very difficult to go past 3/4 levels. The mechanism exploited is a kind of *retiming*, since the f expression is being rebuilt by changing FFs allocation without affecting the original timing. The flow is presented in Figure 4.5, where both the old f' value (named *f_ old* in the schematic) and the new f one are shown for a two-level implementation. In order to find the new function, it is necessary to backtrack from the final signal to find the fanin cone of the last sequential stage, then FFs should be excluded and backtrack should be performed again to find the fanin for each FF in the layer before. In this way a function f is obtained expressing the dependence between the outputs of the FF group 1 and the final payload, ignoring the second group of FFs. This makes it possible to reallocate part of the second group as in figure, changing the two functions f'_1 and f' in f_1 and f. In case new stages have to be added, then it would be necessary to further backtrack from the fanin of group 1. In a real circuit, usually it is impossible to backtrack for more than 3/4 stages, so this poses a limitation for this approach. Besides, attention should be paid to particular hardware structures (as counters) which may need FFs to be allocated in a precise way to work properly.
Since the function f can be written as in Equation 4.3 and each h_i is computed in the previous sequential stage, no PI should be included in f'. If a f' which includes PIs is selected, then these PIs would appear in the expression for each h_i. Because of that, they would be moved to the previous sequential stage, causing the circuit timing to change. This limitation is not very strict, because it is enough to avoid collecting the whole f'_n

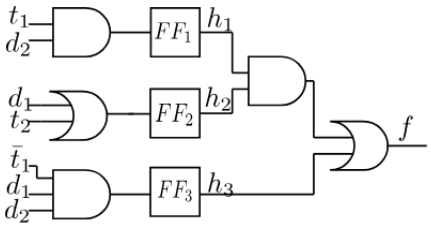

Figure 4.4: A typical payload structure designed to be invulnerable to VeriTrust. It is possible to recognize the form $f = h_1h_2 + h_3$ introduced in Equation 4.3 [3].

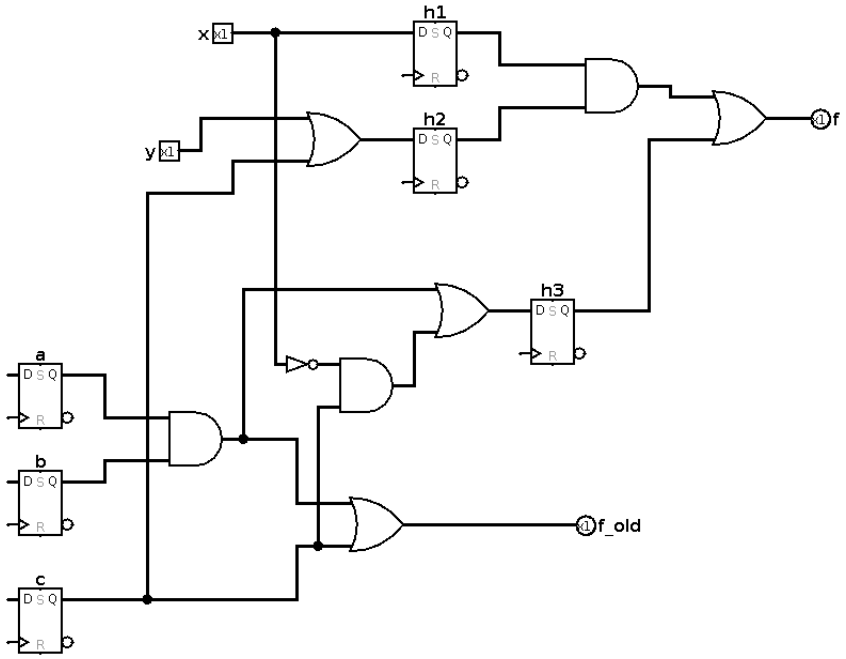

Figure 4.5: A 2-levels payload structure for masking an xy trigger in $f' = ab + c$.

inside h_3. Let us assume that terms including the PIs literals can be separated from the other ones. Equation 4.1 can then be rewritten as follows:

$$f = f''_n + f'''_n + (c_{n_0} p_{n_0} + c_{m_0} p_{m_0})$$

where f''_n is the group of terms which include only PIs and outputs of the second group of FFs, while f'''_n plays the role of h_3. Collecting $h_2 = c^c p^c$ results in the following expression:

$$f = f''_n + h_3 + h_1 h_2$$

where f''_n can be computed in the last sequential stage.
In the following sections it will be discussed how to combine DeTrust against Veritrust with other techniques, circumventing some of the limitations that have been shown.

4.3 Static techniques

Static techniques are the ones which do no rely on simulation, thus overcoming the problems arising from incomplete test suites. The majority of algorithms belonging to this category show a considerable complexity, so performances are usually a critical point as for dynamic techniques. Analyzing the netlist usually consists in defining a metric and computing the corresponding value for each element in the circuit, which could be an actual wire or a structure at an higher abstraction level (as will be presented when discussing *FASTrust* [4], one of these algorithms). In order to mark a circuit element as suspicious or benign, a *threshold* value is needed as reference. Choosing this value is a delicate matter, due to how different circuit functionalities influence the structures, and a poorly chosen value could lead to many *false negatives* (*FN*s) or *false positives* (*FP*s). When it comes to manual inspection, which is usually required after algorithms applications, a huge number of FPs can be much worse than a FN due to the huge time overhead that the former implies. Besides, there is still the opportunity to spot the FN by applying a different algorithm afterward. In the following, four static techniques will be

detailed: *FANCI*, *ANGEL*, *FASTrust* and *COTD*. The toolchain up to now implements trojans which aim to bypass the first two, but an extension to target the remaining ones will be proposed.

4.3.1 FANCI

FANCI [15] has been a reference for static detection algorithms for many years. The main idea behind the algorithm is to identify weakly-affecting inputs through an analysis of the truth table, built on the assumption that such inputs are abundant in trojan structures and very rare in genuine ones. FANCI creators introduced a metric named *control value* (CV), which is computed for each signal and compared with a threshold. The CV of signal x_i with respect to output f is defined as follows [4]:

$$CV(x_i, f) \coloneqq \frac{1}{size(TT)} \sum_{k=0}^{size(TT)} |f(..., x_i = 0, ...) - f(..., x_i = 1, ...)| \tag{4.5}$$

where $size(TT) = 2^n$ is the size of the truth table for f with n inputs. The function f is initially evaluated at index k, then the value of x_i is complemented and a new evaluation is done. To conclude, a comparison between the two f values is performed: if they are equal then the result is increased by 1, otherwise it remains as it was. In synthesis, the CV counts how many times flipping the x_i leads to a change in f value, acting as a metric to quantify how x_i affects f. As a side note, Equation 4.5 counts every flip twice, so only half of the TT should be explored (either where $x_i = 0$ or $x_i = 1$).
Since the previous formula is $\mathcal{O}(2^n)$, a method to compute an approximate version of the CV is needed. The best way to do so is to select K random values from the truth table, redefining 4.5 as:

$$CV(x_i, f) \coloneqq \frac{1}{K} \sum_{k=0}^{K} |f(..., x_i = 0, ...) - f(..., x_i = 1, ...)| \tag{4.6}$$

It is important to notice that the exploration of the truth table is not done in order, so 4.6 is not a simple truncation of 4.5. By employing random selection, it is guaranteed that a potential hacker doesn't know which part of the truth table will be explored.
The native implementation of FANCI works only on combinational levels. For this reason it proves to be very strong when identifying *single-triggered* trojans, the ones whose trigger mechanism is in a single combinational stage. An example of a simple combinational circuit is reported in Figure 4.6, and its malicious counterpart in Figure 4.7 [15]. The former is a simple `MUX`, for which it can be verified that:

$$CV(d_0) = CV(d_1) = CV(d_2) = CV(d_3) = 0.25$$

$$CV(s_0) = CV(s_1) = 0.5$$

The latter is a malicious MUX, whose output is complemented when every trigger evaluates to 1. For this circuit, these relations hold:

$$CV(t_0) = CV(t_1) = ... = CV(t_{63}) = \frac{1}{2^{65}}$$

FANCI will thus mark the 64 triggers as suspicious, identifying the trojan. [14] reports the following inequality for estimating CV bounds in a circuit with n benign inputs and m malicious ones:

$$\frac{1}{2^{n+m-1}} \leq CV(t_i) \leq \frac{1}{2^{m-1}} \tag{4.7}$$

which states an exponential dependency between the number of malicious inputs m and the upper bounds for the control value. Because of this dependency, it is convenient to reduce the number of malicious inputs for each combinational stage. This can be done by separating the triggering logic on multiple sequential stages, as in Figure 4.8. In this case, each stage has two inputs except for the last one which has 3. As a consequence the highest value for CV will be $\frac{1}{2^3}$, which is compatible with values that can be computed in normal circuits.
The next Section will be dedicated to an extension of FANCI which goes under the name of *ANGEL*, which exploits graph-cutting and neighborhood analysis to compute CVs across different sequential stages.

4.3.2 ANGEL

ANGEL [18] has been conceived as a way to extend FANCI without penalizing the performances and overcoming the limitations. Some simple multilevel versions of both FANCI and VeriTrust, namely *FANCIX* and *VeriTrustX*, have been proposed in previous works [58], but they simply bypass FFs without changing the core computation of the CV. As a result, all these methods suffers from spikes in complexity when it comes to big circuits, due to the need to go all the way back to PIs to compute the CVs. ANGEL proposes a completely new algorithm, in which control values are associated with a graph cut c instead of a single signal. In this way it is possible to spot low CV signals that are spatially close, which is a condition typically verified in trojan architectures. This addresses an additional problem of FANCI, which is the abundance of false positives (mainly due to the inability of CV to describe multi-input structures). The cut can also traverse multiple sequential stages, thus capturing some complex trigger patterns that could escape FANCI. The algorithm can summed up as follows:

- for each gate G a feasible cut C is selected. This is done by applying a backward breath-first search in the netlist up to depth d.
- then, the truth table of the cut should be computed, associating each input with each output.
- for each output of the cut it is possible to compute a control value CV. This is done by iterating on the inputs, computing an heuristic for each of them and then accumulating the results in the final CV. If the final value is higher than a threshold, then the cut is marked as suspicious.

According to this flow, ANGEL complexity is $\mathcal{O}(gm2^m)$, where g is the number of gates, m is the number of inputs to the cut. This complexity is exactly as FANCIX complexity,

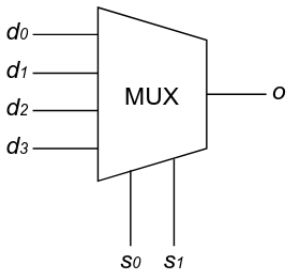

Figure 4.6: A simple `MUX` [3].

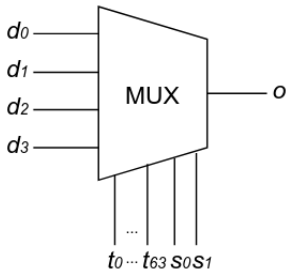

Figure 4.7: A malicious `MUX` with 64 triggers [3].

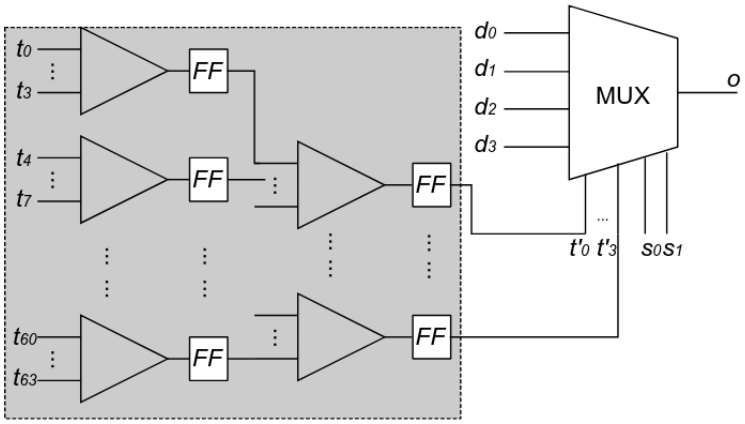

Figure 4.8: A `MUX` equivalent to the one in Figure 4.7, but invulnerable to FANCI [3].

since this last method could be considered as a particular case of ANGEL when the cut is equal to the whole netlist. Both in FANCIX and ANGEL the exponential part is actually bounded, because only a random portion of the table is computed. For this reason, the real complexity is $\mathcal{O}(gm)$. This technique is among the most effective ones against gate-level trojans, but a possible way to elude it will be presented as part of the toolchain. As a side note, HAL uses an heuristics to detect LFSRs, invalidating modern trojan designs based on pseudo-randomicity [58].

4.3.3 FASTrust

FASTrust [4] has been developed as a way to address FANCI and VeriTrust vulnerabilities. It is based on a strategy which differs from the one implemented in ANGEL, since it doesn't work directly on the netlist. In particular, FASTrust is a detection method which performs feature analysis on the *FF-level control-data flow graph* (*FF-CDFG*). The first step when applying this technique is the one to build the FF-CDFG for the circuit of interest, which will show FFs and output nodes as nodes and logical dependencies between them as edges. The authors have analyzed the FF-CDFG of many trojan affected circuits, defining features which identify each trojan class. In the original paper, these features were proposed:

- **Feature 1**: "*all nodes in the trigger circuit of a time-triggered HT form a large loop group*". A *time-triggered HT* is basically a time-bomb, which eludes testing by relying on a very large counter. A *loop-group* is a group of nodes with are connected and present a self-loop each.
- **Feature 2**: "*A single-triggered HT contains a node with extremely large in-degree*". This is a property which identifies purely combinational trigger mechanisms, which receive a huge number of inputs to check if a rare pattern occurred.
- **Feature 3**: "*A sequential-triggered HT contains a loop group whose total in-degree from outside nodes is extremely large*". The obvious reason behind this rule is that a typical sequential trigger captures the values of the same combinational signals on multiple cycles, increasing a counter every time a rare pattern is verified.
- **Feature 4**: "*The nodes in an implicitly-triggered HT have small out-degrees*". This feature describes all the trojan generated through DeTrust, which usually work on separate FFs elements where distinct parts don't interact with each other.

Since the first three types of trojans can be easily detected through basic FANCI or VeriTrust, the following analysis will target only the last feature. Figure 4.9 and Figure 4.10 show the two FF-CDFGs corresponding to DeTrust *MUX* trigger and the implicit trigger respectively. Let us ignore the red edges for now, since they will be explained in due time. FASTrust feature 4 verification is based on the following algorithm:

1. define a *threshold* for the minimum out-degree that passes the check.
2. select all the nodes whose $out\_degree \leq threshold$ and collect them in a list.
3. select all the couples of nodes in the list which are connected by an edge or which share a successor, and include them in separate groups.
4. for each group, check if there is a node in the CFG whose predecessors or successors are all contained in the group: if so, add the node to the group.
5. repeat the previous step until it becomes impossible to add new nodes.

The reference paper written by Yao *et al.* [4] clarifies why the previous algorithm is able to spot every malicious part (the red ones) for both trojan structures in Figure 4.9 and Figure 4.10.
From now on, let us assume $threshold = 1$ to be conservative.
The key problem in the structure on the right is represented by the final convergence, since the various FF_X nodes are the ones already included in step 2) due to their out-degree being equal to one. Since the FF-CDFG doesn't describe the internals of a single combinational stage, it is possible to select as a payload a signal which is in the fanin cone of many FF inputs. In this way the out-degree of each FF can be artificially increased, making it more difficult to populate the list in step 2). As stated in the paper, it could be possible to increase the threshold to cover this case. As a collateral result, however, the list of false positives may become too large. An extension of FASTrust, dubbed *ML-FASTrust* [59], has been created to extend the algorithm at the combinational logic (CL) level, making the analysis a multilevel one. A typical CL-CDFG looks like the one in Figure 4.11. This new approach targets only the positive results, for which the lower level CFG is built and more specific properties are checked. Targeting only a small subset of all the combinational parts is what makes this technique effective and with a low time overhead.
For the DFG on the left, instead, the whole tree structure is vulnerable, because every

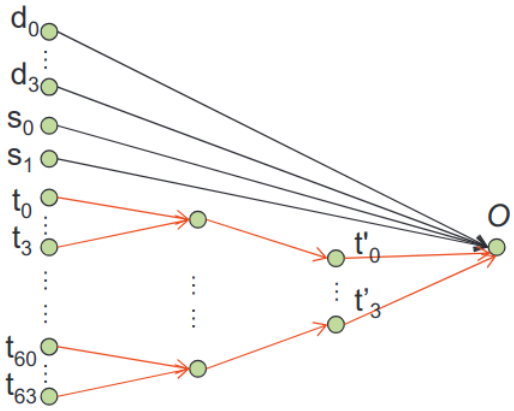

Figure 4.9: The FF-CDFG for the `MUX` trigger structure in Figure 4.8 [4].

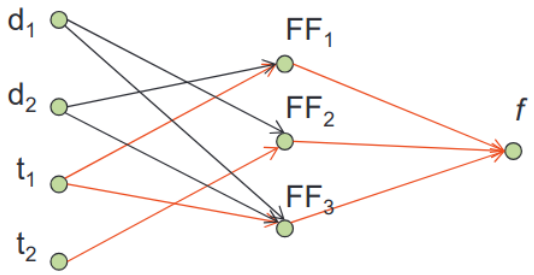

Figure 4.10: The FF-CDFG for the implicit trigger structure in Figure 4.4 [4].

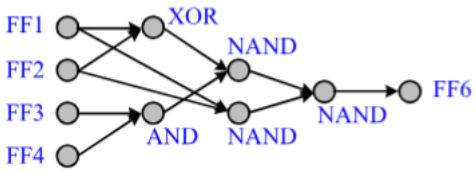

Figure 4.11: A CL-CDFG for a single combinational stage between memory elements [4].

node is connected to a single output. Making the whole design stealthier is definitely harder than before, because there are many more wires whose out-degree should be increased. A possible way to do so could consist in reformulating the logic function for each tree[4], making different trees intersect. In this way, each green node would point to multiple successors. As part of the discussion of the toolchain, a new technique will be presented to make this network more resistant against FASTrust.
(ML-)FASTrust is one of the best algorithms among the state of the art techniques, due to its to ability to spot a wide variety of trojans according to different metrics. Even if it may look like the definitive method, there is still space for some improvement. One of the problems could reside in the necessity to update the features in order to keep track of recent developments in trojan design, so the method depends on how features are defined in order to work correctly. This could make FASTrust less effective with respect to a technique based on an absolute matric, such as FANCIX or VeriTrustX. On the other hand, there is no way to tell if metrics as control values or value/toggle coverage could be applied in describing whichever class of trojans. This vulnerability shows also a good side, because a feature-based approach could ideally be translated in a machine learning model (as for *COTD*). Another problem which rose when applying FASTrust, and was only partially addressed by ML-FASTrust, is the abundance of false positives. The most critical case is related to feature 2, which may potentially identify many regions in a design which performs intensive computation over data (as cryptographic modules). The reason behind this resides in data high parallelism (32 or 64 bits), which leads the aforementioned rule to erroneously mark arithmetic and data manipulation units as suspicious.

