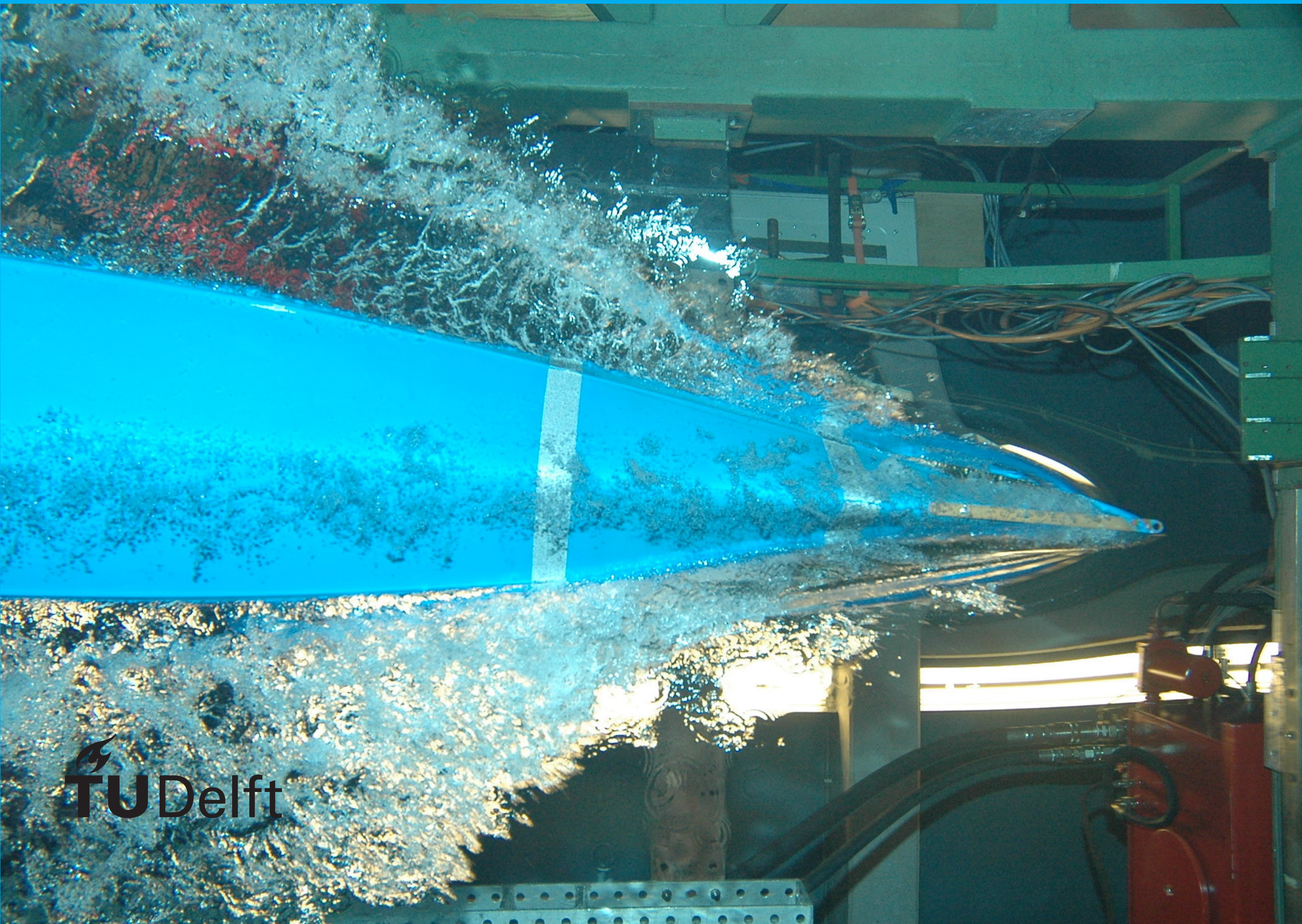
Netlist Level Based Fault Injection Simulation

Davide Belloli



Netlist Level Based Fault Injection Simulation

by

Davide Belloli

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday November 11, 2021 at 1:00 PM

Student number:	5131138	
Project duration:	November 1, 2020 – July 1, 2021	
Thesis committee:	Dr. Ir. S. Wong,	TU Delft, supervisor
	Dr. Ir. M. Taouil,	TU Delft
	Ir. D. Vermoen,	Riscure BV
	Dr. Ir. J. Dauwels	TU Delft
Thesis number:	Q&CE-CE-MS-2021-11	

TU Delft

Abstract

The issue of securing microchip designs against hardware attacks has grown in magnitude as more and more embedded systems are deployed in hostile environments, where security measures have to be taken to prevent attackers from accessing unwanted information.

The first step in solving this problem is gaining awareness of the security vulnerabilities in a design, which can be done through a fault injection campaign. Current solutions tackle the issue by either requiring silicon manufacturing for every prototype, which is expensive, or simulating faults using a model of the device under test and of the possible faults being injected, which takes a considerable amount of time. Some hybrid solutions have been developed to improve this aspect (namely, by using FPGAs to implement the device and injecting faults on it instead), but they still require some form of specialized hardware to operate.

Moreover, a gap still remains between the results from these tools and the work necessary to update the design and mitigate the vulnerabiliies found. Results usually reference cells in the netlist, while the development is mostly done in a high-level hardware descriptive language.

This thesis proposes two improvements to the currently existing workflow: first, it explores the effectiveness of equivalence checking in tracing individual gates in the netlist representation back to the RTL lines that generated them, and second, it builds on the simulation-based fault injection approach by introducing a formal framework to prove the presence or absence of successful faults.

The results show that equivalence checking can significantly increase the number of cells recognized as results of individual RTL lines of code, enabling a designer to better pinpoint which components should be hardened against fault injection attacks. In terms of fault identification, the framework described can reduce the number of faults to be simulated down to 2% of the number necessary to exhaustively check a design for possible vulnerabilities, greatly speeding up existing simulation-based approaches.

Acknowledgements

I would like to thank all of the following people, in no particular order, for helping me during the development of this thesis. Without their support this would not have been possible, especially given the trying times we all had to go through in the past year.

- My daily supervisors Dr. Ir. Mottaqiallah Taouil and Cezar Weidig Reinbrecht, from TU Delft, for their guidance both when developing each topic and when writing the thesis.
- My supervisors Dennis Vermoen and Alexandru Geana from Riscure BV, for their precious insights and experience. Their suggestions and feedback helped me get more acquainted with the field, and their open-mindedness allowed me to explore different directions from which to tackle each problem.
- My parents Dario and Roberta, for unconditionally supporting my decisions, and helping me in my studies especially during the pandemic.

Contents

1 Introduction

This chapter will provide a brief overview of the topic addressed in the thesis: the main driver that motivated the exploration of hardware fault injection attacks, what the current state-of-the-art is with regards to simulation of these faults with or without manufactured hardware, and the results of the exploration in terms of new contributions to the research on the topic. Finally, each chapter will be briefly summarized, to facilitate navigation of the work presented.

1.1. Motivation

As the world becomes more and more interconnected, the necessity for secure means of communication and storage of information continuously increase. In order to keep this information secure, software developers employ encryption schemes to allow only authorized parties to access critical information, write and implement authentication algorithms to verify their identity, and fuzz their applications to remove as many vulnerabilities as possible.

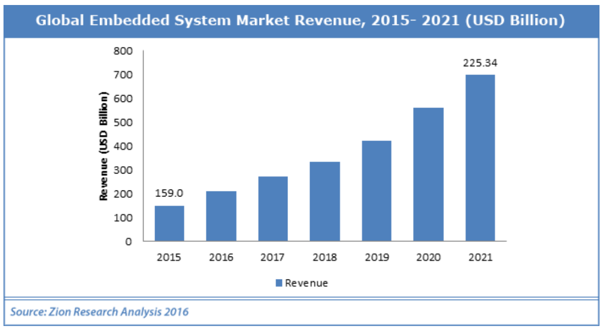


Figure 1.1: Global Embedded Systems total market revenue projection [27]

Almost always, however, they assume that the hardware they run their code on is running as intended, that the algorithm they wrote is faithfully executed by the processor or hardware accelerator of their choice. This assumption is valid most of the time, but if an attacker is motivated enough, they can invalidate it by means of hardware fault injection.

There are multiple ways to cause hardware to misbehave: injecting disturbances into its clock lines, briefly reducing its supply voltage below the rated limits, all the way to more sophisticated attacks such as targeting a specific area of a chip with magnetic or laser interference. These all break the assumption that the code being executed is the one the programmer intended, and can possibly result in dangerous security vulnerabilities.

As a result, hardware designers also had to adapt, in a joined effort to mitigate these faults and ensure that they are eliminated, or at the very least detectable at the software level.

A snapshot of the industry from 2019 by the EE Times and Embedded [13] showed that 1 in 5 products still had no mitigation against hardware attacks, while about 1 in 4 only employed software countermeasures, which are easier to bypass compared to hardware solutions (Figure 1.2).

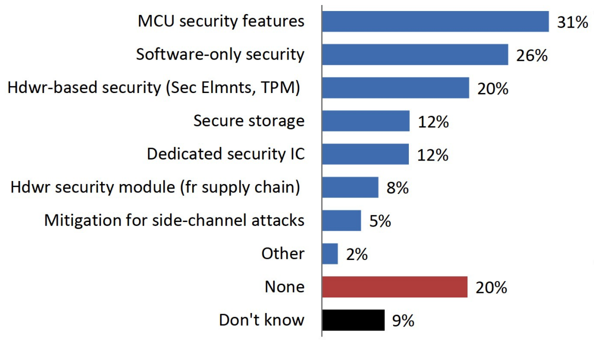


Figure 1.2: Types of fault injection mitigation implemented in the embedded systems market [13]

The tools currently available to find hardware vulnerabilities either require the device under test to be manufactured, which is extremely expensive, or are simulation-based, which is slow and does not provide results that can be re-utilized across runs. This thesis tackles this problem by introducing static analysis to the workflow, resulting in a more efficient vulnerability discovery process with a higher coverage compared to fuzzing techniques.

1.2. State-of-the-art

Hardware fault injection is hardly a new concept: in 1999, an article published on IEEE spectrum states “Dedicated hardware tools are available to flip bits on the instant at the pins of a chip, vary the power supply, or even bomb the system/chips with heavy ions”, indicating that the topic was well known at the time [17].

For this reason, numerous tools have been developed to aid hardware designers in identifying potential faults. They can be roughly categorized into three classes: hardware-based, simulation-based, and emulation-based tools [35][48].

Hardware-based tools require the actual hardware to be manufactured. Once the device is produced, it is instrumented to inject faults, and a fault campaign is executed against it. Tools that fall into this category include FIST [29], which uses heavy ions to irradiate a chip and introduce single or multiple bit-flips, and Messaline [12], which injects faults on each pin of an integrated circuit. These tools provide the most accurate results, with the only unaccounted variable being manufacturing tolerances between chips, but manufacturing costs for each prototype is high, and evaluating the effectiveness of hardware mitigations requires production of each prototype, so using this technique alone can be prohibitively expensive.

Simulation-based tools use a model of the target system, and simulate it applying different faults at different times, modifying the model in order to inject faults at runtime. This relies on the availability of models of the hardware faults that should be injected too. An example of simulation-based tools is MEFISTO [33], which allows a designer to both add saboteurs to a VHDL model and mutate existing components in it by altering their characteristics (e.g. a NAND gate replaced by a NOR gate), manually or automatically. A similar approach of augmenting a design with fault models is followed by VERIFY [46]. Rather than augmenting the model by adding components to it, VERIFY augments hardware components by adding fault injection signals, specifying their rate of occurrence. Compared to hardware-based approaches, the advantages of simulation-based approaches are that there is no associated manufacturing cost, the turnover time between discovering a fault and observing the effects of hardening against that fault is shorter, and they require no specialized equipment. The downsides are that the quality of the results depends on the quality of the fault models and of the simulation, and that the time necessary to complete a fault injection campaign is higher due to the slow nature of simulating a complex model.

Emulation-based tools use a prototype of the system, typically on an FPGA platform, and perform fault injection on it rather than on an ad-hoc manufactured piece of hardware. They address the long execution times of the simulation-based approach by using FPGAs to accelerate circuit simulation, but since the FPGA is not an accurate representation of the final manufactured hardware, the results are not as accurate as a hardware-based approach. Moreover, they still rely on correct modeling of each fault, since the effects are emulated by extra circuitry, and are not generated by using any of the aforementioned hardware faulting techniques an attacker would utilize. One such tool is FITO, which instruments the VHDL design with extra gates for fault injection, then synthesizes and writes the resulting bitstream on an FPGA for real-time fault analysis [14]. Using an FPGA to reduce simulation times has the same advantages and disadvantages as pure simulation-based approaches, trading short execution times for a marginally higher cost. The issue of correctly modeling a fault is still present.

1.3. Contributions

- **Python library to parse, modify, and write YoSYS intermediate language files**: YoSYS is an open source synthesis tool, used during the project to alleviate some of the issues that come with parsing a complex hardware description language such as Verilog. Due to the numerous advantages of the simpler syntax found in its intermediate language representation, developing a library to parse its syntax in order to perform analysis and instrumentation of a given netlist was critical. This resulted in the creation of a Python library able to read and write intermediate language dumps, with the ability to express each cell's output as a boolean function of that cell's inputs.

- **Automated annotation recovery to identify RTL components in a netlist**: injecting faults at the netlist level produces results truer to how the hardware would behave once implemented, but it's hard to perform mitigations at this level of abstraction. For this reason, an algorithm was developed to trace netlist cells back to the lines of code that generated them.

- **Development of a framework to analyze netlist-level faults**: as previously mentioned, netlists are closer to the hardware representation of a component compared to the high-level description components are usually implemented in. For this reason, a workflow was implemented to perform static fault propagation analysis at the hardware level, and then dynamically check which faults would result in unwanted behaviour given an execution trace. This differs from previously attempted approaches due to its formal nature, which has the potential to achieve 100% fault coverage.

- **Successful exploitation of a SecureBoot implementation**: as a proof-of-concept, the framework thus developed was used to perform a fault injection attack on a simple SecureBoot implementation, with the goal of causing it to execute code with an invalid signature.

1.4. Thesis Organization

The rest of the report is divided into 6 chapters.

Chapter 2 will discuss the major fault injection vulnerabilities, how fault injection is performed in the real world, and types of fault that one might expect when observing the effects on the functionality of the device under test.

Chapter 3 will provide an overview of formal verification, a process at the core of the algorithms used to map netlist cells to lines of code in the high-level hardware description language.

Chapter 4 will motivate the important role of optimization when operating on hardware designs. It will first introduce the common notation used when describing the computational load of an algorithm, and then quantify the complexity of the problems that had to be solved by the framework.

Chapter 5 will give a high-level overview of the framework that was created, describing the function of each element as well as the reason why each element was introduced. It will also briefly describe the target chosen for validating the tools.

Chapter 6 will dive more in detail into the netlist-to-RTL backtracking algorithm, presenting the theory behind the algorithm, the expected outputs of the tool, and the results obtained when running the tool against the proof-of-concept target.

Chapter 7 will focus on the fault injection part of the work. It will explain why fault injection at the netlist level was considered valuable, how the task of simulating faults at this level was implemented, and how it was carried out on the proof-of-concept target.

Chapter 8 will summarize the results obtained during the research, discuss the limitations of the developed tools, and indicate future lines of work that could branch from the described toolchain.

2
Hardware Attacks and Fault Injection

Stressing a device outside of its regular operating conditions can result in behaviour that diverges from the expected. These effects can be exploited by an attacker to cause a controlled malfunction in a chip, possibly resulting in security vulnerabilities, such as bypassing of security features, exposure of sensitive information, or writing to protected memory regions. This chapter will introduce the most common types of hardware attacks currently in use. First, the most commonly used classification scheme will be described, which is used to better understand the properties of a given attack. Then, the most commonly used fault injection techniques will be enumerated, specifying which classes they belong to and giving a short description of their operating principles and effects. Some examples of practical fault injection attacks will then be given, so that the reader can better understand how these techniques are used in the real world, as well as the severity of the exploits that can be triggered using fault injection. Finally, the higher-level outcome of hardware faults on the circuitry affected, also known as their profile, will be described.

2.1. Attack Classification

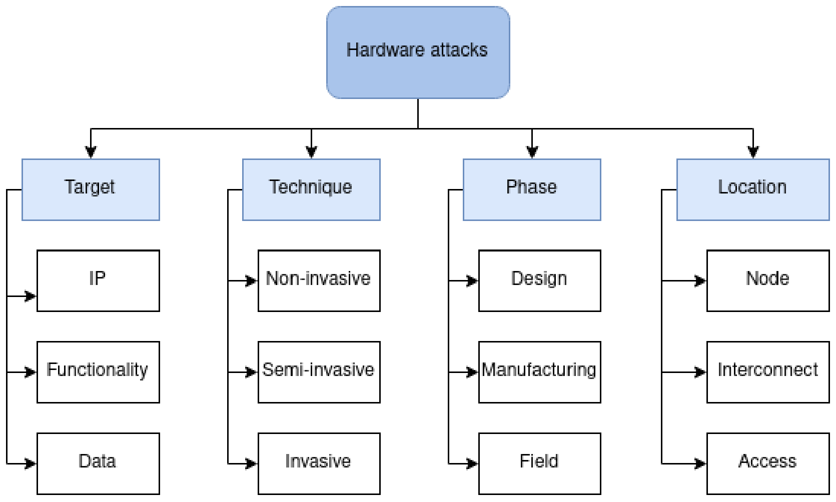


Figure 2.1: Hardware attacks classification scheme

In order to analyze the different effects that fault injection can have on a device, it is useful to separate hardware attacks into classes based on the target of the attack, the technique used, the part of the design that is being affected, and the development phase in which they are performed (Figure 2.1).

The possible target classifications are the following:

- **IP**: attacks that attempt to clone or modify the intellectual property in a chip without authorization from the original manufacturers. One such attack would be to obtain a chip, remove the protective layers around the die, and use an electron microscope to identify each component and reverse-engineer the implemented logic.

- **Functionality**: attacks that try to change the functions being implemented by the chip. Examples include adding extra functionality to a design, known as a hardware trojan, before it is manufactured, so that the product will misbehave in controlled ways after it is deployed.
- **Data**: attacks that try to read or modify data used by the chip during computation. An attack falling into this category could be imaging of a ROM storing private keys on a die: one-time programmable (OTP) ROMs usually store data by burning links between interconnects, and by visually inspecting which links are intact and which are not, the private data contained could be read.

In terms of technique, the following classification is made:

- **Non-invasive**: attacks that do not require modifying the hardware, such as exposing the chip die, in order to be performed. Non-invasive attacks typically cause the chip to misbehave by exceeding its physical ratings, such as operating temperature, voltage, or clock speed.
- **Semi-invasive**: attacks that require exposure of the die in order to be performed. The aforementioned data attack performed by imaging a ROM is one such attack. It would be impossible without removing at least some of the protective layers surrounding the die, but does not require modification of the die itself to be carried out.
- **Invasive**: attacks that require exposure and modification of the die by, for example, cutting interconnects or adding new connections. An example would be to change the contents of an OTP ROM by cutting extra interconnects, resulting in a different private key being used when encrypting and decrypting data.

As for the part of the design affected, there are three possible classes:

- **Node**: attacks that target cells that implement a specific functionality. Exceeding the physical ratings of a product to force it to perform out-of-spec operations is one such attack, as is inserting a trojan at design-time to read stored data that would have otherwise been inaccessible.
- **Interconnect**: attacks that target the connection between cells. An attack that would be classified as targeting the interconnect would be using a very thin probe to read the voltage values of an internal bus, to sniff sensitive data being sent through it.
- **Access**: attacks that target the input and output ports of a design. For example, one such attack could be to use a hardware trojan to mask the existence of undisclosed functionality in a chip during testing, by mimicking the nominal output when it detects that a pre-deployment test is being run.

Finally, in terms of phase, the following categories are used:

- **Design**: attacks that are performed during the design phase of a product. Hardware trojans added into third-party library components are an example: if a company decides to outsource the design of a cryptographic accelerator, the external party involved could add backdoors into this component to leak unencrypted data after deployment.
- **Manufacturing**: attacks that are performed while the chip is manufactured. A different kind of hardware trojan could be added here: instead of inserting the extra functionality in a high-level language that is then synthesized into logic gates and wires, it can be directly added by the manufacturing plant, even if all the parties involved in the design step are not compromised.
- **Field**: attacks that are performed after the chip is manufactured and sold. The previously mentioned attacks that try to cause specific out-of-spec behaviour (by violating temperature, voltage or clock speed constraints) all fall under this category.

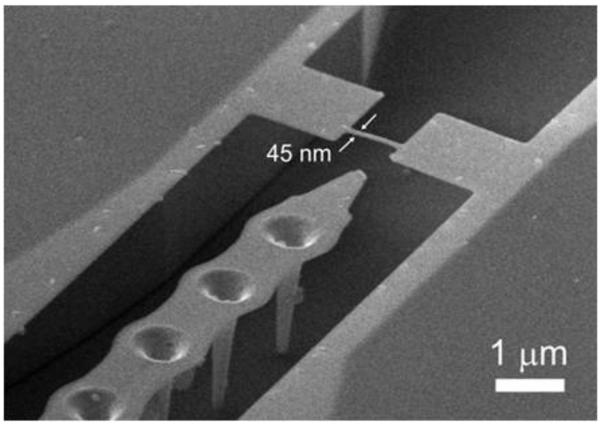


Figure 2.2: Bridge between two pads fabricated using focused ion beam deposition [3]

While this last classification, dependent on the time in the product lifecycle where the attack is performed, is useful when describing hardware attacks in general, its utility is limited when addressing fault injection attacks specifically, as all of them are performed in the field. Hardware attacks performed at design-time or during manufacturing can only be performed by IP vendors or manufacturing companies, and generally try to either add functionality to allow unauthorized access to sensitive data (e.g. private keys, certificates), or to steal intellectual property for financial gain. While protecting against these attacks is no less important than hardening a product against fault injection, the methodologies used to mitigate the former are very distinct from the one used to prevent the latter, and are outside the scope of this project.

2.2. Fault Injection Techniques

Given the aforementioned classes of hardware attacks, it is useful to categorize some common techniques of fault injection to better understand them.

- **Focused Ion Beam**: this technique involves using specialized equipment similar to a scanning electron microscope, that uses ions instead of electrons to bombard a very small area of a chip die. Other than the imaging functionality shared with a regular scanning electron microscope, ion beams can also be used to add and remove material from a chip, cutting traces or creating test points. This allows an attacker to change the functionality of a chip, effectively forcing wires to assume a fixed value regardless of the gates connected to it. It could be used to bypass security checks, or read memory cells that were made inaccessible by blowing one-time-programmable fuses.

  In the classification system, this attack would fall under the following categories: targeting functionality, invasive, affecting nodes or interconnects.

