Transient obfuscation for HLS security : application to cloud security, birthmarking and hardware Trojan defense

Hannah Badier

▶ To cite this version:

Hannah Badier. Transient obfuscation for HLS security : application to cloud security, birthmarking and hardware Trojan defense. Cryptography and Security [cs.CR]. ENSTA Bretagne - École nationale supérieure de techniques avancées Bretagne, 2021. English. NNT : 2021ENTA0012 . tel-03789700

HAL Id: tel-03789700
https://theses.hal.science/tel-03789700v1

Submitted on 27 Sep 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

ENSTA Bretagne

THÈSE DE DOCTORAT DE

L'ECOLE NATIONALE SUPÉRIEURE
DE TECHNIQUES AVANCÉES BRETAGNE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies
de l'Information et de la Communication
Spécialité : *Informatique*

Par

Hannah BADIER

Transient Obfuscation for HLS Security

Application to Cloud Security, Birthmarking and Hardware Trojan Defense

Thèse présentée et soutenue à ENSTA Bretagne, Brest, le 29/04/2021
Unité de recherche : Laboratoire Lab-STICC, UMR 6285

Rapporteurs avant soutenance :

Lilian BOSSUET	Professeur des Universités, Université Jean Monnet, Laboratoire Hubert Curien, Saint-Etienne
Roselyne CHOTIN	Maître de Conférences, HDR, Sorbonne Université, LIP6 Paris

Composition du Jury :

Président :	Régis LEVEUGLE	Professeur des Universités, Université Grenoble Alpes, TIMA
Examinateurs :	Christian PILATO	Assistant Professor, Ecole Polytechnique de Milan, Italie
	Bertrand LE GAL	Maître de Conférences, ENSEIRB Matmeca, Laboratoire IMS, Talence
Dir. de thèse :	Guy GOGNIAT	Professeur des Universités, UBS Lorient, Lab-STICC
Co-dir. de thèse :	Philippe COUSSY	Professeur des Universités, UBS Lorient, Lab-STICC
Encadr. de thèse :	Jean-Christophe LE LANN	Enseignant-Chercheur, ENSTA Bretagne, Lab-STICC, Brest

Contents

Contents iii

List of Figures v

List of Tables vii

Introduction 1

1 Background and Related Work 7

1.1 Hardware Design Process 7

1.1.1 IC Design Cycle 7

1.1.2 HLS 12

1.1.3 Hardware IPs 14

1.1.4 Hardware Design Threats 15

1.2 Hardware IP Protection 18

1.2.1 Foundry 18

1.2.2 Layout Level and Gate-Level Netlist 19

1.2.3 RTL 24

1.2.4 Behavioral 26

1.2.5 Threats and Countermeasures 28

1.3 Software Obfuscation 29

1.3.1 Definition and Principles 29

1.3.2 Threat Models and Use cases 31

1.3.3 Taxonomy and Common Transformations 32

1.3.4 Commercial and Open-source Tools 35

1.3.5 Evaluation Metrics . . . 36
1.4 Conclusion . . . 37

2 Transient Obfuscation against IP Theft during Cloud-based HLS 39
2.1 Threat Model . . . 40
2.1.1 HLS-as-a-Service . . . 40
2.1.2 Threat Model . . . 41
2.2 Related Work and Background . . . 42
2.2.1 Cloud Computing Challenge: Ensuring Data Security . . . 42
2.2.2 Algorithm Level Obfuscation . . . 43
2.3 Proposed Approach: Transient Obfuscation . . . 44
2.3.1 Transient Obfuscation . . . 44
2.3.2 Complete Flow . . . 46
2.4 Proposed Obfuscation Techniques . . . 48
2.4.1 Bogus Code Insertion . . . 48
2.4.2 Control Flow Flattening . . . 52
2.4.3 Further Hiding of the Control Flow . . . 54
2.4.4 Data Obfuscation by Literal Replacement . . . 55
2.5 De-obfuscation: Making the Process Transient . . . 56
2.5.1 Pedagogical Example . . . 57
2.5.2 Naive Key Injection . . . 58
2.5.3 Targeted RTL Modification . . . 60
2.5.4 Full RTL De-obfuscation . . . 61
2.6 KaOTHIC: Key-based Obfuscating Tool for HLS In the Cloud . . . 62
2.6.1 Obfuscation Flow . . . 62
2.6.2 Adding Randomness . . . 63
2.6.3 Obfuscation Parameters . . . 64
2.7 Experimental Setup and Results . . . 65
2.7.1 Test Flow . . . 65
2.7.2 Overhead - Results and Analysis . . . 66
2.7.3 Transience and Security: Discussion . . . 71
2.8 Conclusion . . . 73

3 Transient Obfuscation for BIP Birthmarking 75
3.1 Threat Model and Background . . . 76
3.1.1 Threat Model . . . 76
3.1.2 Watermarking . . . 77
3.1.3 Effects of Code Transformations on High-Level Synthesis . . 78
3.2 Related Work . . . 79
3.2.1 BIP Watermarking Techniques . . . 79
3.2.2 Software Watermarking Techniques . . . 79
3.3 Proposed Approach . . . 80
3.3.1 Complete Flow and Notations . . . 80
3.3.2 Watermark Insertion . . . 82
3.3.3 Watermark Verification: Birthmarking Concepts . . . 83
3.4 Metrics for Watermark Verification . . . 87
3.4.1 Scheduling . . . 87
3.4.2 Dataflow . . . 91
3.5 Implementation and Experimental Setup . . . 96
3.5.1 Experimental Setup and Dataset . . . 96
3.5.2 Scheduling . . . 97
3.5.3 Dataflow . . . 97
3.5.4 Crokus . . . 98
3.6 Results and Analysis . . . 99
3.6.1 Scheduling . . . 99
3.6.2 Dataflow . . . 105
3.6.3 Benchmark Results . . . 108
3.7 Conclusion . . . 110

4 Transient Obfuscation against Hardware Trojan Insertion 111
4.1 Threat Model and Definitions . . . 112
4.1.1 Hardware Trojans . . . 112
4.1.2 Threat Model . . . 113
4.1.3 Examples of HTs in BIPs . . . 116
4.2 Hardware Trojan Countermeasures: Related Work . . . 118
4.2.1 Detection and Run-time Monitoring . . . 118

4.2.2 Design for Security . . . 120
4.3 Proposed Countermeasures for Hardware Trojan Insertion during HLS . . . 121
4.3.1 Payload: Removal by De-obfuscation . . . 121
4.3.2 Combinational Trigger: Detection during De-obfuscation . . . 124
4.4 Experimental Setup and Preliminary Results . . . 127
4.4.1 "Collatz" Hardware Trojan . . . 127
4.4.2 HT Removal . . . 129
4.4.3 Discussion . . . 132
4.5 Conclusion . . . 133

Conclusion **135**

List of published contributions **139**

Bibliography **141**

List of Figures

1.1 Gajski-Kuhn Y chart. 8
1.2 IC design flow. 10
1.3 HLS design flow. 13
1.4 HT attacks at different design stages. (from [Bhu+14]) 17
1.5 Standard and camouflaged cell layouts for NAND [(a) and (c)] and NOR [(b) and (d)] gates. (from [Raj+13]) 21
1.6 Logic locking example on c17 benchmark. The correct key inputs are k0=1, k1=0, k2=1. (from [DF19]) 22
1.7 Active hardware metering design flow. (from [CZZ17]) 23
1.8 FSM watermarking example: (a) original FSM, (b) adding transitions, (c) augmenting input and adding transitions. (from [TC00]) 24
1.9 Modified FSM for key-based RTL obfuscation. (from [CB10]) . . . 25
1.10 Classification of obfuscating transformation mechanisms. (from [Hos+18]) 33

2.1 Vulnerable HLS in the cloud design flow. 42
2.2 HLS in the cloud design flow secured by KaOTHIC, where the external HLS is considered untrusted. 47
2.3 Behavioral and dataflow representation of a code with bogus code insertion. 49
2.4 Basic block example before and after obfuscation: insertion of two bogus expressions. 50
2.5 Different obfuscation techniques applied by KaOTHIC to a function calculating the gcd of two numbers. 51

2.6 Example CFG before and after obfuscation by adding bogus basic blocks. 52
2.7 Example CFG before and after obfuscation by control flow flattening. 53
2.8 Example CFG before and after obfuscation by adding bogus transitions. 55
2.9 Example of a de-obfuscation by forcing input values. 57
2.10 Example RTL datapath before and after de-obfuscation. 59
2.11 Obfuscation flow in KaOTHIC compiler. 62
2.12 Average design overhead per obfuscation level, for bogus expression insertion - ASIC. 68
2.13 Average design overhead per obfuscation level, for bogus expression insertion - FPGA. 68
2.14 Design overhead per branching degree, for bogus basic block insertion - FPGA - **obfuscated**. 69
2.15 Design overhead per branching degree, for bogus basic block insertion - FPGA - **de-obfuscated**. 70
2.16 Cyclomatic complexity increase of Needwun source code after obfuscation per obfuscation level. 72

3.1 Watermarking Threat Model. 76
3.2 Watermarking Flow. 81
3.3 Birthmark Extraction Flow. 85
3.4 Two scheduling examples. 89
3.5 Partial DFG of original and obfuscated code. 91
3.6 Example of dataflow sequences for two designs. 94

4.1 HT Taxonomy - Trust-Hub.org 114
4.2 HT insertion by HLS tool - threat model. 115
4.3 Simple example of a HT inserted in a BIP. (from [VS16]) 117
4.4 HT removal by de-obfuscation flow. 122
4.5 HT detection during de-obfuscation. 125
4.6 No Trojan, these lines are removed during de-obfuscation. 126
4.7 Key1 and Input1 are used as triggers for the Trojan. 126