4.3.4 COTD

Controllability and Observability for Hardware Trojan Detection (*COTD*) is a ML based technique which relies on testability metrics to identify trojan related wires. The powerful feature of this method resides in its ability to detect deTrust masking circuitry, because its way of dividing a combinational function in multiple sequential stages doesn't affect testability. The metrics that are used are taken directly from the *Sandia Controllability and Observability Program* (*SCOAP*), which defines both sequential and combinational observability and controllability values. Controllability values can be additionally computed both for 0 and 1, since forcing a value can be done more easily with respect to forcing the opposite one. Details are provided by Goldstein *et al.* [60], but the general idea is described below. This flow can be applied both for combinational and sequential values, so $X \in \{S, C\}$ will be used in general, except when referring to a particular category:

1. set the default controllability values for the PIs:

$$CC^0(I) = CC^1(I) = 1, SC^0(I) = SC^1(I) = 0$$

[4]From this point on, *tree* will be used as a way to refer to the fanin cone of t'_0,t'_1,t'_2 and t'_3,

and the default observability values for the POs:

$$XO(U) = 0$$

and for the other nodes:

$$CC^0(N) = SC^0(N) = XO(N) = \infty$$

.

2. for the combinational controllability it is necessary to start from the inputs and move toward the output, moving gate by gate and computing the following values for each node (after including a new gate):

$$CC^0(N) = \min_{\forall \mathbf{x}=[x_0,...,x_M] \in S} \sum_{m=0}^{M} CC^{x_m}(X_m) + 1 \tag{4.8}$$

$$CC^1(N) = \min_{\forall \mathbf{x}=[x_0,...,x_M] \in \overline{S}} \sum_{m=0}^{M} CC^{x_m}(X_m) + 1 \tag{4.9}$$

defining $S \coloneqq \{\mathbf{x} \in TT_f \mid f(\mathbf{x}) = 0\}$ and $S' \coloneqq \{\mathbf{x} \in TT_f \mid f(\mathbf{x}) = 1\}$, where f is the function of the M-inputs gate G which drives node N, with the related truth table TT_f. $X_0, X_1, ..., X_M$ are the inputs of the aforementioned gate, and they can be justified to the string of values $\mathbf{x} = [x_0, x_1, ..., x_M]$ (where clearly $x_i \in \{0, 1\}, \forall i$). The last one indicates that G is being added to the path, increasing the overall depth. Since the formula is recursive, the 1s will add up and at the end they will indicate the depth of the node that is being considered. As a side note, the lower the controllability value, the easier it is to control the node. The usage of the *min* operator in Equation 4.8 and Equation 4.9 is thus justified, since the overall controllability value is dictated by the x pattern which is the easiest to drive. Sequential controllability works in the same way, but it considers the influence of sequential nodes (FFs outputs) on each internal node of the circuit. This kind of analysis aims to find the minimum number of cycles needed to justify a specific node to 0/1, but no additional details will be provided because COTD uses only combinational metrics. The observability metrics are computed in the following way, including gate by gate starting from the POs:

$$CO(X_i) = \min_{\forall N_i \in \mathbf{N}} CO(N_i) + \min_{\forall \mathbf{x}=[x_0,...,x_{i-1},x_{i+1},...,x_M] \in S} \sum_{\substack{m=0 \\ m \neq i}}^{M} CC^{x_m}(X_m) + 1 \tag{4.10}$$

where in this case the gate G can have up to N outputs, and:

$$\begin{aligned} S \coloneqq \{\mathbf{x} = [x_0, ..., x_{i-1}, x_{i+1}, ..., x_M] \in TT_f \mid \\ |f([x_0, ..., x_{i-1}, 0, x_{i+1}, ..., x_M]) - f([x_0, ..., x_{i-1}, 1, x_{i+1}, ..., x_M])| = 1\} \end{aligned} \tag{4.11}$$

In this case, both the difficulty in observing the most observable output (among the N_i ones) and the difficulty in forcing a variation on the input X_i to be observed are considered. The +1 plays the same role it has before, even if gates are counted from the POs to the PIs.

3. step 2 is repeated multiple times until convergence is reached. Further insights on the algorithm complexity and the time needed for convergence are reported in [61].

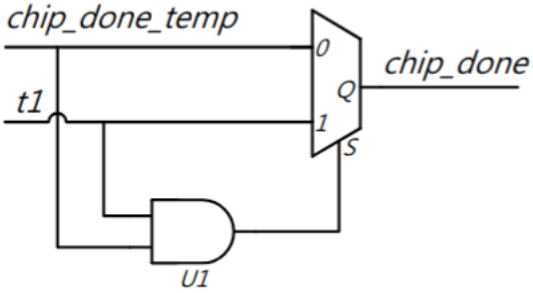

Figure 4.12: A connection implemented in deTest to lower CO [5].

COTD consists in an initial computation of the couple $< CC, CO > (N)$ for each node through the previous definitions, where:

$$CC(N) \coloneqq \sqrt{(CC^0(N))^2 + (CC^1(N))^2}$$

and a norm for $< CC, CO > (N)$ is defined as:

$$| < CC, CO > | \coloneqq \sqrt{CC^2(N) + CO^2(N)}$$

After metrics computation is done, a *k-means clustering*[5] analysis is performed to group all signals in the netlist in three clusters: the first one collecting genuine wires whose centroid will be near $< CC, CO >=< 0, 0 >$, the second one for high CC values and the third one for high CO values. Due to the unsupervised nature of this approach, there is no need to specify parameters such as thresholds. An additional point of strength is represented by the performances on trojans produced by deTrust, since all of them are spotted through this approach. As hinted before, the key problem when applying deTrust is that it doesn't affect testability metrics, and in some cases it actually makes them worse. Zhang *et al.* [5] proposed a new injection approach, dubbed *deTest*, which exploits "dummy conditions" to forcefully increase testability of critical nodes, both in terms of controllability and observability. The main idea to force a decrease in CC^X is the one to `OR` together the rare malicious trigger and the original value to create the payload, forcing a transition on the malicious trigger when the original value is 1 (this transition doesn't affect the final value, but it increases testability for the trigger). Decreasing CO is done by connecting the trigger with a very low CO signal (for example a PO) through a `MUX` as in figure, in this way it is possible to shorten the path between the internal trojan circuitry and the outputs considerably. The authors claim that this technique can be combined with deTrust, but further work is needed to assess full compatibility. For this reason deTest won't be covered in the toolchain, leaving its implementation as future work. As analyzed, deTest-based circuitry mainly consists in `OR` gates and `MUX`es, so it wouldn't be hard to implement. It will still be discussed how to increase testability through other solutions, mainly changing the structure of the trigger with respect to classic deTrust ones. For what concerns the complexity, that's where COTD excels. The overall complexity of COTD is $\mathcal{O}(n)$, because that's the complexity for both CC/CO computation and k-means clustering through Lloyd algorithm[6].

[5]An unsupervised machine learning algorithm which groups a set of N M-dimensional data (in COTD $M = 2$) in k clusters. At each step, the dimension and the position of the clusters are adjusted depending on the distance between data and the centroids of these clusters.

[6]https://en.wikipedia.org/wiki/K-means_clustering

Chapter 5

The toolchain

5.1 Introduction

This chapter will be devoted to detailing the structure of the toolchain, including thorough explanations for each element in the flow. Tools from Chapter 3 will be mentioned to highlight the role they play in the overall structure, underlining why they have been chosen over their competitors. A comprehensive discussion about strength and weaknesses of the proposed solution will be presented in parallel with the explanations, in addition with a sum-up at the end of the chapter.
The first section of this chapter will introduce all the steps that are part of the flow, without providing implementation details or evaluations.
The sections that follow will show every component in details, presenting the main scripts (and tools) and how each one of them could be adjusted or modified to implement additional functionalities. A major standpoint of the toolchain resides in the customizability, due to its core consisting in multiple tools that can be rearranged in different way depending on the use case.
As anticipated, the last section will present some of the features that emerge after the complete analysis.

5.2 General structure

Before actually starting the execution, it is necessary to have some benchmark programs available. In order to simulate random tests, it is possible to generate a new program through RISCV-DV and include it in the benchmarks to be simulated. As anticipated in Section 3.3, the first real step requires selecting the intermediate triggers among rare wires in the post-synthesis netlist. In order to do this, a first simulation over every benchmark is done and coverage reports for every signal are produced. By analyzing these reports, N signals are selected among the ones which show the lowest value/toggle coverage. Then, a script is used to instruct Verilator not to log the other signals (through specific comments) and the simulations are repeated to produce a reduced-size FST/VCD log for each benchmark. This step is necessary because the genetic algorithm will work only on these FST files, so having large files would make parsing last too long and, as a consequence, it would slow down the evaluation for every individual. With the FST files available, the genetic algorithm can be run to select M triggers among the N ones based on the rarest condition that has been generated during the simulation. The main flow is the following one:

1. sets $s_1 = \{t_{00}, t_{01}, ..., t_{0M}\}$ to $s_I = \{t_{I0}, t_{I1}, ..., t_{IM}\}$ are generated, each one associated to a different individual (I is the number of individuals).

2. every set will be parsed to check for the existence of dependencies between triggers. These dependencies would make the whole trigger string weaker, because two values could change in a related way (they could be considered to be equivalent). How this check is actually performed will be detailed in Section 5.5.2

3. (for every individual) every FST file is parsed to determine all the possible assignments for each trigger in each set during the execution, where a generic condition C_{ij} could be the one where every trigger evaluates to 1, so $\{t_{i0} = 1, t_{i1} = 1, ..., t_{iM} = 1\}$.

4. for each set/individual the rarest condition is selected, which is the one corresponding to the lowest number of occurrences. Additional metrics are used to evaluate how this condition is distributed in different benchmarks, as will be explained in Section 5.5.5. Based on these metrics, a fitness value will be produced for each individual.

5. genetic operators are applied to the individuals depending on their fitness values and on *hyperparameters* (as temperature, entropy etc.), producing a new generation of individuals that will go through the whole flow again.

After the flow is concluded, the best individual is selected and the circuit detecting the corresponding condition is added to the post-synthesis netlist. Additionally, the circuit with the trigger inserted can be tested to assess if it behaves as one would expect. The last step is the definition of a proper payload, which should be masked against VeriTrust as presented in Section 4.2.2. Again, this could be done in two different ways:

- P possible payloads could be selected from the netlist, performing topological checks to avoid creating *combinational loops*. After selecting these candidates, the payload circuitry is inserted and a simulation is performed for each of them. During these simulations, the outputs of the core are monitored to assess if the malicious values are observable. This step could easily be considered as the bottleneck of the entire flow, since P re-compilations and re-simulations are responsible for a huge time overhead.

- instead of resimulating the design for each payload, it could be possible to run a SAT solver to find a pattern which is able to propagate the malicious value to the output. If the solver is properly programmed and driven (and an initial state is set), then this flow could be faster than the previous one. However, there is no way to determine if the SAT-generated pattern is actually produced during simulation. Additionally, this pattern could even be a non-functional one, because SAT solvers drive the inputs arbitrarily. It could be possible to introduce ISA-related constraints through a sby script[1], but it would require further customization depending on the particular ISA and pinout.

When both trigger and payload insertions are done, testing and stealthiness evaluation can be performed. Since the majority of the algorithms discussed in Chapter 4 are not directly available, simplified checks have been run to measure trojan metrics.

5.3 Random program generation

Generating a random program through RISCV-DV requires adjusting some parameters, which may be related to both the structure of the program or the instructions that are produced. In the following analysis, only the *pygen/experimental* version from RISCV-DV

[1]Refer to Section 3.10 for details.

Github repository will be considered, since it is the only one that can be run without proprietary tools. The first parameters that can be edited are the ones determining the overall length and presence/absence of subprograms. The program length can be changed by modifying the parameter passed to `gen_program()` when invoking the program generation in `riscv_asm_program_gen.py`:

```
1 instance = riscv_asm_program_gen()
2 instance.add_directed_instr_stream("riscv_load_store_rand_instr_stream",
  ↪  20)
3 instance.gen_program(250)
4 print("Test generated: ./out/test.S")
```

Changing the number of subprograms can be done by editing `num_of_sub_program` definition in `gen_program()`, while modifying `self.sub_program[i].instr_cnt` allows the user to specify program `i` length. Besides, it is possible to insert a generic instruction stream `last_program` in any point of the main program through the following commands:

```
1 self.generate_directed_instr_stream( last_program.directed_instr,
  ↪  last_program.name, last_program.instr_cnt, min_insert_cnt=0)
2 last_program.gen_instr(1,csr_inst=1, not_branch=1, not_system=1,
  ↪  valid_div_and_csr = 1)
3 last_program.generate_instr_stream()
4 self.instr_stream.append(last_program.instr_string_list)
```

An example of this flow is present as a comment in `gen_test_done()`. The parameters at line 2 are switches that specify characteristics of the instruction stream, such as if `CSR`/`SYSTEM` instructions are admitted or if `DIV` instructions can throw exceptions. All these properties can be enforced through constraint definitions to be placed at the desired abstraction level (instruction *stream*, instruction *sequence* etc.), which in our case will be when generating each instruction singularly (`riscv_instr_base.py`). This means that every instruction will be considered as an independent CSP[2] accounting for all the constraints, and it will be solved through the `self.problem.getSolution()`. The ISA-related characteristics, such as instructions and registers support, can be specified in `utils.py`.
The program generated through this flow must then be linked with a *crt0* for the core where it will be run. For the AFTAB, the `crt0.s` available as part of the Github toolchain has been used, which configures the core for both `MACHINE` and `MACHINE/USER` mode. Both linker script and `crt0.s` are available in the *aftab/sw/ref* folder of the toolchain. Due to the lack of support for `SUPERVISOR` level, the AFTAB is currently unable to boot FreeRTOS.
This free version of RISCV-DV shows many limitations with respect to the original one, such as the lack of support for loops (and backward jumping sanitization in general). It will be possible to switch to the real tool only when UVM support will be included in FOSS simulation environments. For now, *pygen* is still enough to produce complete random programs with subroutines, branches, exception and interrupt handling.
To sum up all the modifications that are needed to adapt the flow to a new core:

- in `riscv_asm_program_gen.py`: change the number of instructions of main program, add new subprograms with their own instruction counts and, if needed, create new instruction streams and append them to the main stream.

[2] *Constraint Satisfaction Problem*, refer to Section 3.7.

- in `riscv_instr_base.py`: define new constraints through the flow presented in Section 3.7 and add them to the generation problem. As shown previously, these constraints can be enforced by setting some switches in `gen_instr()` calls. Since different abstraction levels are traversed when moving from `riscv_asm_program_gen.py` to `riscv_instr_base.py`, it may be necessary to adjust all the calls in the middle by adding some extra parameters.
- in `utils.py`: add new supported instructions or registers.

5.4 Simulation and coverage analysis

From now on, the SRV32 toolchain will be taken as a reference. Repurposing this flow for a different core requires changing the simulation environment, so all the steps based on simulation are implementation dependent. Verilator is a widely used tool in the open-source domain, so this flow can still be applied to many cores with few modifications. For this reason, the following explanation will insist on how the tools are involved rather than on how the setup is done in this particular case. The first step when setting up the toolchain for coverage values collection consists in adding `--coverage-toggle --coverage-underscore` to the `verilator` call. This can be done by appending these switches to the `BFLAGS` variable in *sim/Makefile*, which is where the call is executed. After doing that, the simulations can be run and one *dat* file can be collected for each benchmark. The `select_and_insert_triggers.py` script parses all the files included in `filenames` list, looking for the `TRIGGER_NUM` signals with the lowest coverage values. Since it is better to avoid wires which have not been sensitized during benchmarks executions, a low threshold for the coverage value has been set. After locating all these signals, the script produces the RTL circuitry to drive the corresponding intermediate triggers. There can be two types of triggers:

- *level triggers*: which are simply the signal they refer to, they are activated when the signal is set.
- *edge triggers*: the ones that are activated when there is a rising/falling edge over the corresponding signal. Due to Verilator limited support for asynchronous logic, these signals have been implemented through 2 FFs: the first one which stores the old value of the signal (`signal_old`) and the second one which is set for 1 cycle when a transition over this signal happens. In order to implement the desired functionality, the input of the second FF should be `signal_old ^ signal`.

As will be shown later, both kinds of triggers have their advantages and disadvantages. The obvious advantage of the first category relies in the limited area footprint, but it allows to create less complex conditions. However, since additional sequentiality will be introduced later in the toolchain, this kind of triggers can still be solid enough for particular applications. After the new circuit is inserted, the trigger signals should be marked as the only ones to be logged. In order to do that, a `/* verilator tracing_on */` statement is inserted on top of the declarations of these signals and a `/* verilator tracing_off */` afterward. With the new signal scope defined, a new simulation can be run for each benchmark and the corresponding reduced-size FST can be collected. Verilator offers the possibility to log signal traces as VCD files too, but these are less compressed with respect to FST.

5.5 Byron

As anticipated, the evolutionary algorithm implemented through Byron works exclusively on FST files, so there is no need for additional simulations. Such a flow presents mainly two advantages: it doesn't depend on simulation environments and it doesn't introduce time overheads due to expensive re-compilation and re-simulation. For these reasons, this part of the toolchain scales well to different architectures. Byron flow has already been introduced in Section 3.8, where it has been explained how to define the structure of the individual and which evaluators are available. In the following, the focus will shift on the actual evaluator to be used, highlighting both multi-core support and the checks performed by the evaluator. Let us recall the command which should be invoked:

```
evaluator = byron.evaluator.ParallelScriptEvaluator(
↪  './evaluate-triggers_single.sh', "individual.s",
↪  other_required_files = additional_files, timeout = None,
↪  default_result = DEFAULT_RESULT)
```

The `ParallelScriptEvaluator` is able to parallelize evaluation on different individual from the same gneration, replicating the environment that is needed for each individual in different folders inside the */tmp* drive. The files that will be allocated in each folders are the ones in the `additional_files` list. This method is very effective when different evaluations take approximately the same time to complete, otherwise there would be the need to wait for the slowest individual before concluding the generation. This is due to the fact that new individuals are generated after the previous generation has been completely evaluated. As will be shown later, the need to duplicate the environment is one of the reasons why it is unfeasible to implement a Byron toolchain for the payload. The real core of the evaluator is the shell script, `./evaluate-triggers_single.sh`, which is shown in Listing 3. Due to the script being fairly complex in each of its parts, the following paragraphs will be dedicated to discussing all of them separately.

```
1  rm tmp.txt
2  # check if triggers are independent
3  python3 trigger_independence_check.py $1 adjacency_matrix.txt
   ↪  xor_perms.txt > tmp.txt
4  INDEP=$(tail -n 1 tmp.txt)
5  stringarray=($INDEP)
6  if [[ ${stringarray[0]} = "1.0" ]]; then
7      # check the vcd files to determine the values assumed by the
   ↪  selected signals during benchmark executions
8      python3 dumpfst_input_signals.py wave_dhrystone.fst $1
   ↪  signals_map.txt > log1.txt
9      python3 dumpfst_input_signals.py wave_fft.fst $1 signals_map.txt >
   ↪  log2.txt
10     python3 dumpfst_input_signals.py wave_pthread.fst $1 signals_map.txt
   ↪  > log3.txt
11     python3 dumpfst_input_signals.py wave_qsort.fst $1 signals_map.txt >
   ↪  log4.txt
12     python3 dumpfst_input_signals.py wave_queue.fst $1 signals_map.txt >
   ↪  log5.txt
13     # perform metric computation
14     python3 metric_computation_no_window.py $1 > $LOG.txt
```

```
15      rm log1.txt
16      rm log2.txt
17      rm log3.txt
18      rm log4.txt
19      rm log5.txt
20      # python3 metric_computation_new.py $file >> log.txt
21      NUM=$(tail -n 1 $LOG.txt)
22      echo $NUM
23      exit 0
24  else
25      echo $INDEP
26  fi
```

Listing 3: The shell evaluator.