- **Laser Cutting**: similar to the focused ion beam technology, laser cutting stations allow an attacker to modify functionality on a chip by breaking connections, etching test points, or exposing lower layers selectively. It, however, does not allow an attacker to deposit material, so other techniques have to be used if the goal is to re-create broken connections or add new ones.

  The classification of this attack is the same as the one for focused ion beam: targeting functionality, invasive, affecting nodes or interconnects.

- **Voltage glitching**: every chip has an operating range for its supply voltage, reported in its datasheet. Temporarily exceeding this range in either direction can cause timing violations, causing the wrong data to be latched into flip-flops, or cause some wires to cross the 0/1 logic threshold, once again resulting in the wrong data being written to the internal state. Both the timing and the intensity of the violation can be controlled to only affect a specific instruction, but controlling the area in which the glitches will manifest is much harder, as all cells in the same power domain will see a supply voltage range violation.

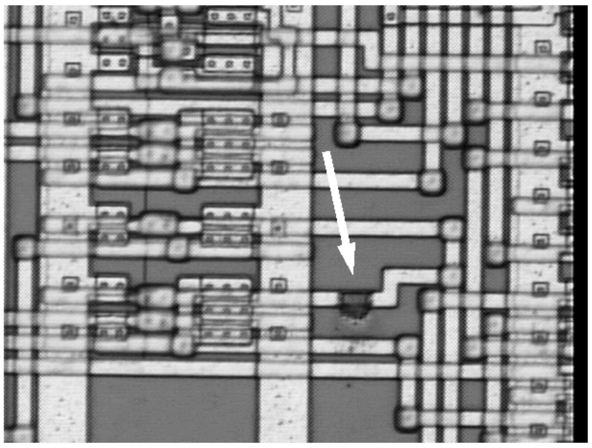


Figure 2.3: Scanning Electron Microscope image of an interconnect cut using a laser [6]

This attack can be classified as follows: targeting data or functionality, non-invasive, affecting nodes.

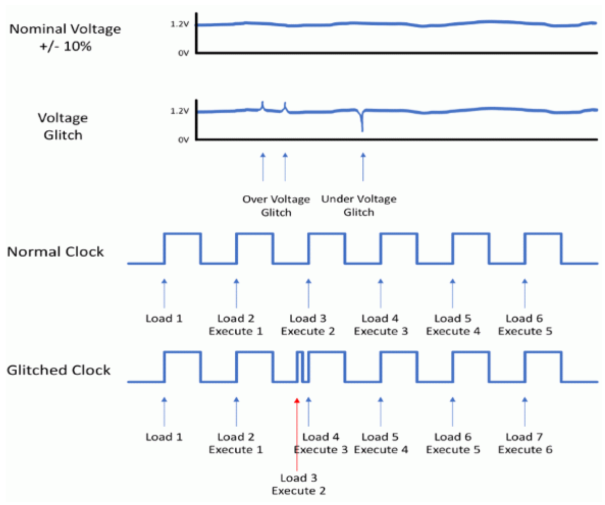


Figure 2.4: Top: voltage glitching, bottom: clock glitching [49]

- **Temperature**: all chips also have an operating temperature range, and exceeding this range can result in faulty behaviour. Overheating can result in bit flips or timing violations, as temperature and propagation delay are correlated [38]. This type of attack has a very low spatial and temporal resolution, since it is impractical to rapidly cool or overheat only a portion of a chip without affecting the rest of it.

  Temperature attacks target data or functionality, are non-invasive, and affect nodes.

- **Clock glitching**: if one of the clocks being used by the chip under exam is supplied externally, one could introduce spurious transitions, or increase the frequency until timing violations occur. If the clock pulse length is smaller than the largest propagation time (corresponding to the critical path), it's possible to affect the data clocked into flip-flops, resulting in unexpected behaviour. Not unlike voltage glitching, the temporal resolution is very high, but the spatial resolution is not: all cells in the same clock domain can be affected. If the target is at the end of a long combinational path, it will be easier to affect it and not other cells, but that is not guaranteed.

  Clock glitching attacks are classified as targeting data or functionality, non-invasive, and affecting nodes.

- **Electromagnetic injection**: by placing inductors close to a chip's interconnects, it's possible to couple the two, and induce electric currents inside the chip by energizing the inductors, causing localized bit flips. This can all be performed with a high time resolution without exposing the die, although spatial resolution will be limited. This allows an attacker to cause transient faults and, once again, change the behaviour of the chip.

  This type of attack targets data, is non-invasive, and affects interconnects.

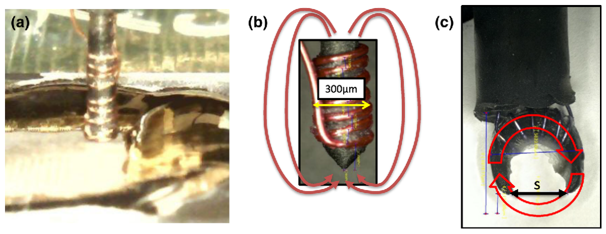


Figure 2.5: Examples of electromagnetic fault injectors [42]

- **Optical**: another attack vector that uses electromagnetic fields involves exposing the bare die to a strong light pulse, such as a laser. Depending on the strength of the pulse, the wavelength chosen, and the width of the beam, it can result in very localized bit flips at any layer, down to a single transistor for a time in the range of nanoseconds.

  Optical attacks target data, are semi-invasive, and affect nodes and interconnects.

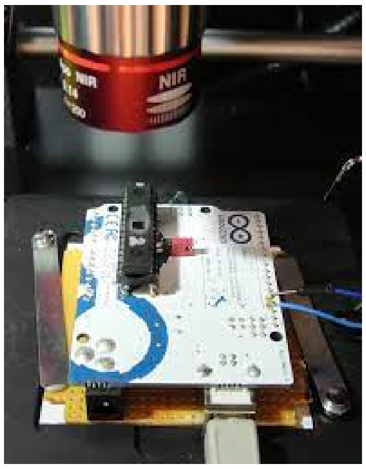


Figure 2.6: Optical fault injection testbench [15]

2.3. Practical Fault Injection Examples

To further describe the real-world effects that can be obtained with the techniques previously described, two different attacks will be analyzed.

2.3.1. Optical Fault Injection Attack

One of the first practical attacks employing strong light pulses to induce faults is described in the paper Optical Fault Induction Attacks, by Sergei P. Skorobogatov and Ross J. Anderson [47].

The target selected was a simple 8-bit Microchip PIC microcontroller, with 68 bytes of static RAM. Once the bare die was exposed, the SRAM region could be located, and individual transistors were targeted by masking all adjacent areas with aluminium foil. Using an inexpensive photoflash lamp, the authors were able to induce single bit flips in any location: looking at Figure 2.7, causing a fault in transistor T_3 resulted in the flip-flop changing state, while targeting T_4 resulted in the opposite state change. By upgrading from a photoflash lamp to a laser pointer, the authors also managed to control the exact timing of the injection, enabling more sophisticated attacks against other devices.

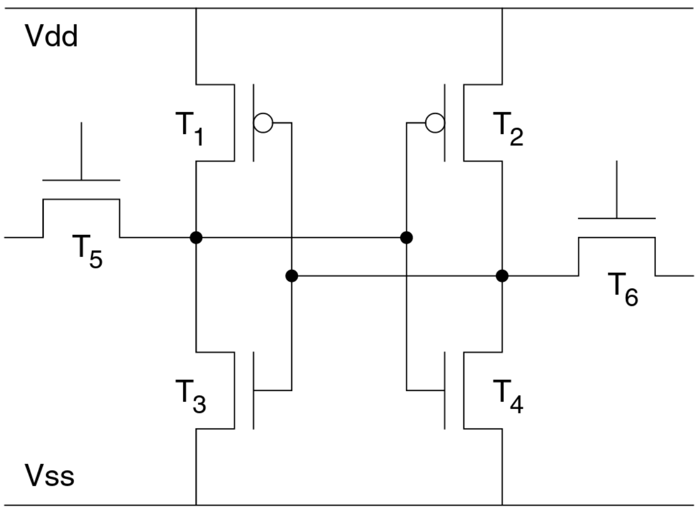


Figure 2.7: Schematic of a CMOS SRAM cell

This attack illustrates how relatively inexpensive equipment can be repurposed to carry out a targeted optical attack, both in terms of affected area and in temporal resolution.

2.3.2. Clock Glitching Attack

Another practical attack, demonstrated in 2017, involves introducing glitches in the clock signal used by an FPGA to attack an AES implementation [55].

AES is a cipher algorithm widely used to encrypt data [22]. It falls under the category of symmetric key encryption schemes, meaning that the same key can be used both to encrypt and decrypt a message. The variant used in this attack uses a 128 bit secret key, and performs 10 rounds of the following operations: SubBytes, ShiftRows, MixColumns, and AddRoundKey. The algorithm was implemented by the authors in VHDL so that each round was performed in 1 clock cycle (for a total of 11 cycles including key preprocessing), and was then loaded onto an FPGA. The researchers then used an external function generator as the reference clock, and by slowly increasing the clock frequency during the MixColumns operation in round 9 (Figure 2.8), they were able to induce timing violations in the FPGA, causing either one byte glitches (model 1) or 4 byte glitches (model 2). Model 1 faults were particularly interesting because the key could be recovered by the authors by observing the output of just 6 different ciphertexts. Choosing an appropriate frequency allowed the authors to exclusively cause faults following this model (Figure 2.9), enabling them to quickly find the secret key.

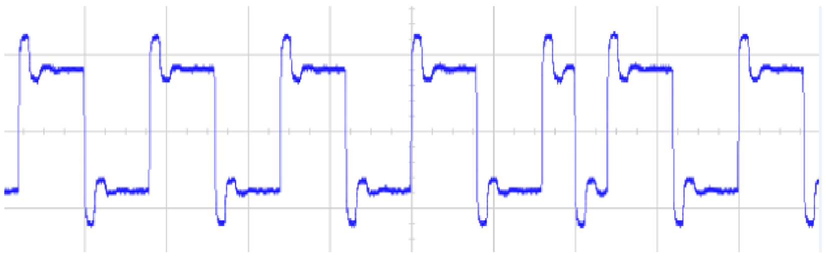


Figure 2.8: Oscilloscope trace of the glitched clock signal [55]

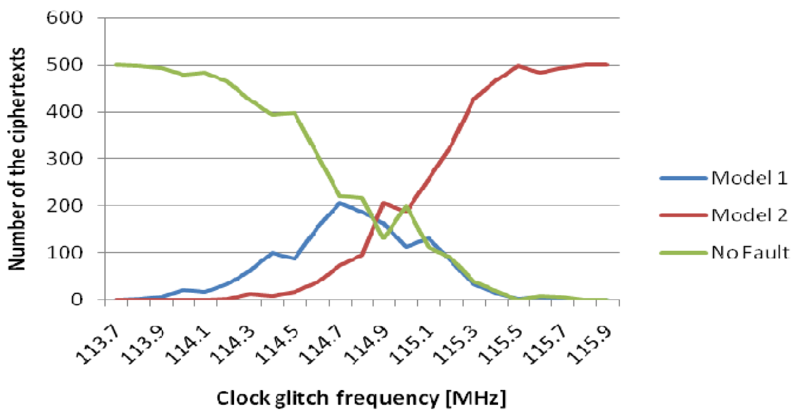
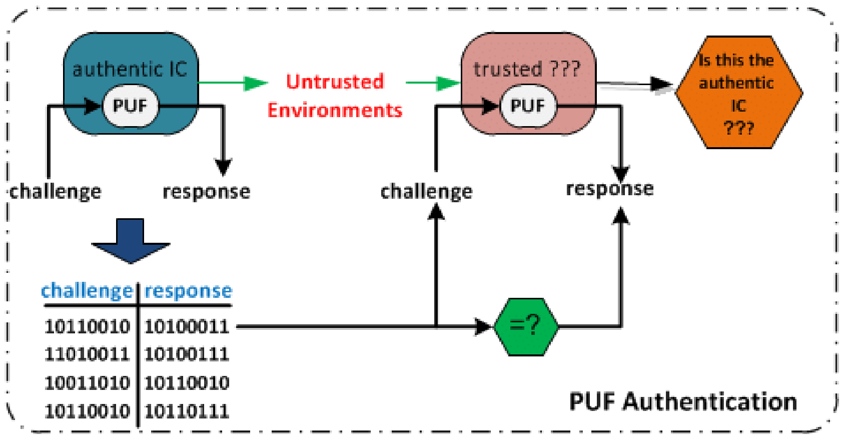


Figure 2.9: Fault profiles at different glitch frequencies [55]

2.3.3. Focused Ion Beam Attack

Yet another practical fault injection attack was demonstrated in 2013, and involved the use of a focused ion beam to replicate physically unclonable functions (PUFs) [30].

In order to prevent counterfeiting of devices, hardware designers have been using the process variations during manufacturing to generate components that have a deterministic output, but at the same time are hard to replicate. These are known as PUFs, and are used for different hardware authentication schemes. Given an external stimulus (also known as a challenge), each device will generate a unique response, which acts as a digital fingerprint. By recording multiple responses to different challenges, the designer can build a table of challenge-response pairs that uniquely identifies each device, and can then verify the authenticity of a deployed device by checking that the response to a given challenge matches the one recorded before deployment.



challenge	response
10110010	10100011
11010011	10100111
10011010	10110010
10110010	10110111

Figure 2.10: Authentication mechanism using a PUF

A simple yet effective implementation of a PUF is an SRAM, consisting of an array of cells each storing a single bit of information. When the device is first powered on, the value assumed by an ideal cell has equal probability of being a logical 0 or 1, and the value assumed by each physical cell depends on physical variations of the transistors that implement them. The manufacturer can then supply the challenge as an address to read, and obtain the response by recording the value at that address.

Since the responses to each challenge are stored in the memory array, if an attacker can gain access to the array and read its contents, they would know the answer to any challenge. The authors of [30] describe a way to then clone the PUF on a different chip, so that its responses will be identical to the original.

The target device chosen is an 8-bit microcontroller by Atmel, namely the ATMega328P. It was first de-capped so as to expose the backside of the chip, and prepared for milling by removing the excess silicon. By observing the infrared emission of the SRAM on the chip, they were able to extract the values of each cell at start-up (Figure 2.11).

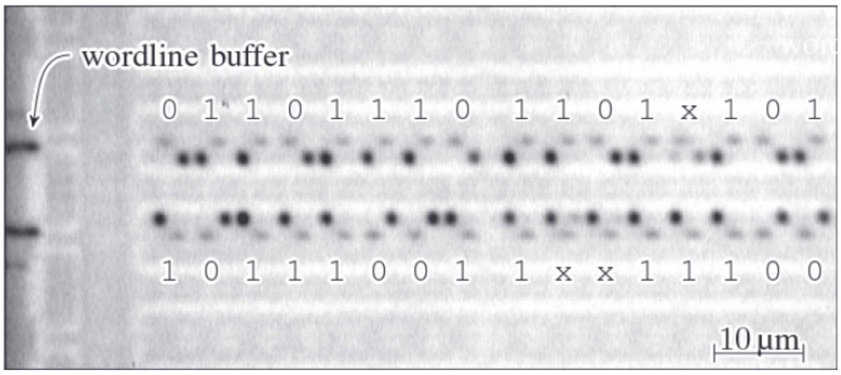

wordline buffer
0 1 1 0 1 1 1 0 1 1 0 1 x 1 0 1
1 0 1 1 1 0 0 1 1 x x 1 1 1 0 0
10 µm

Figure 2.11: Reflected image of the SRAM array to be cloned with overlaid infrared emission image [30]

The next step involved using a focused ion beam on an identical device to clone the start-up value from the target. The new device was prepared similarly to the target, its SRAM start-up value was read, and then portions of silicon were removed with the beam, so that bits that had values opposite to the target were flipped, as seen in Figure 2.12. The result was an array of SRAM cells that had the same start-up behaviour as the target, and that was still fully functional as none of the transistors were completely removed.

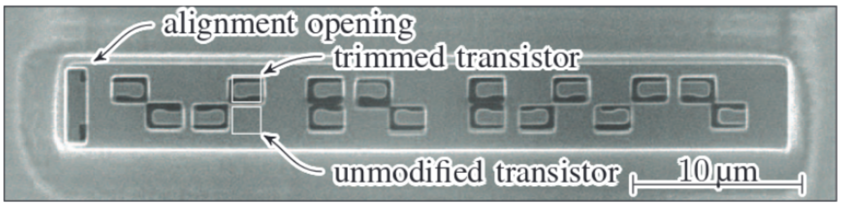


Figure 2.12: Results of FIB milling under an electron microscope [30]

2.4. Fault Profiles

The classification introduced so far focuses on the available means to inject a fault. Another interesting aspect, especially useful when attempting to create a model that can be used during simulation, are the effects that faults have on individual signals and transistors.

A major distinction in this regard can be made between transient and permanent faults.

Transient faults only last for a limited number of clock cycles, as long as the external stimulus that created a faulty condition is maintained, after which the device resumes normal operation. Faults of this kind are generated, for instance, by optical or electromagnetic means, or by glitching the supply voltage or one or more clock domains. Due to their nature, they can manifest themselves only under certain operating conditions. Examples of such faults are:

- **Single Event Upset**: as the same suggests, the fault profile is that a single signal or stored bit is temporarily flipped to the opposite state.
- **Multiple Event Upset**: likewise, if multiple SEUs are concurrently injected into a system, multiple bits will be flipped at the same time.

The kind of fault induced by cutting a trace, as is done with laser or focused ion beams, is not transient, but permanent. In these cases, the value of a signal or a group of signals is irreversibly affected, and the effects are less likely to remain unnoticed during normal operation. Examples include:

- **Single Event Latchup**: the fault profile is that a high current is induced in one of the MOSFETs in a logic port, potentially destroying it. While it can be cleared by resetting the entire device, if left unchecked can result in a stuck-at fault.

- **Single Event Burnout**: a MOSFET exceeds its rated maximum temperature and is permanently damaged, resulting in a stuck-at fault.

Given these different fault profiles, more abstract models can be used to express the effect each of them has on the data carried by the affected wires. These models are useful when simulating fault injection attacks, more so than fault profiles, because they only express the outcome of injecting a fault, regardless of the underlying mechanism that caused it, and that is all the information necessary to simulate a particular fault. For instance, whether a flip-flop is damaged by excessive voltage, or the trace connecting it to the supply voltage is cut with a laser, the result is the same: its output is stuck to a logical value regardless of the data at its inputs. The following fault models express the most common effects of injecting a fault:

- **Stuck-at 0**: typical of single event latchups and single event burnouts, a signal is forced to the '0' value permanently.
- **Stuck-at 1**: signal is forced to the value '1' permanently.
- **Open line**: found when cutting a trace, the value of a signal is neither high nor low, but in a third state of high impedance.
- **Bit flip**: found in single event upsets, the value of a signal is flipped to its complementary for a limited amount of time.
- **Delay**: possible when timing violations occur, such as when performing clock glitching attacks. The value of a signal is delayed by one or more clock cycles.

These fault models can then be used to simulate the effects a fault would have on the hardware realization of the device under test.

3 Formal Verification

Hardware development is an iterative process, where the design is continuously refined to meet requirements by looking at simulation behaviour as well as outputs from automated tools. A high-level overview of the process can be seen in Figure 3.1. Validation and verification are integral parts of this process, as both of them allow designers to identify bugs early and adjust the design accordingly.

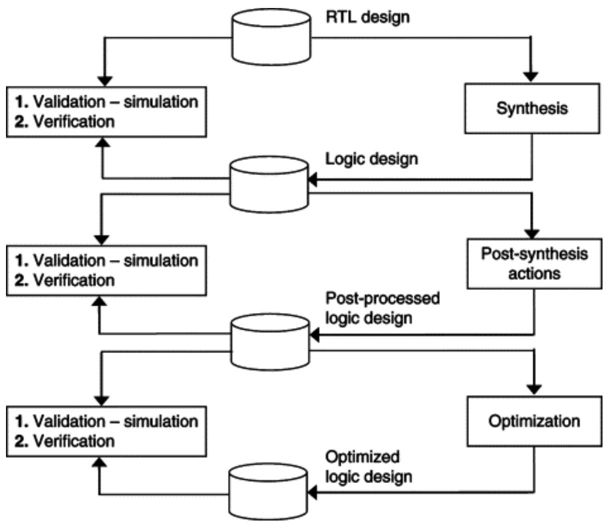


Figure 3.1: Typical hardware design process

This chapters will show the major components of the hardware development process, and the central role played by formal verification in ensuring the correctness of the final product. The formal language employed by formal verification played a central role in the thesis, as it allows one to prove whether two portions of the design at different stages of development implement the same functionality. This can be used to identify high-level constructs at lower levels of abstraction, making the hardening process once a vulnerability is found much simpler.

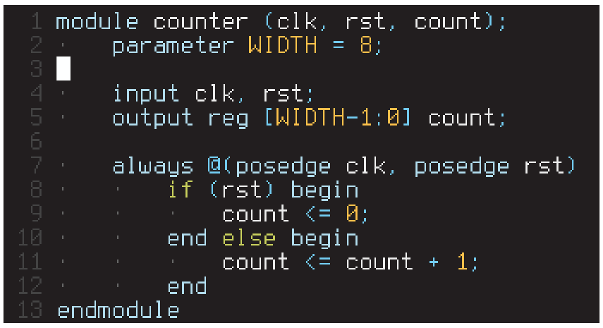
3.1. Hardware Development Process

When designing an integrated circuit, the first step is the delineation of a set of requirements and specifications that the final product should adhere to. These include strictly functional requirements, specifying what functionality the product should implement, as well as non-functional requirements, indicating, for instance, the operating conditions that the product should tolerate, or the physical dimensions that should not be exceeded, or the type of technology that should be used during manufacturing.

From these requirements, a first, high-level hardware implementation is derived, using a language such as VHDL or Verilog. These languages allow designers to describe what the hardware will look like with varying degrees of abstraction, from software-like functional statements down to individual gates.

The resulting description is then tested against the previously defined requirements, and adjustments are made iteratively until the result is satisfactory.

Of note is that designers hardly ever work directly with individual gates at this stage, as that would be impractical and error-prone. Instead, they use higher level descriptions of what the hardware should do, such as those seen in Figure 3.2. Just like software developers do not typically write programs directly in assembly language, hardware developers do not usually describe hardware at the gate level. Instead, they use other tools to obtain a gate-level description of the design, which are collectively classified as synthesizers.



```
module counter (clk, rst, count);
    parameter WIDTH = 8;

    input clk, rst;
    output reg [WIDTH-1:0] count;

    always @(posedge clk, posedge rst)
        if (rst) begin
            count <= 0;
        end else begin
            count <= count + 1;
        end
endmodule
```

Figure 3.2: Verilog description of a synchronous counter

If the high-level description satisfies the given requirements, it is synthesized into a list of gates and interconnects, called a netlist, and the result is, once again, checked to ensure that the specifications are met. If that also succeeds, another transformation is performed on the netlist to obtain a bitstream, which can be used to program an FPGA and obtain a first hardware prototype of the final product. Further checks are performed on this prototype, and then the circuit is ready for manufacturing.