4.8 Code example of a Collatz HT inserted in a greatest common divisor calculation. 128
4.9 Theoretical and experimental HT removal rate. 131

List of Tables

1.1 Common hardware threats and countermeasures. 28

2.1 Average design overhead on ASIC. 66

2.2 Average design overhead on FPGA. 66

3.1 Similarity and containment of scheduling: experimental results. . . 99

3.2 Using **scheduling containment** as classifier: experimental results. 101

3.3 Using **scheduling similarity** as classifier: experimental results. . . 102

3.4 Similarity and containment of scheduling: experimental results with **obfuscation level $\geq$ 50%**. 103

3.5 Using **scheduling similarity and containment** as classifier: experimental results with **obfuscation level $\geq$ 50%**. 103

3.6 Similarity and containment of scheduling: experimental results with **5% bogus operators**. 104

3.7 Using scheduling similarity and containment as classifier: experimental results with **5% bogus operators**. 105

3.8 **Containment** of dataflow: experimental results with obfuscation level $\geq$ 50%. 106

3.9 **Similarity (Levenshtein)** of dataflow: experimental results with obfuscation level $\geq$ 50%. 106

3.10 **Similarity (subsequence)** of dataflow: experimental results with obfuscation level $\geq$ 50%. 107

3.11 Using dataflow containment and similarities as classifiers: experimental results. 107

3.12 Similarity and containment of scheduling: experimental results with obfuscation level $\geq 50\%$ for **adpcm**. 108
3.13 Similarity and containment of scheduling: experimental results with obfuscation level $\geq 50\%$ for **AES**. 108
3.14 Using scheduling similarity and containment as classifier: experimental results for **adpcm**. 109
3.15 Using scheduling similarity and containment as classifier: experimental results for **AES**. 110

4.1 HT Removal Rate: Experimental Results. 130

Introduction

Context

In 1959, Jack Kilby at Texas Instruments and Robert Noyce of Fairchild Semiconductor Corporations both worked independently and received patents for the invention of the first integrated circuit (IC). Made of a large number of transistors and based on semiconductor material such as silicon, the IC quickly became the basis of almost all modern electronics. Today, this invention has changed not only the world of electronics, but every aspect of our society. Computers, mobile phones and other portable devices, gaming consoles, modern cars, and even household appliances all rely on ICs. Semiconductors are now present everywhere, at an incredible amount. No other device in history has been manufactured as often as the transistor[1]. The global IC market has reached $360 billion in 2020[2], and is expected to grow to $468 billion by 2025. The main growth driver nowadays is the increasing use of connected devices and the widespread adoption of the internet of things (IoT) in everyday life. This continuous increase of sales can also be linked to the significant leaps that the industry continues to make in regards to ever higher performances and always smaller size.

However, this frantic market growth and continuing improvements have also lead to tighter time-to-market constraints, while ICs have steadily become more

[1] 13 Sextillion & Counting: The Long & Winding Road to the Most Frequently Manufactured Human Artifact in History. *https://computerhistory.org/blog/13-sextillion-counting-the-long-winding-road-to-the-most-frequently-manufactured-human-artifact-in-history/*

[2] Integrated Circuits Global Market Report 2021. *https://www.thebusinessresearchcompany.com/report/integrated-circuits-global-market-report*

complex, time-consuming and expensive to produce. These factors have lead to a globalization and a fragmentation of the whole IC supply chain, including both manufacturing and the design process.

Most semiconductor companies used to be integrated device manufacturers (IDM), meaning that they were responsible for both design *and* manufacturing. Nowadays, the cost of building new foundries to keep up with technological advances has become prohibitively expensive: market leader TSMC's newest foundry for 3nm technology, expected to start production in 2022, has been estimated to cost up to $20 billion[3].Today, it is no longer economically viable for most companies to build and maintain their own foundries. Only a few IDMs such as Texas Instruments and Intel remain. Most of the industry has evolved to a dual model, where design and manufacturing are completely separated: *fabless* companies, such as Nvidia or AMD, are responsible for design of new ICs, while *pure-play* foundries such as TSMC or UMC manufacture the devices for them. Moreover, test, assembly and packaging are increasingly getting outsourced as well, to so-called outsourced semiconductor assembly and test (OSAT) vendors.

This fragmentation of the supply chain is also directly affecting the design houses. To be able to keep up with increasingly tight time-to-market constraints, many rely on third party intellectual properties (IPs), reusable blocks of hardware designs sold by specialized vendors. In 2020, the worldwide semiconductor IP market is estimated at $5.6 billion[4]. Design teams themselves are more and more spread out around the world, with the creation of one design potentially involving specialists located in different countries or even on different continents. Finally, the complexity of modern ICs has imposed the use of elaborate toolchains involving several distinct tools sold by different specialized electronic design automation (EDA) vendors.

This ongoing globalization has opened up many possibilities for industry advancement, but has also created several vulnerabilities to attacks. IP theft and illegal reuse, reverse-engineering, overproducing and counterfeiting are all known

[3]TSMC Completes Its 3nm Multi-Billion Fab. *https://www.tomshardware.com/news/tsmc-3nm-fab-completed*

[4]Semiconductor Intellectual Property (IP) Market - Global Forecast to 2025. *https://www.marketsandmarkets.com/Market-Reports/semiconductor-silicon-intellectual-property-ip-market-651.html*

challenges that the IC supply chain faces today. For example, it is estimated that counterfeit ICs represent up to 1% of semiconductor sales ([KP13]), leading to a loss of about $100 billion of revenue for electronics companies ([PT06]). Malicious modifications of ICs during design or manufacturing in the form of hardware Trojans are also a growing concern. Several incidents that probably constitute hardware Trojan attacks have already been reported: the critical failure of Syria's air defense system during an Israeli air strike in 2017 is reported to have been intentionally triggered remotely through a so-called kill switch hidden in a microprocessor ([Ade08]).

Research efforts in the past years have been concentrated on securing all steps of the IC design cycle, with special focus brought to manufacturing. However, not much attention has been brought so far to the security issues and protection possibilities linked with behavioral synthesis. High-level synthesis (HLS) is a technique that aims at converting untimed, behavioral level descriptions of circuits (e.g. in C/C++) to register transfer level (RTL) descriptions, written in hardware description languages such as VHDL or Verilog. While HLS has been a promising research subject since the 1980's, it has gained renewed attention in the last ten years and is now increasingly adopted by design houses as powerful leverage for increasing design productivity. However, using HLS also introduces new vulnerabilities and new attack vectors in the IC design cycle. While most works consider HLS to be a secure design step, done in-house with trusted tools, some recent publications have raised concerns about security risks at the behavioral level: [Pil+18a] [Bas+19] [VS17a]. It is thus imperative to address the potential threats introduced in the design cycle by HLS, as well as the risks that behavioral level IPs face.

Contributions

This thesis studies different security issues related to behavioral level IPs and synthesis. It focuses on a novel obfuscation concept and its applications towards security during untrusted, in-house or external HLS, as well as towards securing behavioral IPs before, during and after HLS. The main contributions of this thesis are as follows:

- A novel concept for high-level, key-based hardware obfuscation: *transient obfuscation*, which involves an obfuscation and a de-obfuscation step to protect IPs at design time, specifically *during* HLS.
- Several practical techniques for transient obfuscation, created or adapted from existing software obfuscation techniques.
- A fully automated toolset, KaOTHIC, that implements the previous obfuscation techniques, as well as several tools for de-obfuscation.
- Three applications of transient obfuscation, with for each a detailed threat model, a characterization of the approach, a practical implementation and experimental validation:

  1. A low overhead design flow for untrusted, cloud-based HLS, secured through transient obfuscation.
  2. A method for exploiting transient obfuscation side effects as a birthmark to identify stolen behavioral IPs.
  3. Two methods for using transient obfuscation against hardware Trojan insertion during HLS.

Outline

Chapter 1 begins by presenting the IC design cycle, with the different steps and actors involved. We give a more in-depth explanation of high-level synthesis, which is the main focus of this thesis. We then give an overview of different security threats, as well as a state-of-the-art of existing IP protection methods at different abstraction levels. Finally, since our focus in this work is on hardware protection at the behavioral source code level, we also give a presentation of software obfuscation principles and techniques.

In **Chapter 2**, we focus on the risk of IP theft during cloud-based HLS. We start by explaining why cloud-based HLS is a likely scenario and giving the security risks involved with this model. After a quick overview of relevant related work and background information on cloud security, we introduce our approach:

transient obfuscation. We enumerate several obfuscation techniques and explain different methods for de-obfuscation. This chapter also presents a tool, KaOTHIC, implemented to apply transient obfuscation on C code, and finishes with a series of experiments to test our approach.

Chapters 3 and 4 extend the previous work by showing how transient obfuscation can also be used for other security purposes. In **Chapter 3**, we focus on the subject of stolen behavioral IP identification through watermarking, and in particular birthmarking. After presenting the threat model, as well as background and related work on watermarking, we propose a birthmark-based approach that uses the side-effects of transient obfuscation. We give a detailed presentation of this approach, before providing an experimental setup to validate our method. The end of the chapter presents the experimental results.

Chapter 4 in turn focuses on the subject of protection against hardware Trojans. We start by introducing the threat model, based on the risk of Trojan insertion by a malicious HLS tool, and by presenting related work on Trojan countermeasures. Then we explore two different methods for using transient obfuscation against Trojan insertion. An experiment to validate the first method is described and its results are presented. We also introduce a tool for hardware Trojan insertion in source code, used in the experimental setup.