5.5.1 Some notes on the fitness function

Before delving into implementation details, a brief introduction on the characteristics of the fitness function is necessary. The function that have been chosen for this particular application is a vectorial one $\mathbf{f}$, consisting in multiple components f_i with different meanings. When two individuals are compared during the evaluation, an iterative check is performed comparing the f_{00} and f_{01} values: if $f_{00} = f_{01}$ then the second components f_{10} and f_{11} are compared, otherwise the individual with the highest f_0 value is considered to be the fittest. Let us show some of the properties of good fitness functions, contextualizing them in the overall evolutionary flow:

- a good fitness function introduces a low time overhead, since many evaluations have to be performed during the evolutionary flow. Due to the need to parse every FST file to compute the number of activations, each fitness value computation in the current flow takes around 9 minutes to complete. In general, if the time needed is assumed to be T, then the N-core parallelization introduced by `ParallelScriptEvaluator` significantly amortizes this value, reducing the time per individual to T/N at steady state. It is important to recall, however, that if the number of individuals to be processed in a generation is not a multiple of N, then last execution will waste resources leaving some cores idle.

- it should include every parameter which contributes to define the quality of the solution. As simple as this may seem, in the analysis that follows it will be impossible to estimate the effect of every trojan on the final layout. This stems from the fact that the toolchain works on gate-level netlists, so inferring metrics as the interconnection area overhead requires complex computations (and the results would not be precise). For this reason, including as many parameters as possible introduces a conflict between completeness and efficiency.

- when exploring the design space, local optima should be avoided. Section 3.8 presents a way to do so through *temperature*, but other parameters such as *entropy* are a good way to preserve diversity and to favor wide space exploration.

- in a vectorial function $\mathbf{f}$, components should be properly sorted according to relative importance. According to what has been shown, when evaluating $\mathbf{f}$ the f_{i+1} component is considered only if all components up to f_i are equal. As a consequence, a small increase in component i matter more than a big increase in component $i+1$.

For this reason, building the fitness function and choosing the correct order for the components should be done with care. The best scenario would be the one in which a clear importance scale is present, but this is not always the case. Due to that, fitness function continuity will play a role too, as will be explained later.

- when considering a scalar fitness function f, the function should be as continuous as possible. In this way, arbitrarily small variations in the fitness value can be used to drive the evolutionary process. When it comes to vectorial fitness functions, the degree of continuity may depend on the component being considered. In particular, if there is not a very clear importance scale then a small increase in component i should not be valued more than a big increase in component $i+1$. Besides, sometimes accepting a solution with a huge improvement over $i+1$ could be beneficial on the long term evolution. In order to do that, i granularity/continuity is set to be lower than $i+1$, making i insensitive to small variations and allowing the evaluator to consider $i+1$ value.

5.5.2 Independence check

Since having intermediate triggers which show dependencies can weaken the final condition, a script `trigger_independence_check.py` is called to perform some topological checks. In particular, given a trigger t_0, the script checks if there is another t_i which resides in combinational fanin cone of t_0. As will be shown later, the first fitness component f_0 is driven through this check. The bulk of this procedure consists in using HAL to build a netlist graph and compute the fanin cone of each node, collecting the results in a matrix A written in `adjacency_matrix.txt`. Each element of the matrix a_{ij} will be 1 when signals i and j are dependent (one of them is in the fanin cone of the other), 0 otherwise. The strength of this method relies in the need to use HAL only once, since each individual will simply need to check the `adjacency_matrix.txt` to assess if two signals are dependent. Listing 4 shows the HAL script used to compute the fanin cone, which serves as an introduction to the typical flow that will be used for payload injection. In short, the function receives the id of a net in the circuit and returns its combinational fanin cone. An additional array `net_marker` is used to mark nets that have already been visited, since there is no need to add their fanin gates again (*pruning*). The recursive algorithm consists in these steps:

1. the net is marked as visited by setting the corresponding `net_marker` entry (line 3).
2. assuming the net has a single driver, the endpoint connecting the net to the driver is retrieved (line 8) together with the driver's name (line 9). In case the net is a PI (line 7), a `GND` net or a PPO[3] (line 13), then the input net is returned.
3. the recursion is performed over each input net of the driver, provided that it has not been explored before (line 19).

After computing the fanin for net i through the previous function, the script iterates on the other nets to check if they are included in the cone and populates A accordingly.

```
1 def recursive_fanin(netlist, net_id, net_marker):
2     # mark the path in the fanin of this net as already explored
3     net_marker[net_id] = 1
```

[3]When describing a sequential circuit as an FSM, FF outputs can be named *Pseudo Primary Outputs* (*PPOs*).

```
4
5       fanin_ret = [netlist.get_net_by_id(net_id).get_name()]
6       # retrieve the gate whose output pin is mapped to the signal being
        ↪  considered
7       if not netlist.get_net_by_id(net_id).is_global_input_net() and
        ↪  len(netlist.get_net_by_id(net_id).get_sources()) != 0:
8           e = netlist.get_net_by_id(net_id).get_sources()[0]
9           gate_name = e.get_gate().get_name()
10      else:
11          return fanin_ret
12      # if the gate if a FF you need to stop the recursion, since the
        ↪  fanin considers only the combinational part
13      if (str(list(e.get_gate().get_boolean_functions().values())[0]) ==
        ↪  "IQ" or "gnd" in gate_name):
14          return fanin_ret
15
16      for net in e.get_gate().get_fan_in_nets():
17          # recur on each input to the gate if the fanin of the
            ↪  corresponding net hasn't already been explored
18          if not net_marker[net.get_id()]:
19              fanin_ret += [net.get_name()] + recursive_fanin(netlist,
                ↪  net.get_id(), net_marker)
20
21      return fanin_ret
```

Listing 4: The recursive fanin cone computation.

The decision not to extend the check to previous sequential stages allows for a fast execution, which is crucial due to the duration of the commands that follow in Listing 3. Besides, a dependency which crosses sequential stages would manifest itself in different clock cycles, potentially not producing any effect if the check over trigger values is a combinational one. This will not be the case in the following analysis, where a complete check spanning multiple stages would make the trigger condition stronger. Code from `trigger_independence_check.py` simply consists in checks over the A matrix, so it will not be shown. Another critical point of this implementation resides in the difficulty to find M purely independent triggers. For this reason, an hard threshold `SELECTED_PERMS` has been set as the minimum acceptable number of independent triggers (n_{IND}). For each trigger i, $n_{DEP,i}$ is defined as the number of triggers that i depends on (the ones located in its fanin cone). The actual f_0 value is driven as follows:

$$f_0 = \begin{cases} 1 & \text{if } n_{IND} \geq \texttt{SELECTED\_PERMS} \\ \frac{1}{2}\frac{\texttt{NUM\_TRIGGER} - \min_i n_{DEP,i}}{\texttt{NUM\_TRIGGER}} + \frac{1}{2}\frac{\texttt{SELECTED\_PERMS} - n_{IND}}{\texttt{SELECTED\_PERMS}} & \text{if } n_{IND} < \texttt{SELECTED\_PERMS} \end{cases}. \tag{5.1}$$

As evident from this definition, f_0 is insensitive to every variation in n_{IND} when this value is higher than the threshold. In this way, an increase in value on the higher-indexes components is considered to be more valuable with respect to the addition of another independent signal. When $n_{IND} < \texttt{SELECTED\_PERMS}$, f_0 is driven by a function of both $n_{DEP,i}$ and n_{IND} instead. Both contributions have the same weight, thanks to the $\frac{1}{2}$ which multiply both terms. The main reason why $\min_i n_{DEP,i}$ has been used resides in the need to distinguish between individuals with the same number of indipendent triggers. In particular, the solution which shows a lower $\min_i n_{DEP,i}$ presents the trigger with the

lower number of dependencies from the other ones, meaning that in a few generations this trigger could become independent (thus increasing n_{IND}). This strategy allows for a better driving of the evolution process, because it uses an additional parameter to reward solutions that could improve the result on the medium-long term. A further vectorization of f_0 would have been possible to separate $n_{DEP,i}$ and n_{IND}, but it has been avoided for simplicity. Due to the normalizations, the final value will be in the interval $[0, 1[$, making the piecewise function continuous in $P = (\texttt{SELECTED\_PERMS}, 1)$.

5.5.3 Dumping trigger values from FST files

A preliminary step for evaluating a solution is to log trigger values at each clock cycles, reading them from the FST files. Such files are written according to a particular protocol, which allows for a very high level of compression (even higher than VCDs). In the current thesis work, a tool named *pylibfst* [54] has been used to parse FST files. Due to it being distributed as a Python library, this solution is scalable and doesn't require non-standard environments or proprietary tools. The meaningful parts from `dumpfst_input_signals.py`, the script called for this purpose (from Listing 3), are reported in Listing 5.
A few considerations before analyzing the flow:

- an array `signals_map` is used to map a signal index to the corresponding name. This association is based on the list of signals produced by `select_and_insert_triggers.py` (Section 5.4).
- `individual_lines` is the list containing every trigger index for the current individual. This is the format of the individual generated by Byron, which can be considered a list of M integers ranging from 0 to $N - 1$ (referring to Section 3.8.1).
- signals can be addressed by *name* or by *handle*, where the latter is a unique index identifying the signal inside the FST. Since input and output signals are logged together with the selected triggers, handles can assume values higher than N. For this reason, triggers indexing has been done through `signals_map` instead.

The first step when using pylibfst is to define the signals to be logged, populating a `Signals` structure. In order to do that, first the complete list of `Signal` objects should be retrieved from the FST (line 17). Then, each trigger signal object is retrieved by name from this list (line 25) and inserted in two dictionaries `new_signals_by_names` and `new_signals_by_handles` that will be used to populate a `new_signals` list. With this new list available, `dump_signals` is called. This last function is the one where parsing APIs are called, allowing the user to retrieve all timestamps[4] (line 4) and to reconstruct the value of each trigger signal at that particular timestamp (line 12). The aforementioned flow has been adapted from the one in the pylibfst Github repository, and it could be generalized to perform FST analysis.

```
1 def dump_signals(fst, signals):
2     # get timestamps of all signal changes
3     pylibfst.lib.fstReaderSetFacProcessMaskAll(fst)
4     timestamps = pylibfst.lib.fstReaderGetTimestamps(fst)
5 
6     buf = pylibfst.ffi.new("char[256]")
```

[4]Timestamps can be defined as the simulation instants in which at least a signal changes. They will usually correspond to clock edges.

```
    for ts in range(timestamps.nvals):
        time = timestamps.val[ts]
        for signal in signals.by_name.values():
            val = pylibfst.helpers.string(
                pylibfst.lib.fstReaderGetValueFromHandleAtTime(
                    fst, time, signal.handle, buf)
            )
    pylibfst.lib.fstReaderFreeTimestamps(timestamps)

fst = pylibfst.lib.fstReaderOpen(filename.encode("UTF-8"))
(scopes, signals) = pylibfst.get_scopes_signals2(fst)
new_signals_by_names = dict()
new_signals_by_handles = dict()
Signals = namedtuple("Signals", "by_name by_handle")

for line in individual_lines[1:]:
    elem = signals_map[int(line)]
    name = elem.split()[0]
    signal = signals.by_name.get(name)
    new_signals_by_names[name] = signal
    new_signals_by_handles[signal.handle] = signal

new_signals = Signals(new_signals_by_names, new_signals_by_handles)
dump_signals(fst, new_signals)
```

Listing 5: *pylibfst* code.

5.5.4 Trigger tree structure

In Section 4.3.1, the generic FANCI-invulnerable AND-based trigger tree structure has been introduced. The main idea behind this strategy consists in dividing the trigger string in T groups, where each one of them will be used to build a sequential tree. In this way, trigger set cardinality is reduced from M to T (obviously $T \ll M$). In order to guarantee invulnerability against VeriTrust, imposing $T \neq 1$ prevents the creation of a single wire which is not sensitized at all during testing.
If the following notation is used ($\wedge$ is the logic `AND`):

$$\bigwedge_{i=0}^{N} t_i \coloneqq t_0 \wedge t_1 \wedge t_2 \wedge ... \wedge t_N \tag{5.2}$$

and the M triggers are grouped in T groups, then the generic tree will be defined as follows:

$$t'_i = \bigwedge_{j=0}^{\frac{M}{T}} t_{j+i\frac{M}{T}} \tag{5.3}$$

where t'_i are the trees outputs and t_i the trees inputs (following the same nomenclature in Figure 4.8). M mod $T = 0$ will be assumed from now on, in order to allow for a perfect division in T groups.
The genetic algorithm should optimize over both the selection of the original M triggers

and the definition of the T groups. The problem of selecting all ways in which T groups of g_i elements each can be formed is formalized through the *multinomial coefficient*:

$$\binom{M}{g_0, g_1, g_2, ..., g_{T-1}} = \frac{M!}{\prod_{i=0}^{T-1} g_i!} \tag{5.4}$$

where clearly $g_0 = g_1 = g_2 = ... = g_{T-1} = \frac{M}{T}$. A couple typically used in the toolchain is $M = 24$,$T = 3$. For these values Equation 5.4 evaluates as:

$$\binom{24}{8, 8, 8} = \frac{24!}{8! \cdot 8! \cdot 8!} \approx 9.5 \cdot 10^9$$

which is far too high to allow for an exhaustive search over the space of possibilities. In order to reduce the execution time, only a fixed subset of this number has been selected and each individual is tested against this subset. The size of this subset is given by `NUM_PERMS`, which has been kept very low during the simulation. Executing the toolchain on more powerful machines would make it possible to increase this value, making it possible to explore a larger portion in the space of parameters.
This technique still shows a vulnerable side when inter-sequential stages analysis is performed, either through FANCIX or ANGEL, due to their ability to cross sequential stages by ignoring FFs. FASTrust represents an adversary too, because the FF-CDFG derived from this kind of structure is isomorphic to Figure 4.9 and, as a consequence, presents an high number of low out-degree nodes. In order to address these vulnerabilities, a new way to build this tree will be shown. The key point in building the new tree consists in replacing all the `AND` gates with `XOR` ones, because they lower the CV significantly. Let us define the following notation:

$$\bigoplus_{i=0}^{N} t_i \coloneqq \ t_0 \oplus t_1 \oplus t_2 \oplus ... \oplus t_N$$

where $\oplus$ is the logical `XOR` operator. Assuming the tree structure are kept as before, where every tree is independent from the others, then the generic t'_i in Equation 5.3 can be rewritten as:

$$t'_i = \bigoplus_{j=0}^{\frac{M}{T}} t_{j+i\frac{M}{T}} \tag{5.5}$$

If a CV evaluation is done for each trigger input t_j with respect to output $t'i$, then:

$$CV(t_j, t'_i) = 1, \qquad \forall j \in \{0, 1, ..., i \cdot \frac{M}{T}\} \tag{5.6}$$

This result shows that using `XOR` gates would make the trigger trees invulnerable to most static techniques based on CVs, such as FANCIX and ANGEL. Eluding FASTrust feature 4 can be done too, but it affects robustness against feature 2 and potentially requires duplicating the whole circuit. The procedure to increase the out-degree of each t_j consists in removing all the FFs[5] in the trees and combining t'_i and t'_{i+1} as follows, redefining the outputs of the trees as t''_i:

$$t''_i = t'_i \oplus t'_{i+1}, \quad t''_{i+1} = t'_{i+1} \oplus t'_i \tag{5.7}$$

[5]FFs removal is possible because of condition 5.6, since FFs where originally needed to increase the CV.

In this way, the out-degree of each input belonging to trees t'_i and t'_{i+1} has been doubled. However $t''_i = t''_{i+1}$, so the complexity of the final trigger condition has been halved. If $T = 2$ then it is necessary to add new triggers, because a single condition $t''_0 = t''_1$ would be vulnerable to VeriTrust (if never sensitized during testing). If $T > 2$ then the evaluation depends on how strong the final conditions are, making the addition of new triggers necessary if the the new signals are sensitized too much. Adding new triggers, and new trees, increases the complexity of the payload part as well. As will be shown, this effect could be even more critical, because the payload circuitry lacks the regularity of the trigger tree. As hinted, this technique doubles the out-degree of each t_i, but it also makes the in-degree of t''_i double the one of the corresponding t'_i. For this reason, the new circuit could be vulnerable against FASTrust feature 2. In the general case, the area complexity of this structure is $\mathcal{O}(M)$[6], but FFs should be counted as well if inserted.
The delicate point when applying this technique is related to the `XOR` truth table: for a N input `XOR`, the TT shows exactly 2^{N-1} entries at 1 and 2^{N-1} entries at 0. In particular, changing a single input produces a change in the output value (as suggested by the CV). This is the reason why this technique is strong against static analysis, but at the same time it makes it impossible to find a rare circuit condition. The rarity in this case must be built depending on how signals are activated during benchmarks execution, taking care in selecting the M signals in a way such that the outputs of the trees evaluate at 1 rarely. This technique requires benchmarks that are as complete as possible, because circuit behavior is unpredictable when executing external code. Obviously it is impossible to ensure complete coverage through benchmarks, but it is possible to produce a huge variety of different programs and to force activation[7] only on a very limited subset of them. Additional robustness must be ensured against data-sensitive triggers, since a certain application will surely be run over multiple data during the core life cycle. A possible approach to do that could be the one to simulate the same program with multiple data, forcing activation only on a specific subset of the inputs.
The previous flow could be implemented by defining a matrix B of benchmarks, where B_{ij} is the generic benchmark i executed with the set of input data j. The key objective to ensure rarity is to reward individuals whose trigger activates rarely enough and on a single benchmark B_{ij}. Since ensuring zero activations during functional testing is a priority, programs similar to test suites could be included among benchmarks. Given specific constraints (in terms of B matrix, T and M values), there is no guarantee that the algorithm will be able to find conditions that are rare enough to satisfy them. However, an application of this flow over 5 very different benchmarks produced promising results.
Including `AND` gates in the tree could help in reducing the size of the sub-TT which evaluates at 1. If the last gate in each tree (the one which drives t'_i) is replaced with an `AND`, then Equation 5.5 becomes:

$$t'_i = \bigoplus_{j=0}^{\frac{M}{2T}} t_{j+i\frac{M}{T}} \wedge \bigoplus_{j=0}^{\frac{M}{2T}} t_{j+\frac{M}{2T}+i\frac{M}{T}} \tag{5.8}$$

It is easy to verify that the number of 1 entries has been exactly halved with respect to Equation 5.5, changing each $CV(t_j, t'_i)$ to $\frac{1}{2}$. Since this value is still very high in comparison with CVs from the genuine circuit, it is possible to replace another layer of

[6]The first level of the tree requires $\frac{M}{2}$ gates, then $\frac{M}{4}$ and so on until 1 is reached. The sum of these terms approaches M.