The overall development process is not too dissimilar from that of a software product: a list of specifications is created, a first implementation is generated, then the implementation is tweaked until it matches the given specifications, and using external tools it is converted into a low-level representation that the hardware (or the manufacturing plant in the case of hardware products) can understand. Formal verification, however, is extensively used in hardware development, while in software development it is only used if misbehaviour of the application will have catastrophic consequences [25]. The main reason behind this disparity is cost.

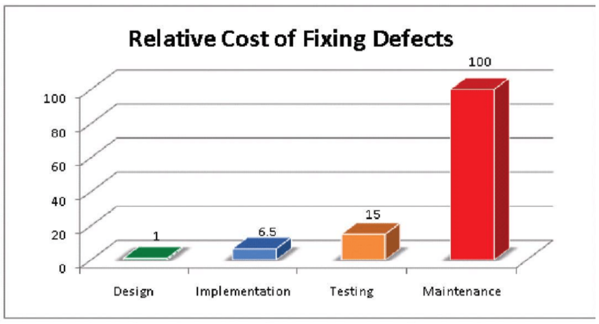


Figure 3.3: Relative cost of fixing defects in various development stages [20]

Catching bugs early is necessary to reduce development costs, as the average cost of fixing a defect exponentially increases the further a product is in its lifecycle, as shown in Figure 3.3. While this growth is fairly constant in software development, hardware development has a much higher jump in cost from the prototype stage to the manufacturing stage. The cost per chip is relatively low for large volumes, but each mask used for chip manufacturing can cost from around 100000 dollars for a 180nm node size up to more than 1.4 million dollars for a 28nm node size [7]. It is imperative, therefore, to ensure that any possible defect introduced during the design stage is identified and removed, something that cannot be accomplished by validation alone.

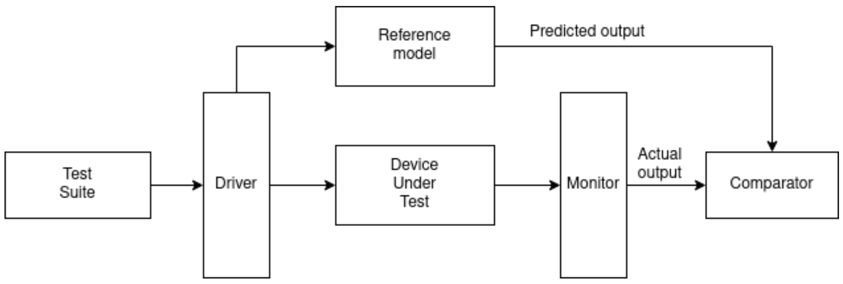


Figure 3.4: Simulation-based testing diagram [26]

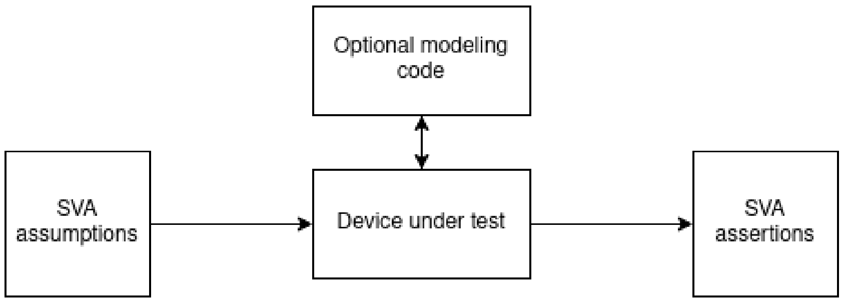


Figure 3.5: Formal verification diagram [26]

3.2. Simulation-Based Validation and Formal Verification

In order to check whether a design is implementing the necessary functions, a way to write values to its input ports and read the results from its output ports is necessary. During simulation, this is done with (usually unsynthesizable) code that interfaces with the device under test by emulating peripherals attached to it, recording the values of each signal, called a testbench.

In a typical functional simulation, every piece of the testbench is written by the designer: a set of stimuli is selected and compiled into a test suite, and each of them is input to the device under test using a driver. Then, the design is simulated, the outputs are captured by a monitor and compared against the output of a reference model, and a pass or fail status is recorded based on whether the two match (Figure 3.4).

This process is useful in the first stages of development, to catch bugs as early as possible, and is extensively used both in hardware and software development due to its generic nature. If a product can be simulated or executed, whether it is hardware or software, functional simulation can be performed.

In order to catch as many bugs as possible, the input stimuli have to be carefully crafted to exercise all parts of the design, including any possible edge cases that require special handling of the inputs. A large number of test cases can be obtained by using random inputs, but the quality of the test results ultimately depend on the skills of the designer and the amount of time spent generating test cases.

More importantly, a design passing all tests does not guarantee the absence of bugs: an input sequence that causes the device under test to malfunction could always exist outside of the test suite. In other words, it is never possible to guarantee that every part of a design is working exactly as intended without checking every possible input. This becomes infeasible very quickly if the device under test stores information in an internal state, as every input has to be checked with every state combination to ensure correct implementation of RTL functionality at the netlist level. This is the gap that formal verification fills.

Formal verification is the act of proving that an algorithm satisfies a certain set of specifications defined at design-time [28]. It has the same goal as functional simulation - to identify bugs in the design. In formal verification, however, no test suite, driver, or monitor is necessary: the tool itself performs checks automatically (Figure 3.5). The designer specifies a set of assumptions that constrain the inputs according to the design specification, and a set of assertions that express properties of the outputs that should always hold. This is much less error-prone than manually specifying test cases, because specifications are formalized in the same assertions-assumptions format during the first stages of the design process.

Since the device under test is essentially a network of logic gates and flip-flops, it can be represented as a set of mathematical equations, which can be used by the tool to prove that the given assertions logically follow from the given assumptions.

The mathematical nature of formal verification allows this procedure to prove the absence of bugs in a design, something that is infeasible by means of functional simulation. The downsides are that the complexity of the tools necessary is higher, and the time necessary to obtain a result longer, which is why simulation-based testing is still used during the development process.

3.3. Equivalence Checking

Equivalence checking is the portion of formal verification concerned with proving that two representations of a design exhibit the same behaviour [39]. It is applied throughout the design process, and on different scopes. One of its uses is to prove that the netlist output of a synthesis tool exactly implements the functions specified at a higher level of abstraction.

The need for equivalence checking post-synthesis comes from the fact that there might be bugs in the synthesis tools, which could lead to an incorrect netlist output from a correct RTL input. The goal of equivalence checking is to identify whether any bugs were triggered during the synthesis process. It does so by mathematically proving the logical equivalence between the input RTL and the output netlist.

There are two kinds of equivalence checking: sequential equivalence checking, where the goal is to prove that two finite state machines are equivalent, and combinational equivalence checking, which aims to prove whether two sets of gates implement the same boolean function.

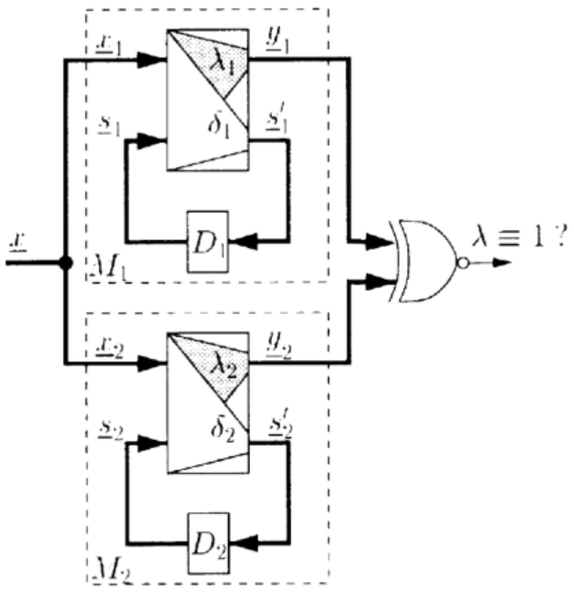


Figure 3.6: Product machine for sequential equivalence checking [36]

Formally, sequential equivalence checking proves whether two finite state machines produce identical output sequences for all valid input sequences [36]. The classical way of verifying this statement for two state machines M_1 and M_2 is to create a product machine as shown in Figure 3.6, and verifying that for any input sequence x, $\lambda = 0$. The problem can be reduced to proving that there is a characteristic function $\rho : S \to \{0, 1\}$, such that:

$$\begin{aligned} &\rho(s_0) = 1 \\ &\rho(s) = 1 \Rightarrow \forall x, \rho(\delta(x, s)) = 1 \\ &\rho(s) = 1 \Rightarrow \forall x, \lambda(x, s) = 1 \end{aligned} \tag{3.1}$$

where δ is the next state function and λ is the output of the combined state machine.

Combinational equivalence checking, on the other hand, aims to prove that two different implementations of the same boolean function are equivalent. It involves the following steps:

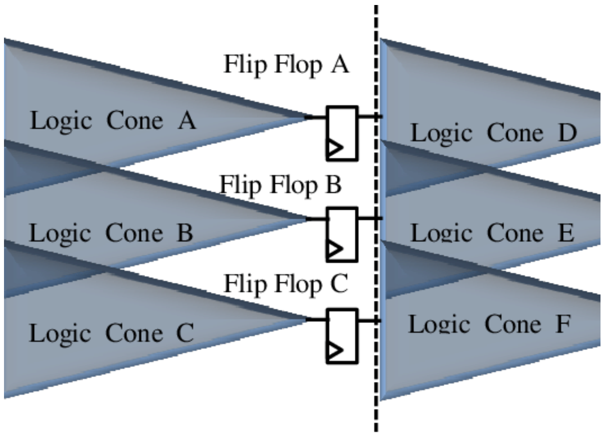


Figure 3.7: Schematic representation of logic cones [43]

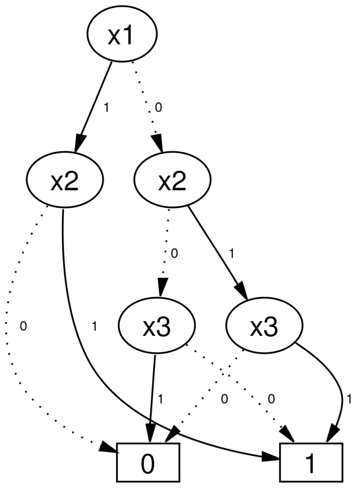


Figure 3.8: Binary Decision Diagram for the boolean function $f(x_1, x_2, x_3) = (x_1x_2) + \neg x_1 \wedge \neg(x_2\dot{\vee}x_3))$

- **Read**: the implemented design and the golden reference are read, and segmented into groups of logic gates bordered by registers, ports, or black boxes called logic cones, as shown in Figure 3.7.
- **Match**: the output of each logic cone is considered a compare point, and compare points are matched between the two designs using both function and non-function based methods, such as name-based matching. This maps logic cones from the reference design to the corresponding logic cones in the implemented design.
- **Verify**: matched logic cones are compared to each other, checking that the functions they implement are equivalent.

The output of the tool will be a set of comparison points with mismatching logic cones. If none are reported, the pre- and post-synthesis representations are mathematically proven to behave the same way for all inputs.

Since the logic cones being compared can be expressed as boolean functions, the problem of combinational equivalence checking can be reduced to a boolean function equivalence problem. There are two basic techniques used for boolean reasoning: binary decision diagrams (BDDs) and satisfiability problem solving (SAT).

BDDs are representations of boolean functions as directed acyclic graphs, where each non-leaf node represents a variable, each leaf represents an output of the boolean function, and each edge represents an assignment of TRUE or FALSE to that variable. An example is shown in Figure 3.8. The output of the function can be determined by traversing the graph starting from the root node, and choosing the next edge based on the value assigned to the variable in the current node. For example, in order to determine the output of the function in Figure 3.8 for $x_1 = 0, x_2 = 1, x_3 = 0$, starting at the top node, the right edge will be followed, then the left edge again, and finally the right edge, landing on the leaf node 0.

The advantageous property of these structures is that they have a canonical form, called Reduced Ordered Binary Decision Diagram, which, given a variable ordering, is shared among all equivalent boolean functions. This makes equivalence checking trivial, since all that has to be done is to check whether the two canonical BDDs are equal [16].

Alternatively, the problem of equivalence checking can be expressed as a reductio ad absurdum SAT problem: two functions are equivalent if there is no input for which they are different. In symbols:

$$f_1 \equiv f_2 \Leftrightarrow \nexists x, f_1(x) \dot{\vee} f_2(x) = True \tag{3.2}$$

where $\dot{\vee}$ indicates the exclusive-or function.

If the expression $f_1(x) \dot{\vee} f_2(x)$ can never be satisfied, then the two functions will have the same output for all inputs, and are equivalent by definition.

While SAT is a known NP-complete problem [34], with no known polynomial-time algorithm for deterministic computers, multiple heuristic solvers exist with varying degrees of optimization, able to solve problems involving tens of thousands of variables [41]. This makes it practically feasible to use SAT solvers to check designs for equivalence.

One such optimizations is the use of cutpoints: internal equivalence points used to decompose large expressions into smaller ones. The existence of cutpoints cannot be guaranteed in general, but experimental analysis of real-world designs showed that the number of internal equivalence points is significant, and that equivalence checking algorithms can and should exploit these similarities to reduce their execution time [37].

4 Algorithm Optimization

While the central part of developing any algorithm is the correctness of the functionality being developed, an important practical aspect is that of computational complexity. In short, for an algorithm to be practical, it should yield results in a reasonable amount of time.

This is of particular importance when treating notoriously hard problems, such as checking whether a boolean function can ever be true, or verifying that two boolean functions are equivalent to one another. Since a large portion of the work described in this thesis revolves around these complex operations, particular care was taken to optimize the algorithms developed to some degree, so that they would be more than theoretical proof-of-concepts.

This chapter aims to provide a cursory understanding of the terminology used when describing the complexity of an algorithm, such as $\mathcal{O}$-notation and NP-completeness.

4.1. Computational Complexity

When evaluating the time and memory required to solve a problem with a given implementation, the number of variables involved is very high. Everything from the language chosen to write the algorithm, to the operating system from which it is launched, to the type of processor it is run on (architecture, cache size, memory bandwidth, clock frequency) can affect how long the program will take to produce an answer.

In order to better compare different algorithms, and choose the ones that are better suited for the problem at hand, all of these parameters have to be abstracted away. To do this, an ideal Random Access Machine, or RAM, is defined. This machine can execute all of the operations typically found in a modern computer in a constant amount of time. These instructions include arithmetic operations such as addition and multiplication, instructions to move data around such as load and store, and control flow instructions such as jump and return. Instructions that require an operand work with limited-size items, so that an arbitrary sized input cannot be processed all at once, which would be unrealistic for any real-world computer.

The random access machine thus defined is deliberately fuzzy in its definition. Some operations may or may not be performed in a constant amount of time by it. For instance, an arbitrary computer cannot perform exponentiation in a constant amount of time, but can however compute 2^k with a single instruction for a limited range of k values. Moreover, it does not model the memory hierarchy found in modern computers at all, something that can heavily influence the computation time in practice. Taking all of these variables into account would result in a more accurate time estimation, but would also make the analysis more difficult to perform. Regardless of these limitations, analyzing the characteristics of an algorithm on this machine can yield useful insights into how a program behaves with different inputs.

4.1.1. Case Study: Insertion Sort

As a simple example, consider the algorithm described in Algorithm 1. Given an array A of numbers, it will sort its elements from smallest to largest. To do so, it looks at each element in the array one at a time, from first to last, and moves each element preceding it to the right until the sub-array is sorted.

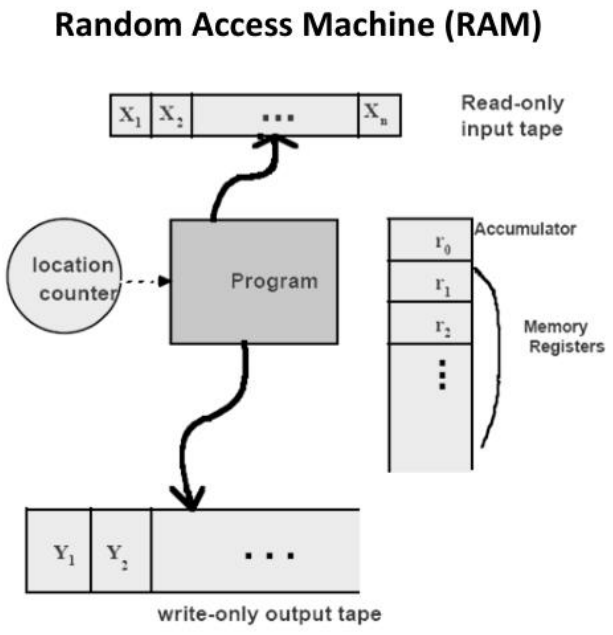


Figure 4.1: Schematic diagram of a random access machine

Algorithm 1 Insertion sort pseudocode

```
Require: $A$ unsorted array of numbers
Ensure: $A$ sorted from smallest to largest
1: for j = 2 to A.length do
2:     $key = A[j]$
3:     $i = j - 1$
4:     while $i > 0 \wedge A[i] > key$ do
5:         $A[i+1] = A[i]$
6:         $i = i - 1$
7:     end while
8:     $A[i+1] = key$
9: end for
```

The time required to execute said algorithm on the RAM depends on the input A: if the array contains 100 numbers, the algorithm will take longer than it would on an array of only 10 numbers. The time taken also depends on the initial position of the elements in the array, as the inner **while** loop will run for fewer iterations if the key is already in the correct position. In general, the time required by an algorithm will grow with the size of the input, so the complexity is most often expressed as a function of the size of its input data.

The definition of “size of its input data” can vary from problem to problem: for sorting algorithms it is the number of elements in the input, for exponentiation it is the number of bits required to represent the input. Other times, it is more appropriate to describe the size of the input with more than one dimension, such as when treating matrix operations. It is important to note, however, that the definition of “size of the input data” is dependent on the problem at hand, not on the algorithm used to solve it. This makes it possible to compare possible solutions, and choose the one that is fastest for a set size.

When analyzing an algorithm, the execution time is defined as the number of primitive operations necessary to calculate the solution. Describing a universal definition of a primitive operation can be hard, as seen when outlining the concept of a RAM, but for practical purposes, we can assume that each line of code requires a constant number of operations. Different lines may require different amounts of primitive operations to execute, but the number of operations is a set constant regardless

Line number	Cost	Number of executions
1	c_1	n
2	c_2	$n-1$
3	c_3	$n-1$
4	c_4	$\sum_{j=2}^{n} t_j$
5	c_5	$\sum_{j=2}^{n} t_j - 1$
6	c_6	$\sum_{j=2}^{n} t_j - 1$
7	0	$\sum_{j=2}^{n} t_j$
8	c_8	$n-1$
9	0	$n-1$

Table 4.1: Complexity analysis for the insertion sort algorithm

of the input data. This hypothesis is compatible with the RAM model, and is also reflected in how real-world computers execute code.

For Algorithm 1, we can associate a cost c_i to each line, indicating the number of atomic operations necessary to execute that line. The total time taken will then be equal to the sum of the costs multiplied by the number of times each line is run. These counts will be expressed as a function of the input size n, so as to obtain a generic formula for the execution time. The results are reported in Table 4.1, where t_j is the number of times the **while** loop will be run for the j-th iteration.

The total execution time will be therefore:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Even for inputs of the same size, this time may vary greatly based on how the input numbers are arranged. In the best-case scenario, the input array is already sorted, and t_j will be equal to 1 for all iterations. The time required will reduce to:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5(n-1) + c_6(n-1) + c_8(n-1)$$

This can be expressed as a linear function of n $an+b$, where a and b depend on the (constant) execution time of each line. When the array is not sorted, however, the time required changes drastically: in the worst-case scenario, where the array is inversely sorted (from highest to lowest), t_j is always equal to j, resulting in the following:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n+1)}{2}\right) + c_6\left(\frac{n(n+1)}{2}\right) + c_8(n-1)$$

This function can be expressed as $an^2 + bn + c$, which is a quadratic function with constants depending, once again, on the execution time of each line of code. As n increases, this function increases much more quickly than the one obtained in the best case, highlighting how the input size is not the only parameter to consider when calculating the execution time requirements of the algorithm.

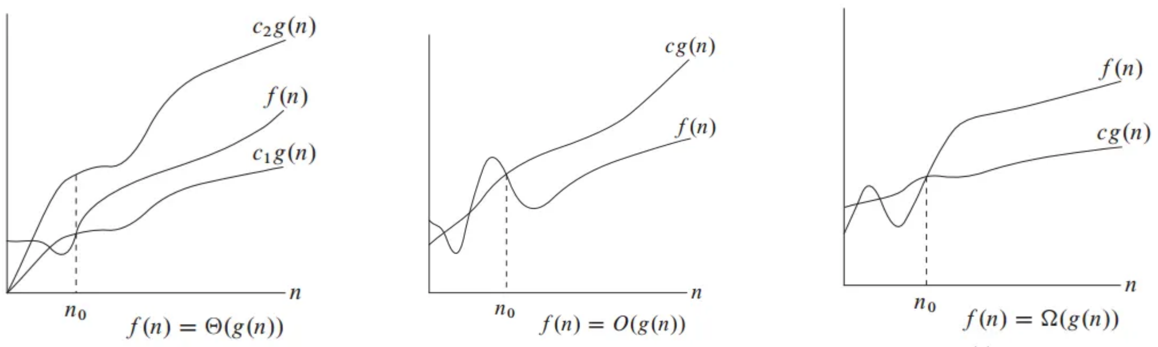
In most practical applications, the metric used to determine the performance of an algorithm is the worst-case execution time. There are multiple reasons behind this choice:

- The worst-case execution time is an upper bound on the execution time of an algorithm for any input. Knowing this value, the algorithm is guaranteed to return an answer within this time period, without any other hypothesis on the input data.
- For most algorithms, the worst case execution time is much more likely to occur than the best-case or average-case execution time. For example, when searching a hash in a database, the worst-case execution time occurs every time the hash is not in the database.
- The average-case execution time is often in the same order of magnitude as the worst-case execution time. Take for instance the insertion-sort algorithm: on average, half of the elements will have to be checked by the **while** loop in each iteration to determine where to place each element. Thus $t_j \approx j/2$, resulting in a quadratically increasing execution time.

- The average-case execution time often requires some level of statistical analysis on the expected input data, as well as more assumptions on what the average case looks like for a given problem.

Yet another simplifying assumption used in practice is considering only the fastest-growing term in the execution time function. For insertion-sort, the best-case execution time is said to grow linearly, while the worst-case execution time grows quadratically, disregarding all terms with a lower degree. This is because the interesting behaviour is that observed for very large values of n, where execution times might be measured in the order of hours or more.