Chapter 1

Background and Related Work

The hardware design process is more fragmented and complex than ever. The integrated circuit (IC) design cycle involves a multitude of design and validation steps at different abstraction levels, several actors working together, as well as complex toolchains and methodologies. In this chapter, we start by presenting background information about the IC design cycle, as well as the actors and concepts involved. We also give a short introduction into high-level synthesis, which is the step in the hardware design cycle we focus on in this work. The aforementioned complexity of the hardware design ecosystem has lead to the identification of several security threats. We present three of the main threats: reverse-engineering, piracy and hardware trojan insertion. To thwart these security issues, many methods have been developed. We give an overview of the protection techniques at different abstraction levels, while pointing out works that are relevant to the subject studied in this thesis, security during HLS. Finally, since our work aims at protecting high-level source code, we draw parallels with the software protection world by presenting software obfuscation principles and methods.

1.1 Hardware Design Process

1.1.1 IC Design Cycle

Integrated Circuits (ICs), also called chips, are electronic devices based on semiconductors (silicon) and mainly made of transistors. Nowadays, with the advances

of very-large-scale integration (VLSI), there are well over ten billion transistors on one chip, with the most recent progress in technology nodes as of 2020 being 3nm transistors. Several different types of ICs are fabricated. SoCs (system on chip) combine all or most components of a computer on a single chip. They contain a processor, memory units and advanced peripherals such as a GPU. ASICs (application specific integrated circuit) were initially ICs without a processor, designed for one specific application, for example in routers or switches. Nowadays most ASICs also have a SoC-type architecture, with one or several processors embedded on the chip. Their topology is decided in advance and not reconfigurable, but they can be parametrized to provide higher flexibility. They are usually manufactured in high volume, since their design cycle is time and resource consuming. In recent years, there has also been increasing interest for FPGAs (field programmable gate array). Contrary to ASICs, FPGAs are reconfigurable and programmable. They are for example used in digital signal processing (DSP) and for prototyping other ICs.

Design Levels

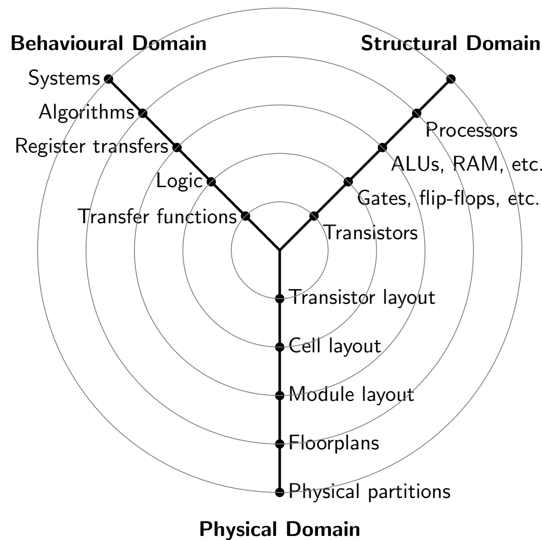


Figure 1.1 – Gajski-Kuhn Y chart.

IC design cycles are complex and can be separated into several steps at different

abstraction levels. These levels have been popularized by Gajski and Kuhn in 1983 in the so-called *Y diagram*, depicted on Figure 1.1. This chart shows three different domains used to describe hardware development: behavioral, structural and physical. Each axis on the diagram represents one domain in a top-down approach.

On Figure 1.2, the full design cycle with following abstraction levels is depicted:

- **System level**: during system specification, the functional requirements and external interfaces of the system (IC) are defined.
- **Behavioral level**: at this level, the function of the IC is described as an algorithm using high level languages such as C/C++ or SystemC. High-level synthesis (see Section 1.1.2) is used to convert this description into RTL.
- **Register-Transfer Level (RTL)**: the complete IC design is captured using hardware description languages (HDLs) such as Verilog or VHDL. At this level, the circuit is modeled using sequential logic (registers) and combinational logic. The circuit is timed and its interfaces (input/output pins) and connectivity are defined. Logic synthesis is used to convert a RTL description into a gate-level netlist
- **Gate-level netlist**: at this level, the IC is implemented for a specific technology library in terms of interconnected logic gates, memory elements (flip-flops) and inverters. The netlist contains all the components of the circuit and their connectivity. A physical synthesis is then used to further specify the design.
- **Layout level**: during physical synthesis, several technology-specific steps such a floorplanning, partitioning, place-and-route are performed. These result in a low-level netlist, and finally in a graphical database system (GDSII) file, a binary file used to describe the full IC which can be sent to the foundry.

IC Design Actors

The ever-growing complexity and worldwide distribution of the IC supply chain means that there is a growing number of distinct actors involved in the design of

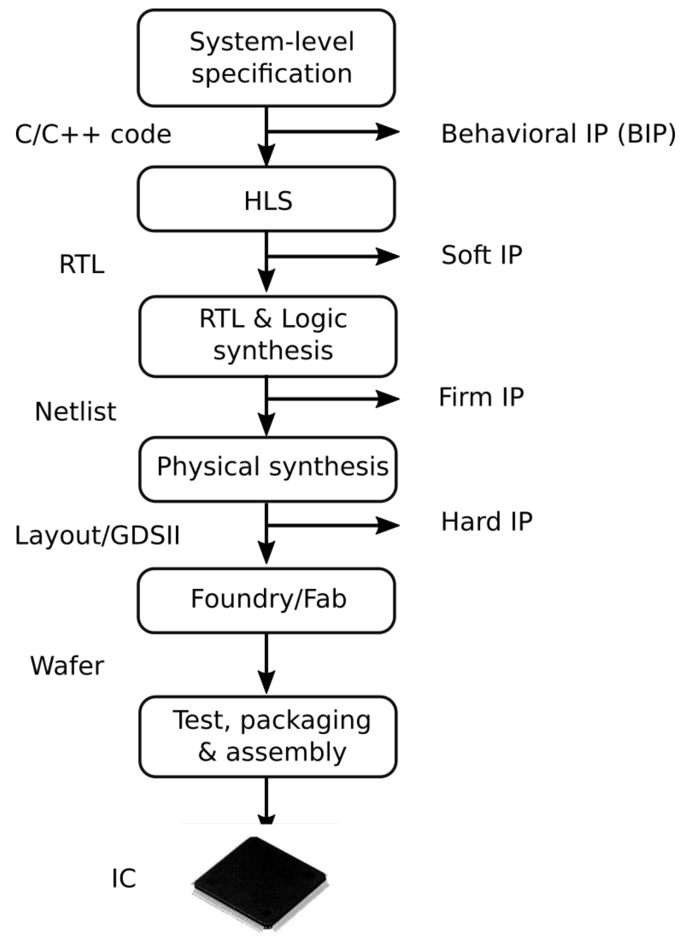


Figure 1.2 – IC design flow.

an IC. Each of these actors adds new vulnerabilities to the whole design flow.

- **Customer**: the customer orders an IC by giving the specifications, usually at system level, to a design house.
- **Design house**: the design house defines the IC's specifications and performs the different design steps up to the physical layout, including several testing and verification steps. During this process, reusable intellectual property (IP) blocks from external vendors can be bought and combined with in-house components. The biggest design houses include Broadcom, Qualcomm, Nvidia and AMD.
- **Third-party IP (3PIP) vendors**: these vendors sell specialized IP blocks at different abstraction levels (see Section 1.1.3). The IPs are already optimized and are widely used nowadays: the global IP market is expected to reach $8.81 billion by 2026[1]. By using 3PIPs, the design house can reduce design costs and reach better time to market thanks to faster design cycles.
- **Foundry**: Foundries are responsible for manufacturing wafers and dies. The cost of building a new fab has been steadily increasing, reaching over $1 billion. The latest foundries for 3nm technology have been estimated to cost up to $20 billion[2]. Due to this high entry cost, most design houses have moved to a *fabless* model, meaning that they outsource IC fabrication to specialized foundries instead of having an in-house foundry. Most of the foundries, e.g. TSMC, are located in China and Taiwan.
- **Assembly**: After fabrication, a separate actor is often responsible for packaging and testing of the IC, before shipping it to the design house or directly to market.
- **EDA tool vendors**: The growing complexity of IC design has led to the need for specialized tools such as high-level and logic synthesis tools or sim-

[1] Global Semiconductor Intellectual Property (IP) Market Analysis 2020. *https://www.researchandmarkets.com/reports/5236592/global-semiconductor-intellectual-property-ip*

[2] TSMC Completes Its 3nm Multi-Billion Fab. *https://www.tomshardware.com/news/tsmc-3nm-fab-completed*

ulation tools. Electronic design automation (EDA) tool vendors sell their tools to design houses. The best known vendors include Synopsys, Cadence and MentorGraphics.

1.1.2 HLS

To deal with the ever growing complexity of hardware systems and need for better design productivity, abstraction levels have been raised several times over the years, along with an increased automation of design methods and tools. Since the 1980s, low-level design at the gate and layout level has been replaced with the use of dedicated hardware description languages (HDL), mostly at register transfer level (RTL). The two most used HDLs are VHDL and Verilog. More recently, there has been a trend towards using new HDLs based on domain specific languages (DSLs) hosted by mainstream programming languages [Ker+19][LBK20]. At even higher abstraction level, high-level synthesis (HLS) has been a serious research subject since the 1990s, when the first commercial tools were made available [Kna96]. HLS tools transform an algorithmic level, usually untimed, description of an application into a fully timed hardware implementation at RTL. The resulting architecture, usually described using a HDL such as Verilog or VHDL, contains a controller, a data path, memory banks and communication interfaces [Cou+09].