[7]"forcing" indicates that the individual showing this particular behavior will be rewarded with an higher fitness value.

XOR gates obtaining:

$$t'_i = \bigwedge_{k=0}^{4} \bigoplus_{j=0}^{\frac{M}{4T}} t_{j+k\frac{M}{4T}+i\frac{M}{T}} \tag{5.9}$$

Again decreasing the control value for each input by half, obtaining $\frac{1}{4}$. In theory, it is possible to apply this flow until a CV compatible with the ones from genuine circuitry is reached. The generic tree where L layers have been replaced with AND gates will look like this:

$$t'_i = \bigwedge_{k=0}^{2^L} \bigoplus_{j=0}^{\frac{M}{2^L T}} t_{j+k\frac{M}{2^L T}+i\frac{M}{T}} \tag{5.10}$$

Another interesting case is the one in which the XOR replacement is done over the whole first layer, the one driven directly by the t_j triggers. In this case the expression becomes:

$$t'_i = \bigoplus_{j=0}^{\frac{M}{2T}} (t_{2j+i\frac{M}{T}} \wedge t_{2j+1+i\frac{M}{T}}) \tag{5.11}$$

and if the same replacement is applied for L layers moving upward in the tree then:

$$t'_i = \bigoplus_{k=0}^{\frac{M}{2^L T}} \bigwedge_{j=0}^{2^L} t_{2^L k+j+i\frac{M}{T}} \tag{5.12}$$

This second technique doesn't decrease the CV as much as the first one, but it reduces area occupation considerably[8]. Both techniques can be used to manipulate the pure XOR-based logic function to exclude some minterms, replacing the appropriate number of XOR gates depending on CV constraints and on the number of minterms to exclude. Including this kind of manipulation as part of the design space would make the overall worst-case complexity $\mathcal{O}(N^M \cdot M! \cdot 2^M)$, where:

- computing all sets of M triggers among N has a complexity which is $\Theta(N^K)$.
- grouping all the M triggers in T groups has a complexity of $\mathcal{O}(M!)$ when $T \approx M$. In the case of the thesis, however, $T \ll M$ (usually $T = 2$ or $T = 3$) so the complexity can be considered $\mathcal{O}(M^T)$ as before.
- as mentioned previously, the spatial complexity of the trees is $\mathcal{O}(M)$. Choosing which gates have to be replaced with AND may require evaluating every gate, so the complexity for this part will be $\mathcal{O}(2^M)$.

The evolutionary algorithm explores part of this design space looking for the optimum, using the fitness function as an heuristic. The next step in improving this flow could be the one to include both the trees groupings and the AND/XOR mapping in each individual, adding two Byron macros to the individual definition (referring to the flow in Section 3.8). The problem would become a tradeoff between:

- keeping the control value high enough, since adding AND gates may decrease it considerably (depending on the position of the gate in the tree).

[8]Assuming an XOR gate is bigger than an AND gate (it occupies more area), which is definitely true for CMOS tecnology.

- excluding 1-minterms which are activated too often. In general, the more the minterms to be excluded, the more `AND` gates are needed to address them specifically.

Due to the existing tree structure acting as a constraint, it may not be possible to zero out some specific minterms (given the M triggers and the T groups). For this reason another degree of complexity could be added, which consists in adding some gates to any tree branch. This procedure can be shown by considering the following logic function:

$$f = a \oplus b \oplus c \oplus d$$

Let us suppose that during execution the 1-minterms $(0, 0, 0, 1)$ and $(0, 0, 1, 0)$ are sensitized too much, so eliminating them would make the f evaluate to 1 more rarely. One possible way to do so consists in changing the function as follows:

$$f = a \oplus b \wedge \overline{c \oplus d}$$

Apparently this introduces an additional `XNOR` gate besides `AND` and `XOR`, but this can be circumvented by rewriting the expression as:

$$f = a \oplus b \wedge c \oplus d \oplus 1$$

since $\overline{c \oplus d} = c \oplus d \oplus 1$. Thus, adding an `XOR` gate with an input wired to 1 on the (c, d) branch can change the function. It is possible to demonstrate that this can be done in an arbitrary way, eliminating whichever minterm from the final function. The root of this behavior resides in the functional completeness[9] of the $\{\texttt{AND}, \texttt{XOR}, 1\}$ set, which follows from:

$$\overline{a} = a \oplus 1, \quad \overline{a \wedge b} = (a \wedge b) \oplus 1$$

and from $\{\texttt{NAND}\}$ set being functionally complete.

5.5.5 The actual trigger tree

Three versions of the trigger tree structure have been implemented: a purely `AND`-based circuit, another one including only `XOR` gates and the last one which mixes up different trees to create a more complex condition. The script `metric_computation.py` which is called from Listing 3 assumes that the trigger circuit to be inserted is coherent with the second case, so it splits the trigger string in T trees and computes the `XOR` tree output for each one of them. There are two additional scripts, `metric_computation_and.py` and `metric_computation_sequential_xor.py`, which implement the same flow for the first and the third solution respectively. It is important to recall that every computation is done over the FST files produced during simulations, so the netlist is not manipulated in this step. The actual injection of the trigger circuit will be performed in a separate script, `inject_trees_window.py`, which can be customized to perform injection both for `OPERATOR` "∧" (Verilog `XOR`) and "&" (Verilog `AND`). The remaining parts of the toolchain are independent from the kind of trigger tree that has been used, making them scalable. Before proceding with the explanation, it is necessary to introduce another element of the trigger structure. In order to create a sequential condition, a k-bits counter has been added for each rare signal s_i identified during the flow in Section 5.4. Each counter c_i will be set when a condition happens on the corresponding s_i, which could be s_i being equal to 1 (in case of a level trigger) or s_i showing a falling/rising edge (in case of an edge trigger). This counter remains on for 2^k cycles, and while it is on the actual trigger t_i

[9] A set of gates is said to be *functionally complete* when it is possible to write whichever logic function by combining an arbitrary numbers of gates from the set.

is set. In this way, the t_i becomes a sequential trigger which is forcefully kept high for 2^k cycles after the triggering condition happened. This interval of time will be called *on-window*, and it may allow for the creation of stronger trigger conditions (expecially in the `AND`-based tree). In case a new triggering condition happens over s_i during the on-window, then the timer is reset and the window starts again. Obviously this new structure introduces additional area overhead, which is in the order of $M \cdot K$ FFs.
The actual code written in `metric_computation.py` will not be reported in this document, but its flow is presented here:

1. first a subset of all possible groupings of T elements among M is selected. This `NUM_PERMS`-sized set will define all possible assignments for trigger trees that will be analyzed. Each one of these groupings is in the following form (assuming $T = 3$):

$$(a_1, a_2, ..., a_{\frac{M}{3}}) - (a_{\frac{M}{3}+1}, ..., a_{\frac{2M}{3}}) - (a_{\frac{2M}{3}+1}, a_2, ..., a_M)$$

   where the generic a_i is a trigger among the M ones.

2. the whole file is scanned to compute t'_i values at each clock cycles, computing how many times each string $(t'_0, t'_1, ..., t'_T) = (k_0, k_1, ..., k_T)$ with $k_0, k_1, ..., k_T \in \{0, 1\}$ appears during simulation (for each grouping). These strings will be named $S_0, S_1, ..., S_{2^T-1}$, and only the rarest one will be considered to build the final trigger condition over the t'_i values. The results are collected in a dictionary named `perm_vals`, which associates each grouping in step 1 with an array of 2^T values representing the occurrences of each string S_i. While parsing the file, it is important to consider the existence of on-windows over which the value of a trigger t_i is kept high (even if the corresponding s_i is no more in the triggering state).

3. when the previous flow is applied over each FST file, the results are scanned to assess in which programs the rarest S_i string is produced at least once. This check is performed to enforce *variety*, in case the objective is the favor trigger activation on multiple benchmarks, or *exclusivity*, for example when trying not to activate the trigger on the benchmark which emulates functional testing. In case the objective is to produce the same S_i in 2 programs but it is produced only in 1 of them, then it is possible to compute other metrics to drive the evolutionary process. The one that has been used in the toolchain is the *Hamming distance* (HD)[10], which is computed between the best solution S_i and the best solutions in the other programs. If these solutions show a low HD with respect to S_i, then changing a few t_i triggers (through genetic operators) could be enough to obtain a new common solution S_j.

4. the fitness function for the individual is computed, relying on the parameters computed before. The whole fitness function is an array consisting in 3 elements, whose first element has already been determined through the independence check (Section 5.5.2). The second component is driven as follows:

$$f_1 = k_1 \cdot \left(-\frac{1}{2} \cdot |\texttt{best\_sol\_n\_programs} - k_2 \cdot \texttt{NUM\_P}| + 1 \right) + (1 - k_1) \cdot \texttt{drive\_intersection} \tag{5.13}$$

[10] Given $a = (a_0, a_1, ..., a_N)$ and $b = (b_0, b_1, ..., b_N)$ the *Hamming distance* between a and b is defined as $HD(a, b) \coloneqq \sum_{i=0}^{N} |a_i - b_i|$, which counts how many corresponding elements differ.

where drive_intersection (abbreviated as d_i) is defined piecewise:

$$\texttt{d\_i} = \begin{cases} 1 & \text{if } \texttt{best\_sol\_n\_programs} = k_2 \cdot \texttt{NUM\_P} \\ \frac{\texttt{best\_sol\_n\_programs}}{\texttt{NUM\_P}} & \text{if } \texttt{best\_sol\_n\_programs} < k_2 \cdot \texttt{NUM\_P} \\ \frac{\texttt{NUM\_P} - \texttt{best\_sol\_n\_programs}}{\texttt{NUM\_P}} & \text{otherwise} \end{cases} . \tag{5.14}$$

best_sol_n_programs indicates in how many programs the best solution S_i is produced during execution, NUM_P is the number of benchmarks/programs and consequently the number of FST files, k_1 and k_2 are hyperparameters that can be chosen by comparing results from multiple executions. The main idea behind Equation 5.13 is to use $k_1 \in [0, 1]$ to modulate the strength of two different terms: the first one considers how far the solution is from the desired number of programs $k_2 \cdot \texttt{NUM\_P}$ ($k_2 \in [0, 1]$ and $k_2 \cdot \texttt{NUM\_P} \in \mathbb{N}$), while the second one drives the evolution in different ways depending on if best_sol_n_programs is bigger than the target or not (obviously by trying to make it smaller if it is bigger and vice versa).
As a side note, if the trigger is purely AND-based, it could be beneficial to add to Equation 5.13 another term consisting in $\frac{\texttt{trigger\_max}}{\texttt{NUM\_TRIGGER}}$, where trigger_max is the number of t_i triggers which are at 1 in the trigger condition and NUM_TRIGGER corresponds to M. In this way, a triggering condition with more ones is rewarded, due to it being more solid.
The third component is driven as follows:

$$f_2 = k_3 \cdot \frac{\texttt{NUM\_TREES} - \texttt{lowest\_hamming}}{\texttt{NUM\_TREES}} + (1 - k_3) \cdot \texttt{third\_element} \tag{5.15}$$

where third_element (abbreviated as t_e) is defined piecewise as:

$$\texttt{t\_e} = \begin{cases} \mathrm{e}^{-k_4 \cdot (\texttt{min\_val} - \texttt{TARGET})^{k_5}} & \text{if } \texttt{min\_val} > \texttt{TARGET} \\ \mathrm{e}^{-k_6 \cdot (\texttt{min\_val} - \texttt{TARGET})^{k_7}} & \text{if } \texttt{min\_val} \leq \texttt{TARGET} . \end{cases} \tag{5.16}$$

NUM_TREES corresponds to T, lowest_hamming is the lowest Hamming distance between the best solution S_i and the other solutions in different programs, min_val is the number of occurrences for the best solution S_i, TARGET is the desired number of occurrences for the best solution[11], k_3 to k_7 are additional hyperparameters. As before, f_2 consists in two parts whose strength is modulated by k_3: the first one is used to impose a lowest_hamming as low as possible, while the other one drives evolution differently depending on how big min_val is with respect to the target. The structure for the second term in Equation 5.16 has proven to be useful for pure AND-based triggers, provided that $k_7 > k_5$. In this way, the exponential function decreases very fast when $\texttt{min\_val} \to 0$, while it decreases at a slower rate when $\texttt{min\_val} \to \infty$. This constraint penalizes solutions with very low number of occurrences, with the following rationale:

- not every trigger activation produces a malicious payload which propagates to the outputs, mainly due to the program flow and the positioning of the payload. Forcing a TARGET high enough gurantees more opportunities to find at least a pattern which influences the program in a meaningful way.
- a very low number of activations could indicate a very specific conditions, which may be difficult to replicate in a real-world program (not prepared *ad hoc*).

[11] Propagation of each malicious value to the POs is not guaranteed, so it is better to impose a TARGET > 1.

When pure `XOR`-base trees are considered, finding a solution with a very small number of occurrences becomes more difficult. For this reason, a simple `third_element` as follows will suffice:

$$\texttt{third\_element} = \frac{1}{\texttt{min\_val}}$$

As anticipated, there is the possibility to operate on another kind of trigger structure by running `metric_computation_sequential_xor.py`. The main idea behind this new structure is to keep all the FFs and to introduce backward or forward connections, inside the same tree, or to mix up different trees by connecting them together in random points. As hinted in Section 4.3.1, the purpose of FFs is to shield the circuit against static techniques as FANCI without actually changing the logic functionality of the trigger (they simply delay the output by $\lfloor log_2 M \rfloor$ cycles, the number of layers in the tree). Introducing new connections helps in making the condition more complex, because values produced in different sequential stages are compared together. In order to do that FFs should be left in the circuit, compromising the strategy described in Section 5.5.4 to make the circuit invisible to FASTrust. On the other hand, making the sequential components of the tree more complex makes the counter circuitry useless, reducing the overall area complexity of the trojan.
The actual injection of the trigger circuitry is performed in `inject_trees_window.py`, which expands the netlist adding both the counters for defining the on-window and the sequential trigger trees. Another Verilator simulation could be done over this file to assess if the behavior is the desired one, producing the VCD/FST logs for the final triggers and checking if they are coherent with the results provided by `metric_computation.py`. Over the course of the thesis, a program named *GTKWave* [62] has been used as waveform viewer to perform manual analysis of simulation results.

5.6 Payload injection

Once the netlist with the trigger circuit is produced by `inject_trees_window.py`, payload injection can be performed. Defining the characteristics of the payload is a problem which shows many degrees of freedom, such as: payload positioning, functionalities and how the malicious circuit is embedded in the genuine one.
A very important topic to be tackled is related to the influence of a malicious value over program execution. In the following discussion, only *modify payload*[12] trojans will be considered. Defining in which way an unintended value affects the control flow of a program is not trivial, since the effect is something that can be quantified only by the user. When implementing *leak payload* trojans the effects are explicitly defined by the trojan itself, while the functionalities of *modify payload* ones depend on the circuits around as well. For these reasons, a simple formal metric have been defined based on the values of the POs. Let us assume that the same C cycles long simulation is performed on both the original DUA[13] and the infected one, which have O outputs named $Y_0, Y_1, ..., Y_{O-1}$ and $Y'_0, Y'_1, ..., Y'_{O-1}$ respectively. These outputs assume values $Y_i(j)$ $Y'_i(j)$ at cycle c. A payload can be defined as *externally observable* if and only if:

$$\exists i \in \{0, 1, ..., O-1\}, \exists c \in \{0, 1, ..., C-1\} \mid Y_i(c) \neq Y'_i(c) \tag{5.17}$$

This is a very broad definition, since it includes every PO of the circuit. However, there is no guarantee that a variation over data in the data bus produces an effect, since it may

[12] Refer to the taxonomy in Section 2.2.3.
[13] *Design Under Analysis.*

happen in a cycle when these data are not meaningful the core (data bus lines are driven only during *store* instructions). A more restrictive definition could be built checking only the address bus, since a variation would indicate a different PC[14] value and thus a change in the control flow. Again, the address bus could be undriven during some cycles depending on the type of processor. As it should be evident, it is impossible to define observability in a black-box environment without information about the core inner behavior. For this reason, the simple definition introduced above will be used without further analysis.
In the current toolchain, payloads have been implemented according to the deTrust metodology introduced in Section 4.2.2. Let us revise the flow detailing the actual steps to be followed:

1. the first step consists in the identification of suitable payloads. In order to do that, a fanin cone computation of the POs have been performed. Wires that are closer to the outputs are prioritized, under the assumption that variations over their values are easier to propagate. An additional metric that should be considered is the number of FFs inputs in the fanout cone of these wires, since this affects the vulnerability against FASTrust (Section 4.3.3).

2. depending on the number d of sequential stages over which the payload circuit is distributed, a sequential fanin cone for the selected payload wires has to be computed. This fanin ignores all the FFs up to depth d. For simplicity, if a PI appears while computing this fanin for a candidate payload then this payload is discarded (since PIs cannot be delayed through the addition of FFs, as anticipated in Section 4.2.2). Starting from the the base of this fanin cone consisting in signals $\{s_0, s_1, ..., s_N\}$, the boolean expression of the candidate payload $p_i(s_0, s_1, ..., s_N)$ can be computed.

3. by applying the flow described in Section 4.2.2 it is possible to modify p_i to distribute it over multiple sequential stages, making it invulnerable against VeriTrust and FANCI.

4. for each candidate payload, the engineered circuit is injected and a simulation is performed against each benchmark. The FST files produced are then compared with the golden ones produced during simulations over the original design, performing the check in Equation 5.17 to assess if the trigger is externally observable.