4.1.2. Θ Notation



(a) Example of $\Theta(g(n))$ (b) Example of $\mathcal{O}(g(n))$ (c) Example of $\Omega(g(n))$

Figure 4.2: Graphical representations of different complexity notations

As previously mentioned, the most widely used metric for determining the time required for an algorithm to run is its asymptotic complexity. A more rigorous definition of asymptotic complexity is the following: for a given function $g(n)$, $\Theta(g(n))$ indicates the set of all functions such that:

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 : 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \quad \forall\, n \geq n_0\}$$

A graphical representation is shown in Figure 4.2a. In less formal terms, a function $f(n)$ is in the $\Theta(g(n))$ set if and only if there are two constants c_1 and c_2 such that $f(n)$ is between $c_1g(n)$ and $c_2g(n)$ past a certain value n_0. As an example, the function $T(n)$ for insertion sort in the worst-case is $\Theta(n^2)$, because as n approaches infinity, $0.5n^2 < T(n) < 2n^2$, as the lower-degree terms in $T(n)$ contribute a smaller and smaller fraction compared to the second degree term. Note that the constants associated with the higher-degree term are also irrelevant, since for any function $f(n) = an^k$, it is sufficient to choose $c_1 < a, c_2 > a$ in the definition of $\Theta(g(n))$ to find functions that are part of the set.

4.1.3. $\mathcal{O}$ Notation

The Θ notation just described bounds the function $f(n)$ both from above and below. When the goal is to only find an upper bound on the given function, an alternative notation is used, called big-o or $\mathcal{O}$ notation. The definition is the following:

$$\mathcal{O}(g(n)) = \{f(n) : \exists c, n_0 > 0 : 0 \leq f(n) \leq cg(n) \quad \forall\, n \geq n_0\}$$

As can be seen from the example in Figure 4.2b, this notation limits the function only from above. This is a superset of the functions in $\Theta(g(n))$, which have an extra condition for the lower bound. For example, taking the $T(n)$ for the worst-case execution in insertion sort, it still holds that $T(n) \in \mathcal{O}(n^2)$. It also holds true that any linear function is in $\mathcal{O}(n^2)$: given a generic $f(n) = an + b$, one can choose $c = a + |b|$ and $n_0 = max(1, -b/a)$ to ensure that $cn^2 \geq f(n) \quad \forall\, n \geq n_0$.

This notation is more commonly used than the more stringent Θ notation because it is simpler to prove, and still provides enough information to determine which algorithm has better performance (a $\mathcal{O}(n)$ algorithm is always asymptotically better than a $\mathcal{O}(n^2)$). It will be the notation of choice when describing the computational complexity of the algorithms described in this document.

4.1.4. Ω Notation

Just as the lower bound requirement can be removed from the Θ notation to obtain the $\mathcal{O}$ notation, the upper bound requirement can be lifted to obtain another notation, called Ω-notation. The definition then changes as follows:

$$\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0 : 0 \leq cg(n) \leq f(n) \quad \forall\, n \geq n_0\}$$

In this notation, the function is only bounded from below, and can grow arbitrarily large at any rate, as shown in Figure 4.2c. $\Omega(g(n))$ is also a superset of $\Theta(g(n))$, because once again the definition of $\Theta(g(n))$ has an extra condition for the upper bound.

This notation is only sporadically used, when for instance the goal is to show that an algorithm is at least as bad as another. Since it does not provide an upper bound, both $f_1(n) = 2^n$ and $f_2(n) = n^2$ can be shown to be $\Omega(n^2)$, even though the former will grow at a much quicker rate than the latter.

4.2. P and NP

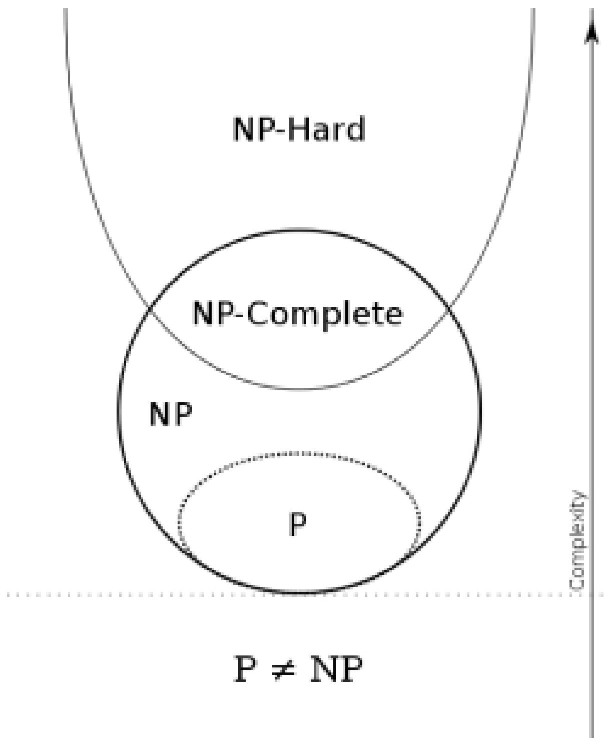


Figure 4.3: Relationship between P and NP problems

Most algorithms have a worst-case execution time of $\mathcal{O}(n^k)$ for some constant value of k. Examples include insertion sort ($\mathcal{O}(n^2)$), linear search in an array ($\mathcal{O}(n)$), and search in a sorted binary tree ($(O)(log(n))$). All of these problems are said to belong to the P class, or class of problems that have polynomial-time solutions.

However, not all problems can be solved in polynomial time. A classical example is the “halting problem”: given a Turing machine, a program and an input sequence, it is impossible to determine whether the machine will halt, regardless of the time available [54]. Some other problems can be solved, but not in $\mathcal{O}(n^k)$ [32].

There is another class of problems for which there is no known polynomial solution, but that are also not proven to have strictly exponential computation time. These are known as NP problems.

The NP class contains all problems that can be verified in polynomial time: given a candidate solution, it takes $\mathcal{O}(n^k)$ time to verify whether it is an actual solution, where n is the length of the input to the problem. This means that every problem in the P class is also part of the NP class: obtaining solutions can be done in polynomial time by definition, and comparing the solution with the given candidate can also be done in polynomial time. It is still unknown, however, whether P is a

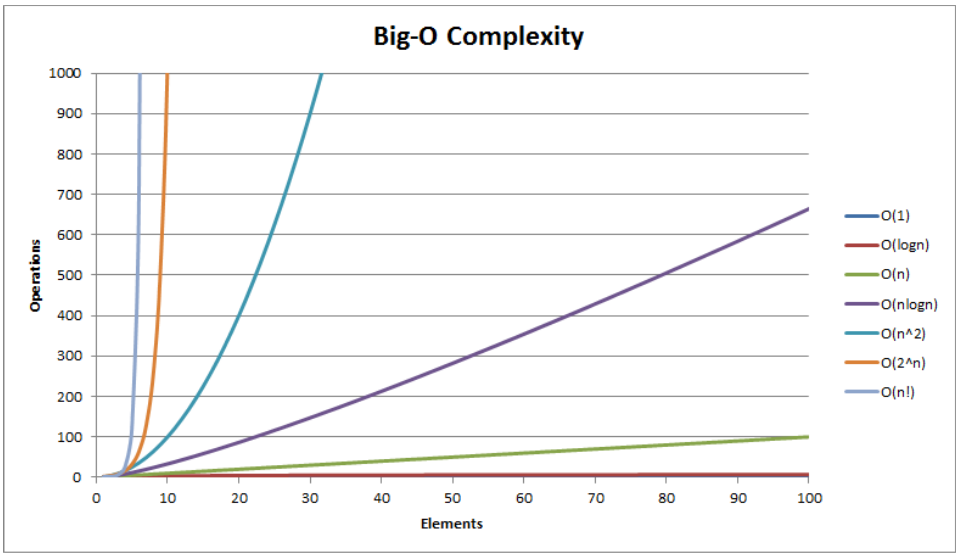


Figure 4.4: Examples of growth of complexity curves. NP problems have curves similar to the ones orange and sky blue

proper subset of NP or $P = NP$. If $P = NP$ was proven true, it would mean that all problems currently considered intractable have a polynomial-time solution, but currently NP problems are still exceptionally hard to compute (Figure 4.3 and Figure 4.4).

4.2.1. Reducibility

A very useful tool to identify whether a problem is in P or NP is that of reduction. Intuitively, a problem Q can be reduced to another problem Q' if it's possible to rewrite all instances of Q as instances of Q'. For example, the problem of solving a linear equation $ax + b = 0$ can be rewritten as the problem of solving a quadratic equation $0x^2 + ax + b = 0$, and solving this problem yields the answer to the original equation. The insight provided by reduction is that, if Q can be reduced to Q', it is at worst as hard to solve as Q'.

A problem Q is said to be reducible in polynomial time to Q' if there exists a function f that maps solutions to Q into solutions to Q', and x is a solution to Q if and only if $f(x)$ is a solution to Q'. Polynomial reduction is a powerful tool to prove that a problem belongs to P: if Q can be reduced to Q' in polynomial time and Q' belongs to P, then Q also belongs to P.

4.2.2. NP-Completeness and Circuit Satisfiability

Among all problems in the NP class, the hardest ones are known as NP-complete. These problems have the added characteristic of being at least as hard as every other problem in NP, or that they can be reduced in polynomial time to any other problem in the NP class. If these were proven to have a polynomial-time solution, it would imply that all problems in NP also have a polynomial-time solution, hence their relevance in the $P = NP$ discourse.

One such problems is that of circuit satisfiability: given a set of interconnected logic gates, prove that there is a set of inputs that causes the output to be TRUE. The naïve solution is to extensively check all possible combinations for inputs and see whether the output is ever TRUE. Since each boolean variable can be either TRUE or FALSE, the number of test cases is 2^n, resulting in a $\Omega(2^n)$ complexity (assuming that the number of gates is polynomial with respect to the size of the input).

To prove that the problem is NP-complete, it must first belong to the NP class, meaning that it can be verified in polynomial time. This can be done using the following algorithm: given the logic circuit C and a set of wire assignments K, the algorithm checks for each logic gate whether the output found in K is correct given the inputs in K. If this is the case, and the output of the overall circuit is TRUE, the algorithm returns TRUE, else it returns FALSE. The algorithm thus described executes a number of operations linear with the number of gates, which is assumed to be polynomial with respect to the number of inputs. It therefore runs in polynomial time, so the circuit satisfiability problem is in NP.

The second part is to prove that the circuit satisfiability problem is at least as hard as every other problem in NP, or that every problem in NP can be reduced in polynomial time to a circuit satisfiability problem. The entire proof can be found in Introduction to Algorithms by Cormen et. al. [18], but the intuitive idea is to write a given NP problem as an algorithm to run on a simplified computer, composed of a working memory and a combinational circuit that operates on it. It is possible to create a function that reduces a NP problem into an algorithm that can be run by this computer in polynomial time, and the computer itself will run the algorithm in polynomial time. This proves that the circuit satisfiability problem is at least as hard as any other NP-class problem, and is therefore NP-complete.

The proven difficulty of treating circuit satisfiability problems has guided the optimization efforts during the development of the entire framework. The most tangible improvements were obtained by reducing the number of circuits that had to be satisfied, as the computation time was dominated by the $\mathcal{O}(2^n)$ term introduced by satisfiability algorithms.

5 Framework Overview

In order to obtain concrete results from the tools developed in this thesis, a number of different tools had to be used, ranging from Verilog simulators to compilers for the processor architecture chosen. The goal of this chapter is to provide an overview of how all of these components fit together at a high level, followed by a short description of the external tools that were selected to realize some of these components. A description of the design targeted during the thesis will also be given, both in terms of the generic architecture selected and in terms of the specific implementation chosen.

5.1. Functional components

Two main research directions were followed during this project. The first was concerned with establishing relations between netlist components and RTL components, while the aim of the second was to implement a more formal approach to netlist fault simulation, reducing the computational requirements of netlist-level fault injection simulations at the same time. Figure 5.1 shows how the two components fit together into the framework. The setup phase uses readily available components to generate a netlist, a preprocessed RTL with a simplified structure, and an execution trace for the target application running on the netlist. The execution phase uses the developed scripts to perform annotation recovery and netlist fault simulation. The two are completely independent, and were treated as such during development.

Generation of the annotated netlist uses the synthesis output and a pre-processed version of the RTL design, and matches parts of the former to parts of the latter using equivalence checking. The inner workings of the algorithm are discussed in chapter 6.

Fault injection is performed in two steps. First, the synthesized netlist is statically analyzed to determine which glitches would generate faults that will affect the state of the processor. For each fault, the accessory conditions necessary for this to happen are also calculated. Once the static analysis is complete, the trace from simulating the target application is used to check when the propagation conditions are met, and to track which states will be affected in subsequent clock cycles. Given a set of conditions that, if met, would result in a vulnerability, the output will be a set of faults that result in those conditions being met. More in-depth discussion of the algorithms used will follow in chapter 7.

5.2. External tools

In order to facilitate development and reduce the chances of errors in the core algorithms used to implement the aforementioned functional components, some external tools were used, namely YoSYS, PyEDA, and Icarus Verilog.

5.2.1. YoSYS

Previous research attempts directly used Verilog design descriptions to perform hardware vulnerability analyses, but struggled with supporting all the features that are part of the Verilog language specification [50]. This resulted in multiple preprocessing steps, necessary to rewrite unsupported constructs in more simple forms.

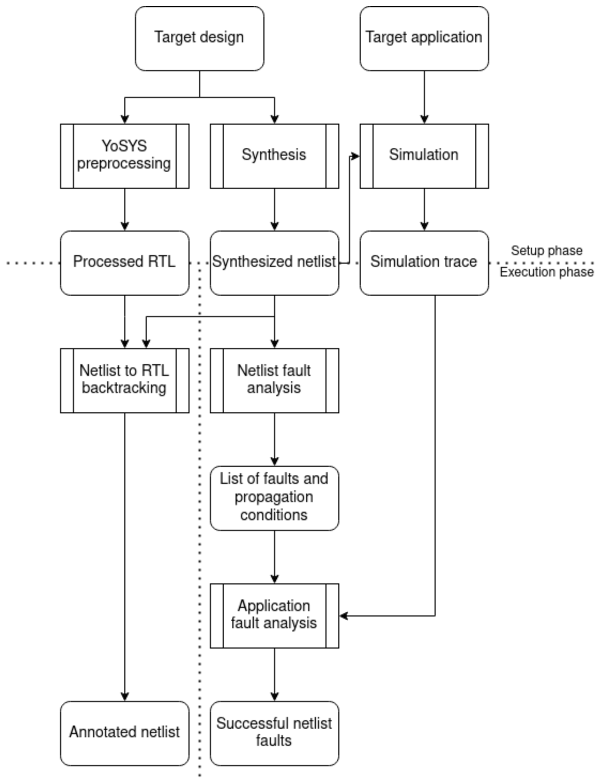


Figure 5.1: Framework functional block diagram

Object name	Function
Module	Describe a collection of cells and wires, exposed to other components via ports
Cell	Instantiate modules
Wire	Represent a physical wire interconnecting two or more cells
SigSpec	Connect cell ports with wire objects, constants, or a concatenation of both

Table 5.1: List of synthesizable constructs in the YoSYS intermediate language specification

A more robust alternative is to use existing tools to perform the necessary preprocessing step. One such tool is YoSYS, the Yosys Open SYnthesis Suite.

YoSYS is a vendor-agnostic open-source synthesis framework [11]. Many other tools exist to perform synthesis of an RTL description into a netlist, both free and commercially licensed. Examples include Genus [5], developed by Cadence, and Vivado [10], developed by Xilinx. Each of them has unique features, and optimizations for specific FPGA layouts. The unique feature of YoSYS used throughout the development of the tools is its ability to represent Verilog hardware descriptions into a simpler intermediate language, called the YoSYS intermediate language, RTL intermediate language, or RTLIL. The specification for this language is also freely available [53].

The RTLIL is functionally equivalent to Verilog, and conversion to and from Verilog can be easily performed using YoSYS's command pairs `read_verilog`/`write_ilang` and `read_ilang`/`write_verilog`.

The synthesizable portion of the RTLIL consists of a very small set of simple objects, which can be parsed with ease: modules, cells, wires, and sigspecs. A short description of each is reported in Table 5.1.

Both the RTL description and the netlist synthesized equivalent can be represented using these objects, with the only difference being the abstraction level of the cells being instantiated, described as high-level functions first, and then gradually mapped to a given cell library during the synthesis process.

Synthesis is controlled using scripts, where a design is read first, then elaborated with one or more commands, and finally written to disk. The script used throughout this project to convert an RTL design into a netlist is reported in section A.1.

5.2.2. PyEDA

Since the entire work presented here revolved around analyzing logic networks, it was necessary to find a library that supported evaluation and manipulation of boolean functions, the mathematical abstraction of collections of logic gates.

Boolean function operations were performed using PyEDA, a Python library for electronic design automation [21].

This library provides routines to simplify functions, evaluate them given a set of inputs, compose them, convert them to their canonical binary decision diagram representation, and check their satisfiability, among others.

The library had to be slightly modified to be able to load and store the results from one script to the next. More specifically, it was adapted to correctly parse stored literals, necessary to successfully reconstruct boolean functions from a file into memory. The changes made are reported in section A.2.

A limitation of PyEDA is that boolean variable names can only contain a subset of all printable characters, so wire names could not be used directly in boolean expressions. The solution adopted was to map each wire to a unique name only containing allowed characters, and storing the mapping between Verilog names and PyEDA variable names.

5.2.3. Icarus Verilog

The tool used to simulate Verilog code was Icarus Verilog [23]. This is an open-source simulation and synthesis tool, able to compile and simulate Verilog IEEE-1364 components.

Icarus Verilog compiles Verilog source code into executable programs for simulation, using `vvp`, or other netlist formats for synthesis and further processing. Here only the simulation part will be used, to generate a VCD trace containing the value of each wire in the design at every simulation time instant.

5.3. Target description

The target selected for validation of the developed tools was PicoRV32, an open-source, configurable RISC-V processor [52]. It is optimized for size and maximum clock frequency, and can be configured to, among other things, perform hardware multiplication, division, and interrupt handling.

5.3.1. RISC-V ISA

RISC-V is an open standard instruction set architecture, based on reduced instruction set computer (RISC) principles [9]. It implements a common load-store architecture, with a small number of instructions that allow for simpler, more efficient designs. It is designed to be modular and extensible, with a small core ISA and multiple standard extensions (Figure 5.2).

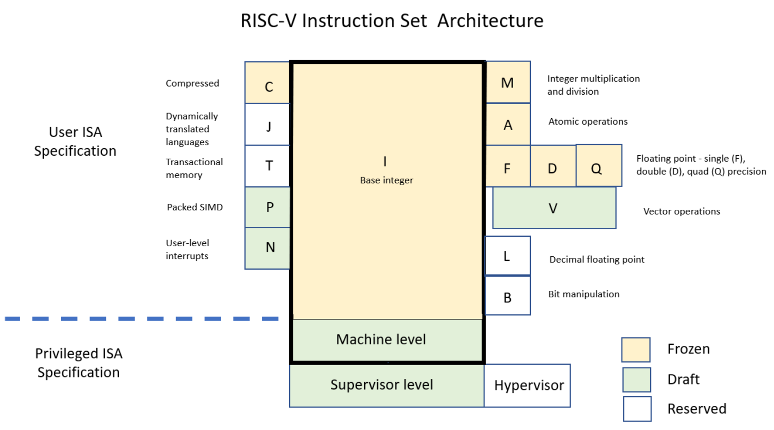


Figure 5.2: RISC-V instruction set architecture extensions [44]

Register name	Symbolic name	Description	Saved by
x0	zero	Always zero	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5	t0	Temporary/alternate return address	Caller
x6-7	t1-2	Temporary	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function argument / return value	Caller
x12-17	a2-7	Function argument	Caller
x18-27	s2-11	Saved register	Callee
x28-31	t3-6	Temporary	Caller
f0-7	ft0-7	Floating point temporary	Caller
f8-9	fs0-1	Floating point saved register	Callee
f10-11	fa0-1	Floating point argument / return value	Caller
f12-17	fa2-7	Floating point argument	Caller
f18-27	fs2-11	Floating point saved register	Callee
f28-31	ft8-11	Floating point temporary	Callee

Table 5.2: RISC-V integer and floating point registers, and ABI usage

Contrary to other architectures, RISC-V was developed with the explicit intent of being an open-source, academic-friendly architecture, with no royalties to be paid. The base specification is intentionally simple, describing instruction encodings, control flow, registers, and integer operations. It is sufficient to implement a general-purpose processor, with a general-purpose compiler to develop software to run on top of it.

The base specification includes 32 general-purpose integer registers, which can be coupled with 32 more floating-point registers if the F extension is implemented. A description of their usage, which can be found in the application binary interface (ABI), is shown in Table 5.2.

As previously mentioned, accessing memory is performed with a load-store architecture: instructions can only operate on registers, and specific load and store instructions are used to read values from and write them to memory, respectively. The byte ordering is little-endian, with the least significant bit coming first and the most significant bit last.

The standard instruction encoding is fixed in size to 32 bits, simplifying the instruction decoding process. The downside is that the resulting code size is larger than other instruction sets. To address this issue, the C extension was added as part of the specification, allowing for a commonly used subset of instructions to be encoded in 16 bits instead [51].

5.3.2. PicoRV32

PicoRV32 is a good candidate for a proof-of-concept target due to its non-trivial complexity and modest size. Being a processor, its runtime functionality can be affected without changing the hardware, allowing for exploration of multiple failure modes. At the same time, it is not prohibitively complex, as that would increase the time required to verify the functionality of each component in the framework. Moreover, if the toolchain can be proven to work on this processor, it is very likely that it will also work on non-programmable designs, with functionality that is fixed in hardware and that do not require any firmware to run.

A block diagram of the main functionality of the core is shown in Figure 5.3. It features a simple, single-stage pipeline, so instructions require multiple clock cycles to execute (its CPI is 3-6 clock cycles per instruction for simple arithmetic, up to 40 cycles for multiplication [52]). It does not support out-of-order execution nor does it feature IEEE-754 floating point support. It connects to memory via a standard AXI-4 interface [1].

The processor configuration chosen included support for hardware multiplication (M extension) and compressed ISA (C exension). The following is a full list of the parameters that were toggled compared

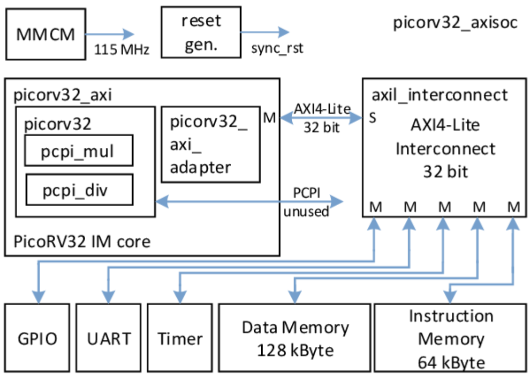


Figure 5.3: Block diagram of the PicoRV32 RISC-V core [31]

AND2X2 cells	584
AOI21X1 cells	1293
AOI22X1 cells	291
INVX1 cells	2065
MUX2X1 cells	254
NAND2X1 cells	2169
NAND3X1 cells	450
NOR2X1 cells	2190
NOR3X1 cells	71
OAI21X1 cells	5456
OAI22X1 cells	327
OR2X2 cells	160
XNOR2X1 cells	86
XOR2X1 cells	57
Total cells	18356

Table 5.3: Synthesis statistics for PicoRV32 in the chosen configuration

to the default configuration:

- **COMPRESSED_ISA**: enables support for the C RISC-V extension.
- **ENABLE_MUL**: enables support for the M RISC-V extension, adding a hardware multiplier to the design.
- **ENABLE_DIV**: enables support for hardware division, also part of the M extension.
- **ENABLE_IRQ**: enables support for external interrupts.