As shown on Figure 1.3, HLS tools perform the following tasks: parsing the source code, generating internal representations (IR, CDFG), allocation, scheduling, binding, and RTL generation. Allocation, scheduling and binding are usually performed in sequence, but the particular order can vary depending on the HLS tool [WC12]. These steps are detailed below.

Front- and Middle-end

The first HLS step starts with a algorithm's source code written in a high level language such as C, C++ or SystemC. A compiler front-end parses this code into an abstract syntax tree (AST), and then elaborates an intermediate representation (IR), which is chosen to ease development of most optimization passes. IRs such as LLVM or Gimple present themselves as assembly code independent from the final target. They provide an unlimited amount of registers and usually take a static

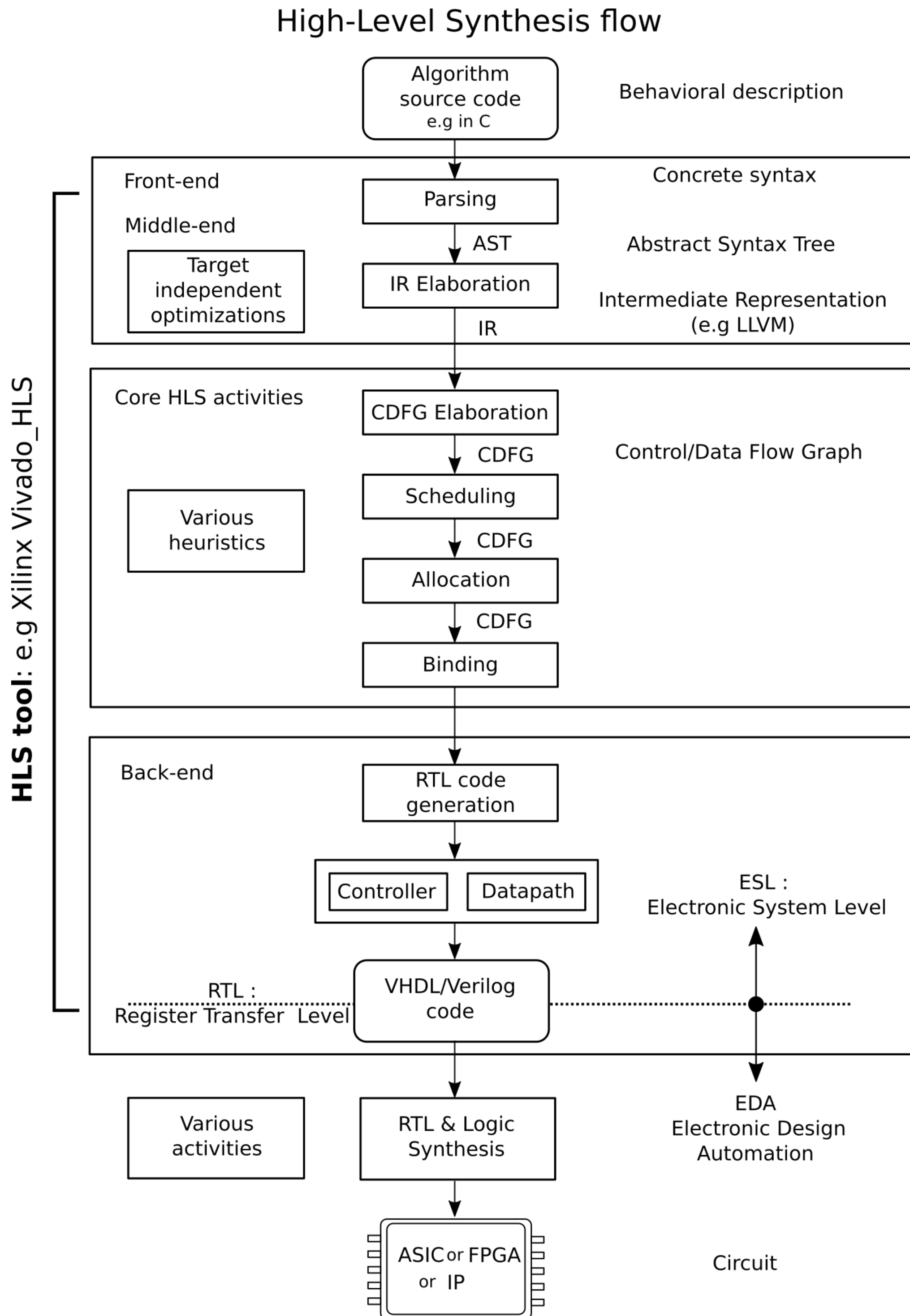


Figure 1.3 – HLS design flow.

single assignment (SSA) form, which explicitly describes dataflow relationships between variable definitions, assignments and uses. At this level, several optimization passes such as for example dead code removal or loop unrolling are performed. Finally a control and data flow graph (CDFG) is built to formally represent all data dependencies (in the form of basic blocks) and control dependencies (in the form of edges between the basic blocks).

Scheduling, Allocation and Binding

During the scheduling step, each operation is scheduled into one or several clock cycles. Several operations can be scheduled in parallel as long as there are no data dependencies between them. During allocation, the hardware resources necessary for the design are selected from an RTL component library. These resources include functional units and storage units. During binding, each operation is bound to one of these functional units, and each variable to one of the storage units. Resources can be shared by several operations or variables. Detailed presentations of these steps and the algorithms used for optimizing them can be found for instance in [Gaj+92].

Back-end

During the back-end step, the RTL architecture in form of a datapath and a controller is generated according to the design decisions taken during scheduling, allocation and binding. The datapath is made of functional units and registers and other storage elements, while the controller is a finite state machine (FSM) which controls the data flow. Finally, RTL code, usually written in a HDL such as Verilog is generated.

1.1.3 Hardware IPs

Intellectual Property (IP) blocks, i.e. reusable components, are now widely integrated into hardware design cycles.

Different IP block types have been defined at different abstraction levels, as shown on Figure 1.2:

- Hard IPs: these IPs are mapped to a specific technology and are fully optimized in terms of performance. They are usually sold as low level circuit descriptions, for example in the form of fully placed and routed netlist, or full physical layouts.
- Firm IPs: these IPs are less specific than hard IPs, while still optimized for performance. They are sold as detailed floorplans combined with a netlist and usually synthesizable RTL. Contrary to hard IPs, firm IPs are not routed and are less technology dependent.
- Soft IPs: these IPs are defined at high level and usually sold in the form of synthesizable HDL (Hardware Description Language) code. They are more flexible and easier to modify and adapt than other IPs, but also offer less predictable performances in terms of timing, area, power consumption etc.
- Behavioral IPs (BIPs): BIPs have similar properties to soft IPs, but are sold as a description at behavioral level.

1.1.4 Hardware Design Threats

The complexity and distributed character of hardware design cycles, with the involvement of several different actors, creates several different threats on design data and ICs.

Reverse-Engineering

Reverse-engineering an IC means trying to identify its functionality, design and structure. It can have several goals such extracting the transistor-level or gate-level netlist [VD01], identifying what technology is used, or understanding the functionality. Several tools and techniques have been developed to facilitate reverse-engineering. Reverse-engineering can be performed by several actors:

- IC user: the end user can de-package and delayer an IC, capture images of these layers and then using them to extract the netlist.
- Foundry: an attacker in the foundry can use the IC layout to reverse-engineer it.

- Higher level: during earlier design steps, for example during SoC integration, external IPs can also be reverse-engineered.

Design steps earlier in the flow are particularly vulnerable to reverse-engineering, for example BIPs and soft IPs, due to their better readability and high level of abstraction. Once reverse-engineering has been successfully performed, the IC or IP can easily be modified, copied and/or illegally resold. A design house or IP vendor can also use it for corporate espionage and to gain knowledge about a competitor's technology.

IC Piracy, Overbuilding and Counterfeiting

The high complexity and cost of fabrication has lead to a fabless model where design companies are no longer able to afford an in-house foundry: instead, they rely on external foundries. This model leads to several security risks during fabrication. IC descriptions can be stolen during fabrication and adjusted for another foundry [RKM08], which can use them to illegally produce ICs. The attacker can then claim ownership of the stolen ICs. At higher level, IP vendors can also be victims of illegal copying of their IPs, which can be resold on the market or copied and modified by their competitors.

Another possible threat is overbuilding or overproducing, where a foundry simply produces more chips and sells them on the black market without authorization from the design company.

On top of the previous risks, counterfeiting, where faulty or outdated ICs are sold as new, is also a threat. According to the authors in [Gui+14], more than 80% of all counterfeit ICs are recycled components that are sold as new after being remarked and repackaged. These counterfeit ICs represent up to 1% of all IC sales [KP13], causing an estimated loss of revenue of $100 billion [PT06].

Hardware Trojan Insertion

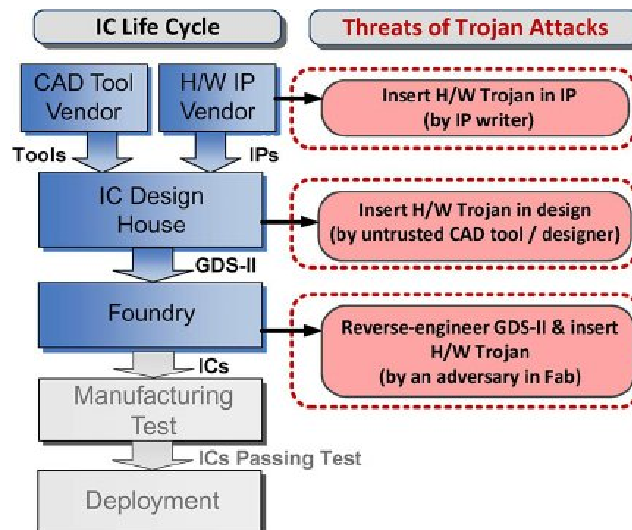


Figure 1.4 – HT attacks at different design stages. (from [Bhu+14])

Hardware Trojans are malicious modifications of a circuit, where the attacker surreptitiously inserts additional logic into the circuit. Hardware Trojans (HTs) can have several goals, such as leaking sensitive information, lowering the performance of the circuit, or creating unwanted behavior and causing errors. HTs can be inserted during manufacturing, but also at earlier design stages, by EDA tools or in 3PIPs, as illustrated on Figure 1.4. HTs are hard to detect during validation and testing, and can have significant impact on the circuit.