Every one of these steps will be explained afterward, detailing the algorithms that have been used and how HAL has been integrated in the flow. As a side note, it would be possible to build a Byron toolchain for the payload definition as well. However, there are two critical issues which should be addressed in the process:

- determining if the POs are affected after injecting a payload is a *hit or miss* problem, since it produces to a discrete binary outcome {`AFFECTED`,`UNAFFECTED`}. Such problems are not fit to be used in a fitness function, because they are unable to drive the evolution in a specific direction (two individuals which produce a `UNAFFECTED` are always considered to be equal). It is possible to make the fitness function more continuous by tracking how many signals in the fanin cones of the POs are affected by the malicious payload. It is possible to extend the definition in Equation 5.17 to introduce *internal visibility*. Let us assume that a simulation is performed on both the genuine DUA and the infected DUA, whose outputs

[14] *Program Counter.*

$\{Y_0, Y_1, ..., Y_{O-1}\}$ and $\{Y'_0, Y'_1, ..., Y'_{O-1}\}$ have fanin cones at sequential depth d $\{FC|_d(Y_0), FC|_d(Y_1), ..., FC|_d(Y_{O-1})\}$ and $\{FC|_d(Y'_0), FC|_d(Y'_1), ..., FC|_d(Y'_{O-1})\}$ respectively. If all the FFs up to depth d are removed from the circuit and given $C|_d(Y_i^{(\prime)})$[15] as the circuit (set of nodes $a_{ij}^{(\prime)}$) which drives $Y_i^{(\prime)}$ starting from the FFs at depth d, then $FC|_d(Y_i^{(\prime)})$ can be defined as:

$$FC|_d(Y_i^{(\prime)}) \coloneqq \{a_{ij}^{(\prime)} \mid a_{ij}^{(\prime)} \in C|_d(Y_i^{(\prime)})\} \tag{5.18}$$

The portion of each fanin of the malicious circuit POs which includes the payload circuitry should be excluded from $FC|_d(Y'_i)$, because payload circuitry is not included in the benign DUA. Given all these actors, a candidate payload is said to be *internally observable* over $a_{ij}^{(\prime)} \in C|_d(Y_i^{(\prime)})$ if and only if:

$$\exists i \in \{0, ..., O-1\} \mid \exists j \in \{0, ..., \#FC|_d(Y_i^{(\prime)})\}, \exists c \in \{0, ..., C-1\} \mid a_{ij}(c) \neq a'_{ij}c \tag{5.19}$$

where $\#FC|_d(Y_i^{(\prime)})$ is the cardinality of sequential fanin cone i. The definition of internal observability makes it possible to add an high amount of granularity to the fitness function, because instead of observing if the malicious value propagates to the POs it is possible to perform the same check over every signal a_{ij} in the fanin cone of the POs. In this way, 2 payloads which are not able to affect the POs can still be distinguished by how they affected the intermediate a_{ij} signals. Incorporating this term in the fitness function can be done in different ways, for example by rewarding the solution which managed to go closer to the POs or by favoring a wide exploration of the fanin cones. The important element to keep into account is the need to log all these signals in the FST files, which will be surely responsible for an increase in size and a longer parsing time.

- combining parallel execution through `ParallelScriptEvaluator` and a re-simulation for each one of the N individuals requires having N distinct copies of the Verilator toolchain in the */tmp* drive (referring to the flow at Section 5.5). This could produce a non-negligible memory demand overhead, which is compensated by the decrease in the overall time due to parallelization. In general, a solution which didn't require one simulation per individual would be preferred, but it probably doesn't exist due to the nature of the payload problem.

5.6.1 Identification of candidate payloads

This preliminary step is performed by the `select_payloads.py` script, which explores the sequential fanin of each output and logs to a file `payload_ids.txt` all the nodes which satisfy particular conditions. Each entry written to file has the following format:

p_i name - $p_i(FC|_d(p_i))$ - $FC|_d(p_i)$ - maximum gate depth d^*

where $FC|_d(p_i)$ is the sequential fanin cone (at depth d) of the signal p_i as defined in Equation 5.18, $p_i(FC|_d(p_i))$ is p_i expressed as a function of its fanin cone, d^* is the depth of the deepest node in the cone [16].
Candidate payloads are produced according to the following flow:

1. for each netlist output, a recursive fanin computation is performed up to depth `FANIN_DEPTH`. A set is filled with the nodes which belong to this cone, and it is

[15] From this point on, the notation $X^{(\prime)}$ will be used to refer both to X and X'.
[16] This *depth* is computed simply as the length of the gate sequence between this node and p_i, so it is the traditional depth of a circuit (but computed starting from the POs and going backward).

used in the next step as a pool from which candidate payloads are selected. The code to populate this set will not be shown here, since it is very similar to the one in Listing 4. There is however a noticeable difference, which resides in how FFs are handle. Since the fanin to be computed is a *sequential* one, which means that it should traverse multiple sequential stages, FFs should by bypassed through the code in Listing 6. The first step when bypassing them is to locate the input pin `D` of the FF iterating through all the input endpoints (line 2). Once it has been located, the signal driving this pin should be checked to determine if it is a PI: if it is, then recursion should be stopped (line 10), otherwise the new net should be added to the cone and the corresponding endpoint is set as `e` (line 6).

```
1     if str(list(e.get_gate().get_boolean_functions().values())[0])
      ↪  == "IQ":
2         for endpoint in e.get_gate().get_fan_in_endpoints():
3             if endpoint.get_pin().get_name() == "D":
4                 if not endpoint.get_net().is_global_input_net():
5                     fanin_ret.append([endpoint.get_net().get_id(),
                      ↪  level])
6                     e = endpoint.get_net().get_sources()[0]
7                     gate_name = e.get_gate().get_name()
8                     break
9                 else:
10                    return fanin_ret
```

Listing 6: Code to bypass FFs.

2. for each node among the selected ones, the $p_i(FC|_d(p_i))$ function is computed. d corresponds to the `NUM_LEVELS` constant, and `DEPTH` marks the maximum acceptable depth for nodes in the expression. The complete code is reported in Listing 7, which is again very similar to the original flow even if it presents some key differences.
   The first one is related to the necessity to mark functions which depend on PIs (or whose nodes are deeper than `DEPTH`) as invalid, done through the injection of "dummy" characters as "%" or "$" in the expression (lines 6, 19, 37).
   The second key difference is given by how the recursion is handled to compute the expression. Given endpoint `e`, the logic function of the driving gate is retrieved from the liberty file at line 43. This logic function is in the following form:

$$f = P_0 x P_1$$

   where P_0,P_1 are pin names and x is the gate operator. With this function available, a recursive call is performed over each input pin P_i of the gate, collecting the result in `to_replace` (line 50). In the end, the name of P_i in `logic_expr` is replaced with `to_replace`, building a new expression which includes P_1 fanin cone as well.
   The last element to underline is the check at line 28, which stops the recursion when the maximum sequential depth d/`NUM_LEVELS` is reached.

```
1  def recursive_expression(netlist, net_id, symbol_list, depth,
   ↪  level):
2
3      level_curr = level
4
```

```
    if depth > 10:
        return "%"

    gate_name = ""

    # retrieve the gate whose output pin is mapped to the signal being considered
    if not netlist.get_net_by_id(net_id).is_global_input_net() and len(netlist.get_net_by_id(net_id).get_sources()) != 0:
        e = netlist.get_net_by_id(net_id).get_sources()[0]
        gate_name = e.get_gate().get_name()
    elif not netlist.get_net_by_id(net_id).is_global_input_net():
        symbol_list.append(netlist.get_net_by_id(net_id).get_name())
        return netlist.get_net_by_id(net_id).get_name()
    else:
        # invalidate the return value in case you reach an input, because inputs cannot be delayed
        return "$"
    # if the gate if a FF you need to bypass it, unless you have reached the max sequential depth NUM_LEVELS
    if (str(list(e.get_gate().get_boolean_functions().values())[0]) == "IQ"):
        for endpoint in e.get_gate().get_fan_in_endpoints():
            if endpoint.get_pin().get_name() == "D":
                # move to the next level
                level_curr += 1
                symbol_list.append(endpoint.get_net().get_name())
                # if you reached the FF marking the last layer to be considered then you have to return
                if level_curr == NUM_LEVELS:
                    return endpoint.get_net().get_name()
                # update the endpoint and the gate_name definitions, since you are now considering the FF input
                # but remember to stop when you hit a global input
                if not endpoint.get_net().is_global_input_net():
                    e = endpoint.get_net().get_sources()[0]
                    gate_name = e.get_gate().get_name()
                else:
                    # invalidate the return value in case you reach an input
                    return "$"

    if "gnd" in gate_name:
        symbol_list.append(netlist.get_net_by_id(net_id).get_name())
        return netlist.get_net_by_id(net_id).get_name()

    logic_expr = str(list(e.get_gate().get_boolean_functions().values())[0])
    for net in e.get_gate().get_fan_in_nets():
        for dest in net.get_destinations():
```

```
            if dest.get_gate().get_name() == gate_name:
                gate_pin = dest.get_pin().get_name()
                break
        # recur on each input to the gate if the fanin of the
        ↪  corresponding net hasn't already been explored
        to_replace = recursive_expression(netlist, net.get_id(),
        ↪  symbol_list, depth + 1, level_curr)
        logic_expr = logic_expr.replace(gate_pin, to_replace)

    return logic_expr
```

Listing 7: Recursive computation of $p_i(FC|_d(p_i))$.

3. after building the expression, its length is checked together with the number of literals included. For the sooner an upper limit is imposed as `NUMBER_OF_CUBES`, for the latter both a lower and a higher threshold have been defined. The main idea behind this check is to avoid selecting payloads with complex expressions, preventing a potential huge increase in area. In order to assess the number of cubes, the solution must be converted to a SOP form through `sympy.to_dnf()`[17]. In case the expression respects the constraints, then it is added to the list of candidate payloads to be written in `payload_ids.txt`. Besides, it is necessary to mark the declarations of selected payloads with the Verilator directives */* verilator tracing_on */* and */* verilator tracing_off */*.

As anticipated, it could be possible to extend this selection by considering the number of FFs inputs in the fanout cone of each candidate payload. By making sure that this number is high enough, additional resistance against FASTrust would be enforced.

5.6.2 Design of the sequential stages for the payload

This step follows the strategy in Section 4.2.2, which will be briefly shown below to highlight how the actual sequential tree is built. Let us start from a generic SOP form for the payload logic expression $p_i(FC|_d(p_i))$:

$$p_i(FC|_d(p_i)) = \bigvee_{c=0}^{C} \bigwedge_{l=0}^{L} a_l^* \tag{5.20}$$

where $\bigvee$ is the logic `OR` (defined as a compact iterative `OR` as done for the `AND` in Equation 5.2); a_l^* indicates that in each cube each literal a_l could be adfirmed, complemented or absent; C is the number of cubes. If the trigger sequence is added then:

$$p_i^T(FC|_d(p_i)) = \left(\bigvee_{c=0}^{C} \bigwedge_{l=0}^{L} a_l^* \right) \vee \bigwedge_{j=0}^{T} (t'_j)^* \tag{5.21}$$

T being the number of trigger trees, and thus the number of final literals in the trigger cube. The t'_j values are the outputs of the trigger trees, as defined previously. In order to merge the trigger cube with the original circuit, it is necessary to duplicate a genuine cube as follows:

$$p_i^T(FC|_d(p_i)) = \left(\bigvee_{\substack{c=0 \\ c \neq c'}}^{C} \bigwedge_{l=0}^{L} a_l^* \right) \vee \left(t'_k \wedge \bigwedge_{l=0}^{L} a_l^* \vee \overline{t'_k} \wedge \bigwedge_{l=0}^{L} a_l^* \right) \vee \bigwedge_{j=0}^{T} (t'_j)^* \tag{5.22}$$

[17]The SOP form is also named *Disjunctive Normal Form* (*DNF*)

exploiting $t'_k \vee \overline{t'_k} = 1$. The literal t'_k corresponds to one of the trigger literals, to be chosen according to the criteria in Section 4.2.2. It is then possible to collect t'_k from both the duplicated genuine cube and the trigger cube:

$$p_i^T(FC|_d(p_i)) = \left(\bigvee_{\substack{c=0 \\ c \neq c'}}^{C} \bigwedge_{l=0}^{L} a_l^* \vee \overline{t'_k} \wedge \bigwedge_{l=0}^{L} a_l^* \right) \vee t_k \wedge' \left(\bigwedge_{l=0}^{L} a_l^* \vee \bigwedge_{\substack{j=0 \\ j \neq k}}^{T} (t'_j)^* \right) \tag{5.23}$$

where $(\bigvee_{\substack{c=0 \\ c \neq c'}}^{C} \bigwedge_{l=0}^{L} a_l^* \vee \overline{t'_k} \bigwedge_{l=0}^{L} a_l^*)$ was called f'_n and defined as h_1, while $h_2 \coloneqq t'_k$ and $h_3 \coloneqq \bigwedge_{l=0}^{L} a_l^* \vee \bigwedge_{\substack{j=0 \\ j \neq k}}^{T} (t'_j)^*$. As suggested in the original analysis, it is possible to go on applying the same procedure to mask different triggers on different sequential levels. The program which implements this flow supports this generalization. According to the discussion in Section 4.2.2, introduing at least a sequential level is needed to bypass VeriTrust. Additional sequential level make the payload more distributed, potentially making the design stealthier against FANCIX and VeriTrustX (considering the whole $p_i^T(FC|_d(p_i))$ is becomes more computationally expensive when d is higher).
Due to the need to refactor p_i expression without affecting the circuit around, new circuitry should be added to drive the trojan affected version of p_i. The area complexity of this circuit will be affected by the number of cubes and the number of literals of the new expression, which depends on the sequential depth of the final result (how many iterations were made). The algorithm in charge of masking the triggers against genuine cubes is completely implemented inside the `recursive_cube_masking()` function, which receives a list of candidate cubes `selected_cubes` together with a list of all cubes in $p_i^T(FC|_d(p_i))$ named `cubes`. Since the function consists mainly in the manipulation of p_i expression as a string, its code won't be reported here.
The same applies for the `recursive_code_insertion()` function, which rewrites the original netlist adding the RTL code of the payload. The actual synthesis is performed afterward, in order to make the whole flow technology independent. This partially sacrifices the efficiency, because translation into standard cells can be done in a suboptimal way. A viable alternative could be the one to rely on the *Netlist Writer* included in HAL to automatically insert the new circuitry, since the implementation of the netlist graph is strongly technology dependent and a manipulation of this model would implicitly consider the underlying liberty file.
As anticipated, there are two critical points about this implementation, both related to the need to have an *internally* or *externally* observable payload. In particular, trigger condition propagation can be divided in two phases:

- the propagation up to p_i. This step should not be given for granted, because as shown in Equation 5.25 the trigger cube is combined with benign functionality through a simple `OR`. As a result, if both $\bigwedge_{j=0}^{T} (t'_j)^*$ and $\bigvee_{c=0}^{C} \bigwedge_{l=0}^{L} a_l^*$ are '`1`' then the trigger will not be observed.

- the propagation from p_i to the POs Y'_i (when evaluating external observability) or to elements in their respective fanins $FC|_d(Y'_i)$ (when evaluating internal one). This step will be abundantly covered in the next sections.

A few additional notes on the first point are needed. Intuitively, it could be possible to force a propagation to p_i by employing an `XOR`. Using an `XOR` would affect p_i even when both the values are '`1`', because in that case the final p_i would be '`0`'. Equation 5.25

would change as follows:

$$p_i^T(FC|_d(p_i)) = \left(\bigvee_{c=0}^{C}\bigwedge_{l=0}^{L} a_l^*\right) \oplus \bigwedge_{j=0}^{T}(t'_j)^* \tag{5.24}$$

If the `XOR` operator is unrolled to obtain two cubes $\overline{a}b + a\overline{b}$, then:

$$p_i^T(FC|_d(p_i)) = \overline{\left(\bigvee_{c=0}^{C}\bigwedge_{l=0}^{L} a_l^*\right)} \wedge \bigwedge_{j=0}^{T}(t'_j)^* \vee \left(\bigvee_{c=0}^{C}\bigwedge_{l=0}^{L} a_l^*\right) \wedge \overline{\bigwedge_{j=0}^{T}(t'_j)^*} \tag{5.25}$$

It is possible to expand the second term $a\overline{b}$ through De Morgan Theorem[18] However, it is impossible to do the same with the first one, because it introduces a logic multiplication. As a consequence, the whole first term results to be even rarer than the original trigger cube (it requires both the original function to be '`0`' and the trigger cube to be '`1`'). This would make it sensitive to VeriTrust analysis, invalidating any possible masking operation that can be done over the second term.

5.6.3 Simulation and "fitness" evaluation

After the new design is produced, a simulation is performed over every benchmark and the FST files are collected. *pylibfst* is then used to read every logged value from the simulation and to compare them with the ones from the golden simulations (the one done over the genuine circuit). Depending on which values are logged, pylibfst can be used to verify either internal observability (over a node in $FC|_d(Y_i)$) or external observability (over the POs Y_i). A fitness value is produced for each candidate p_i, which quantifies how much it favors observability of the overall trojan. Since no Byron toolchain has been prepared for the payload, this number doesn't play a role in evolving solutions. The fitness function is again a vectorial one $\mathbf{f}$, where the components are defined as follows:

- f_0 can be either '`0`' or '`1`' depending on if the payload circuitry forced a malicious value over p_i. In the previous paragraph, a brief discussion on how this is not always possible was provided. This component shows a high degree of discontinuity, so it is not fit to be a real fitness function in a Byron environment. However, due to the problem of modifying p_i being *hit or miss*, there is no way to make this component more continuous. Luckily, the wide majority of payloads that are produced from the toolchain is able to propagate at least to p_i.

- f_1 is built through an exponential function very similar to the one in Equation 5.16, where a certain `TARGET` is defined as the desired number of occurrences of wrong values over a specific output Y_i. This part of the function is the one which computes external observability over output Y_i. The output pin should be chosen according to the criteria mentioned in Section 5.6, in particular outputs that act as control signals and address bus pins should be preferred. The reason behind this is that they act as good indicators to assess if the execution flow has been compromised.

- f_2 is the defined as the count of occurrences of wrong values over every node in $FC|_d(Y_i)$, weighted by the depth of the node which was found to be different. As will be shown later, the formula rewards nodes that are closer to the outputs with a larger fitness value.

[18] De Morgan Theorem consists in the equality $\overline{\bigwedge_{j=0}^{T}(t'_j)^*} = \bigvee_{j=0}^{T}\overline{(t'_j)^*}$.

The signals comparison is performed in the `compute_fitness.py` script, which reads and parses every FST file one by one and compares them with the respective golden versions. Then, based on the comparison results, it computes the fitness function.
The FST read operation and the signals dictionary population is done through the snippet in Listing 8. The strategy to follow is very similar to the one in the general pylibfst flow in Listing 5, but it has to be replicated for both FST versions. This snippet introduces all the actors that will be involved in the comparison process, for example: `sim_fst[i]` and `golden_fst[i]`, which are the FST files produced during the payload-affected and the golden simulations; `new_signals` and `new_golden_signals`, the two dictionaries for the signals from payload-affected and the golden simulations; `input_lines`, an array populated with the signals to be logged; `comparison_result`, a four elements array whose components are used to compute the three components of the fitness function.

```
for i in range(NUM_PROGRAMS):

    new_signals_by_names = dict()
    new_signals_by_handles = dict()

    for name in signals_lines + output_lines + [payload]:
    # for name in [payload]:
        for key, val in signals[i].by_name.items():
            if key.split()[0] == name:
                break
        signal = signals[i].by_name.get(key)
        new_signals_by_names[name] = signal
        new_signals_by_handles[signal.handle] = signal

    new_signals = Signals(new_signals_by_names, new_signals_by_handles)

    new_signals_by_names = dict()
    new_signals_by_handles = dict()

    for name in signals_lines + output_lines + [payload]:
    # for name in [payload]:
        for key, val in golden_signals[i].by_name.items():
            if key.split()[0] == name:
                break
        signal = golden_signals[i].by_name.get(key)
        new_signals_by_names[name] = signal
        new_signals_by_handles[signal.handle] = signal

    new_golden_signals = Signals(new_signals_by_names,
    ↪  new_signals_by_handles)

    comparison_result = list(map(add,compare_signals(sim_fst[i],
    ↪  golden_fst[i], new_signals, new_golden_signals, output_lines,
    ↪  signals_depths, payload),comparison_result))
```

Listing 8: Code to populate the signals dictionary for both golden simulations and actual ones.