To better indicate the size of the design under exam, post-synthesis metrics are reported in Table 5.3.

Generating software for the processor to run was done through the GCC toolchain, an open-source collection of compilers for different languages and target architectures [4]. The two lanugages used were C and RISC-V assembly, and the target architecture was `riscv32imc`.

6 Netlist to RTL Backtracking

Since netlist-based simulations are closer to the hardware layer compared to RTL-based ones, it is preferable to perform fault injection campaigns at this level, as the results would be more comparable to how the chip will perform once it's etched in silicon.

Netlist fault simulations, however, pose a challenge for a designer interested in hardening their product: because the abstraction level is lower, adding mitigations at this level can be a tedious task. The functional blocks developed during the design process are no longer clearly distinguishable, making it harder to assess what the effects of a fault in a specific cell would be on the overall functionality of the chip.

Of note is that some tools do partially preserve the RTL to netlist mapping information during synthesis. YoSYS, for example, can keep track of flip-flops across each transformation step, and reports the mapping between them and register declarations in the RTL with Verilog annotations. This information is however quite limited: all other logic gates are no longer mappable to the RTL lines that generated them, and distinguishing functional blocks by only looking at the flip-flops they are connected to might be impossible.

The proposed solution to this problem is to reconstruct the mapping information lost during synthesis by means of equivalence checking. The core principle is the following: assuming that the synthesized design exactly implements the functionality described at the RTL level, if the output of a netlist cell and that of an RTL cell are functionally the same, then the two are equivalent, and that portion of the netlist was obtained by synthesizing the corresponding RTL section. This approach has the added benefit of being agnostic to the synthesis tool utilized: its only requirement is that the RTL and netlist are functionally equivalent, which can be verified using formal verification tools.

This chapter will guide the reader through the implementation details of this step: how to find sections of netlist that match lines of RTL code, how to use this information to reconstruct netlist metadata, and the optimization techniques that had to be used in order to reduce the computation time to a reasonable amount. After describing some of the optimization attempts that failed, the results of the tool will be described, comparing the metadata of a pristine netlist with one processed by the tool.

6.1. Implementation

In order to prove that two wires are functionally equivalent, it is necessary to express them as functions of common input variables, which can be mapped via other means between the two levels of abstraction. Once that is done, known algorithms for checking the equivalence of boolean functions can be employed, and when an equivalence is found, an annotation can be added to the netlist wire referring back to the corresponding RTL wire.

The input variables chosen were flip-flops and top-level inputs. The reason is that flip-flops store the value at their input only on the rising or falling edge of the clock, so their output can be considered fixed when determining the boolean function corresponding to a wire. Similarly, top-level inputs are, by definition, inputs to any function expressed by the gates in the netlist, so they too are endpoints when generating boolean functions.

The boolean functions themselves are obtained by recursively backtracking the netlist until only

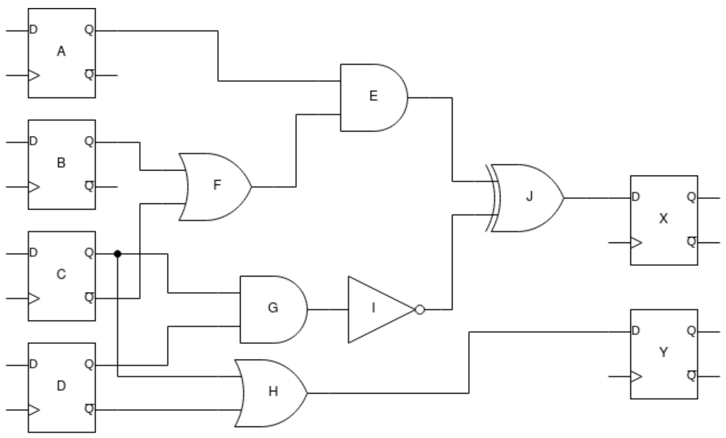


Figure 6.1: Example section of a netlist

input variables are hit, as shown in Algorithm 2. In short, each variable in a given function that is not an input is expanded into an expression that only contains inputs, then the original function is composed with the newly expanded expression.

Algorithm 2 Recursive backtracking

```
Require: $E$ list of variables where backtracking should end
Require: $f$ function corresponding to the wire under exam
Ensure: $f$ is expanded to be only a function of the endpoints
  for $v \in f$ do
    if $v \notin E$ then
      $f_v \leftarrow get\_function(v)$
      $f_{exp} \leftarrow backtrack\_recursive(E, f_v)$
      $f \leftarrow f \cdot f_{exp}$
    end if
  end for
```

As an example, let's assume that a part of the netlist looks like Figure 6.1. A, B, C and D have been identified as backtracking endpoints due to them being flip-flops. Finding the logic function that describes the input of flip-flop X will be performed as follows:

$$\begin{aligned} X &= J && \text{Expand J} \\ &= E\dot{\vee}I && \text{Expand E and I} \\ &= (F \wedge A)\dot{\vee}G && \text{A is an endpoint, expand F and G} \\ &= ((B \vee \neg C) \wedge A)\dot{\vee}(C \wedge D) && \text{All variables are endpoints, no expansion needed} \end{aligned}$$

Similarly, the function at the input of Y is derived as follows:

$$\begin{aligned} Y &= H && \text{Expand H} \\ &= C \vee D && \text{All variables are endpoints, no expansion needed} \end{aligned}$$

This operation is repeated for every wire in the netlist, until all of their input functions are determined. The time complexity of this algorithm is $\mathcal{O}(v_c)$, proportional to the size of the combinational network the initial variable is part of, for each wire analyzed.

Given the core principle under which the mapping will be performed, creating an algorithm that will check each possible pair of wires in the RTL and netlist for equivalence is fairly simple: for each possible pair of netlist and RTL wires, get their corresponding boolean variables, express those as function of only input variables, then check if the two are equivalent (Algorithm 3).

This algorithm is quite inefficient however. Checking whether two functions are equivalent, as explained previously, has a time complexity exponential with the number of variables involved, and this

Algorithm 3 Naïve annotation recovery

```
Require: $N$ list of netlist wires, $R$ list of RTL wires
Require: $E_n$ list of netlist endpoints, $E_r$ list of RTL endpoints
  for $w_r \in R$ do
    $v_r \leftarrow get\_variable(w_r)$
    $f_r \leftarrow backtrack\_recursive(E_r, v_r)$
    for $w_n \in N$ do
      $v_n \leftarrow get\_variable(w_n)$
      $f_n \leftarrow backtrack\_recursive(E_n, v_n)$
      if $f_n \equiv f_r$ then
        $annotate(w_n, w_r)$
      end if
    end for
  end for
```

algorithm has to perform $\mathcal{O}(|R| \cdot |N|)$ checks for equivalence, a number proportional to the product of the number of wires in the RTL and netlist. This situation worsens as the complexity of the RTL and netlist increase: not only are the functions to be compared larger on average, but there will be more of them, as every sub-section of each has to be checked against every other. In order to improve the time necessary to analyze a given design, since the equivalence checking algorithm cannot be performed in sub-exponential time, it is of utmost importance to reduce the number of comparisons as much as possible.

6.1.1. Algorithm optimization

A first major improvement can be obtained by using the mapping information that is preserved by the synthesis tool: in general, registers and ports declared at the RTL level are kept unchanged in the resulting netlist, and their names in the netlist are based on their names in the HDL code. This information can be used to build an initial equivalence relation between the synthesized design and the RTL, which can in turn be used to speed up annotation recovery.

By using Algorithm 4, it is possible to obtain a set of all variables obtained when backtracking from a given starting point towards input variables. The initial equivalence relation can then be exploited as shown in Algorithm 5.

Algorithm 4 Variable collection algorithm

```
Require: $f$ function to collect the variables of
Require: $E$ set of endpoints to stop variable collection at
Ensure: $V$ set of all variables encountered when backtracking to variables in $E$
  $V \leftarrow \{\}$
  for $v \in f$ do
    $V \leftarrow V \cup v$
    if $v \notin E$ then
      $f_v \leftarrow get\_function(v)$
      $V \leftarrow V \cup collect\_variables(f_v, E_n)$
    end if
  end for
```

By starting from wires that are known to be equivalent, the variables collected during backtracking will be the only ones that can potentially be equivalent to each other. This reduces the search space for a RTL wire from the entire netlist down to the logic cone behind an equivalent flip-flop in the netlist.

Yet another optimization was obtained by using the notion of cut-points in each logic cone. If two cones being compared share two equivalent sub-cones, the backtracking step can be halted early, since the two output variables of the sub-cones are known to be equivalent. The two shared comparison points are known as cut-points.

The concept of cut-points is used extensively by all formal verification tools to tame the exponential

Algorithm 5 Faster annotation recovery

```
Require: $N$ list of netlist wires, $R$ list of RTL wires
Require: $E_n$ list of netlist endpoints, $E_r$ list of RTL endpoints
Require: $Eq$ list of $(rtl, netlist)$ pairs of a priori equivalences
  for $(w_r, w_n) \in Eq$ do
    $v_r \leftarrow get\_variable(w_r)$
    $v_n \leftarrow get\_variable(w_n)$
    $R_v \leftarrow collect\_variables(v_r, E_r)$
    $N_v \leftarrow collect\_variables(v_n, E_n)$
    for $v_r \in R_v$ do
      for $v_n \in N_v$ do
        $f_r \leftarrow backtrack\_recursive(E_r, v_r)$
        $f_n \leftarrow backtrack\_recursive(E_n, v_n)$
        if $f_n \equiv f_r$ then
          $annotate(w_n, w_r)$
        end if
      end for
    end for
  end for
```

complexity inherent in the equivalence checking algorithm, and can be extremely effective at accelerating the verification process.

As an example, consider a netlist logic cone with n inputs being checked against an RTL section with the same number of inputs n. Checking these two for equivalence will require at worst $\mathcal{O}(2^n)$ variable assignments, and assuming a constant evaluation time, will have an overall complexity of $\mathcal{O}(2^n)$. Now assume that an equivalence point pair (e_r, e_n) is found within the two, corresponding to a sub-cone involving k variables in the netlist and a similar amount $\Theta(k)$ of variables in the RTL (a reason why the quantities might differ is that a simplification occurred in the synthesis process and removed a portion of the RTL that did not affect the value of the point under exam). Knowing the existence of this equivalence between sub-cones, it is no longer necessary to calculate the value of the function leading up to those points, and (e_n, e_r) can be considered an input variable pair, just like any flip-flop at the input of the logic cone. The result is that $\mathcal{O}(2^k)$ operations are saved for each comparison involving e_n and e_k.

To better visualize the advantage of this approach, consider the two logic functions represented in Figure 6.2. Under normal conditions, verifying whether the two implement the same function would require expanding all the gates in the RTL block F prior to evaluation. If, however, the red wires in the RTL and the netlist are found to be equivalent, the number of operations can be reduced: the expansion does not have to be performed for one of the inputs of gate 2.

The process of determining which pairs of variables are valid cut-points does not require any extra processing steps compared to the previous implementation: sub-cones are checked for equivalence regardless, so all this step requires is to change the order in which points are checked. By looking at variables that are fewer gates away from the input wires before those that are closer to the outputs, cut-points can be found and exploited to accelerate the annotation recovery process.

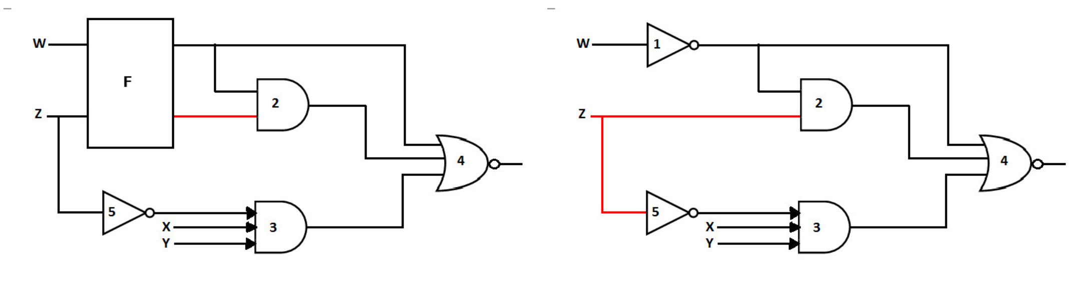


Figure 6.2: Sample RTL and hypothetical netlist

In order to implement this, the backtracking step for the netlist was halted at both known-equivalent variables and variables that may have an equivalent, while the backtracking for the RTL had to be performed fully. Then, each netlist function that only contained known equivalents was checked against all RTL functions. If no equivalence was found for a given variable, all netlist functions that had that variable as one of their inputs was updated, substituting the variable with its function. This process was repeated until no netlist variables were left to be checked. The resulting algorithm is shown in 6.

Algorithm 6 Cut-point based optimized annotation recovery

```
Require: $N$ list of netlist wires, $R$ list of RTL wires
Require: $E_n$ list of netlist endpoints, $E_r$ list of RTL endpoints
Require: $Eq$ list of $(rtl, netlist)$ pairs of a priori equivalences
  for $(w_r, w_n) \in Eq$ do
    $v_r \leftarrow get\_variable(w_r)$
    $v_n \leftarrow get\_variable(w_n)$
    $R_v \leftarrow collect\_variables(v_r, E_r)$
    $N_v \leftarrow collect\_variables(v_n, E_n)$
    $N_{unchecked} \leftarrow N_v$
    $B_n \leftarrow \{\}$
    for $v_n \in R_n$ do
      $B_n \leftarrow B_n \cup \{backtrack\_recursive(E_n \cup N_{unchecked}, v_n)\}$
    end for
    for $f_n \in B_n$ do
      if $\forall v_n \in f_n, v_n \in E_n$ then
        $eq \leftarrow 0$
        for $v_r \in R_v$ do
          $f_r \leftarrow backtrack\_recursive(E_r, v_r)$
          if $f_n \equiv f_r$ then
            $annotate(w_n, w_r)$
            $E_n \leftarrow E_n \cup \{f_n\}$
            $eq \leftarrow 1$
          end if
        end for
        $N_{unchecked} \leftarrow N_{unchecked} \setminus \{v_n\}$
        if $eq = 0$ then
          for $f_n \in B_n$ do
            $f_n \leftarrow backtrack\_recursive(E_n \cup N_{unchecked}, f_n)$
          end for
        end if
      end if
    end for
    for $v_r \in R_v$ do
      for $v_n \in N_v$ do
        $f_r \leftarrow backtrack\_recursive(E_r, v_r)$
        $f_n \leftarrow backtrack\_recursive(E_n, v_n)$
        if $f_n \equiv f_r$ then
          $annotate(w_n, w_r)$
        end if
      end for
    end for
  end for
```

To exemplify, consider the netlist from Figure 6.1. The algorithm will start by expanding all variables one step back: $J = E \dot{\vee} I, E = A \wedge F, F = B \vee \neg C, I = \neg G, G = C \wedge D, H = C \wedge \neg D$. The only known equivalents at the beginning of the algorithm are the flip-flops A, B, C, D, X, Y, and the only functions that only contain these variables are $F = B \vee \neg C$, $G = C \wedge D$ and $H = C \wedge \neg D$. These three functions will be checked against all known RTL functions in the relevant logic cone for equivalences.

Let's assume that a known equivalence is found for F and H, but not for G. The set of known equivalences will then be updated to contain A, B, C, D, X, Y, F, H. Next, each function that was not analyzed will be expanded once again, resulting in $J = E \dot{\vee} I, E = A \wedge F, I = \neg(C \wedge D)$. Note that E and I are not expanded in the functions that contain them, because it is still unknown whether they are equivalent to any part to the RTL. The functions checked for equivalence in this iteration will be $E = A \wedge F, I = \neg(C \wedge D)$. Note that the function for E does no longer have to be expanded fully: F was found to be equivalent, so the complexity of the function being analyzed now is lower.

Once again, let's assume that an equivalence is found for E but not for I. The new set of known equivalences is $A, B, C, D, X, Y, F, H, E$. The last function to be analyzed, $J = E \dot{\vee} I$, is expanded into $J = E \dot{\vee} \neg(C \wedge D)$, and checked for equivalence against the RTL.

The worst-case computational complexity is unchanged compared to the previous algorithm: if no equivalences are found, each wire's boolean function representation will have to be expanded until it only contains flip-flops, and checked for equivalence. The average case, however, was found to be much faster than the previous algorithm. This is because the netlist is derived from transformations applied to the RTL, so the probability of an intermediate equivalence point being found is high. Due to the exponential complexity associated with checking the equivalence of two functions, finding even one point of equivalence greatly reduces the time necessary to complete the analysis.

6.1.2. Equivalence checking with Conformal

Since equivalence checking was essentially at the core of the algorithm used to perform annotation recovery, and it took up a significant portion of the total runtime, an effort was made to use the equivalence checking core available in Cadence, called Conformal [2]. This, however, did not succeed, as the inputs to the tool have to be standalone modules and the objects being checked for equivalence are just portions of modules. The only way to perform this step would have been to extract each logic tree from the netlist and the RTL, build dummy modules around them, and pass those to Conformal, which would have taken a significant amount of time to implement.

6.1.3. Binary Decision Diagrams

As mentioned in chapter 3, BDDs are a viable way to check for equivalence: given two logic cones, equivalence between them can be proven by representing both as BDDs, then bringing them into a canonical form, and checking whether the two are the same. The comparison process itself has a complexity of $\mathcal{O}(n)$, linear with the number of nodes in the BDD.

The issue with this approach is that the complexity of the BDD associated with a boolean formula is greatly dependent on the variable ordering, and the process of reducing a BDD into a canonical form has a complexity dependent on the complexity of the initial BDD. Moreover, "Some functions cannot be represented efficiently with [a BDD] representation regardless of the input ordering. Unfortunately, the functions representing the output bits of an integer multiplier fall within this class" [16].

While these issues do not immediately preclude the ability to use BDDs to perform equivalence checking, they caused the library used to operate on boolean functions to seemingly stop working for long periods of time, so the SAT approach was used instead.

6.2. Results

6.2.1. Evaluation parameters

In order to evaluate the results from the annotation recovery step, the number of annotations present before and after synthesis was collected first, and then compared to the number of annotations after the algorithm was run. The target selected was PicoRV32, configured to include support for the compressed ISA, hardware multiplication and division, and external interrupt requests (IRQs).

The thus configured RTL was then synthesized by YoSYS using the script shown in section A.1, which also dumped intermediate outputs during the synthesis process. The number of source annotations during the synthesis process was then extracted by counting the number of occurrences of `attribute \src` in each file

Finally, the number of unique RTL lines referenced at each stage was extracted using the following command, which first extracts all the source annotations, then extracts the quoted string from them containing the lines referenced, then splits that line so that each reference is treated separately, and finally removes any leading or trailing spaces and counts the unique occurrences:

Stage	Source annotations	Unique line references
0_postbegin.ilang	2906	916
1_postcoarse.ilang	2023	623
2_postopt.ilang	1949	606
3_posttechmap.ilang	15982	615
4_postopt.ilang	15982	615
5_postlibmap.ilang	1751	221
6_postopt.ilang	1751	221
7_postabc.ilang	4761	221
8_postflatten.ilang	4753	217
9_postclean.ilang	1657	138
netlist.ilang	1657	138

Table 6.1: Number of annotations and unique line references during synthesis

```
for file in *.ilang; do echo -n "$file "; grep "attribute \\\\src" $file | cut -d"\""
    -f2 | sed -e ’s/|/\n/g’ | awk ’{$1=$1};1’ | sort -u | wc -l; done
```

The results of running the two commands are summarized in Table 6.1.

Contrary to expectations, the number of referenced lines is not monotonically decreasing as the synthesis process is run to completion: from step 2 to step 3 both the number of annotations and the number of unique references increases. By analyzing the output of step 3, it can be seen that multiple references are added to an external file, `techmap.v`, as YoSYS is mapping the high-level modules from the RTL to an internal cell library, which is then used to perform subsequent transformations. This explains why the number of annotations increased during that step. Similarly, step 7 saw a large increase in source annotations, while the number of unique references remained the same. This is because step 7 involves the use of ABC, a tool that performs sequential synthesis and optimization[40]. Looking at the differences compared to the output of the previous step, the most likely reason for this increase is that YoSYS copies all the annotations from the cells input to ABC to the resulting outputs, which are more numerous because of the sequential synthesis performed.

Given this baseline, an evaluation of the performance of the algorithm from section 6.1 can be done by looking at the two metrics above for the netlist output by the tool: higher metrics are better, and the ideal case would result in the same number of unique RTL references as the first intermediate file in the synthesis process.

6.2.2. Tool output

The intermediate file used as the RTL reference was `4_postopt.ilang` for two reasons. First, it contains only modules, cells, wires and sigspec objects, which is a requirement for the tool to parse the given intermediate language file and is not the case for the first intermediate file. Secondly, it uses a known set of modules from the YoSYS cell library, all of which can be expressed as boolean functions. This allows the expression of any signal in that file as a boolean function of inputs to the module and flip-flop states, which is necessary to perform equivalence checking.

The number of signals matched based on their name was 2191, which was lower than expected since the design had 2495 flip-flops. This discrepancy was caused by multiple flip-flops in the multiplier module having names that could not be found in the RTL. Manually inspecting the two files revealed that two register banks, `rs1` and `rs2` were renamed to `next_rs1` and `next_rs2` during the synthesis process. Other variables were renamed during synthesis too, preventing them from being matched by comparing names directly. This limited the number of equivalences that can be found, as the number of known equivalent points was reduced.

In total, 330 equivalences were found between RTL and netlist sub-cones, Of these, 281 resulted in new source annotations being added to the netlist, and the number of unique RTL lines of code referenced increased by 61. This brought the number of cells with source annotations from 0.75% to 1.08%. The collected statistics for the annotated output are shown in Table 6.2.

Stage	Source annotations	Unique line references
Netlist	1657	138
Post-annotation	1938	199

Table 6.2: Number of annotations and unique line references post-annotation

6.3. Discussion

Source annotation recovery by means of equivalence checking seems to be a promising avenue to aid in design hardening. The results shown indicate that there is room for improvement, since the number of new annotations added by the tool was only 44% over the ones found in the original netlist. This could be improved by addressing limitations of the current tool, such as its inability to recognize flip-flops that were renamed by the synthesis process.

Despite these limitations, the results prove that this is a worthwhile method of remapping portions of the netlist to parts of HDL code. The main appeal of this approach is that it is vendor-agnostic, so the netlist output of any synthesis tool could be augmented for hardening purposes. It would enable netlist-level fault injection results to drive RTL-level hardening in a more effective way, without changing the existing synthesis workflow.