1.2 Hardware IP Protection

To prevent IP theft, counterfeit and other security issues, several different protection methods have been extensively studied. These methods can be classified according to the technique used: watermarking, obfuscation, locking, etc. They can also be divided into passive and active methods, where passive methods are used to detect if an illegal action such as copying of the circuit has occurred, while active methods are used to prevent such actions. In this section, we give an overview of different design protection methods classified by abstraction level. We focus mainly on active methods. A more in-depth review of some protection mechanisms, in particular at behavioral level, can be found in Section 2.2.2, Section 3.2.1 and Section 4.2.

1.2.1 Foundry

To prevent IP theft and other attacks at the foundry level, split manufacturing has been proposed as a patent in [JM07]. With this technique, designs layouts are split into two parts: Front End Of Line (FEOL) layers and Back End Of Line (BEOL) layers, which are then fabricated in two different, independent foundries. The FEOL is made of the lower layers such as transistors, capacitors and resistors. These expensive layers need to be fabricated in a high-end foundry capable of fabricating the finest components. Most, if not all design houses, have to outsource this fabrication to an untrusted foundry. On the other hand, the BEOL layers (top metal layers, interconnects) can be manufactured in a lower end, but trusted foundry. Larger design houses can even use their own, in-house foundry. The BEOL can either be built directly on top of the previously fabricated FEOL, or both can be manufactured separately and then combined. By splitting designs in such a manner, the untrusted foundry does not have access to the full design, and specifically to the interconnect network.

However, research has demonstrated that it is possible for an attacker to recover the missing BEOL connections by reverse-engineering and analyzing the FEOL. This *proximity attack* [RSK13] relies on the following heuristic used in most floorplanning and replacement tools: FEOL features are usually connected close to one

another, in order to reduce the wiring. With this attack, up to 96% of the BEOL interconnects can be recovered [RSK13]. Even more accurate results have been achieved by using deep learning techniques [Li+19a]. Split manufacturing thus needs to be combined with other defense mechanisms to ensure proper security.

1.2.2 Layout Level and Gate-Level Netlist

Physical Unclonable Functions

Physical unclonable functions (PUFs) are entities that, when given a *challenge*, i.e. an input and conditions, produce a *response* as output. Each PUF has different physical characteristics, due to random variations during the manufacturing process. These characteristics are impossible to duplicate and lead to each PUF giving a unique and unpredictable response to a challenge. The PUFs thus each have a set of unique *challenge-response pairs* (CRP). Due to fhe variations introduced by manufacturing, it is impossible to obtain two PUFs with the same CRP. This means that a PUF can be used as a means to uniquely identify and authenticate an IC. This method is called *challenge-response authentication*.

PUFs were first introduced in 2002 in [Gas+02] and have been gaining more and more interest over the last 20 years. Several desirable properties have been defined for PUFs [CZZ17]:

- **Unclonability**: each PUF has a number of unique CRPs, which cannot be duplicated in another PUF.
- **Unpredictability**: the other CRPs of a PUF cannot be used to predict the response to a challenge.
- **Reliability**: the PUF should always produce the same response to a given challenge.
- **Physical unbreakability**: the PUF should be tamper-proof.

Several different types of PUFs have been proposed such as arbiter PUFs [Lim+05], ring-oscillator PUFs [SD07] or SRAM PUFs [HBF08]. Detailed surveys and state of the art studies can be found for example in [MV10] or [BS19].

In recent years, several vulnerabilities and weaknesses of PUFs have been studied. For instance, PUFs are vulnerable to side-channel attacks [DV13], as well as machine learning (ML) based attacks [Her+14], where an algorithm is trained with a subset of CRPs of a PUF to predict responses to unknown challenges.

Watermarking

Watermarking relies on applying slight modifications to a design by inserting a watermark, then later extracting and verifying this watermark as a means of claiming ownership of a stolen IP (for a more detailed definition, see Section 3.2.1). Watermarking can be applied at all abstraction levels, with the watermark propagating to later design stages.

With *constraint-based* watermarking [Kah+98], the design house's signature is converted into a set of constraints. These constraints are added to the original design's constraints. This reduces the solution space available for a given optimization problem. The resulting design after synthesis and optimizations is a unique solution that it is highly unlikely to obtain without the watermarking constraints. Constraints can be added for several different optimization problems, such as:

- Adding clauses to a satisfiability problem such as 3SAT [Kah+98].
- Adding unique timing constraints to the timing constraints of a path by breaking it into subpaths [Jai+03].
- Modifying the routing of a design, for example by modifying the number of bends used to route the nets of the design [Nar+01].

The main drawback of low-level, constraint-based watermarks is that they are hard to detect and verify after fabrication of the IC [ATA03].

Fingerprinting

Fingerprinting is a method used to trace stolen IPs [Cal+04]. Each instance of the IP core is assigned a unique identifier, contrary to watermarking where all instances of the IP share the same identifier. In case of a stolen IP, this identifier can then be used to find who stole the IP. In [Cal+04], fingerprinting is combined

with a previous constraint-based watermarking scheme. The IP buyer transmits a signature (public key) to the IP designer. This signature is converted into fingerprinting constraints and added to the design house's watermarking constraints. These constraints are taken into account during synthesis. The resulting design has both the designer's watermark and the buyer's fingerprint.

Camouflaging

The goal of camouflaging is to prevent reverse-engineering of an IC by hindering image-based extraction of the netlist. Several different techniques for camouflaging are possible:

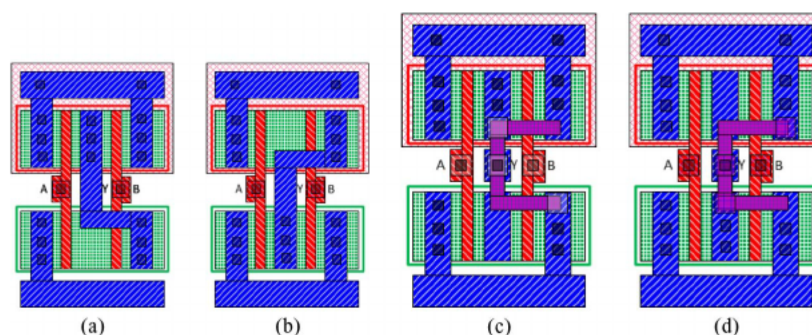


Figure 1.5 – Standard and camouflaged cell layouts for NAND [(a) and (c)] and NOR [(b) and (d)] gates. (from [Raj+13])

- Using filler cells to fill any empty space in the IC [Cho+12].
- Adding dummy contacts [CBC07] to the layout: these dummy contacts fake a connection between two layers but have a gap in the middle to prevent forming a real electrical connection.
- Using logic gates that are all designed to look identical.

An example for logic cells with identical layouts is given on Figure 1.5 as depicted in [Raj+13]. The figure first depicts standard NAND (Figure 1.5(a)) and NOR (Figure 1.5(b)) gates. These cells have different layouts and are easy to differentiate during reverse-engineering. On the other hand, (Figure 1.5(c)) and (Figure 1.5(d)) show camouflaged layouts for NAND and NOR gates which look identical.

Obfuscation and Logic Locking

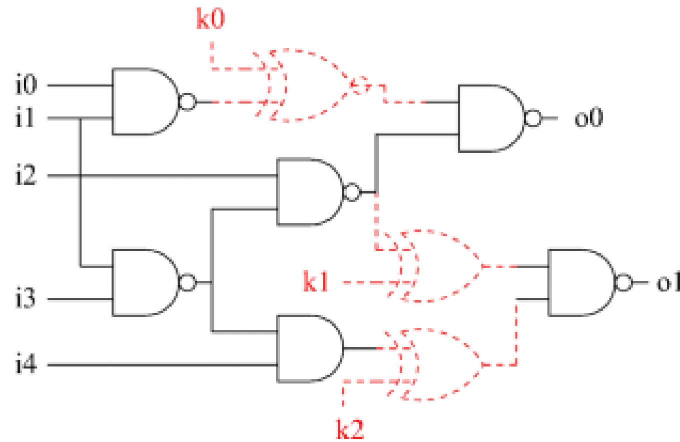


Figure 1.6 – Logic locking example on c17 benchmark. The correct key inputs are k0=1, k1=0, k2=1. (from [DF19])

Hardware obfuscation aims at hiding the functionality and implementation of designs. With logic locking, the circuit operates in a locked mode until the correct input key is applied and restores functionality. A simple example can be found on Figure 1.6. The terms "logic obfuscation" , "logic locking" and "logic masking" are often used interchangeably, depending on the author. A formal definition and clearer distinction between these techniques can be found in [CBH16]. In the rest of this work, we will refer to logic locking for techniques applied at gate level, and obfuscation for higher level techniques.