The function which performs the actual comparison is `compare_signals()`, which is reported in Listing 9 instead. Again, the code is built on the same flow presented in Listing 5, but in this case two FST files are read at the same time. The comparison is done at every timestamp, corresponding to the two edges in each clock cycle. The key part in implementing the check consists in the `if` statement which extends between line 30 and 39. In particular, `results` update is done differently depending on if the signal compared is the payload, an external signal or a signal in $FC|_d(Y_i)$. An important element to note is how `results[2]` depends on the circuit depth of the signal which was found to be different. In this way it is possible to generalize the function for a Byron flow, since it would be possible to evolve solutions which get closer and closer to the final outputs. Another possibility could be to produce a fitness function which exclusively depends on the wrong node found to be closer to the outputs, since this function would represent the propagation flow in a better way. An analysis on how continuous this function is should be done, in order to assess if it would fit an evolutionary flow.

```
1  def compare_signals(fst, golden_fst, signals, golden_signals,
   ↪  output_lines, signals_depths, payload):
2
3      # allocate the array where to build the result
4      results = [0] * 4
5
6      # get timestamps of all signal changes
7      pylibfst.lib.fstReaderSetFacProcessMaskAll(fst)
8      pylibfst.lib.fstReaderSetFacProcessMaskAll(golden_fst)
9      timestamps = pylibfst.lib.fstReaderGetTimestamps(fst)
10     timestamps_golden = pylibfst.lib.fstReaderGetTimestamps(golden_fst)
11     results[3] = timestamps.nvals
12
13     buf = pylibfst.ffi.new("char[256]")
14     index = 0
15     for ts in range(timestamps.nvals):
16         index += 1
17         time = timestamps.val[ts]
18         for signal in signals.by_name.values():
19             val = pylibfst.helpers.string(
20                 pylibfst.lib.fstReaderGetValueFromHandleAtTime(
21                     fst, time, signal.handle, buf
22                 )
23             )
24             golden_val = pylibfst.helpers.string(
25                 pylibfst.lib.fstReaderGetValueFromHandleAtTime(
26                     golden_fst, time,
                        ↪  golden_signals.by_name[signal.name].handle, buf
27                 )
28             )
29             # compare them, weighting them differenly depending on if
               ↪  they are internal signals or output ones
30             if val != golden_val:
31                 if signal.name == payload:
32                     # you managed to force a change over the value of
                       ↪  the signal where the payload is deployed
```

```
                    results[0] += 1
                if signal.name in output_lines:
                    # the outputs have been affected
                    results[1] += 1
                else:
                    # an internal signal close to the outputs have been
                    ↪ affected
                    results[2] += (FANIN_DEPTH -
                    ↪ signals_depths[signal.name] + 1)

    pylibfst.lib.fstReaderFreeTimestamps(timestamps)
    return results
```

Listing 9: Comparison between golden and actual values.

5.6.4 An alternative: SAT-based flow for malicious payload propagation

As anticipated, the flow presented up to now is potentially very slow due to the need to simulate every candidate payload. For this reason, an alternative solution which includes a SAT solver engine is presented. The main idea is to produce a pattern able to propagate the original malicious value to the circuit outputs. A general take on this problem will be provided, with the actual implementation being suggested as future work. The major problem about this solution is that it invalidates the benchmark-based approach which was presented before, since there is no guarantee for a SAT-generated pattern to be produced during test benchmarks execution. Besides, exactly as scan techniques, the SAT is not instructed on *functional* or *non-functional* input configurations, so it could potentially produce non-functional input sequences that are not possible to replicate at all.
Implementing the aforementioned flow requires a way to excite p_i before the actual SAT-based propagation from p_i to Y_i. It is possible to do so only through a simulation, which apparently invalidates the premises. However, it must be noted that in this case there is no need to compute $FC|_d(Y_i)$. For this reason, a simpler version of the netlist could be employed in the simulation, such as the one which is synthesized but not mapped. In this way the simulation duration is sped up considerably, because the unmapped netlist has a much lower complexity (in terms of number of processes and concurrent assignments). In order to produce this netlist, the synthesis command in Listing 1 should be modified removing the `dfflibmap` and `abc` passes. This introduces a limitation over the available p_i nodes, because these must be present in the synthesized netlist.
The flow to be followed is this one:

1. first, a simulation of the circuit up to the moment in which the trigger condition reaches p_i is performed. All the `reg` values in the circuit should be logged, since the state at the moment when p_i is found to be wrong must be imported in the SAT solver (as a starting point for the SAT-based generation). The output of this phase is an FST/VCD file with all the reg values (and the payload p_i) logged.

2. then, the FST/VCD file is parsed to determine the values of each reg value at the exact cycle when p_i is found to be wrong. This works under the hypothesis that the trigger condition is able to affect p_i, which is simple to verify when working on the synthesized netlist due to the limited complexity of the simulation. The values found in this way are set to be the `initial` ones of netlists regs. As an alternative, Yosys

could be used directly relying on `sat --set-init <signal> <value>`[19], which allows to drive whichever reg signal with the initial value *<value>*.

3. a SAT solving instance is started from the netlist with the initial values set, aiming to find the input pattern which produces a discrepancy on an output Y_i with respect to the golden netlist. It is possible to perform this test through a miter, which can be created as in Equation 2.1.

A partial implementation of point 1 has been implemented for the AFTAB, consisting in:

- `ubus_test_levels_log_vals.py` to parse the VCD and to populate a file `vals_log.txt` with the final values. For the AFTAB, Modelsim™ (*vsim*) can be used for simulation, since the small pre-mapping netlist satisfies the constraints introduced by the free version. In order to parse vsim generated VCDs through vcd_parsealyze a preprocessing is needed, because these VCDs are not fully compatible with the parsing tool. This preprocessing is implemented in `adjust_vsim_vcd.py`.
- `build_initial.py` to refactor the netlist in order to insert `initial` directives over the `reg` signals.
- an sby script to perform SAT solving, following the guidelines in Section 3.10.

This could be joined with a simplified miter implementation which has been written for the SRV32. This miter uses SystemVerilog `assert` statements to assess if the values produced by the reference are equal to the ones produced by the UUT[20]. In case they aren't, the execution is interrupted due to an error being thrown by the assertion. Another possibility to implement a miter could be the one to `AND` together the results of the `XNOR` operations between corresponding outputs, building a final circuit with a single output. In particular, given Y_i and Y_i' the generic outputs from the golden circuit and the UUT respectively and $\mathbf{x}$ the common inputs, then the final miter output will be:

$$M = \bigwedge_{i=0}^{N_Y} \overline{Y_i(\mathbf{x}) \oplus Y_i'(\mathbf{x})} \tag{5.26}$$

or equivalently, according to de Morgan:

$$M = \overline{\bigvee_{i=0}^{N_Y} Y_i(\mathbf{x}) \oplus Y_i'(\mathbf{x})} \tag{5.27}$$

The entity implementing the miter is defined in `miter.sv`, while the SymbiYosys script interfacing with the SAT engine is `srv32_miter.sby`.
A critical point of this technique is given by the amount of time required to perform the SAT-based generation of the input pattern, which could strongly vary depending on the depth of the payload signal. Placing the payload closer to the outputs could be beneficial, because it would reduce the number of cycles needed for the propagation. Besides, such a payload could also decrease observability metrics defined in Section 4.3.4, guaranteeing more stealthiness against COTD.

[19] Referring to `https://yosyshq.readthedocs.io/projects/yosys/en/latest/cmd/sat.html`.
[20] *Unit Under Test.*

5.6.5 A review of strengths and weaknesses

Since the presentation of the toolchain has been completed, it is necessary to tell how the complete product is able to elude detection techniques, addressing also limitations of the current version. Viable solutions will be proposed as part of the discussion on how to extend the toolchain, included in Chapter 7. Let us revise all the detection techniques introduced in Chapter 2, clarifying how the toolchain respond to each one of them:

- due to the need to activate the trojans in some specific benchmarks, it would be possible to include these benchmarks in the toolchain and use the genetic algorithm to evolve solutions which activate only in these specific cases. In order to minimize the probability of spurious activations, the number of benchmarks should be increased as much as possible. Besides, some benchmarks should include patterns close enough to the ones ATPG-generated and applied during functional testing. In this way it would be possible to increase the probability of bypassing simulation-based testing techniques. This proves to be particularly critical when an `XOR`-based trigger tree is chosen, since the `XOR` truth table is responsible for unpredictable behavior over patterns non included in benchmarks. Replacing some `XOR` gates with `AND` gates could address this problem, reducing area occupation in the process. The whole flow is detailed in Section 5.5.4.

- scan chains based testing techniques are responsible for reducing the sequential complexity of a circuit to a combinational one, allowing FF outputs to be externally driven (as pseudo PIs). Including trojan circuitry in scan-chains would make the trigger condition a trivial one, allowing ATPG patterns to model a *stuck-at-X fault* (*SaX*) [21]the trigger wire and to generate patterns to excite it. Through the structure proposed in Section 4.3.1 and detailed in Section 5.5.4, the final trigger condition is never explicitated as a single wire (the T trees are never merged), thus shielding against a SaX fault model targeting a single wire. This is certainly true when analyzing the trigger tree, but it doesn't hold when considering the payload structure. In particular, converting all the registers hosting the t'_i values to scan registers and setting their values at '`1`' would potentially force a wrong value '`1`' over p_i after D cycles (where D is the sequential depth of the payload structure). The worst case consists in all registers in the payload structure (the ones dubbed as h_{1i}, h_{2i} and h_{3i} in Section 4.2.2) being converted to scan ones. In this case it would be even easier to force the p_i to erroneously evaluate at '`1`', because it could be enough to set $h_{20} = 1$ and $h_{30} = 1$. For these reasons, it is necessary not to include payload-related FFs in scan chains. FFs included in the trigger tree can be included in scan chains only when the testing is a *full scan* one or when $D > T_C$, where T_C is the number of clock cycles for which each test is run in a *partial scan* approach. In this case, a malicious condition reaching the registers hosting t'_i values wouldn't be able to propagate to the POs or to the PPOs. From this point of view, having a very high value for D would make scan chain based testing harder. Assuming T to be a power of two, then the number of FFs included in the trigger tree is:

$$N_{FF,tree} = M - T - 1 + M \cdot K \tag{5.28}$$

  where M is the number of partial triggers and K is the size of the window whose role has been explained in Section 5.5.5. On the other hand, the number of FFs in

[21]A stuck-at-X fault is an hardware fault that consists in a wire being set at $X = \{0, 1\}$. It can be tested through an input pattern which forces the opposite value on the wire and propagate it to the output. In the case of the thesis, a pattern testing a Sa1 fault would spot the trojan.

the sequential payload structure is given by:

$$N_{FF,sequential} = 3 \cdot D$$

since every sequential stage adds 3 FFs for h_{1i}, h_{2i} and h_{3i}. Since $D \in \{2,3,4,5\}$ and $M \in \{2,4,8,16,32,64\}$ [22], the main contribution is clearly due to $N_{FF,tree}$. For this reason, if FFs from the tree are included in scan chains then a very small number of non-scan FF will remain. This proves to be useful to evade SEM imaging techniques, which are able to spot large groups of non-scan FFs [26].

- in case the activation over a benchmark is considered easy to spot, it is possible to include multiple benchmarks which execute the same program receiving different inputs. This proves to be useful when the benchmark is included in functional test suites, because it would be unfeasible to test the same program over every possible set of data. The resulting benchmark suite would consist in a matrix $M \in \mathbb{P}^{N_B \times N_D}$, where $\mathbb{P}$ is the set of all programs, N_B is the number of different benchmarks and N_D is the number of different input patterns for each benchmark (they could be generated randomly). The problem of imposing selective activation may be formalized as forcing the trojan to be triggered only on M_{ij}, a specific program i executed with data-set j. This flow is a formalization of what was suggested in Section 5.5.4.

- a very important property of this toolchain is given by the possibility to scale it to include different solutions. Due to the modularity, designing a new trigger requires changing only the fitness function computation and the trigger insertion script. When modifying the payload, only the final part of the toolchain should be edited. The benchmark.based approach is considered to be the foundation of this work, so changes which affect this architecture would obviously require substantial modifications. Nevertheless, every kind of modification over the trojan architecture can still be performed by changing only specific files.

- when discussing about resistance against static (and dynamic) analysis it is necessary to refer to the currently implemented trojan architecture. In the following, all the techniques that have been introduced will be analyzed one by one, detailing if the implementation guarantees a good degree of stealthiness.
  The first technique to be analyzed was UCI, a dynamic technique for identifying unused circuit through equality checks between different wires. Invulnerability against UCI is essentially guaranteed by both the payload and the trigger tree structure. In particular, dividing the single tree in T trees and combining each tree in the payload structure as in Equation 5.23 makes the h_2 and h_3 parts not redundant provided that:

$$\exists c \in \{0,1,...,C\} \quad | \quad t_k(c) = 1 \wedge \bigwedge_{l=0}^{L} a_l^*(c) = 1$$

  which can be made easier by choosing the most commonly activated trigger and the most commonly activated $\bigwedge_{l=0}^{L} a_l^*$ cube. In case the payload structure must have a depth $D > 2$, then it is even possible to collect multiple cubes instead of a single one. This mechanism works independently on the trigger structure, provided that the final output still consists in $T > 1$ trees.

[22] These are typical values used in the current benchmark, but they can be modified depending on the characteristics of the UUT.

The second technique is VeriTrust (and its X variant), which still belongs to the spectrum of dynamic techniques. How the payload and the trigger structures cooperate in bypassing VeriTrust has been abundantly explained in Section 4.2.2 and Section 5.5.4, so it won't be analyzed again.
The basic technique in the catalogue of the static ones is surely FANCI, which again has been thoroughly through the course of this thesis. However, it may still be beneficial to determine resistance against its X variant, FANCIX. The following analysis will assume a purely `AND`-based trigger tree, since an `XOR` or `XOR-AND` tree proved to be invulnerable to FANCIX thanks to the low control values. As explained in Section 4.3.2, the main vulnerability of FANCIX is due to spikes in complexity when extending the CV computation to multiple levels. For this reason, masking the dependance $p_i(t'_0, t'_1, ..., t'_{T-1})$ over as many sequential stages as possible would make the the 2^m term in the $\mathcal{O}(gm2^m)$ raise exponentially (or even super-exponentially, depending on how m changes when passing from one stage to another). Besides, the necessity to randomly partition the truth table as introduced in Section 4.3.1 adds a degree of uncertainty on if it is possible to locate the low CV input at all. An additional detail to be considered is the placement of trigger tree outputs t'_i in the sequential structure. In particular, the assignments of $h_{20}, h_{21}, ..., h_{2(D-1)}$ should be done according to increasing rarity. In this way, when the cut is extended then high CV triggers are met first and the overall CV doesn't decrease too much. This works assuming the analysis starts from the output p_i and goes backward.
ANGEL has already been covered, and it has been shown how `XOR`-based trees are the only viable solutions to elude it.
FASTrust can be addresses in different ways in both the trigger tree and the payload structure. In particular, recombining different trees through the strategy suggested in Section 5.5.4 could be a viable way to artificially increase the fanout cone of each t_i wire. However, combining trees in couples as in Equation 5.7 creates couples of equivalent triggers. For this reason, it may be necessary to double the number of triggers (and trees) to obtain the same complexity as before. The resulting number of triggers will be given by:

$$M' = M \cdot 2^{(F-1)}$$

where F is the desired fanout (needed for each t_i) to bypass FASTrust. The complexity of the trees scales linearly with M [23], so the area overhead remains linearly bounded too. It must be noted that every FF in the tree should be removed to make this strategy possible. This can be done without side effects, since the `XOR` gates are invulnerable to FANCI anyway (so there is no need to separate them on multiple stages). The payload structure requires identifying how the sensitive point is given by the last sequential stage, the one connecting the $h_{\{1,2,3\}}$ FFs with payload p_i (it is evident in Figure 4.10). In particular, if the payload is a PO or a PPO then the FF-CDFG will show each FF sourcing only a single node. This can be addressed by selecting a payload as close to PIs or PPIs as possible, since those are the nodes with the highest fanout. All these countermeasures are potentially uneffective against ML-FASTrust, since building a gate-level CDFG would spot them. However, ML-FASTrust operates only on potential false-positives nodes labelled by FASTrust. For this reason, if the fanout threshold selected by FASTrust is low enough then trojans built in this way could still bypass detection. Increasing the threshold for detection too much would produce too many false positives, making the gate-level analysis more difficult.
Handling COTD will be left as future work, together with the implementation of a

[23] It must be observed that $N_{\texttt{XOR/AND},tree} = N_{FF,tree}$, from the topology of the tree.

deTest coherent framework [5]. However, one thing must be noted. The CC values of an `XOR` gate and an `AND` one, both with inputs a and b, are as follow:

$$CC^1_{AND} = CC^1(a) + CC^1(b) + 1 \tag{5.29}$$

$$CC^0_{XOR} = \min(CC^0(a) + CC^0(b), CC^1(a) + CC^1(b)) + 1 \tag{5.30}$$

$$CC^1_{XOR} = \min(CC^0(a) + CC^1(b), CC^1(a) + CC^0(b)) + 1 \tag{5.31}$$

where clearly in the `AND` case only the the CC^1 is needed, while the `XOR` tree can be active also when intermediate gates are at '`0`'. When Equation 5.30 and Equation 5.31 are analyzed, it must be considered that both $CC^0(a)$ and $CC^0(b)$ are reasonably low when $a = t_i$ and $b = t_{i+1}$ (which means for the first layer in the tree). This holds because trigger candidates must be rare signals, so their controllability metrics should be low enough. For this reason, the generic $CC^1(\{a, b\})$ can be assumed to be low enough too, since the most common configuration for triggers should be $(t_0, t_1, ..., t_{M-1}) = (0, 0, ..., 0)$. For this reason, the `XOR` related CC values are much lower than the `AND` related CC^1. This makes the `XOR`-based trigger intrinsically stronger than the `AND`-based one against COTD.

- imaging techniques consist in analyzing the circuit layout to spot suspicious components. Since the core of these techniques involves a comparison, a golden model of the circuit is needed. The rule-of-thumb to elude imaging is to reduce area occupation as much as possible, building trojans which consist in fewer gates (or simpler ones). Estimating the trojan size requires knowing the *liberty file* used in the mapping phase, which defines all the gates that can be used in the mapping process. It is still possible to perform an approximate analysis, considering typical area profiles for CMOS gates. In general, the following relationship holds:

  $$A_{AND2} < A_{XOR2} \ll A_{FF}$$

  which is mainly due to the fact that `XOR` functions requires more transistors than a simple 4-transistors `AND` in CMOS technology, and FFs are much more complex than both `AND` and `XOR`. Since in the `AND`-based tree there is also the need for FFs to bypass FANCI, the solution with `XOR` gates is the one with lower area occupation. In general, the relationship in Equation 5.28 is valid also for `XOR`/`AND` gates, the size of the trigger tree can be evaluated in advance. Unfortunately, it is impossible to do the same for the payload structure, whose area footprint depends on how complex $p_i(FC|_d(p_i))$ is. As a consequence, there is also a clear dependency between the area overhead and the value of D, the sequential depth of the fanin cone. For these reasons, more detailed analysis will be performed in Chapter 6.
  Another important point to be considered is related to the length of interconnections between trojan subcircuits. As evident, reducing the overall length of these interconnections is beneficial for the trojan footprint, but it makes it necessary for the trojan to be attached to a specific subcircuit of the core (potentially limiting the trigger strength). Over the course of the current analysis, triggers have been selected from as many different sources as possible, so area occupation due to interconnections has not been considered. A more accurate evaluation of this effect would require the trojan injection to be performed on the actual layout, where interconnections are actually routed.