7 Fault Injection

In this chapter, the second part of the framework will be described: how to identify hardware faults that will result in successful exploitation and potential security vulnerabilities. First, the results expected by the fault injection campaign will be described, indicating the limitations that had to be introduced to prevent an explosion in complexity. Then, three different layers of abstraction will be described. These were employed to better manage the difficulty of the problem at hand, as well as to gain insight into the vulnerability of the target at different levels. Following that, a description of the envisioned workflow, that ties the analysis processes mentioned up to that point together, will be given. Finally, for each layer, the implementation details will be discussed, together with optimization techniques employed, failed attempts, and the results of applying the workflow described against the given target.

7.1. Outcome Characterization

An ideal output of a fault injection campaign would describe, for each clock cycle and each possible injectable glitch, the result in terms of high-level netlist functionality, such as whether the netlist halted execution, its execution flow changed, or it entered an infinite loop. This allows a designer to quickly identify faults that may result in security vulnerabilites, reducing the number of glitches to investigate more in depth and potentially mitigate.

Identifying whether a glitch caused a netlist to halt or enter an infinite loop, commonly referred to as deadlock and livelock conditions, in general requires a priori knowledge on which states result in the netlist entering these states. This is because determining if a Turing machine has halted is an undecidable problem, and a processor is approximately a Turing machine for all intents and purposes [19]. For simpler netlists, which implement smaller finite state machines, a complete state space exploration can be performed to identify states that result in a halt, but that approach cannot be generalized to more complex designs such as processors.

In order to gain useful insights from static analysis, and bypass the issues introduced by the halting problem, a different, lower-level indicator of successful faults was chosen: a fault was considered successful if the netlist trace deviated in any way from the reference, non-glitched one. This deviation would then be dynamically analyzed against a target application later in the glitch injection campaign to determine what the effect of the glitch was on the execution flow.

7.2. Abstraction Layers

Sub-dividing the fault injection process into multiple layers was helpful in reducing the complexity of this step: instead of having to exhaustively search the entire state space of the netlist for all glitches that are successful, or randomly glitching the board looking for possible fault injection vulnerabilities while it's running, static analysis can be performed first to gain insight into which glitches are worth looking into, reducing the time necessary and increasing coverage compared to the random search approach. This process has the added benefit that the information learned in lower-level layers is independent from the higher layers, making it possible to test, for example, multiple applications running on the same processor without re-analyzing the effects of each glitch at the hardware level.

A high-level overview of the different layers of abstraction is represented in Figure 7.1.

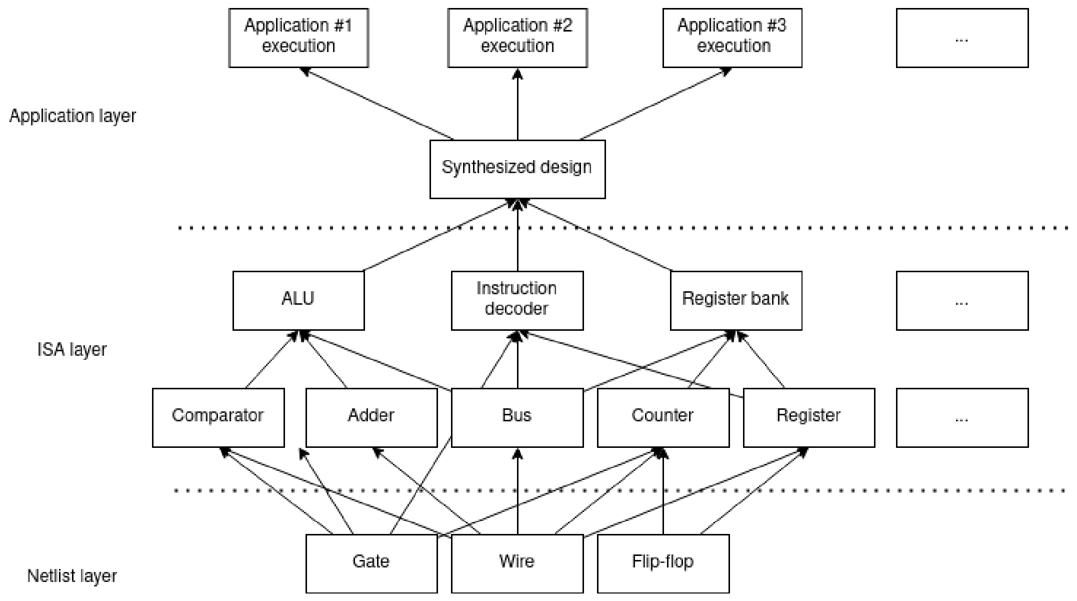


Figure 7.1: Examples of division of components into abstraction layers

7.2.1. Netlist Layer

At this layer, all that exists are individual flip-flops and gates. No higher-order groupings are present.

The propagation of physical faults injected is performed at this level of abstraction. Each wire is glitched based on some predefined strategy (one at a time or multiple at a time, only for one clock cycle or for multiple clock cycles), and each attempt is considered successful if the resulting state and/or output is different from the one reached by the golden reference. The reason why the outputs are also important is that transient faults on the outputs could be clocked into external memory cells or flip-flops, making the fault somewhat permanent. This depends on the type of component connected to the outputs, so some of the transient faults at the outputs could be ignored, but for simplicity here it is assumed that transient faults on any output are potentially successful.

Since glitches may or may not be masked by the state at the time of the glitch, a set of potential states can be provided. These valid states would have to be determined in higher levels of abstraction, where each flip-flop is assigned a meaning and invalid combinations of bits can be identified. Alternatively, a set of constraints on the state that must be satisfied for the glitch to be successful can be created during the injection process. These constrains will be on a set of individual wires at first, and can be expanded to be requirements purely on the state variables using the same process adopted when performing annotation recovery.

The types of faults that can be injected in a design are multiple. Since analyzing the effects of all of them would be impractical, the only fault profile analyzed was that of single event upsets, which result in bit flips. These can be modeled by forcing the affected wire to assume the opposite value, and attempting to propagate this state flip to a state, creating constraints on the fly in order for this propagation to occur.

The process of propagating the fault to the outputs may cause conflicting requirements on the state. In that case the glitch cannot occur at all (if the chosen strategy only allows glitching one wire per clock cycle), or requires another glitch to be injected in order to manifest itself in the next clock cycle.

It may also cause requirements on the glitch itself, forcing it to be a specific value in order to be propagated to the state. This would be interpreted as an ability for an attacker to glitch the value of a specific state only when the wire they are trying to glitch assumes a certain value, which is perfectly valid.

7.2.2. ISA Layer

At this layer, cells and interconnects are grouped into higher-order constructs, such as registers, wire buses, and adders, and assigned a logical interpretation based on those groupings. With the knowledge of larger constructs, we can define which states are legal, i.e. reachable during normal operation, and which are not. Examples of illegal states include: multiple bits set on a one-hot encoded bus, conflicting redundant states, or counters out of normal execution bounds.

Fault injection at this level would be performed by manipulating the state of the board rather than the values of individual flip-flops. A fault is considered successful if the netlist does not reach an illegal state after injection. Of note is that the definition of “legal state” used here is quite loose. Ideally, all states in which the board is not completely stuck would be considered legal: some faults might temporarily cause the board to temporarily enter a state that is not reached during nominal execution, but not halt. The resulting definition would however be hard to verify in practice: the looser the definition, the harder it becomes to determine whether a state is legal a priori. If the goal was to define which states are legal according to the aforementioned definition, an exploration of the entire state space would have to be performed, noting which state combinations cause the board to halt indefinitely.

To simplify the analysis, it is assumed that the netlist under exam is working correctly if the state is legal, and all states that contain inconsistencies or invalid values cause the board to halt and are considered illegal. In theory, however, any definition of legal and illegal state could be used.

7.2.3. Application Layer

This is the topmost layer of abstraction analyzed, where the application that the netlist is expected to run is executed. Examples include a set of instructions in the case of a programmable processor, or a set of states in the case of a hard-coded finite state machine.

Fault injection at this level aims to find which operations are susceptible to hardware faults and could lead to successful exploitation of the board. For this reason it is specific to each application, and the states reached during its execution. Example of successful exploitations include: a conditional branch instruction jumping to the target address when it shouldn’t, an instruction becoming a no-op, or a finite state machine transitioning to an unlocked state without performing the necessary checks.

In order to identify the effects of a fault on the high-level operations being executed by the board under exam, some level of insight into the functions implemented by the netlist is necessary. More specifically, it is nontrivial to determine whether a glitch causes the board to completely halt, or to execute a sequence of operations in an infinite loop. Without this insight, it is also impossible to know whether a fault that causes the execution to diverge from the reference is successful under the narrower success definition of “causes the processor to misbehave in ways that compromise the security of the design”, which is more reflective of the typical definition of a successful glitch.

For this reason, it is assumed that the designer knows which signals in the netlist are symptoms of a indefinitely halted state, and which signals are critical for secure execution at which timestamps. These would both be supplied to the application-level analysis algorithm, which can thus identify faults that cause the netlist to halt and highlight those that affect critical signals.

7.3. Workflow

Fault injection could be performed on each layer separately, and the results combined at the end to obtain a list of potential vulnerabilities. This is not particularly efficient however, and unnecessarily complex, since a lot of insight gained in a layer this way may not be applicable to the other layers due to constraints defined in those layers.

A better approach is to perform fault injection on one layer, then propagate the results up or down the layers, progressively adding constraints to the faults found. An example of a possible workflow through the layers of abstraction is the following:

- **Netlist layer**: perform a full fault injection campaign, simulating the effects of any possible glitch of interest and recording both the targeted states and the constraints necessary for the glitch to propagate to those states.

- **ISA layer**: given the list of faults generated in the previous step, remove glitches the requirements of which cannot be satisfied by any legal state, or that bring the netlist into an illegal state.

- **Application layer**: given the list of glitches from the previous steps, and a set of states that should not be reached during normal operation (or transitions that should not occur), simulate the application on the netlist, and verify whether those states can be reached by means of fault injection.

This will be the workflow implemented in the following sections. The only discrepancy is that, due to time limitations, the ISA layer was not implemented, as the results obtained at the netlist layer could be immediately utilized during simulation at the application layer. The disadvantage of this approach is that some of the glitches reported as successful at the netlist layer will never result in successful exploitation at the application level, as they might depend on an invalid state to propagate in the first place. This was not a major issue in the selected test platform thanks to its limited size, but could have the potential of slowing down the application layer analysis as the size of the design under examination increases.

7.4. Netlist Layer

7.4.1. Implementation

The method used to obtain a set of faults that can propagate to a state or output variable is based around path sensitization, i.e. given a candidate wire that should be glitched, determine which values the other wires should assume for it to propagate to a target state.

As an example, consider the simple combinational network shown in Figure 7.2.

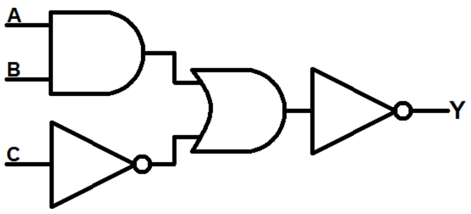


Figure 7.2: Simple combinational network

Assume that Y is the input to the target state, and A is the wire that will be targeted by a bit flip. In order for A to propagate to Y, it should first propagate through the AND gate, then through the OR gate, and finally through the inverter. To propagate its value through the AND gate, it must be that $B = 1$, otherwise the output would always be 0 due to the way the logical AND function operates. Similarly, for the bit flip to affect the output of the following OR gate, it must be that $\overline{C} = 0$, meaning that $C = 1$. The final propagation through the inverter does not add any extra requirements. The final list of requirements for a bit flip on A to propagate to Y is therefore $B = 1 \wedge C = 1$.

This procedure is known as path sensitization [8], and is used both in hardware and software testing tools to find which condition is necessary for a specific line of code to execute, or for a specific wire to assume a predetermined state.

The simplest approach to testing a digital circuit would be to generate all possible input patterns, and exhaustively check whether each can excite the target location. The problem is that the same issue found when performing equivalence checking is present here: the time complexity increases exponentially with the number of input variables. Unlike in equivalence checking, however, the scope of testing is more narrow: given a fault on wire x, the goal is to prove that there is a combination of states that cause this fault to propagate to a predetermined target. Rather than considering the logic network as an single black box, one can proceed one gate at a time, identifying whether there is a set of inputs to that gate that causes the fault to appear at its output.

This path sensitization procedure is at the core of the algorithm used at this stage. In order to extend the process to any gate, the algorithm reported in Algorithm 7 was implemented. The library function $satisfy\_all$ was used to generate a comprehensive list of all input combinations that resulted in the output being equal to 1 and 0. Then, the lists were compared against each other, and combinations that were present in both with only the targeted variable flipped were collected. The obtained sets were then trivially converted into conjunctive normal form expressions (given the set $\{\{a = 0, b = 1\}, \{a = 1, c = 1\}\}$, the resulting expression would be $\overline{a} \wedge b \vee a \wedge c$), and OR'd together to obtain the final requirement for a given variable to propagate to the output of a given cell.

The aforementioned procedure to sensitize a cell's output Y to a glitch in an input wire V was then applied to all cells with Y as an input, and the constraints obtained for each were then AND'ed with the pre-existing constraints on Y to obtain the constraints necessary for V to propagate one layer further. This process was repeated until the affected output variables were all inputs to states or output

Algorithm 7 Path sensitization algorithm

```
Require: $f$ boolean function of the output of the given cell.
Require: $v$ variable that will be glitched.
Ensure: $C$ set of all input variables that sensitize the output of $f$ to $v$.
  $P \leftarrow satisfy\_all(f)$
  $N \leftarrow satisfy\_all(Not(f))$
  // Invert the value of $v$ if present in any $n \in N$, so that $P$ and $N$ can be compared
  for $n \in N$ do
    if $\{v = 0\} \in n$ then
      $n \leftarrow n \setminus \{v = 0\} \cup \{v = 1\}$
    end if
  end for
  $C_+ \leftarrow \{\}$
  $C_- \leftarrow \{\}$
  for $n \in N$ do
    if $\exists p \in P, n \subset p$ then
      $C_+ \leftarrow C_+ \cup (n \setminus \{v = 0\})$
    end if
  end for
  for $p \in P$ do
    if $\exists n \in N, p \subset n$ then
      $C_- \leftarrow C_- \cup (p \setminus \{p = 1\})$
    end if
  end for
  $C \leftarrow$ CNF$(C_+) \vee$ CNF$(C_-)$
```

variables.

The final product of this analysis was a set of constraints indicating the conditions necessary for a glitch in any given variable to reach all states combinationally connected to it. These constraints are not functions of just state variables, but of intermediate variables as well (generated by combining states using logic functions). Because of this, some constraints may not ever be satisfiable, and the glitch may never propagate to the designated state: the relation between different intermediate variables may be such that they cannot assume the required values at the same time, but that relation can only be determined by backtracking each variable using the approach delineated in chapter 6.

Removing these false positives, however, requires checking each constraint for satisfiability: it is necessary to prove that the expression obtained by AND'ing all constraints cannot be verified by any set of inputs. This requires processing time exponential with the number of variables in the expression under exam. Moreover, the values of the wires inside each combinational network were recorded by the simulation tool selected, and could be accessed directly when checking which glitches can propagate for a specific trace, lifting the requirement of expressing each constraint only in terms of inputs and flip-flop variables.

7.4.2. Static Instruction set Analysis

The first idea to approach the problem of analyzing an application running on the processor was to statically analyze the fault profile of each instruction in the ISA separately, and then perform fault injection directly at the software level.

This would have had the advantage of being a fully static analysis, requiring no simulation at the hardware level from external tools and reducing the time necessary to analyze any given application. It could have also produced insight into which instruction were most vulnerable at the hardware level, which in turn could have been exploited at the software level by performing checks right before and after these instructions.

The main issue with this approach is that glitches depend not only on the instruction being decoded, but may also succeed or fail based on the data the instruction is operating on, the address from which the instruction is read, and other states inside the processor, especially for a pipelined design (this was not the case for PicoRV32, but the goal was to create a workflow that would work on any processor).

7.4.3. Semi-static Instruction Set Analysis

Once the infeasibility of a purely static analysis was established, the plan of action shifted to identifying which wires are only dependent on the instruction being decoded, and using those to reduce the constraints to simpler expressions. In order to do this, the following step-by-step solution was created:

1. Assume that the CPU is at the beginning of the fetch phase. This means assuming values for some states in the CPU which govern the instruction phase it's in. For PicoRV32, the state `picorv32_core.cpu_state` was assumed equal to `b01000000`, a value that can be found at `picorv32.v:1152`.
2. Mark the data bus from memory as instruction-dependent.
3. Perform one simulation step.
4. Check which registers only depend on previously marked values, mark those as instruction-dependent.
5. Repeat steps 3 – 4 until `picorv32_core.cpu_state` changes.

The resulting list of marked states would be the set of instruction-dependent values after the fetch phase. Effectively, it would have been a list of the output wires for the instruction decoder, the value of which could be computed independently for each possible instruction.

All the propagation constraints for each glitch would then be restricted against the flagged values, and this would result in one of the following:

1. The constraint was reduced to TRUE: this would mean that the given glitch would always propagate regardless of data for that instruction.
2. The constraint was reduced to FALSE: this would indicate that the glitch under exam would never propagate when executing that instruction regardless of data.
3. The constraint was reduced to another boolean expression, or did not change at all: in this case, the glitch would be truly data-dependent when executing the given instruction.

Cases 1 and 2 would have given the same level of information that the previous attempt would have given: glitch propagation information dependent only on the instruction being decoded. Glitches that resulted in case 3 would have to be analyzed dynamically however, filling in the values of the remaining variables in the constraints based on a simulation of the netlist (hence the semi-static nature of this analysis).

The main issues faced with this approach were during the identification of the values that were purely instruction-dependent. First and foremost, marking the data bus as instruction-dependent at the beginning of the propagation was not enough, as the output of the instruction decoder depends on other internal states as well, such as whether the CPU is processing an interrupt, or if it has thrown an exception. These internal states cannot be assumed to have specific values for each instruction without loss of generality for the analysis, meaning that no constraint can be truly only instruction-dependent.

7.4.4. Simulation Optimization

As previously mentioned, Icarus Verilog was the simulation tool of choice. It is reliable and provides the necessary output. It is not, however, particularly fast, and can only run on a single core. Since simulating the injection of a single glitch took up to 30 minutes on a Core i5-7200U processor, an alternative, better optimized simulator was sought.

A promising project is Verilator, an alternative open-source simulator that claims to "outperform many commercial simulators" [24]. It does so by translating Verilog descriptions into C++, and then compiling the resulting source into an executable optimized for the platform where the simulation is performed. The advantage is that by doing so the processor is no longer interpreting the design as it is simulated, but interprets it once and then compiles it into native machine instructions. It also supports running the simulation on multiple threads, scaling linearly with the number of concurrent threads.

Preliminary results showed a large performance benefit, reducing the time taken to simulate the board without glitches by up to 90%. The fatal limitation of this tool, however, is that the trace it generates does not include all intermediate combinational signals, but only flip-flop outputs and top-level ports. Intermediate signals are mandatory for the application-level analysis to be correctly performed however, and the only way to obtain these values without changing the source code of Verilator itself was to create a new top-level port connected to all of these signals, thus forcing the simulation tool to also record the states of these signals.

Adding this extra port significantly decreased the simulation performance, resulting in a simulation time in line with the one produced by Icarus Verilog. The most likely explanation is that most of the performance improvements obtained by Verilator are in simplifications applied to the combinational networks in the design, resulting in all of the intermediate signals being inaccessible. Connecting all wires to a top-level port prevented Verilator from applying these optimizations, resulting in next to no performance benefit.

Similarly, adding an XOR gate to each wire, and tying the other input of each gate to a top-level port in order to arbitrarily flip bits in the design during simulation, caused the performance improvements introduced by Verilator to all but disappear. For this reason, Icarus Verilog was used both during the collection of golden reference traces and during the fault injection verification process.

7.4.5. Results

While the results are meant to be used as a basis for further analysis, insight into the vulnerability of a design can be inferred from netlist-level analysis alone.

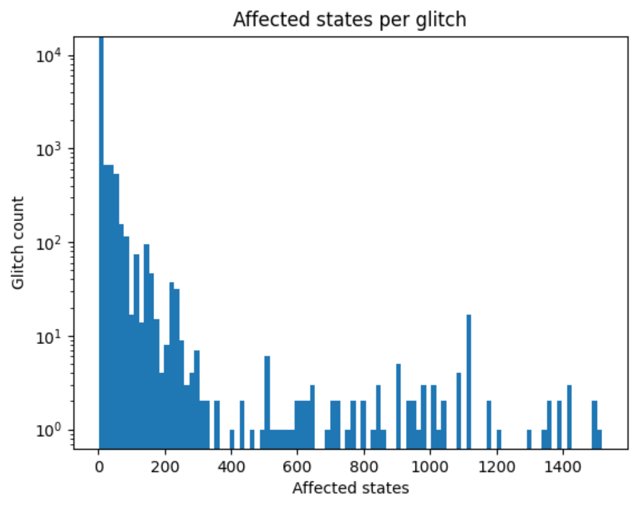


Figure 7.3: Potentially affected states per glitch

Some insight can be gained by plotting a histogram of the number of state variables affected, as shown in Figure 7.3. The mean number of states affected per glitch was 16.0, and the median 2.0. The fact that the distribution is this heavily skewed towards the left side suggests that most faults only affect a small number of states.

The spike above 1000 states are due to glitches that affect most flip-flops in the register file, which has $32 \cdot 32 = 1024$ flip-flops. Examples include glitching `latched_rd`, which contains the index of the register that should be modified by the current instruction. All of these glitches also affect other internal states that are not part of the register file, such as `cpu_state`, hence why there is no defined spike at $x = 1024$.

Each filp-flop can be individually affected by glitching its data input, and since there are 2495 flip-flops in the design, around 25% of the glitches in the first bin are due to these interconnects. The rest of them, on the other hand, comes from glitching combinational logic interconnects, and could be protected against by adding redundancy to computations, either in terms of duplicated logic, or by adding parity checks on the outputs.

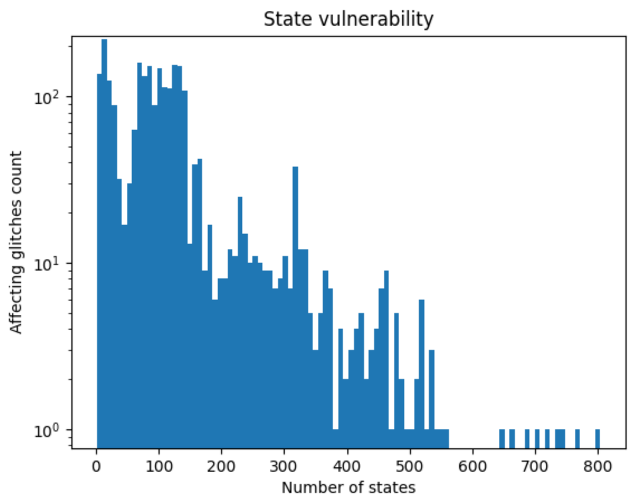


Figure 7.4: Number of glitches affecting each state

Plotting the inverse histogram (glitches per state rather than states affected by each glitch) yields some more insights: by checking how many glitches can affect a single state, a rough estimate of the vulnerability of the design is possible. The further left the distribution is, the fewer glitches can affect each variable, making it harder for an attacker to affect multiple states in a single clock cycle.