In [RKM08], the authors propose a combinational logic locking method. Support for public-key cryptography, a public key and some circuitry to support a locking mechanism are added to the RTL description. This enriched RTL is then synthesized into a gate-level netlist. At this level, combinational locking is performed on the circuit by adding XOR and XNOR gates. A random key is generated and sent to the IP rights holder. After fabrication, a key exchange mechanism with the foundry leads to the right locking key being produced by the chip. Once this correct key appears, the circuit is unlocked and behaves as expected. With any other key, the circuit malfunctions. Because this technique is vulnerable to attacks such as oracle-based attacks ([Raj+12]) and SAT attacks ([SRM15]), several improvements have been proposed. A detailed overview of such techniques can be

found in [DF19]. Another systematic analysis of the back-and-forth between research on locking techniques and new attacks, as well as a critique of logic locking evaluation, is presented in [Tan+20].

Instead of combinational locking, other works such as [CB09] or [AKP07] propose sequential locking, by adding a Finite State Machine (FSM) to the circuit. At power-up, this FSM locks the design. The correct sequence of inputs is necessary to unlock it. In [CB09], the proposed method, on top of locking the circuit until authentication, also obfuscates it's design to provide protection against reverse-engineering.

Hardware Metering

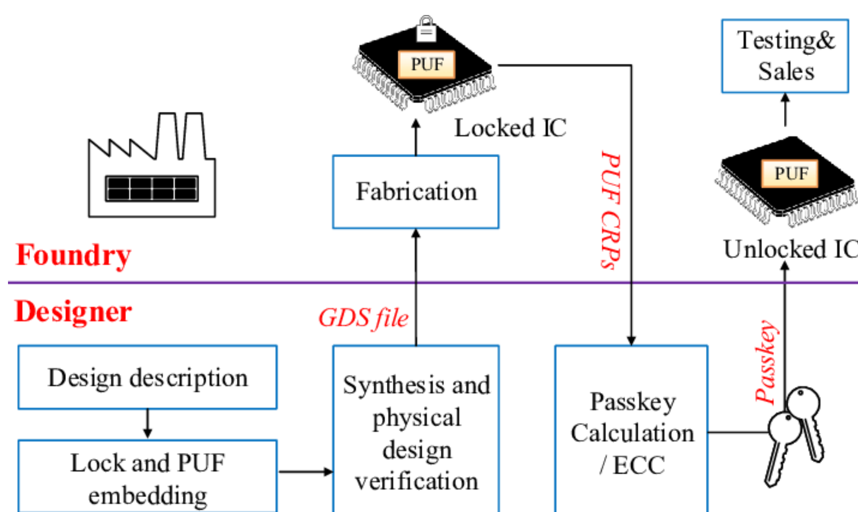


Figure 1.7 – Active hardware metering design flow. (from [CZZ17])

With hardware metering, the design house has access to the ICs after fabrication and can identify each IC individually. This method is used to track all manufactured ICs and can thus detect overproduced and cloned ICs. With active metering approaches, each IC is locked after fabrication and can only be unlocked by the design house. A unique key is used for each IC, contrary to logic locking where all produced ICs are locked with the same key. An example of such a flow is shown on Figure 1.7, as depicted in [CZZ17]: a lock and a PUF are embedded in the circuit before layout generation. After fabrication, a challenge given by

the designer is applied to each IC and the response to this challenge is sent to the design house. Here, the PUF is used to uniquely identify each IC. With this response, the designer can authenticate each IC and send an activation key back to the foundry. The IC is only unlocked with the correct key.

1.2.3 RTL

Watermarking

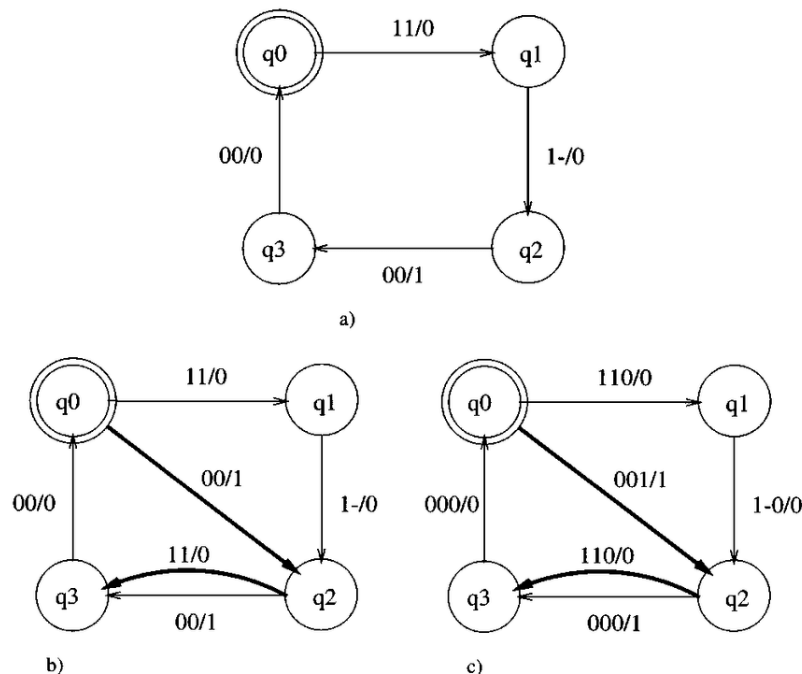


Figure 1.8 – FSM watermarking example: (a) original FSM, (b) adding transitions, (c) augmenting input and adding transitions. (from [TC00])

Watermarks can also be embedded at RTL, for example to protect soft IPs. One added challenge compared with lower level watermarking schemes is that the watermark must resist synthesis without being removed or modified by the tool. Some techniques rely on modifying the finite state machine (FSM) of a design. The approach introduced in [TC00], relies on adding bogus transitions on unused input/output pairs, as illustrated on Figure 1.8(b). On Figure 1.8(c), the previous method is extended by first augmenting the number of input bits and then adding

bogus transitions. In [Oli01], the FSM is also used to hide a watermark. With this approach, the owner's signature is encrypted, hashed and divided into a sequence of input combinations to the design. Then the state transition graph (STG) is modified by adding extra states and transitions, so that when this sequence of inputs is followed, the design reaches a particular sequence of states, exhibiting a specific property. Ownership is proven by providing both the input sequence and the specific property defined by the owner.

Instead of modifying the FSM, in [Cas+07], the authors propose to form a watermark by spreading a signature throughout the design, in memory structures or existing combinational logic. To be able to extract the watermark at layout level, their method requires adding some dedicated circuitry to the existing design.

The test circuit of an IP can also be used for watermarking. In [FT03], a watermark generating circuit is added to the test circuit before logic synthesis. After fabrication, when the IP is run in test mode, the watermark generating circuit is activated and a watermark is sent out with the test pattern. This method has the advantage that the watermark extraction process is much simpler than for other techniques. However, it fails if the attacker simply removes the test circuit [ATA04].

A full survey of IP watermarking techniques can be found for example in [ATA04].

Obfuscation

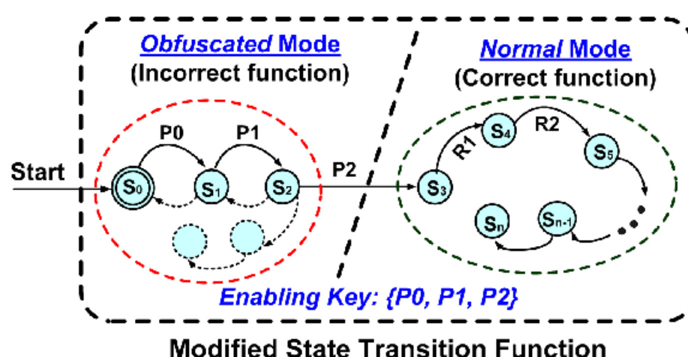


Figure 1.9 – Modified FSM for key-based RTL obfuscation. (from [CB10])

Similarly to logic locking at the gate level, key-based obfuscation can also be applied at RTL. In [CB10], the CDFG of an RTL IP is extracted. A "mode-control FSM" is then added to the IP, see Figure 1.9. This ensures that the circuit falls by default into an obfuscated mode, and only reaches normal mode if the correct input key sequence is applied ($P0 \implies P1 \implies P2$ on Figure 1.9).

Other approaches focus on obfuscating the HDL source files to reduce their readability, by reusing classic software obfuscation techniques: [BY07], [Mey+11]. However, these techniques do not prevent illegal copying and reuse, provide little resistance against automatic software de-obfuscation tools, and are sometimes removed by logic synthesis tools.

1.2.4 Behavioral

In this subsection, we quickly present an overview of a few approaches for IP protection at the behavioral level, before and during HLS. More detailed analyses of existing methods are given in Section 2.2.2, Section 3.2.1 and Section 4.2.

Watermarking

In [KHP05], the authors present a method to perform constraint-based watermarking during different steps of the HLS process. In particular, they detail how to insert a watermark during register allocation. However, they note that this method can also be used for other HLS tasks such as scheduling or partitioning. For register allocation, an interval graph is used to represent all the program variables as nodes, and their overlapping lifetimes as edges. A graph coloring algorithm is then used to solve the problem. The watermark is embedded by first encrypting and encoding the IP owner's signature. Then, edges are added to the interval graph according to this encoded signature. The additional edges result in additional constraints for the graph coloring problem. These specific constraints result in a design that is highly unlikely to obtain by coincidence, thus forming a watermark. [SBM16] proposes to improve the previous method by decreasing the watermark embedding cost and increase the resistance to attacks. By using particle swarm optimization (PSO) to explore watermarking solutions, the authors are able to find a low-cost optimal solution.