- side-channel analysis involves measuring all the physical quantities that are affected by executing a program on the core, involving temperature, radio channels, cache

delays, currents etc. A variation over these quantities may indicate trojan presence, or it may even be exploited willingly, in the so-called side-channel attacks. Since this thesis doesn't cover this last class of attacks, only the unwanted side-effects of trojan injection will be analyzed.
The most commonly affected quantities are power and temperature, where an increase in the latter usually follows from an increase in the former. A general relationship to compute power consumption associated to a certain node is the following one:

$$P_c = a\,C\,V_{DD}^2\,f_{clk} \;+\; a\,t_{dis}\,V_{DD}\,I_{peak}\,f_{clk} \;+\; V_{DD}\,I_{leak}$$

where all parameters except a, the *switching activity factor*, will be considered to be unaffected by trojans. This assumption is surely an oversimplification in many contexts[24], but it still holds for the majority of gate-level trojans. The switching activity factor a can be considered as the probability of observing a transition ($0 \to 1$ or $1 \to 0$) on that node, and thus it is computed as follows:

$$a = P_{0\to1} + P_{1\to0} = 2\,P_1(1 - P_1)$$

Assuming the node is the output of a 2-inputs gate, whose inputs are A and B, then P_1 will obviously depend on the truth table of the gate logic function. In particular, for the `AND` gate and the `XOR` one:

$$a_{AND} = 2\,P_A\,P_B(1 - P_A\,P_B) \tag{5.32}$$

$$a_{XOR} = 2\,(P_A(1 - P_B) + P_B(1 - P_A))[1 - (P_A(1 - P_B) + P_B(1 - P_A))] \tag{5.33}$$

where P_A and P_B are the probabilities of A and B being '`1`' respectively. An evaluation of this probabilities can be done through STA or by logging them during benchmarks executions. From the relationships above, it is possible to demonstrate that:

$$a_{AND} < a_{XOR}, \quad \forall P_A, P_B \in [0, 1]$$

For this reason, implementing an `XOR`-based tree may increase power consumption considerably, making the trojan more sensitive to side-channel analysis. Increasing the trojan complexity is usually responsible for an increase in power consumption as well, because more gates can switch together in the same amount of time (this applies in case M or D are increased).

[24] As described in Section 2.2.4, layout and manufacturing level trojans may affect capacitances and currents (both I_{peak} and I_{leak}). When merging trojan circuitry with benign one, load capacitances over genuine gates can be affected by gate-level trojans too, but this effect won't be considered.

Chapter 6

Experimental Results and Discussion

6.1 Introduction and general environment

This section is devoted to presenting some of the results which have been obtained from the application of the flow. As introduced in Section 3.2, the simulation environment has been built around a particular instance of a RISC-V core: the SRV32. For this reason, the whole toolchain has been tested exclusively against this core. It is still possible to modify the simulation related parts in order to include a new core, which requires to partially rewrite some of the scripts.
The metrics evaluated as a part of this analysis are the following ones:

- the duration of the whole flow, highlighting which are the possible bottlenecks;
- how variations of the (hyper)parameters may affect duration and quality of results;
- resistance to static analysis techniques, mainly focusing on FANCIX;
- resistance to dynamic analysis techniques, including VeriTrust and UCI;
- area overhead, mainly in terms of number of gates.

Due to the need to measure timings, a specific machine has to be chosen as a reference. For this thesis work, the toolchain has been run on a *Intel(R) Core™ i7-8565U*, a quad core 2-threads per core CPU. The underlying kernel is an Ubuntu 22.04.4 LTS Linux distribution, where the `time` command has been used for time measurements and ad-hoc Python scripts have been written for additional evaluations. The tools involved in the simulations are the ones mentioned in Chapter 3, consiting mainly in Verilator, Yosys and HAL.

6.2 Time measurements

Table 6.1 shows a summary of the timings measured during the execution of each part of the toolchain, including variations over the hyperparameters regulating each step. Some of the most critical timing results have been marked in bold, in order to show where it may be necessary to improve the flow. Before discussing the results, it is necessary to explain the nomenclature and to identify each step in the complete description provided in Chapter 5. A brief overview over each part of the flow is presented here:

- design synthesis and mapping has been done according to the Yosys flow presented in Section 3.5, adapting the script for the SRV32. In order to make technology mapping faster, a very simple liberty file `cmos_cells.lib` has been included in

Table 6.1: Detailed time profiling of the toolchain.

Step	Duration (1)	Duration (2)	Duration (3)	Executions
synthesis	59m 50.411s	n.a.	n.a.	1
compilation	**77m 2.836s** (cov)	7m 20.819s (fst)	n.a.	2
dhrystone	0m 48.057s (cov)	0m 10.372s (fst)	n.a.	5
fft	1m 55.228s (cov)	0m 23.047s (fst)	n.a.	5
pthread	1m 46.183s (cov)	0m 23.815s (fst)	n.a.	5
qsort	0m 51.745s (cov)	0m 12.391s (fst)	n.a.	5
queue	1m 41.985s (cov)	0m 23.334s (fst)	n.a.	5
coverage report	0m 2.744s	n.a.	n.a.	1
select triggers	0m 4.760s	n.a.	n.a.	1
Byron flow	**72m 46s** (10g)	**152m 49s** (20g)	**308m 13s** (30g)	2
			$>$ **12h** (300g)	3
inject trees	0m 0.424s	n.a.	n.a.	1
select payloads	1m 31.674s (2l)	5m 35.179s (3l)	25m 29.134s (4l)	5
w/ PIs	2m 2.2s (2l, no dnf)	**129m 48s** (3l)	$>$ **12h** (4l)	1
insert payload	0m 4.350s (2l)	0m 4.519s (3l)	n.a.	10
payload flow	9m 14.357s (1p)	n.a.	n.a.	5
	$>$ **12h** (100p)	n.a.	n.a.	1

the synthesis process. This *lib* file consists in 6 gates, implementing common logic operators in a single variant[1]. An extension of this liberty file has been implemented to support post payload injection synthesis, where `XOR` gates are needed in order to build the `XOR`-based tree in the most efficient way possible. This new version of the file has been dubbed `cmos_cells_XOR.lib`.

- Verilator compilation has to be done multiple times over the course of the flow, but it is possible to distinguish two major cases: the first is the one where FST files have to be produced after the simulation, the second when coverage reports have to be produced instead. These two cases are identified by the *fst* and *cov* labels in the table.

- *coverage report* is the step in charge of producing the final report starting from the outcomes of the simulations. It hasn't been mentioned earlier since it acts as an intermediate step to feed the `select_and_insert_triggers.py` script.

- *select triggers* consists in the execution of `select_and_insert_triggers.py`, which selects the N triggers from the coverage report (according to the metrics defined in Section 5.4), inserts the circuitry needed by edge-based triggers and places the Verilator directives to limit the simulation scope (to log only the N signals, limiting the FST size).

- the Byron flow has been thoroughly explained in Section 5.5. A major hyperparameter which can be configured is the number of generations (g), which has been set to 10, 20, 30 and 300.

[1]In actual liberty files there may be multiple variants of the same logic function, accounting for low-power/high-performance constraints or driving efficiency. As an example, the `AND2` gate could be present in two flavors: `AND2_X2` and `AND2_X4`. In this case, the former gate has a lower driving strength than the latter.

- *inject trees* involves injecting the actual `AND`/`XOR`-based trees circuits in the netlist. It can be performed through different scripts, depending on if the structure requires trigger time windows and if the design has to be optimized for synthesis.

- *select payload* is performed through the `select_payload.py` script, which is available in two versions: one which doesn't accept PIs at all and the other one which accepts them without any kind of sanitization. According to the final remarks in Section 4.3.1, it could be possible to extend FANCI(X) formulation to consider expressions $p_i^T(FC|_d(p_i))$ containing PIs (introducing additional constraints). Since this hasn't been done in this thesis, a simple unconstrained script has been prepared as a way to approximately estimate performances. As introduced in Section 5.6.1, counting the number of cubes of the final p_i expression is a good way to estimate the complexity of the payload. Doing so requires rewriting p_i in SOP form, which may take a lot of time. For this reason, a *no dnf* variant has been introduced to bypass this check.

- *insert payload* consists in the insertion of the circuit related to a specific payload among the ones selected in the previous step.

- in the end, each payload should be tested to assess if it makes it possible to propagate a wrong value up to the POs. A fitness value is produced to describe how fitting each candidate payload is, depending on the metrics introduced in Section 5.6.3.

As evident from the table, the Byron flow represents the main bottleneck in the whole process, affecting the performances considerably even when run over a small number of generations. Some of the simulations exceed the 12h timeout, so the corresponding duration has been logged in the table as "$>$ 12h". An important metric to be kept into account when evaluating each Byron solution is how it satisfies the constraints imposed at the beginning, both in terms of trigger rarity and selective activations. Table 6.2 shows the quality of results (QoR) for the most meaningful among Byron runs, marking the best solutions in bold. The unsurprising result is that by increasing the number of generations it is possible to positively affect the QoR, both in terms of low activation numbers and coherence with the target (which is the desired number of programs over which the trigger should be activated). To be precise, executions over 30 generations are able to produce triggers that are rare enough, but they still lack the selectivity which is shown by 300-generations runs (which usually hit the target). As anticipated, the fitness function in Section 5.5 could be adapted to impose precise constraints in terms of number of activations too. It would be possible to improve the overall flow by parallelizing it further over a larger number of threads, provided that the number of individuals per generation is increased (typically named λ).
Another step that could potentially be configured is the Verilator compilation with coverage

Table 6.2: Quality of results produced by Byron.

N° gens	Duration	Best occurrences	N° programs	Target
10	79m 56.617s	156	5	3
20	138m 45.172s	62	3	3
30	294m 5.019s	36	3	1
30	**322m 21.975s**	**18**	**3**	**3**
300	$>$ 12h	12	3	3
300	$>$ **12h**	**5**	**1**	**1**
300	$>$ **12h**	**8**	**3**	**3**

metrics on, which takes around 1h. Unfortunately, removing the `--coverage_underscore` option from the Verilator call marginally reduces the compilation duration, even in correspondence of a sensitive reduction of the pool of signals to be logged. As a consequence, changing the parameter N will produce a marginal effect too (it reduces the signals pool even less).
The payload selection is surely another critical points, where some further adjustments should be performed. In particular, both the flow to include the PIs and a new metric for estimating complexity should be added. For what concerns the former, guidelines have already been traced in Section 4.3.1, while the latter requires eluding the `sympy.to_dnf()` call, which is very time consuming for deep payload fanins.
As anticipated, the payload flow could potentially be much more time consuming than the trigger one, due to the need to resimulate every payload to evaluate the effects. Unfortunately, few modifications could be introduced over this flow, except for a possible parallelization which would make it necessary to replicate the whole toolchain. The efficiency of this part of the toolchain is still debatable, mainly because only 2 valid payloads have been identified over a set of 100 candidates during a complete execution. For this reason, a Byron-based approach would be the best solution.

6.3 Resistance against static analysis

As known, static analysis techniques are a widely used tool to spot hardware trojans in an effective way without the need for a simulation. Over the course of this thesis, many algorithms for static analysis have been introduced, ranging from FANCI to ANGEL. In order to assess the quality of the trojans produced, a simplified version of FANCIX has been implemented in `cv_computation.py`. This script computes a D-depth CV for the actual payload signal and multiple random signals from the netlist, allowing the user to compare them to check if the one related to the payload shows a particular pattern which would make it recognizable. The same flow has been applied for both the `AND` and the `XOR`-based trees to verify how much the latter improve resistance against static analysis. As explained in Section 4.3.1, computing the whole truth table for each cut depth d is unfeasible even for low d values, because of the truth table size being $\mathcal{O}(2^{\#FC|_d(s_i)})$. For this reason, only a small part of the truth table is actually considered, sized `TT_SIZE` in the script.
Let us consider P payloads $\mathcal{P} = \{p_0, p_1, ..., p_{P-1}\}$ and S random signals $\mathcal{S} = \{s_0, s_1, ..., s_{S-1}\}$. According to the nomenclature introduced in Chapter 5, the generic fanin cone of signal s_i at depth d is $FC|_d(s_i)$, and the same applies for the fanin cone of payload p_i at depth d $FC|_d(p_i)$. Table 6.3 shows a comparison between the following values:

$$\overline{CV}|_d(\mathcal{P}) \coloneqq \frac{1}{P \cdot \#FC|_d(p_i)} \sum_{i=0}^{P} \sum_{j=0}^{\#FC|_d(p_i)} CV(x_j, p_i) \tag{6.1}$$

$$\overline{CV}|_d(\mathcal{S}) \coloneqq \frac{1}{S \cdot \#FC|_d(s_i)} \sum_{i=0}^{S} \sum_{j=0}^{\#FC|_d(s_i)} CV(y_j, s_i) \tag{6.2}$$

where $x_0, x_1, ..., x_{\#FC|_d(p_i)-1} \in FC|_d(p_i)$ and $y_0, y_1, ..., y_{\#FC|_d(s_i)-1} \in FC|_d(s_i)$. The two operations basically correspond to a double average of all the CVs, averaging first between elements in the fanin cone and then between different payloads. The circuits considered in this case include exclusively `XOR`-based trigger trees. An average value for the size of

Table 6.3: Comparison between average payload CV and average random signal CV, with the size of the corresponding cut.

d	$\overline{CV}\|_d(\mathcal{P})$	$\#\overline{FC}\|_d(\mathcal{P})$	$\overline{CV}\|_d(\mathcal{S})$	$\#\overline{FC}\|_d(\mathcal{S})$
1	0.5003	2	0.6846	2
2	0.2535	3	0.4440	3
3	0.1547	6	0.2768	5
4	0.1529	9	0.1832	8
5	0.1335	13	0.1213	14
6	0.0985	22	0.0920	23
7	0.0810	28	0.0566	37
8	0.0829	32	0.0401	57
9	0.0812	39	0.0303	86
10	0.0645	47	0.0113	103

the fanin cones has been computed as follows:

$$\#\overline{FC}|_d(\mathcal{P}) \coloneqq \frac{1}{P}\sum_{i=0}^{P}\#FC|_d(p_i) \tag{6.3}$$

$$\#\overline{FC}|_d(\mathcal{S}) \coloneqq \frac{1}{S}\sum_{i=0}^{S}\#FC|_d(s_i) \tag{6.4}$$

As expected, an increase of d is responsible for a decrease of the control values, because it becomes more difficult for each signal in $FC|_d(s_i)$ or $FC|_d(p_i)$ to control the value of s_i or p_i respectively. The $\overline{CV}|_d(\mathcal{P})$ values for small d are not significant, because they are related to the payload circuitry which do not include the trigger trees. In correspondence of high values for d, it is evident how the payload-related CVs are similar to the ones related to random signals. This makes the payload hard to spot through static techniques. A close inspection shows that the $\overline{CV}|_d(\mathcal{P})$ values are actually higher than the corresponding $\overline{CV}|_d(\mathcal{S})$. It could be possible to exploit this margin by replacing some `XOR` gates in the trees with `AND` ones, reducing the $\overline{CV}|_d(\mathcal{P})$ values but making it easier to force more selective trigger activations. As a consequence, the number of generations in the Byron flow could be reduced without sacrificing too much the QoR. It is important to notice that including these `AND` gates involves solving a constrained optimization problem too, because an optimal placement must be found without exceeding the existing margin $\overline{CV}|_d(\mathcal{P}) - \overline{CV}|_d(\mathcal{S})$.
In order to assess how effective the `XOR`-based trees are in keeping the control value high, circuits including such trees have been compared with the ones including purely `AND`-based triggers. The outcomes of this comparison have been summarized in two tables:

- Table 6.4, which presents the comparison between the CVs computed starting directly from the outputs of the trees (and then averaged together). Given $\mathcal{T} = \{t'_0, ..., t'_{T-1}\}$ the outputs of the trigger trees, then the values to be compared are $\overline{CV}|_d(\mathcal{T})$. In this case, only the CVs computed for signals that are part of the trees have been shown. Results which are not meaningful have been marked as "n.m.";

- Table 6.4, which shows the results of a proper comparison starting from the payloads. The values to be computed are the same present in Table 6.3.

Table 6.4: Comparison between the $\overline{CV}|_d(\mathcal{T})$ for the AND and XOR-based trees.

d	$\overline{CV}\|_{d,AND}(\mathcal{T})$	$\overline{CV}\|_{d,XOR}(\mathcal{T})$
1	1.0	1.0
2	0.508	1.0
3	0.503	1.0
4	0.125	1.0
5	0.128	0.885
6	0.007	n.m.
7	0.006	n.m.

Table 6.5: Comparison between the $\overline{CV}|_d(\mathcal{P})$ for the AND and XOR-based trees.

d	$\overline{CV}\|_{d,AND}(\mathcal{P})$	$\overline{CV}\|_{d,XOR}(\mathcal{P})$
1	0.490	0.515
2	0.223	0.240
3	0.160	0.148
4	0.144	0.140
5	0.122	0.131
6	0.089	0.109
7	0.073	0.082
8	0.079	0.081
9	0.055	0.086
10	0.048	0.074
11	0.033	0.058
12	0.029	0.052
13	0.022	0.045
14	0.016	0.035
15	0.012	0.024

As shown in Table 6.4, CV results for the XOR trees are all ones, because each input x_i of an XOR2 gate is such that $CV(x_i, f) = 0$. The CVs for the AND-based trees should follow this formula:

$$\overline{CV}|_{d,AND}(\mathcal{P}) = \begin{cases} 2^{1-2^{\frac{d}{2}}} & \text{if} \quad d = 2i, \quad i \in \mathbb{N} \\ \overline{CV}|_{d-1,AND}(\mathcal{P}) & \text{otherwise} \end{cases}. \tag{6.5}$$

Since the liberty file doesn't include a AND2 gate, the AND logic function must be implented as a series of NAND and NOT. When the NOT gate is traversed then the CV won't change, because a NOT has maximum controllability as the XOR. The resulting pattern is shown in the table, where CVs change when the cut includes a new NAND gate while it remains the same when a NOT is included instead. The closed formula in Equation 6.5 follows from the fact that every extension of the cut duplicates the number of inputs (because of the tree structure) and each new input reduces by half each control value [2]. Both tables show how the XOR-based solution ensures better protection against static analysis, doubling the average of the control values in correspondence of high depths. If the analysis is extended to include also the occurrences of very low CVs for each cut, it is possible to show how these values are widespread in the AND-based solution and very rare in the XOR one, making the latter much more resistant against techniques which involve inspecting the single control values.