As can be seen from Figure 7.4, the PicoRV32 design analyzed has a large number of states that can be affected by multiple glitches. The mean value was 113.9 glitches per state, and the median 97.

To push the distribution further to the left, a designer should intervene on interconnects that, if glitched, have the largest impact on the overall state of the design, hardening by either moving those interconnects further down the layers stack, or splitting the function being computed so that each part affects a smaller subset of states.

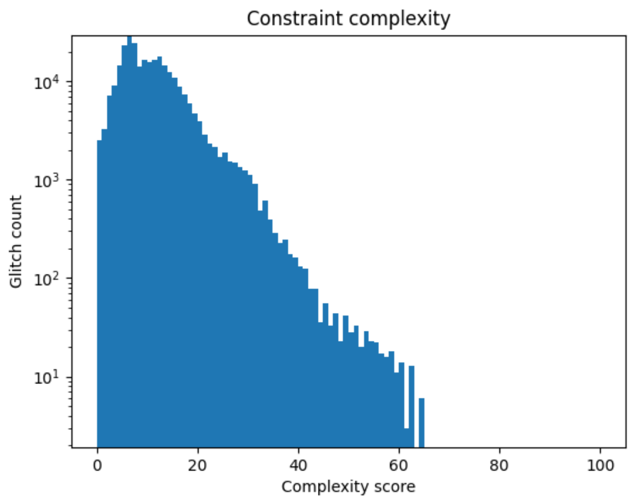


Figure 7.5: Constraint complexity for each glitch to propagate to a state

Another measure of hardening effectiveness is how probable it is for a glitch to propagate. This can be measured by looking at the number of variables involved in the propagation constraints for that glitch: the larger the set of states that are involved, the more statistically improbable it is for the glitch to propagate to a state. To have a better estimate of the actual constraint complexity, the following indicator was computed on its boolean function representation:

- If the function is OR or NOR, the complexity is equal to the minimum between the complexities of its arguments, since verifying any of them results in the overall function being verified.
- If the function is AND, NAND or XOR, the complexity is the sum of the complexities of its arguments, as all of its inputs have to be verified for the output to be verified.
- The complexity of an individual variable is 1, and the complexity of a constant (TRUE or FALSE) is 0.

The results are shown in Figure 7.5. The mean constraint complexity was 10.7, and the median 9.

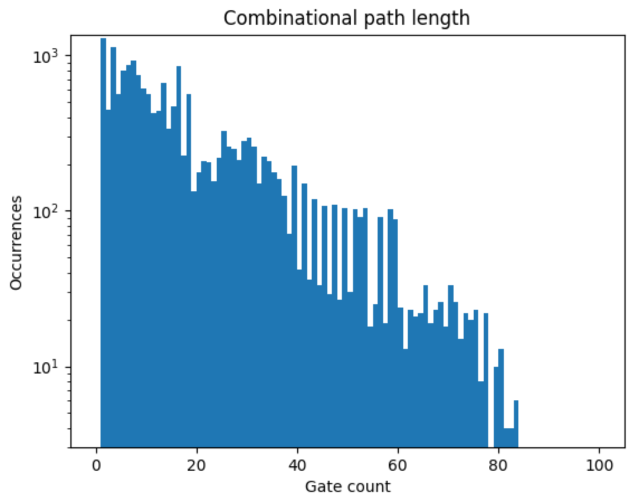


Figure 7.6: Combinational path length distribution

Compared to the previous two histograms, the shape of this distribution is much more regular, and closely resembles that of a log-normal random variable with a mode of 6. A possible explanation is that propagating through each gate requires constraining one variable, and since these individual requirements are ANDed together to obtain the final constraint, the constraint complexity is linear with the length of the combinational path from the glitched wire to the targeted state. Since combinational path length also roughly follows a log-normal distribution, as seen in Figure 7.6, the same shape is reflected in the constraint length.

Assuming that all input variables are independent, and that they follow a Bernoulli distribution with $p = 0.5$, the average probability of a glitch successfully propagating is $0.5^{10.7} = 0.06\%$. This is just a rough estimate however, and these results have to be followed up by more in-depth validations. The probability of a constraint being verified depends not only on its complexity, but also on the probability distributions of the input values. They cannot be assumed to be independent, nor do they have a 50% chance of being either 0 or 1. A more accurate estimate could be obtained by determining the actual distribution by looking at an execution trace, but the results would be application-dependent, since different applications may use different parts of the processor for different amounts of time.

This is where the ISA layer analysis would come into play. By looking at constraints put on the variables by design limitations, such as a one-hot encoded bus where only one out of a set of variables can be TRUE at any given time, a probability distribution for the constraint variables can be determined, and a more accurate vulnerability score can be assigned to each injectable fault. The advantage of obtaining the distribution this way, rather than with experimental data, is that these results would still be application-independent, and could be used directly to perform hardware mitigations. The disadvantage is that not all variables may be constrained at design-time, meaning that the distributions will still be inaccurate, and should be refined still by means of application-dependent analysis.

7.5. Application Layer

7.5.1. Implementation

Given the results of the analysis at the netlist level, a trace from the target running under nominal conditions is obtained. Then, at each time step in the simulation, the conditions for each potential glitch to propagate are checked, and successful injections are recorded.

During the development of this abstraction layer, care was taken to remove false positives, or glitches that cause changes in the state of the netlist but only temporarily. An example of this would be a glitch that causes a bit flip in a register that is never read, or that is overwritten by a subsequent instruction without being read.

In order to observe the effects a glitch has on the execution trace without simulating the entire netlist from the injection time onwards, states affected by a fault were marked as glitched, and these states were propagated in subsequent clock cycles, tracking the states affected in turn. If at any point during propagation the set of affected states became the empty set, the glitch was discarded as a false positive, since its effects on the execution trace had vanished before reaching the target.

This addition to the analysis further extended the results from the initial implementation: glitches that indirectly affected the critical states by propagating from state to state were also identified, and not only those glitches that directly affected the critical states. If, during propagation, any of the affected states was part of the critical set, either success or failure was reported for that glitch, depending on whether it would cause the board to halt or compromise its security, respectively. The pseudocode implementation of the algorithm is shown in Algorithm 8.

A limitation of the implementation is that, once a glitch is injected, it is assumed that no other glitch will occur in that simulation timeline. At any given clock cycle, all glitches that cause the netlist state to diverge from the reference are recorded, and their effects are propagated assuming that the netlist will behave as dictated at design-time from then on. While theoretically possible, analyzing the effects of injecting multiple faults at different times would be a lot more computationally intensive, requiring $\mathcal{O}(n^a)$ time (where n is the number of potential glitches per clock cycle and a is the number of glitches allowed in each simulation timeline).

Notably, this approach could be used not only for netlists that execute an application, but for fixed-function designs as well. As long as a netlist and a trace collected from running the netlist are provided, the same analysis could be theoretically carried out with no modifications.

7.5.2. Hash Functions and HMAC

The authentication algorithm used by the target application is a well-known method to ensure both the integrity and the authenticity of a message (or, in this case, a user application). This method is called HMAC (Hash-based Message Authentication Code), and employs specific mathematical functions, called hash functions, to generate a signature for the message.

Hash functions all have the same property: regardless of the size of their input, their output size is always fixed. A trivial example is the constant function $f(x) = 1$, but most commonly used hash functions generate different values for different messages, and their values are used to index large amounts of data, allowing applications to find items in large databases in near constant time. Hash functions used with HMAC are known as cryptographically secure hash functions, a subset of all hash functions that are used to ensure the integrity of a given message.

In order for a hash function to be considered cryptographically secure, it must also obey the following properties [45]:

- Given a hash, it is infeasible to generate a message that yields the same hash value
- Given a message and its hash, it is infeasible to generate another message, different from the one given, that yields the same hash value
- A small change in the input message results in the corresponding hash being uncorrelated with its original value.

These three properties are sufficient to ensure integrity: a sender would compose a message, calculate its hash using a function obeying these properties, and then send both to the intended recipient. If a part of the message is corrupted during transmission, the entire hash will change, and its extremely unlikely that any change introduced in the message will not change the value of its hash. Moreover, if

Algorithm 8 Application-level fault analysis

```
Require: $T$ netlist trace obtained by simulating the execution of a target application with no faults.
   Assumed to be an array of dictionaries, one for each clock cycle. Each dictionary contains the state
   of all wires in the netlist.
Require: $G$ collection of glitches from the netlist layer analysis.
Require: $C$ collection of critical states that, if modified, would cause the netlist to halt.
Require: $S$ collection of success conditions.
  $A_g \leftarrow \{\}$    Set of glitched states resulting from earlier injections
  for $state \in T$ do
    Evaluate which states would be affected next by the active glitches in $A_g$
    for $A \in A_g$ do
      $A \leftarrow states\_affected\_by(A)$
      if $A = \{\}$ then
        $A_g \leftarrow A_g \setminus A$
      end if
    end for
    Evaluate which glitches would be successful at the current simulation step
    for $glitch \in G$ do
      $A_{g,new} \leftarrow \{\}$
      if $eval(glitch.constraint, state) = True$ then
        $A_{g,new} \leftarrow A_{g,new} \cup glitch.affected\_states$
      end if
    end for
    $A_g \leftarrow A_g \cup A_{g,new}$
    Check for glitches that affected critical states
    for $A \in A_g$ do
      if $A \cap C \neq \{\}$ then
        $A_g \leftarrow A_g \setminus A$
      else if $any\_verified(S, A)$ then
        Glitch reached a success condition
      end if
    end for
  end for
```

Contents	Start address	End address	Size (bytes)
SecureBoot code	`0x0000`	`0x7FFF`	32768
Application signature	`0x8000`	`0x8020`	32
Application code	`0x8020`	`0xFFFF`	32736

Table 7.1: SecureBoot ROM memory layout

an attacker purposefully changed any part of the message, it would be infeasible for him to generate a new message with the same hash. The recipient would then calculate the hash of the corrupted message, and by comparing it against the original hash, would know that the message was tampered with.

Hash functions alone cannot, however, be used to ensure the authenticity of the message. After all, if the attacker can change the message, they could also change the hash to match, and the recipient would be unable to verify whether what he received was what the original sender transmitted. In order to also ensure the authenticity of the message received, the two parties should first exchange some secret value, only known to them, and then combine this secret with the hash, so that the other party can verify that the sender also knew the secret.

It is extremely important, however, that the attacker cannot retrieve the secret by knowing the hash and the message, and that he cannot forge new messages from a given hash/message pair. For example, concatenating the secret with the message and computing the hash of both is not sufficient: most cryptographically secure hash functions repeat the same operation on fixed-length blocks of data, using the result of previous rounds to compute the next, until the output is obtained. Knowing $H(k||m)$, where k is the secret and m is the message, an attacker could compute $H(k||m||m')$ by using the original hash as the starting point for his calculation, obtaining a message that still passes authentication at the receiver. One secure way to use hash functions for authentication is HMAC.

The HMAC of a message is computed as follows. Given a message m, a secret key K, and a cryptographically secure hash function H:

$$\begin{aligned}\text{HMAC}(K,m) &= H((K' \oplus opad)||H((K' \oplus ipad)||m))\\ K' &= H(K)\text{if K is larger than the block size, else}K\end{aligned}$$

$opad$ and $ipad$ are fixed values: $opad$ is the byte `0x5c` repeated to fit the size of the secret, and $ipad$ is the byte `0x36` repeated. An attacker cannot append an arbitrary message anymore, as the message value is inside the inner hash function, not outside. Likewise, he cannot recover the value of the secret, since that requires breaking the hash function which is, by definition, secure.

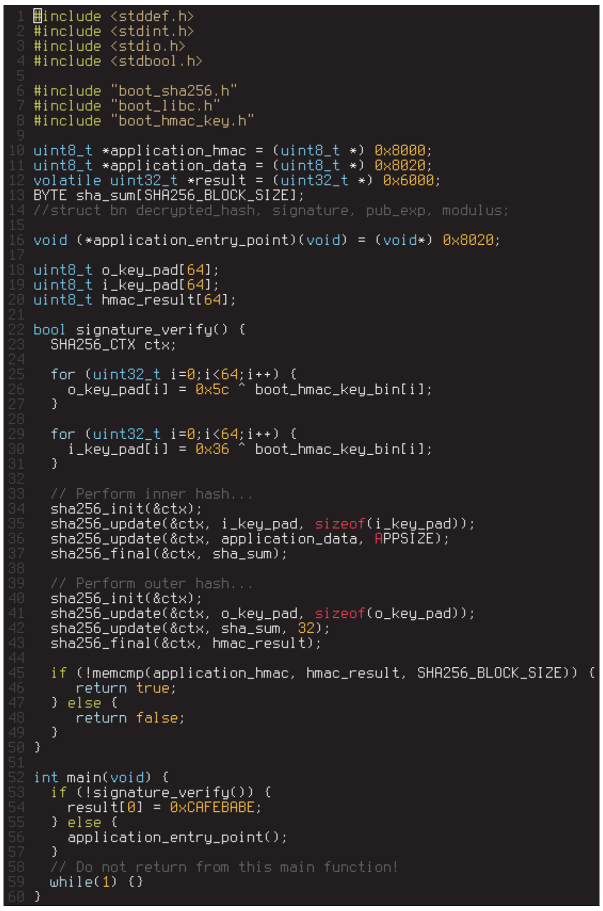
7.5.3. Target Application Analysis

The target application chosen to validate the effectiveness of this analysis was an implementation of SecureBoot compiled for RISC-V processors.

The SecureBoot implementation used is a simple bootloader that checks whether an application has a valid signature before executing it. This signature is calculated based on a HMAC authentication scheme, using SHA256 as the base hash function, and if it matches the one stored in ROM, the processor begins executing the application. If, on the other hand, the check fails, the application is not executed, and the processor enters an infinite loop. A memory map of the whole ROM binary image is reported in Table 7.1.

The source code for the application is shown in Figure 7.7. As shown, the core function `verify_signature()` first calculates the values of the key XOR'd with the outer and inner pads, then calculates the inner hash (hash of the inner pad concatenated with the application data) and finally computes the outer hash (hash of the outer pad concatenated with the inner hash), returning TRUE or FALSE depending on whether the signature matches.

Assuming that the secret key resides in a memory region inaccessible to an attacker, the only way for the bootloader to validate and execute an arbitrary application is to ensure that the signature of the new application matches that of the old code, which is infeasible due to the cryptographic hash function being used. There is, however, another weak point that could be attacked, outside of the verification function.



```c
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdbool.h>

#include "boot_sha256.h"
#include "boot_libc.h"
#include "boot_hmac_key.h"

uint8_t *application_hmac = (uint8_t *) 0x8000;
uint8_t *application_data = (uint8_t *) 0x8020;
volatile uint32_t *result = (uint32_t *) 0x6000;
BYTE sha_sum[SHA256_BLOCK_SIZE];
//struct bn decrypted_hash, signature, pub_exp, modulus;

void (*application_entry_point)(void) = (void*) 0x8020;

uint8_t o_key_pad[64];
uint8_t i_key_pad[64];
uint8_t hmac_result[64];

bool signature_verify() {
  SHA256_CTX ctx;

  for (uint32_t i=0;i<64;i++) {
    o_key_pad[i] = 0x5c ^ boot_hmac_key_bin[i];
  }

  for (uint32_t i=0;i<64;i++) {
    i_key_pad[i] = 0x36 ^ boot_hmac_key_bin[i];
  }

  // Perform inner hash...
  sha256_init(&ctx);
  sha256_update(&ctx, i_key_pad, sizeof(i_key_pad));
  sha256_update(&ctx, application_data, APPSIZE);
  sha256_final(&ctx, sha_sum);

  // Perform outer hash...
  sha256_init(&ctx);
  sha256_update(&ctx, o_key_pad, sizeof(o_key_pad));
  sha256_update(&ctx, sha_sum, 32);
  sha256_final(&ctx, hmac_result);

  if (!memcmp(application_hmac, hmac_result, SHA256_BLOCK_SIZE)) {
     return true;
  } else {
     return false;
  }
}

int main(void) {
  if (!signature_verify()) {
    result[0] = 0xCAFEBABE;
  } else {
    application_entry_point();
  }
  // Do not return from this main function!
  while(1) {}
}
```

Figure 7.7: SecureBoot source code

Looking at the disassembly of the code that will be simulated on the CPU (Figure 7.8), it can be seen that the last step in the verification process is a branch instruction, which either causes the program counter to jump to the application's entry point, or enter an infinite loop. If the left branch is executed, at address `0x760`, the processor writes `0xCAFEBABE` to memory location `0x6000` and enters an infinite loop. If the right branch is executed, at address `0x778`, the processor loads the entry point of the application into register `$a5` and then jumps to it.

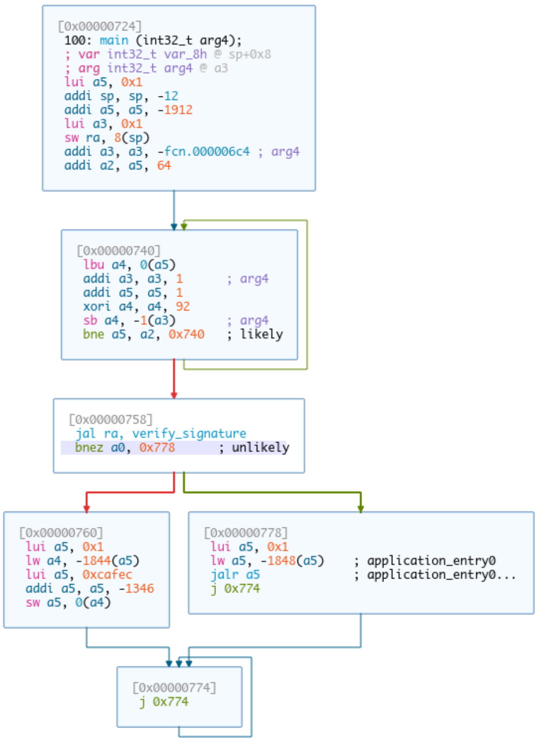
Rather than attacking the core cryptographic function, all an attacker has to do is to change the value returned by it from FALSE to TRUE for the main function to jump to the opposite branch, allowing execution. This was the vulnerability targeted at the application level.

Success can be reliably determined by checking the value of the program counter after branching. A value of `0x778` implies that the application will be executed, a value of `0x760` indicates that the check failed and the application will not run.

7.5.4. Results

Since it is easier to cause an application to fail the signature check when it should succeed rather than the opposite (after all, there is only one correct signature and many incorrect ones), the first proof-of-concept involved using the developed tools to cause a valid application to not execute.

If the application-level fault injection analysis was to be performed on an unknown application, or one where identifying a single point of failure would not have been obvious, it would have had to run on the entire trace. There is no intrinsic limitation in the approach described that would prevent this from succeeding, but given the limited time devoted to optimizing the application-level analysis, this would have taken a considerable amount of time, more than was available at the time. Without loss of generality, the analysis was therefore started 250ns before the target branch instruction's execution. This limitation means that there could be more, undetected glitches that cause the signature check to



```
[0x00000724]
100: main (int32_t arg4);
; var int32_t var_8h @ sp+0x8
; arg int32_t arg4 @ a3
lui a5, 0x1
addi sp, sp, -12
addi a5, a5, -1912
lui a3, 0x1
sw ra, 8(sp)
addi a3, a3, -fcn.000006c4 ; arg4
addi a2, a5, 64
```

```
[0x00000740]
lbu a4, 0(a5)
addi a3, a3, 1       ; arg4
addi a5, a5, 1
xori a4, a4, 92
sb a4, -1(a3)        ; arg4
bne a5, a2, 0x740    ; likely
```

```
[0x00000758]
jal ra, verify_signature
bnez a0, 0x778       ; unlikely
```

```
[0x00000760]
lui a5, 0x1
lw a4, -1844(a5)
lui a5, 0xcafec
addi a5, a5, -1346
sw a5, 0(a4)
```

```
[0x00000778]
lui a5, 0x1
lw a5, -1848(a5)     ; application_entry0
jalr a5              ; application_entry0...
j 0x774
```

```
[0x00000774]
j 0x774
```

Figure 7.8: SecureBoot disassembly: the highlighted branch determines whether the target application runs

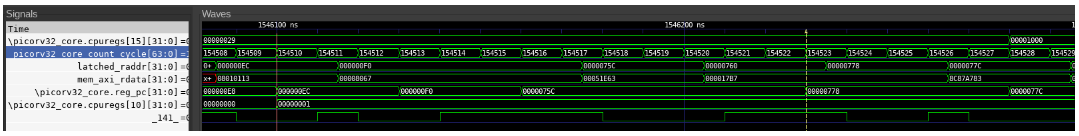

Signals
Waves
Time
\picorv32_core.cpuregs[15][31:0]
picorv32_core.count_cycle[63:0]
latched_raddr[31:0]
mem_axi_rdata[31:0]
\picorv32_core.reg_pc[31:0]
\picorv32_core.cpuregs[10][31:0]
_141_
1546100 ns
1546200 ns

Figure 7.9: SecureBoot trace: correct signature, no glitches

fail, although the probability decreases the further back in time the glitch occurs.

Figure 7.9 shows the critical section of the execution trace, when the signature-dependent branch is executed. Note the value of `picorv32_core.reg_pc` at the position of the yellow marker, which is 0x778 at $t = 1546.23us$ indicating that the signature check passed.

Running the application-level analysis resulted in 18263 faults reported as successful, with the first one occurring at $t = 1545.98us$. This amounts to around 3% of all possible injections during the 250ns timeframe. Simulation-based fault injection was then used to verify the results of each individual glitch reported, to validate whether they had the expected effects. Multiple glitches did not, but the first one to cause the application to fail the signature check was a glitch on wire _141_ at time $t = 1546.12us$. The resulting trace is shown in Figure 7.10: at $t = 1546.23us$ the program counter is 0x760, hence the signature check failed.