Several approaches for watermarking at the behavioral level targeting specifically digital signal processing (DSP) have been published, e.g. in [CD00] and [Ras+99]. In both cases, the watermark is embedded by performing minor changes in the requirements of filters. Verification is done by observing the slight modifications in the filter's response. These approaches are however highly sensitive [ATA04] to low-level design fluctuations as well as watermark removal attacks, since a minimal change in the filter's response can be enough to hide the watermark.

Obfuscation

Obfuscation can be applied at behavioral level, either on the algorithmic-level source code, or during HLS using a modified HLS tool. In [VS17a], the authors use traditional software obfuscation techniques applied on the source code before HLS. They then study the impact of this method on the HLS quality of results and propose two methodologies to find the optimal obfuscation. However, they only test basic layout obfuscation techniques, which make the code harder to read but do not affect functionality. Their techniques do not provide protection against IP theft and reuse, only reverse-engineering.

Similarly, in [SR17], the authors use compiler-based, high-level transformation to obfuscate IP cores. Their techniques are applied on the DFG and include redundant operation elimination, tree height reduction and logic transformation, where functions are replaced with other, logically equivalent functions. These techniques all change readability and hide the structure of the IP, but do not modify functionality.

In both [IK18] and [Pil+18b], the authors propose to modify an in-house HLS tool to apply obfuscation during HLS. This results in an obfuscated RTL description, which is protected both against IP theft at high level and against theft and reverse-engineering at foundry level. In [IK18], the obfuscation passes are applied on the CDFG during scheduling and datapath generation. In [Pil+18b], obfuscation is performed during all steps of the HLS flow, at front-end, mid-level and back-end. Both approaches offer *key-based* obfuscation, which, similar to logic locking at lower design levels, provides protection against illegal reuse.

1.2.5 Threats and Countermeasures

	Reverse-engineering	Hardware Trojans	IP Piracy & Overbuilding	Counterfeiting
Possible attackers				
Customer & End User	✓			
Design House & SoC Integrator	✓	✓	✓	✓
3PIP Vendor		✓		
Foundry	✓	✓	✓	✓
Test & Assembly	✓			✓
EDA Tool		✓	✓	
Countermeasures				
Split Manufacturing	✓	✓		
PUF				✓
Watermarking			✓	✓
Fingerprinting			✓	✓
Camouflaging	✓		✓	
Hardware Metering			✓	✓
Obfuscation	✓	✓	✓	✓

Table 1.1 – Common hardware threats and countermeasures.

The previously presented hardware IP protection methods can be used at different abstraction levels and against different threats. Table 1.1 sums up the different threats and their countermeasures. It also indicates which actors can be involved in each attack.

In this work, we focus on threats introduced by the use of EDA tools, in particular HLS tools. In Chapter 2, we present an obfuscation-based solution against IP piracy during cloud-based HLS. In Chapter 4, we focus on the risk of hardware trojan insertion by a malicious HLS tool. Finally, in Chapter 3, we show how our methods can also be applied as a watermarking countermeasure against reverse-engineering, IP piracy, overbuilding and counterfeiting at later design stages.

1.3 Software Obfuscation

Our focus in this work is on how to protect IPs against theft, counterfeiting and other threats at the *behavioral* level. Since behavioral IPs are usually provided in high level source code such as C/C++, protection methods can be strongly similar to those used for software protection. Software obfuscation specifically is of particular interest to our work. In this section, we thus give an overview of software obfuscation principles and techniques.

1.3.1 Definition and Principles

Informally, obfuscating a program means rendering a program unintelligible while keeping it functionally and semantically equivalent with the original program. Obfuscation is based on the paradigm of *security through obscurity*. The idea was introduced in 1984 at the International Obfuscated C Code Contest, and first mentioned in literature by Collberg et al. in [CTL97] in 1997.

More formally, O is an obfuscator if O takes as input a program P and produces as output a program $O(P)$ such as:

- $O(P)$ has the same functionality as P
- $O(P)$ is *harder to understand* than P

Program obfuscation is mainly used with the goal of protecting software intellectual property. It can be used to prevent reverse-engineering, but also to protect software watermarks or hide critical constants or predicates.

Most well known and used software obfuscation techniques rely on the predicate *security through obscurity*. The security of these techniques cannot be proven because they do not rely on a formal definition of obfuscation, since the notion of *harder to understand* has no clear formal foundation.

Practical program obfuscation is also called *code-based obfuscation*, opposed to model-based obfuscation [Xu+17]. Code-based obfuscation techniques focus on usability and are not provably secure. No metrics are known as of today to formally evaluate the security of these techniques. On the other hand, *model-based obfuscation* techniques are based on a strong theoretical background and

focus strongly on security. However, this comes with significant overhead and a lack of usability.

A formal description of what is called *virtual black box obfuscation* is given by Barak et al. in [Bar+12]. According to this definition, if a program is obfuscated, it should not be possible to gain more information from seeing the obfuscated code than from interacting with the unobfuscated code in a black box. In other words, other than observing inputs and outputs, there should be no information whatsoever to gain about the obfuscated program. With this definition, an obfuscator O can be formally defined by two conditions:

- The obfuscated program $O(P)$ satisfies the previously mentioned virtual black box property.
- The obfuscated program $O(P)$ has the same functionality as the unobfuscated program P.

In [Bar+12], the authors prove that black box obfuscation is impossible, in the sense that it is impossible to have one obfuscator that can obfuscate all programs of a certain class. However, they also give a new possible definition for obfuscation, *indistinguishability obfuscation*:

- The obfuscated program $O(P)$ has the same functionality as the unobfuscated program P.
- Indistinguishability: given two functionally equivalent programs P_1 and P_2 of approximately the same size, the obfuscated programs $O(P_1)$ and $O(P_2)$ are computationally indistinguishable from one another.

While this definition is slightly weaker than full *black-box obfuscation*, research has revealed encouraging progress in this direction: a first candidate for building indistinguishable obfuscators was published in 2013 [Gar+13].

While there are some papers proposing implementations of indistinguishability obfuscators, they are as of now unusable in a real-life application [Xu+17]. A strong gap can thus be found between theoretical software obfuscation research and practical obfuscation implementations. Furthermore, it should be noted that

these two subjects involve two different research communities: code-oriented obfuscation papers are usually published by the software security research community, while model-oriented obfuscation is more a focus in cryptography and theoretical computation research communities. To conclude, there are nowadays no obfuscation approaches that are both provably secure and usable. In the rest of this work, we will focus only on practical, code-oriented obfuscation.

1.3.2 Threat Models and Use cases

There are two main users of obfuscation: software developers trying to protect the intellectual property of their programs from theft and misuse, and malicious actors trying to protect their malware from detection.

Defensive Obfuscation

Obfuscation is mainly used by software developers and companies to prevent so-called Man-At-The-End (MATE) attacks [Fal+11][Akh+15]: with this type of scenario, the attacker has physical access to the software and can inspect it, modify it and run it at will. This includes any software running on devices such as personal computers, smartphones etc., but also remote execution in potentially hostile environments such as cloud computing [HE12]. Several threats are involved with MATE attacks:

- Reverse-engineering:
  - Extracting intellectual property such as proprietary algorithms and implementations.
  - Finding critical information such as cryptographic keys, static integers [SL12].
  - Discovering vulnerabilities in the program that can be exploited in future attacks, for example code injection where malicious code is injected into the program [BS05].

- Cloning:
  - Making and distributing illegal copies.
  - Circumventing Digital Rights Management (DRM) measures.
  - Finding and breaking license checking mechanisms [KM14].
- Tampering [Got+01]: modifying the software in ways unintended by the developer.

Black-hat Obfuscation

Obfuscation is also used by malware authors as a protective measure [YY10] against defensive threats such as:

- Detection by automated defense mechanisms such as anti-virus scanners. Obfuscation can be used to hide key-words, code snippets or patterns known in malware databases.
- Reverse-engineering and analysis by security researchers: understanding how a malware works is often an essential step for preventing its execution and spread. It can also be used to identify the malware authors.
- Take-down of the malware: by searching for specific strings such as URLs, researchers can identify what server the malware communicates with and how it spreads.

1.3.3 Taxonomy and Common Transformations

Obfuscating transformations have long been classified by their target, in other words by what they are modifying. In [CTL97], the authors proposed a now widely used taxonomy where obfuscating transformations target either **layout**, **data** or **control flow** of the program. A detailed classification of common obfuscating transformations following this taxonomy can be found in Figure 1.10.