To conclude, implementing a trigger tree that is purely XOR-based ensures better protection against CV-based static analysis techniques like FANCI, FANCIX and ANGEL. In order to improve resistance against FASTrust, it would be possible to remove all the intermediate FFs in the tree and to combine trees outputs together, as suggested in Section 5.5.4. However, this newfound technique adds a non negligible degree of vulnerability against dynamic/simulation based analysis. The targeted insertion of AND gates would compensate for this effect, excluding some regions of the truth table from the on-set.

[2] It can be verified that given x_i the generic input of the old cut $\mathcal{C}$ and x_i' the input of the new cut $\mathcal{C}'$, then $CV(x_i', f) = (\frac{1}{2})^{\#C' - \#C} CV(x_i, f)$ where $\#C'$ is the number of inputs of the new cut, $\#C$ is the old one. The reason behind this behavior is that each input to be added is responsible for the TT size to be doubled, but the entry which gives $f = 1$ is still one (since the function is a AND).

6.3.1 Resistance to dynamic analysis

As known, dynamic techniques involve simulating the design to find out rare wires through coverage analysis. Over the course of the thesis, different strategies have been presented to detect potentially unuseful circuit through this kind of analysis. The one that will be analyzed in the current section is VeriTrust, which aim to identify uncovered wires. It is possible to verify invulnerability to UCI as well, according to the flow that follows. Assuming that the original circuit is not marked as suspicious by UCI, it is enough to demonstrate that:

- each tree output t'_i is sensitized to `1` at least once during execution;
- each `XOR` input is sensitized to both 0 and 1, including the intermediate wires in the tree that are inputs to the `XOR` gates in the following levels.

This analysis has been performed by simulating the post-injection synthesized design to collect coverage values, according to the same steps listed in Section 5.4. Some of the results collected are listed in the *coverage* subfolder of the *srv32* repository, where outcomes from different runs are available. Since each toolchain run potentially exploits different trigger conditions, there is no way to produce a comprehensive metrics to sum up all the results. The only element that should be taken into account is the absence of zero coverage values, which would be spotted would result in trojan circuitry being marked as suspicious by both VeriTrust and UCI. From the aforementioned reports it is possible to verify that no value in the payload circuit shows a zero coverage, making this part of the trojan invisible to the dynamic techniques considered in this thesis. The trigger tree shows a zero coverage for `trigger15_curr` instead, but it would be possible to replace this trigger with another one to solve the problem. This behavior may arise because of imprecise tracking of the coverage during the initial simulations, where an overestimation of the coverage amount may be done. As a proof of this behavior, the trigger values to be affected by this effect the most are the ones which have been assigned a low index, since they are the ones with low coverage values just above the lower threshold[3] (they are sorted in order of increasing coverage values).
As a side note, the previous analysis satisfies the second condition for invulnerability against UCI as well. The reason behind this is that the coverage values refer to toggles, so they already imply transitions $0 \rightarrow 1$ or $1 \rightarrow 0$.

6.3.2 Area overhead

The estimation of the are overhead has been done on solutions with payloads spread over two or three sequential levels. From now on, the area overhead will be quantified in terms of number of gates without considering the actual area occupation of the gate. This guarantees the results to be independent from the specific technology library used, but it produces only a rough estimation of the actual area values. However, a physical layout would be needed in order to obtain a comprehensive estimation of the actual occupation, so an approximation would be needed in any case.
The average results obtained from the analysis are reported in Table 6.6, where various configurations are shown:

- *original*: the original DUA with no circuitry inserted;
- *+t, no w*: the DUA plus the trigger circuitry, without including the FSMs to implement the active windows;

[3] Refer to Section 5.4 for further details.

Table 6.6: Average area overhead for different configurations.

D	original	+t, no w	+t,w	+t+p, no w	+t+p,w	tot,w	tot, no w
2	74975	75336	76146	75355	76165	**1190**	**380**
3	74975	75336	76146	75366	76176	**1201**	**391**

- *+t,w*: as before, but including the FSMs;
- *+t+p, no w*: as *+t, no w* but including the payload circuit as well;
- *+t+p,w*: as *+t+p, no w* but including the FSMs.

The total gate counts are reported in the last two columns. It is clearly shown how the FSMs account for the wide majority of the trojan area, resulting in a decrease by almost 200% in the gate count when they are removed. However, a solution without windows could potentially be much weaker due to the reduced sequentiality. A possible way to compensate for this effect could be to modify the structure of the tree, keeping the FFs but rearranging them in different ways. This solution has been suggested in Section 5.5.5, where it has been highlighted how it could compromise the invulnerability against FASTrust. Since the introduction of the `XOR` makes the trigger activation much more frequent, there is currently no way to make trees whose conditions are both rare enough and resistent against FASTrust (except for some particular benchmarks).
It is important to notice that the area occupation of the trigger trees remains constant, while the one associated with the payload structure increases in correspondence of an increase in D [4]. However, the difference between payload areas for $D = 2$ and $D = 3$ is negligible, making it necessary to increase D considerably to make the payload overhead sensibly different from the base case ($D = 2$). For this reason, area optimization should focus exclusively on the trigger structure.

[4]The nomenclature used is the one in Section 5.6.5.

Chapter 7

Conclusions

7.1 Final comments

Over the course of the thesis, a thorough discussion on how trojan detection tools could benefit from the introduction of automated injection environments has been presented [9]. The present toolchain aims to do that, providing a configurable, scalable solution for producing meaningful trojans able to elude the detection algorithms introduced in Chapter 4. The approach followed is a novel one, based on imposing selective activation on specific code benchmarks. This selectivity is ensured by generating the trigger condition through a genetic programming tool, which produces increasingly fitting solutions over time.
Properties of the malicious circuits produced through this flow have been demonstrated through an analysis targeting stealthiness, in terms of area/power overhead, frequency of activation and resistance to static and dynamic techniques. In particular, trojans have been tested against FANCIX [16], which is a cross sequential stage static algorithm. Thanks to a `XOR`-based trigger tree design, the designs show very high control values (almost double the normal ones, as in Table 6.3 and Table 6.5) combined with a reduced activation frequency (for tens of generations: $N_{act} < 10$) and low visibility on the POs, making them invisible on the benchmarks considered.
These trojans could integrate the designs already present on famous HUBs, like TrustHUB, providing meaningful samples for training AI-based detection models. Besides, they could help security experts in developing new strategies to detect `XOR`-based trojans which intentionally keep the CVs low. Due to the completely automated injection flow, which consists in a few Python, Bash and Make scripts, it would be possible to generate a huge number of different benchmarks. The variety is guaranteed by the payload structure, which depends on the netlist topology and the highly configurable hyperparameters introduced in Chapter 5 (the sequential depth D, the cubes to be selected, the number of trees etc.).
The last important note is related to the open source nature, which makes it possible for a wide audience to replicate the results on their own Linux machines. However, the toolchain shows a considerable amount of dependencies following from the high complexity of the tools involved. Still, the free tools make the product competitive against proprietary solutions. Besides, the features used as part of the thesis are in common with more popular alternatives, facilitating the learning process for beginners and hobbyists.

7.1.1 Future work

The version of the toolchain proposed as part of this thesis is a simple one, which still accounts for all the steps needed when designing a trojan. It would be possible to improve

it considerably by either refining the injection strategies or by implementing new ones, producing stealthier and more varied circuits. Here some possible improvements are presented, in order of decreasing priority:

- the toolchain should be extended to the physical layout, relying on open place and route tools from OpenLane [17]. In this way it would be possible to consider low level trojan features like interconnection length and area distribution, making it possible to refine the power and time overhead estimations through measurements performed directly on the final circuit.

- new injection techniques could be included in the toolchain, ensuring vulnerability against additional techniques from the state of the art. In particular, the strategies implemented in deTest [5] to bypass observability and controllability based detection could be replicated in the current toolchain as well, powering up the elusivity introduced through the deTrust [3] suite.

- improving the current purely `XOR`-based tree by including `AND` gates is advised, in order to minimize the risk of unwanted activations on code not included in the benchmarks. As shown in Section 5.5.4, `AND` gates can selectively eliminate minterms or implicants from the on-set. This results in a stealthier design from the point of view of dynamic analysis, but it involves an undesired decrease in the CVs as well. For this reason such modifications should be done with caution, designing an algorithm capable of performing on-set resizing under strict constraints over the control values.

- the flow should be tested over multiple processors, possibly RISC-V ones to preserve the compatibility with tools like RISCV-DV. In particular, it is advised to repeat the experiments with more complex cores to assess how the area overhead may vary.

- the structure of the toolchain should be refined, to increase the amount of configurable parameters and to make it simpler to use.

- benchmarking against proprietary environments may be performed, evaluating the QoR both in terms of trojan stealthiness and time needed to produce working solutions.

Bibliography

[1] "https://miro.medium.com/v2/resize:fit:786/format:webp/1*QohZrdOTm96g2trVYysm9w.png".

[2] Cynthia Sturton, Matthew Hicks, David Wagner, and Samuel T King. Defeating uci: Building stealthy and malicious hardware. In *2011 IEEE symposium on security and privacy*, pages 64–77. IEEE, 2011.

[3] Jie Zhang, Feng Yuan, and Qiang Xu. Detrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware trojans. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 153–166, 2014.

[4] Song Yao, Xiaoming Chen, Jie Zhang, Qiaoyi Liu, Jia Wang, Qiang Xu, Yu Wang, and Huazhong Yang. Fastrust: Feature analysis for third-party ip trust verification. In *2015 IEEE International Test Conference (ITC)*, pages 1–10. IEEE, 2015.

[5] Ning Zhang, Zhiqiang Lv, Yanlin Zhang, Haiyang Li, Yixin Zhang, and Weiqing Huang. Novel design of hardware trojan: A generic approach for defeating testability based detection. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 162–173. IEEE, 2020.

[6] Jarrod A Roy, Farinaz Koushanfar, and Igor L Markov. Circuit cad tools as a security threat. In *2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 65–66. IEEE, 2008.

[7] Rajat Subhra Chakraborty, Seetharam Narasimhan, and Swarup Bhunia. Hardware trojan: Threats and emerging solutions. In *2009 IEEE International high level design validation and test workshop*, pages 166–171. IEEE, 2009.

[8] Hassan Salmani and Mohammed Tehranipoor. Analyzing circuit vulnerability to hardware trojan insertion at the behavioral level. In *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, pages 190–195. IEEE, 2013.

[9] Alexander Hepp, Tiago Perez, Samuel Pagliarini, and Georg Sigl. A pragmatic methodology for blind hardware trojan insertion in finalized layouts. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.

[10] Nishant Gupta, Mohil Sandip Desai, Mark Wijtvliet, Shubham Rai, and Akash Kumar. Delta: Designing a stealthy trigger mechanism for analog hardware trojans and its detection analysis. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 787–792, 2022.

[11] Kaiyuan Yang, Matthew Hicks, Qing Dong, Todd Austin, and Dennis Sylvester. A2: Analog malicious hardware. In *2016 IEEE symposium on security and privacy (SP)*, pages 18–37. IEEE, 2016.

[12] Julien Francq. Hardware trojans: Taxonomy and detection methods. http://www.cs.ucr.edu/ nael/ee260/lectures/hardware-trojans.pdf, 2014.

[13] Miron Abramovici and Paul Bradley. Integrated circuit security: new threats and solutions. In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, pages 1–3, 2009.

[14] Jie Zhang, Feng Yuan, Lingxiao Wei, Zelong Sun, and Qiang Xu. Veritrust: Verification for hardware trust. In *Proceedings of the 50th Annual Design Automation Conference*, pages 1–8, 2013.

[15] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. Fanci: identification of stealthy malicious logic using boolean functional analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 697–708, 2013.

[16] Hassan 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(2):338–350, 2016.

[17] "`https://github.com/efabless/openlane2`".

[18] Marc Fyrbiak, Sebastian Wallat, Pawel Swierczynski, Max Hoffmann, Sebastian Hoppach, Matthias Wilhelm, Tobias Weidlich, Russell Tessier, and Christof Paar. Hal—the missing piece of the puzzle for hardware reverse engineering, trojan detection and insertion. *IEEE Transactions on Dependable and Secure Computing*, 16(3):498–510, 2018.

[19] Matthew Hicks, Murph Finnicum, Samuel T King, Milo MK Martin, and Jonathan M Smith. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *2010 IEEE symposium on security and privacy*, pages 159–172. IEEE, 2010.

[20] Jeyavijayan Rajendran, Efstratios Gavas, Jorge Jimenez, Vikram Padman, and Ramesh Karri. Towards a comprehensive and systematic classification of hardware trojans. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 1871–1874. IEEE, 2010.

[21] Samuel T King, Joseph A Tucek, Anthony Cozzie, Chris Grier, Weihang Jiang, and Yuanyuan Zhou. Designing and implementing malicious hardware. *Leet*, 8:1–8, 2008.

[22] Jie Zhang and Qiang Xu. On hardware trojan design and implementation at register-transfer level. In *2013 IEEE international symposium on hardware-oriented security and trust (HOST)*, pages 107–112. IEEE, 2013.

[23] Jie-Hong R Jiang, Victor N Kravets, and Nian-Ze Lee. Engineering change order for combinational and sequential design rectification. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 726–731. IEEE, 2020.

[24] Tiago Perez, Malik Imran, Pablo Vaz, and Samuel Pagliarini. Side-channel trojan insertion-a practical foundry-side attack via eco. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2021.

[25] Z Chen et al. Hardware trojan designs on basys fpga board (virginia tech). *CSAW Embedded System Challenge*, 2008.

[26] Ayush Jain, Ziqi Zhou, and Ujjwal Guin. Survey of recent developments for hardware trojan detection. In *2021 ieee international symposium on circuits and systems (iscas)*, pages 1–5. IEEE, 2021.

[27] Nidish Vashistha, M Tanjidur Rahman, Haoting Shen, Damon L Woodard, Navid Asadizanjani, and Mark Tehranipoor. Detecting hardware trojans inserted by untrusted foundry using physical inspection and advanced image processing. *Journal of Hardware and Systems Security*, 2:333–344, 2018.

[28] Yuriy Shiyanovskii, F Wolff, C Papachristou, D Weyer, and W Clay. Exploiting semiconductor properties for hardware trojans. *arXiv preprint arXiv:0906.3834*, 2009.

[29] Tiago D Perez and Samuel Pagliarini. Hardware trojan insertion in finalized layouts: From methodology to a silicon demonstration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(7):2094–2107, 2022.

[30] Yier Jin, Nathan Kupp, and Yiorgos Makris. Experiences in hardware trojan design and implementation. In *2009 IEEE international workshop on hardware-oriented security and trust*, pages 50–57. IEEE, 2009.

[31] Christian Krieg, Michael Rathmair, and Florian Schupfer. A process for the detection of design-level hardware trojans using verification methods. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS)*, pages 729–734. IEEE, 2014.

[32] Kento Hasegawa, Masao Yanagisawa, and Nozomu Togawa. A hardware-trojan classification method utilizing boundary net structures. In *2018 IEEE international conference on consumer electronics (ICCE)*, pages 1–4. IEEE, 2018.

[33] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 42–47. IEEE, 1993.

[34] Jonathan Cruz, Farimah Farahmandi, Alif Ahmed, and Prabhat Mishra. Hardware trojan detection using atpg and model checking. In *2018 31st international conference on VLSI design and 2018 17th international conference on embedded systems (VLSID)*, pages 91–96. IEEE, 2018.

[35] Franck Courbon, Philippe Loubet-Moundi, Jacques JA Fournier, and Assia Tria. Semba: A sem based acquisition technique for fast invasive hardware trojan detection. In *2015 European Conference on Circuit Theory and Design (ECCTD)*, pages 1–4. IEEE, 2015.

[36] Nidish Vashistha, Hangwei Lu, Qihang Shi, M Tanjidur Rahman, Haoting Shen, Damon L Woodard, Navid Asadizanjani, and Mark Tehranipoor. Trojan scanner: Detecting hardware trojans with rapid sem imaging combined with image processing

and machine learning. In *International Symposium for Testing and Failure Analysis*, volume 81009, pages 256–265. ASM International, 2018.

[37] Qihang Shi, Nidish Vashistha, Hangwei Lu, Haoting Shen, Bahar Tehranipoor, Damon L Woodard, and Navid Asadizanjani. Golden gates: A new hybrid approach for rapid hardware trojan detection using testing and imaging. In *2019 IEEE international symposium on hardware oriented security and trust (HOST)*, pages 61–71. IEEE, 2019.

[38] Nitin Varshney, Haoting Shen, Olivia Paradis, and Navid Asadizanjani. He-ion beam imaging for accurate hardware trojan detection. *Microscopy and Microanalysis*, 26(S2):188–190, 2020.

[39] Ted Huffmire, Jonathan Valamehr, Timothy Sherwood, Ryan Kastner, Timothy Levin, Thuy D Nguyen, and Cynthia Irvine. Trustworthy system security through 3-d integrated hardware. In *2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 91–92. IEEE, 2008.

[40] "https://github.com/kuopinghsu/srv32".

[41] "https://github.com/kuopinghsu/FreeRTOS-RISCV".

[42] "https://github.com/YosysHQ/yosys".

[43] "https://github.com/chipsalliance/riscv-dv".

[44] "https://github.com/cad-polito-it/byron".

[45] "https://cad-polito-it.github.io/byron/history".

[46] "https://www.corewars.org/".

[47] "https://github.com/emsec/hal".

[48] "https://github.com/YosysHQ/sby".

[49] "https://yosyshq.readthedocs.io/projects/yosys/en/latest/cmd/flatten.html".

[50] "https://github.com/ghdl/ghdl".

[51] "https://github.com/ghdl/ghdl-yosys-plugin".

[52] "https://itsembedded.com/dhd/verilator_1".

[53] "https://verilator.org/guide/latest/connecting.html".

[54] "https://github.com/mschlaegl/pylibfst".

[55] "https://github.com/umarcor/vcd_parsealyze".

[56] "https://pypi.org/project/python-constraint".

[57] Seyed Mohammad Sebt, Ahmad Patooghy, Hakem Beitollahi, and Michel Kinsy. Circuit enclaves susceptible to hardware trojans insertion at gate-level designs. *IET Computers & Digital Techniques*, 12(6):251–257, 2018.

[58] Syed Kamran Haider, Chenglu Jin, Masab Ahmad, Devu Manikantan Shila, Omer Khan, and Marten van Dijk. Advancing the state-of-the-art in hardware trojans detection. *IEEE Transactions on Dependable and Secure Computing*, 16(1):18–32, 2017.

[59] Xiaoming Chen, Qiaoyi Liu, Song Yao, Jia Wang, Qiang Xu, Yu Wang, Yongpan Liu, and Huazhong Yang. Hardware trojan detection in third-party digital intellectual property cores by multilevel feature analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(7):1370–1383, 2017.

[60] Lawrence H Goldstein and Evelyn L Thigpen. Scoap: Sandia controllability/observability analysis program. In *Proceedings of the 17th Design Automation Conference*, pages 190–196, 1980.

[61] Lawrence Goldstein. Controllability/observability analysis of digital circuits. *IEEE Transactions on Circuits and Systems*, 26(9):685–693, 1979.

[62] "`https://github.com/gtkwave/gtkwave`".