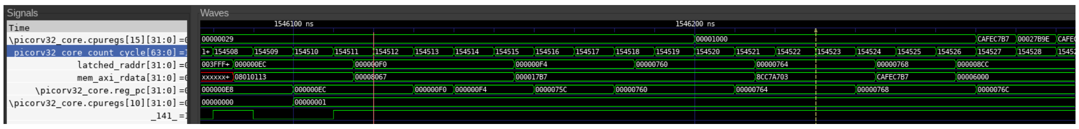

Signals
Waves
Time
\picorv32_core.cpuregs[15][31:0]
picorv32_core.count_cycle[63:0]
latched_raddr[31:0]
mem_axi_rdata[31:0]
\picorv32_core.reg_pc[31:0]
\picorv32_core.cpuregs[10][31:0]
_141_
1546100 ns
1546200 ns

Figure 7.10: SecureBoot trace: correct signature, glitch on wire _141_ at t=1546.12us

Many of the faults reported had a different effect when simulated, resulting in either non-faulty behaviour, or different states being affected at the time the target branch was executed. The main reason is that the framework, as it currently stands, has no way to simulate devices attached to top-level ports in the design, including ROM and RAM. If a glitch affects the address bus at a timestamp previous to the one used for success-failure classification, the data read from or written to memory would be different, possibly causing the entire execution trace to diverge. The analysis tool, however, has no way of knowing what this data would be, and assumes that it will be the same as in the non-faulty trace.

The next attempt was to boot an application with the wrong signature. This is more in line with what an attacker would attempt: to load unsigned code and trick SecureBoot into executing it.

A different application was compiled for the target, and the binary loaded into memory at simulation time was edited so that the target application section contained the new, unsigned binary. The source code for the new application can be found in section A.4. The signature section of ROM was unchanged compared to the original binary, so it did not match the code in the application section of the binary. This meant that the signature check performed by the bootloader would fail, and the application not run, under nominal operating conditions.

The target instruction and success conditions were kept the same as in the previous attempt, as the goal in both cases was to cause the branch instruction to execute the wrong jump, changing the address of the instruction accessed after the branch. The only difference was that the analysis was run on a different trace, where the modified binary was executed.

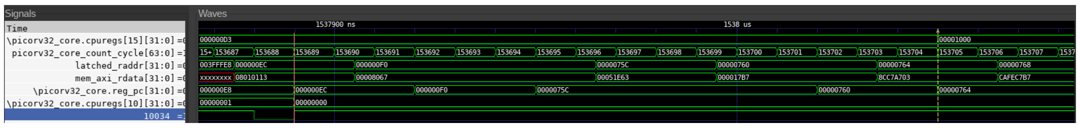

Signals
Waves
Time
\picorv32_core.cpuregs[15][31:0]
picorv32_core_count_cycle[63:0]
latched_raddr[31:0]
mem_axi_rdata[31:0]
\picorv32_core.reg_pc[31:0]
\picorv32_core.cpuregs[10][31:0]
10034
1537900 ns
1538 us

Figure 7.11: SecureBoot trace: wrong signature, no glitches

Compared to the previous trace, the timestamp at which the target instruction executed was 1538050ns, as shown in Figure 7.11. The time difference compared to the previous experiment's baseline is due to the different number of instructions in the target application, which affects the time taken to hash the application. Once again, note the value of `picorv32_core.reg_pc` equal to `0x760` after the aforementioned timestamp, indicating that the check failed.

The application layer script was adjusted to check for the success condition at the updated timestamp, and then run against the given trace. Once again, to reduce the time necessary to obtain the results, the analysis was started 250ns before the target branch instruction.

The first reported successful injection was at $t = 1537.8us$, which is the first time step at which fault analysis was performed. In total, 16623 successful injections were reported in the given time period, which is around 2% of the total number of possible injections during that timeframe. As with the previous results, many of these were false positives, or resulted in different wires being glitched compared to the ones reported, since the framework cannot simulate external devices at this time.

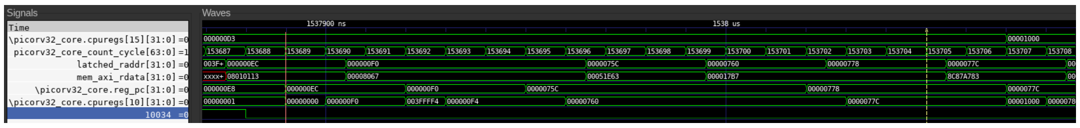

Signals
Waves
Time
\picorv32_core.cpuregs[15][31:0]
picorv32_core_count_cycle[63:0]
latched_raddr[31:0]
mem_axi_rdata[31:0]
\picorv32_core.reg_pc[31:0]
\picorv32_core.cpuregs[10][31:0]
10034
1537900 ns
1538 us

Figure 7.12: SecureBoot trace: wrong signature, glitched wire _10034_ at t=1537.98us

To validate the results and remove false positives, a traditional simulation-based analysis was performed afterwards, injecting only where the analysis indicated that the glitch would be successful. The first glitch that resulted in a success condition was performed on wire `_10034_` at time $t = 1537.89us$, and generated the trace shown in Figure 7.12.

In order to verify that no glitches outside of the given subset were successful at causing the SecureBoot check to succeed with the wrong signature, a random set of possible glitches outside of it were injected, and none of them resulted in the success state being reached. An exhaustive search would have definitely proven the soundness of the analysis performed, but was not feasible due to time limitations.

7.6. Discussion

The results obtained on the target platform indicate that static analysis allows existing simulation tools to achieve full fault injection coverage (for the limited timeframe chosen) by only checking a small subset of all possible glitches, greatly reducing the time necessary to obtain results.

Moreover, the netlist-level analysis described above can be used on its own to guide preliminary hardening against hardware faults, as it provides insight into which nets and which flip-flops are more susceptible to this kind of attacks. While the application-level analysis algorithm is sound, and successfully identified faults that achieved a given goal, further optimization is required to achieve a significant performance advantage over blind simulation.

There were numerous false positives in the final application level analysis, more than one might expect at first. The reason is quite simple: while the application analysis itself was thorough, the simulation of the processor was less so: external peripherals, including the memory where code and data reside, could not be included in the simulation. The result was that, if a glitch reached the address bus, it would not affect the data retrieved at the next clock cycle, while in reality it did. Nonetheless, the number of possible faults that had to be verified with an external simulator was greatly reduced, and so was the time required to find vulnerabilities in the design.

8
Conclusion

8.1. Summary

Simulation-based fault injection is a cost-effective way to catch hardware vulnerabilities in a design, and has a much shorter turnaround time compared to purely hardware-based approaches. The main downside is that each new injection test takes a considerable amount of time, so an exhaustive search cannot be performed, and compromises have to be made.

The abstraction gap between netlist and RTL, which makes implementing mitigations more cumbersome, can be somewhat bridged by recovering annotations from the RTL into the netlist, tying individual cells to the lines of code that generated them. This can be performed by means of equivalence checking, although more research is required to increase the number of annotations recovered above the 1% figure reported.

By statically analyzing the effects of each fault on the hardware first, results can be re-used across runs, and the number of test cases necessary to achieve 100% coverage is greatly reduced. This is also more insightful compared to a purely simulation-based approach, as parts of the analysis are only dependent on the hardware, and mitigations derived from this analysis will be effective regardless of the software run on the board.

Chapter 1 introduced the concept of hardware fault injection, describing how its importance has grown in recent years. It presented some existing solutions to identify possible weak points in a hardware product during the design stage, and highlighted the pros and cons of each approach. It also presented a list of the products of the work in terms of contributions, which were described further in detail in the following chapters.

Chapter 2 gave a brief overview of formal verification: what it is, why it is important, and how it is usually integrated in a development workflow. It highlighted its major differences when compared to validation, focusing on why formal verification is necessary alongside simulation-based validation, and why they are both necessary to ensure that a design satisfies a given set of design constraints. It then described an aspect of formal verification named equivalence checking, which was used in Chapter 4 to perform source annotation recovery. It briefly described what equivalence checking is, and two possible ways to implement it on a design.

Chapter 3 described hardware attacks, the typical classification system used to sub-divide attacks based on their features, and then introduced multiple examples of fault injection techniques performed by attackers in the field, classifying them and describing the effects each one can have on the targeted device. Finally, it introduced the concept of fault profiles, which are formal models used to describe the effects a fault injection attack has on a device in an abstract way.

Chapter 4 introduced the basic concepts most commonly encountered when applying optimization techniques: computational complexity and NP-completeness. These ideas were used extensively in the development phases of the work, due to the inherent problems of working with boolean functions and the limited computational power available.

Chapter 5 gave an overview of the framework developed during the thesis, and of the target used as a case study.

Chapter 6 described the first part of the thesis work: reconstructing links between netlist cells and RTL lines of code. It described the algorithms used to perform this step, the different optimization techniques that were attempted to accelerate the equivalence checking process, and the results obtained in the final implementation.

Chapter 7 focused on the second part of the work: performing fault injection at the netlist level efficiently. It described the different layers of abstraction that were used to reduce the complexity of the task into multiple sub-tasks, followed by a description of the workflow envisioned using these abstraction layers. It then described the two layers implemented: the netlist layer and the application layer. In both cases, the core algorithm was described, as well as the failed attempts encountered along the way, and then showed the results for each abstraction layer. In the case of the netlist layer, different statistics were presented to indicate how vulnerable the PicoRV32 design was to hardware attacks. For the application layer, on the other hand, a SecureBoot binary was used as a proof-of-concept, showing how the framework could be used to bypass the signature checking algorithm implemented in it.

8.2. Future work

In this section, a list of possible extensions that could be made to the presented framework is highlighted, ranging from pure optimizations to new features that could improve the results obtained.

- **Equivalence checking comparison points**: results have shown that name-based matching was not enough to map all flip-flops in the RTL to flip-flops in the netlist. This limited the effectiveness of the approach, since some of the logic cones could not be compared to each other. The solution would be to find another function to map the remaining flip-flops. Commercial tools can overcome this issue, demonstrating that it is possible, but their algorithms are not publicly available.
- **Retiming detection**: during synthesis, long combinational paths could be broken down into smaller paths separated by buffer flip-flops, so that timing requirements are met. This breaks the RTL logic cones into smaller cones, and they are no longer recognized as equivalent in the netlist by the framework in its current state. Adding retiming detection logic can resolve this issue, increasing the number of matches in the combinational paths and, in turn, the number of annotations recovered. This could be done, for instance, by removing the added flip-flops, and extending the logic cones up to the original flip-flops instead.
- **Different fault profiles**: only single event upsets were analyzed in this framework. Implementing other faults, such as stuck-at or delay faults, can be performed by changing the netlist and application layer analysis. This could also be extended by modeling different fault profiles, if flipping individual bits is unrealistic.
- **Multiple faults**: simulating more than one fault on the same execution path would have resulted in exponentially increasing computational complexity ($O(n^k)$ where n is the number of possible faults and k is the number of faults per execution path). This was infeasible because of poor optimization, but it could be performed if the framework was ported to a compiled language. Further optimizations to the algorithm could take advantage of the constraint system in place: if a glitch cannot propagate due to a constraint not being satisfied, it could be possible to inject another glitch that causes the constraint to be satisfied, and the original glitch to succeed.
- **Peripheral simulation**: as mentioned in the application-level analysis, many of the reported faults caused the processor to behave differently in simulation compared to what the static analysis predicted. The main reason is that the framework currently cannot emulate external devices, the output of which changes depending on the values presented on their inputs by the processor. If a glitch affects one of these inputs, the output of the peripheral could change as well (e.g. if the peripheral is a ROM and the address bus changes, the data at its outputs will change as well). Emulating this behaviour will reduce the amount of false positives reported.

A
Code Listings

A.1. YoSYS synthesis script

```
# Import cell library
read_liberty -lib -ignore_miss_dir -setattr blackbox ../../osu018_stdcells.lib
setundef -zero


# Read main verilog source
read_verilog ../picorv32/picorv32.v

# Synthesize
synth -top picorv32_axi -run begin
write_ilang 0_postbegin.ilang
synth -top picorv32_axi -run coarse
write_ilang 1_postcoarse.ilang
opt -fast -full
write_ilang 2_postopt.ilang
memory_map
techmap
opt -fast
write_ilang 3_posttechmap.ilang
opt -fast
write_ilang 4_postopt.ilang
abc -fast
opt -fast

# Map FFs
dfflibmap -liberty ../../osu018_stdcells.lib
write_ilang 5_postlibmap.ilang

# Optimize
opt
write_ilang 6_postopt.ilang

# Map combinational cells
abc -keepff -liberty ../../osu018_stdcells.lib -script +strash;scorr;ifraig;retime,{D};
    strash;dch,-f;map,-M,1,{D}
write_ilang 7_postabc.ilang

# Flatten hierarchy
flatten
write_ilang 8_postflatten.ilang

# Cleanup
clean -purge
write_ilang 9_postclean.ilang

# Final optimization and cleanup
opt
clean
```

```
rename -enumerate

write_ilang netlist.ilang
write_verilog netlist.v
```

A.2. PyEDA modification: modified ast2expr() in pyeda/boolalg/-expr.py

```
def ast2expr(ast):
    """Convert an abstract syntax tree to an Expression."""
    if ast[0] == 'const':
        return _CONSTS[ast[1]]
    elif ast[0] == 'var':
        return exprvar(ast[1], ast[2])
    elif ast[0] == 'lit':
        return _LITS[ast[1]]
    else:
        xs = [ast2expr(x) for x in ast[1:]]
        return ASTOPS[ast[0]](*xs, simplify=False)
```

A.3. Original, signed SecureBoot target application

```
#include <stddef.h>
#include <stdint.h> #include <stdio.h>

volatile uint32_t *result = (uint32_t *) 0x6000;

int main(void) {
        result[0] = 0x8badf00d;
}
```

A.4. Modified, unsigned SecureBoot target application

```
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>

volatile uint32_t *result = (uint32_t *) 0x6000;

int main(void) {
        uint32_t f1, f2, tmp;
        int i;

        result[0] = 0x8BADF00D;

        /* Filler code: Fibonacci sequence */
        f1 = f2 = 1;
        for (i=0; i<32; i++) {
                tmp = f1 + f2;
                f1 = f2;
                f2 = tmp;

                result[0] = f1;
        }
}
```

Bibliography

[1] Amba 4 axi4-stream protocol specification. URL https://zipcpu.com/doc/axi-stream.pdf.

[2] Cadence conformal overview. URL https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/conformal-overview.html.

[3] Focused ion beam - litography and patterning. URL https://www.nffa.eu/offer/lithography-patterning/installation-1/fib/.

[4] Gcc, the gnu compiler collection. URL https://gcc.gnu.org/.

[5] Genus synthesis solution. URL https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html.

[6] Laser cutting. URL https://www.semitracks.com/reference-material/failure-and-yield-analysis/failure-analysis-die-level/laser-cutting.php.

[7] Semiconductor wafer mask costs. URL https://anysilicon.com/semiconductor-wafer-mask-costs/.

[8] Test generation principles in dft. URL https://technobyte.org/test-generation-principles-dft-vlsi/.

[9] Risc-v specifications. URL https://riscv.org/technical/specifications/.

[10] Vivado design suite user guide. URL https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug901-vivado-synthesis.pdf.

[11] Yosys open synthesis suite. URL http://www.clifford.at/yosys/.

[12] J. Arlat, Y. Crouzet, and J.-C. Laprie. Fault injection for dependability validation of fault-tolerant computing systems. In [1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers, pages 348–355, 1989. doi: 10.1109/FTCS.1989.105591.

[13] Aspencore. 2019 embedded systems market study. URL https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf.

[14] Dr Bhavanam and Vasujadevi Midasala. Implementation of fpga based fault injection tool (fito) for testing fault tolerant designs. International Journal of Web Engineering and Technology, 4: 522–526, 10 2012. doi: 10.7763/IJET.2012.V4.424.

[15] Jakub Breier and Dirmanto Jap. Testing feasibility of back-side laser fault injection on a microcontroller. In Proceedings of the WESS'15: Workshop on Embedded Systems Security, WESS'15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336673. doi: 10.1145/2818362.2818367. URL https://doi.org/10.1145/2818362.2818367.

[16] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. 1986. URL https://www.cs.cmu.edu/~bryant/pubdir/ieeetc86.pdf.

[17] J. V. Carreira, D. Costa, and J. G. Silva. Fault injection spot-checks computer system dependability. IEEE Spectr., 36(8):50–55, August 1999. ISSN 0018-9235. doi: 10.1109/6.780999. URL https://doi.org/10.1109/6.780999.

[18] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. Introduction to Algorithms. McGraw-Hill Higher Education, 2nd edition, 2001. ISBN 0070131511.

[19] M. Davis. Computability and Unsolvability. Dover Books on Computer Science. Dover Publications, 2013. ISBN 9780486151069. URL https://books.google.it/books?id=nbOqAAAAQBAJ.

[20] Maurice Dawson, Darrell Burrell, Emad Rahim, and Stephen Brewster. Integrating software assurance into the software development life cycle (sdlc). Journal of Information Systems Technology and Planning, 3:49–53, 01 2010.

[21] Charles Drake. Python eda documentation. URL https://pyeda.readthedocs.io/en/latest/.

[22] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced encryption standard (aes), 2001-11-26 2001.

[23] Stephen Williams et al. Icarus verilog, . URL http://iverilog.icarus.com/.

[24] Wilson Snyder et al. Verilator, . URL https://www.veripool.org/verilator/.

[25] Jędrzej Fulara and Krzysztof Jakubczyk. Practically applicable formal methods. In In SOFSEM '10: Proceedings of the 36th Conference on Current Trends in Theory and 143 of Computer Science, pages 407–418. Springer, 2010.

[26] Subramani Ganesh. A gentle introduction to formal verification. URL https://www.systemverilog.io/gentle-introduction-to-formal-verification.

[27] Zion Research Group. Security for embedded electronics. URL https://semiengineering.com/security-for-embedded-electronics/.

[28] Ian Grout. Chapter 2 - electronic systems design. In Ian Grout, editor, Digital Systems Design with FPGAs and CPLDs, pages 43–121. Newnes, Burlington, 2008. ISBN 978-0-7506-8397-5. doi: https://doi.org/10.1016/B978-0-7506-8397-5.00002-7. URL https://www.sciencedirect.com/science/article/pii/B9780750683975000027.

[29] U. Gunneflo, J. Karlsson, and J. Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In [1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers, pages 340–347, 1989. doi: 10.1109/FTCS.1989.105590.

[30] Clemens Helfmeier, Christian Boit, Dmitry Nedospasov, and Jean-Pierre Seifert. Cloning physically unclonable functions. In 2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), pages 1–6, 2013. doi: 10.1109/HST.2013.6581556.

[31] Roland Höller, Dominic Haselberger, Dominik Ballek, Peter Rössler, Markus Krapfenbauer, and Martin Linauer. Open-source risc-v processor ip cores for fpgas — overview and evaluation. In 2019 8th Mediterranean Conference on Embedded Computing (MECO), pages 1–6, 2019. doi: 10.1109/MECO.2019.8760205.

[32] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? Journal of Computer and System Sciences, 63(4):512–530, 2001. ISSN 0022-0000. doi: https://doi.org/10.1006/jcss.2001.1774. URL https://www.sciencedirect.com/science/article/pii/S002200000191774X.

[33] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into vhdl models: the mefisto tool. In Proceedings of IEEE 24th International Symposium on Fault- Tolerant Computing, pages 66–75, 1994. doi: 10.1109/FTCS.1994.315656.

[34] Richard M. Karp. Reducibility among combinatorial problems. In Complexity of Computer Computations, New York, NY, USA. Springer, Boston, MA. URL https://doi.org/10.1007/978-1-4684-2001-2_9.

[35] Maha Kooli and Giorgio Di Natale. A survey on simulation-based fault injection tools for complex systems. In 2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS), pages 1–6, 2014. doi: 10.1109/DTIS.2014.6850649.

[36] Andreas Kuehlmann and Cornelis A J Eijk. Combinational and sequential equivalence checking. In Logic Synthesis and Verification, pages 343–372. Springer US, Boston, MA, 2002.

[37] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In Proceedings of the 34th Annual Design Automation Conference, DAC '97, page 263–268, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897919203. doi: 10.1145/266021.266090. URL https://doi.org/10.1145/266021.266090.

[38] Benoit Lasbouygues, Robin Wilson, Nadine Azemard, and Philippe Maurine. Temperature- and voltage-aware timing analysis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 26(4):801–815, 2007. doi: 10.1109/TCAD.2006.884860.

[39] L. Lavagno, G. Martin, and L. Scheffer. Electronic Design Automation for Integrated Circuits Handbook - 2 Volume Set. Industrial Information Technology. Taylor & Francis, 2006. ISBN 9780849330964. URL https://books.google.it/books?id=y4RrnQAACAAJ.

[40] Alan Mishchenko. Abc: A system for sequential synthesis and verification. URL https://people.eecs.berkeley.edu/~alanmi/abc/.

[41] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = lazy clause generation. In Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, LNCS. Springer-Verlag, 2007.

[42] Sebastien Ordas, Ludovic Guillaume-Sage, Karim Tobich, Jean-Max Dutertre, and Philippe Maurine. Evidence of a larger em-induced fault model. 11 2014. ISBN 978-3-319-16762-6. doi: 10.1007/978-3-319-16763-3_15.

[43] Athanasios Papadimitriou, David Hely, Vincent Beroulle, Paolo Maistri, and Regis Leveugle. A multiple fault injection methodology based on cone partitioning towards rtl modeling of laser attacks. pages 1–4, 03 2014. doi: 10.7873/DATE2014.219.

[44] Richard Quinnell. Risc-v initiative leverages standard platform for custom designs, 2019. URL https://www.embedded.com/risc-v-initiative-leverages-standard-platform-for-custom-designs/.

[45] Russel J. Bradford Saif Al-Kuwari, James H. Davenport. Cryptographic hash functions: recent trends and security notions. Cryptology ePrint Archive, 2011.

[46] V. Sieh, O. Tschache, and F. Balbach. Verify: evaluation of reliability using vhdl-models with embedded fault descriptions. In Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing, pages 32–36, 1997. doi: 10.1109/FTCS.1997.614074.

[47] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, Cryptographic Hardware and Embedded Systems - CHES 2002, pages 2–12, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-36400-9.

[48] Ningfang Song, Jiaomei Qin, Xiong Pan, and Yan Deng. Fault injection methodology and tools. In Proceedings of 2011 International Conference on Electronics and Optoelectronics, volume 1, pages V1–47–V1–50, 2011. doi: 10.1109/ICEOE.2011.6013043.

[49] Bart Stevens. Fault injection attacks: a growing plague. URL https://www.eeweb.com/fault-injection-attacks-a-growing-plague/.

[50] Pavan Talluri. Fault classification and vulnerability analysis of microprocessors, 2020.

[51] Andrew Waterman. Improving energy efficiency and reducing code size with risc-v compressed. Master's thesis, EECS Department, University of California, Berkeley, May 2011. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-63.html.

[52] Clifford Wolf. Picorv32 - a size-optimized risc-v cpu, . URL https://github.com/cliffordwolf/picorv32.

[53] Clifford Wolf. Yosys manual, . URL http://www.clifford.at/yosys/files/yosys_manual.pdf.

[54] Ben Wood. Computability and the halting problem, 2015. URL `https://cs.wellesley.edu/~cs251/f20/notes/halt.html`.

[55] L. et al. Yifei Q., Zhaojun. Clock glitch fault injection attacks on an fpga aes implementation. In Journal of Electrotechnology, Electrical Engineering and Management, 2017.