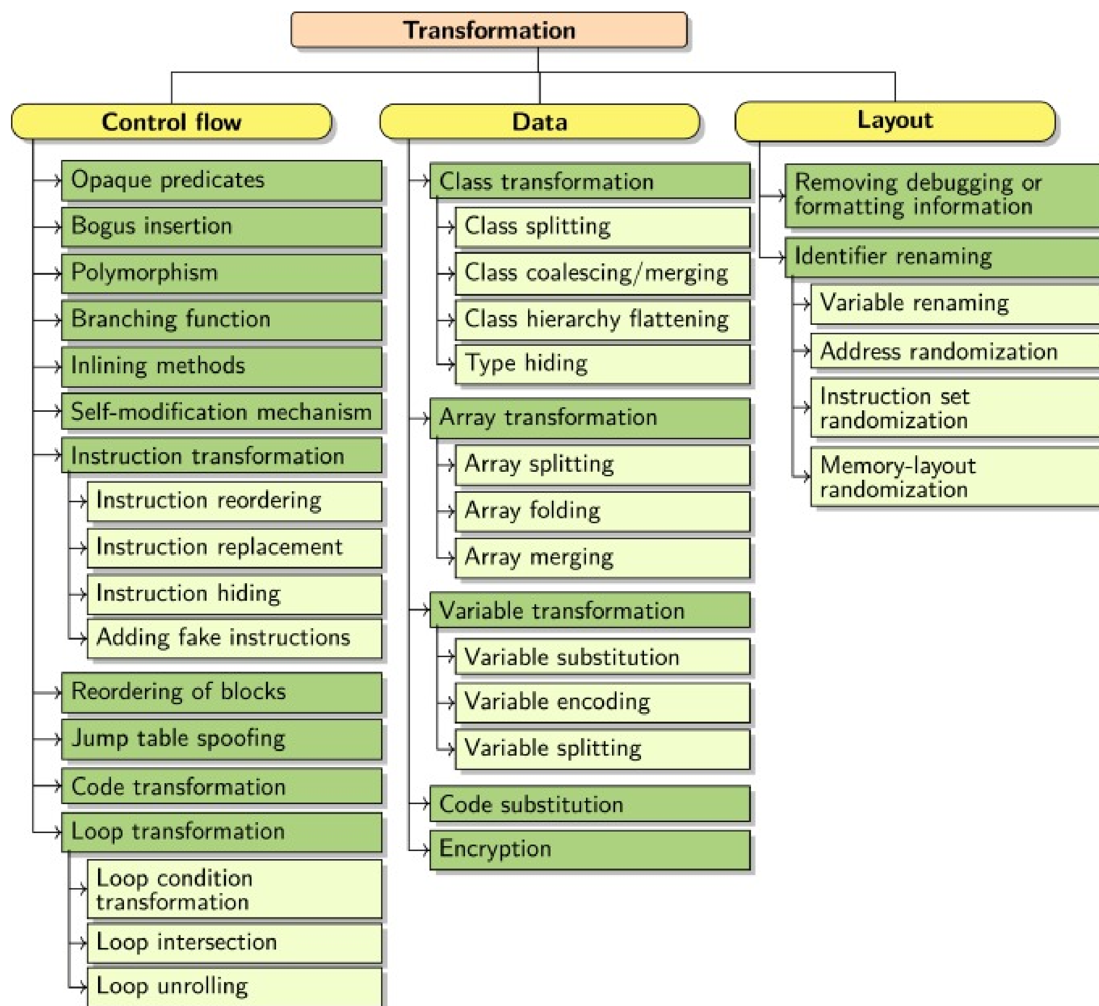


Figure 1.10 – Classification of obfuscating transformation mechanisms. (from [Hos+18])

Layout Obfuscation

Layout obfuscation mainly aims at making the code harder to read for humans. It targets a program's structure and decreases readability by for example removing any formatting (line breaks, indentation, etc) from the source code, changing identifier names or deleting comments. Layout obfuscation is considered irreversible, since the original formatting cannot be recovered once removed. These techniques have no significant influence on the size of the obfuscated program or on the speed of execution. However, they are usually not resistant against automated tools such

as disassemblers.

Control Flow Obfuscation

Control flow obfuscation aims at altering and/or hiding the control flow of a program. This can make the code harder to read and understand for a human, but can also prevent disassembly and reverse engineering with automated tools. Among the most common control flow transformations, following can be cited:

- Dead or redundant code insertion: bogus operands, statements, blocks, control flow transfers, classes, etc.
- Breaking computations up or merging them together
- Loop modifications: unrolling, extending termination condition
- Reordering instructions or computations
- Inlining functions
- Opaque predicates: predicates whose value is known at obfuscation time, but hard to evaluate for an attacker.

Data Obfuscation

Data obfuscation aims at hiding data and data structures. It usually [CTL97] affects the ordering, aggregation, storage or encoding of data. A few common data obfuscation techniques include:

- Splitting variables into two or several variables
- Aggregating several variables into one variable or array
- Restructuring arrays: merging, splitting, folding (increasing the number of dimensions of the array), flattening
- Reordering declarations, variables or arrays
- Modifying the encoding of variables
- Encrypting parts of the data

1.3.4 Commercial and Open-source Tools

Several academic and/or open-source obfuscation tools at varying levels of maturity can be found online. Among the better known tools, the following should be cited:

- **Obfuscator-LLVM** [Jun+15]: Swiss project based on the LLVM compilation suite. It provides a fork of LLVM that adds obfuscation to the code. The obfuscation passes are added at IR (Intermediate Representation) level. A commercial version with more advanced obfuscation techniques was sold by Strong.codes (now bought by Snapchat). The available obfuscation passes include control flow flattening, bogus control flow (adding a new basic block containing an opaque predicate that makes a conditional jump to the original basic block) and instructions substitutions (replacing binary operators with more complicated instructions).
- **Tigress** [Col]: source-to-source obfuscator for C mainly developed by Christian Collberg, written in OCaml. This tool is not open source, but can be freely downloaded for tests. Tigress has several classic obfuscating transformations such as control flow flattening, function splitting and merging, control flow splitting and encoding of literals and data. It also offers the possibility to virtualize a function by turning it into an interpreter, or to transform a function so that it is dynamically compiled to machine code (just-in-time compilation).

Commercial obfuscation tools are also sold by several companies. In most cases, the exact obfuscation techniques used are not publicly revealed and information about the inner workings of these tools is kept secret. Following tools are well known:

- **Stunnix** [Stu]: obfuscator for C/C++, Perl, VBScript and JavaScript. This tool mainly performs layout obfuscation, e.g. renaming symbols, removing comments, removing spaces and tabs, etc.
- **Cloakware** [Ird]: obfuscator sold by Irdeto, a company specializing in digital security.

- **Quarks AppShield** [Qua]: formerly known as Epona, solution sold by Quarkslab. Among several other protection mechanisms, it offers more than 30 different obfuscation techniques for C/C++.

1.3.5 Evaluation Metrics

To evaluate the quality of obfuscation techniques, 4 metrics have been proposed in [CTL98] and are now used in most works on obfuscation:

- **Potency**: how much more complex to understand the obfuscated program is compared with the unobfuscated program. Several software complexity measures such as McCabe complexity, cyclomatic complexity, data structure complexity can been used to estimate potency of an obfuscation technique. Empirical studies where for example groups of students [Cec+14] are asked to attack obfuscated code have also been conducted to evaluate potency.

- **Resilience**: difficulty faced by an automatic de-obfuscator or disassembler in breaking the obfuscation. This metric is complementary to potency: obfuscated code can have high complexity but still be very easy to de-obfuscate with an automated tool.

- **Stealth**: how well the obfuscated code blends in with the rest of the program. If code added during obfuscation *looks* very different from the original code, it can be easy to detect and remove for a human attacker.

- **Cost**: computational overhead added to the obfuscated program. This overhead mainly comprises increases in run time, memory usage, CPU usage and program size [Hos+18].

1.4 Conclusion

In this chapter, we presented background and related work relevant for this thesis. In particular, we introduced the hardware design cycle, with an overview of the different steps and actors involved, as well as the possible security threats. We also focused on software obfuscation for protecting programs against reverse-engineering and theft, giving a presentation of the principles and techniques involved.

Finally, we presented various approaches for protecting hardware against different threats. Most threat models focus on risks during or after manufacturing, or at lower abstraction level. At the behavioral level, many methods rely on the HLS tool to add a protection mechanism such as watermark or obfuscation. None of the techniques focus on a threat model where the HLS tool itself or its runtime environment is a security risk. In this work on the other hand, we focus specifically on the scenario of an untrusted HLS, either because the tool is hosted in the cloud (cf. Chapter 2), or because the tool itself is potentially compromised (cf. Chapter 4). Any protection technique relying on HLS for insertion thus cannot be used here. Furthermore, the techniques presented in this chapter that can be applied at behavioral level all rely on idea that the security is transmitted throughout all design steps. This process makes sense since the main goal is to provide protection at foundry level. However, in our case, since we only focus on protection *during* HLS, we propose a method called *transient obfuscation* that does not persist at lower design levels and thus does not have an impact on the final design in terms of functionality or performance. This method can be used both as protection against BIP theft and reverse-engineering (Chapter 2), and as defense against hardware trojan insertion during HLS (Chapter 4). The side-effects of this method can additionally be used to watermark BIPs in a novel manner (Chapter 3).

Chapter 2

Transient Obfuscation against IP Theft during Cloud-based HLS

Modern hardware designs have reached a fantastic degree of complexity. To sustain this industrial challenge, design flows are increasingly distributed, with companies relying on third parties for IP development, manufacturing and testing. This leads to substantial cost reductions, but also a growing amount of security issues. In particular, threats at early design stages are coming more and more into focus: the widespread use of third-party Behavioral IPs (BIPs), as well as the trend towards external Computer-Aided Design (CAD) tools, have created new attack surfaces.

In this chapter, we aim at ensuring the security of BIPs, and in particular at protecting them against theft *during* HLS. We focus on the case of a cloud-based HLS service. With the recent surge in cloud computing capabilities, and the growing trend of complex computations being outsourced to dedicated Software-as-a-Service (SaaS) platforms, a HLS as a Service scenario becomes more likely. However, broad adoption of such a service is slowed down by legitimate security concerns. While encryption is widely used to ensure data security in the cloud, a complex operation such as HLS cannot be performed on encrypted code without decrypting the code at some point during the operation.

We propose *transient obfuscation*, a new method combining software obfuscation techniques with key-based hardware protection techniques to provide a temporary protection during untrusted, cloud-based HLS. This method relies on

a two-step process: applying obfuscation to behavioral level source code, *before* HLS, and de-obfuscating the resulting design *after* HLS, at a trusted point in the design flow.

In this chapter, we introduce the concept of *transient* obfuscation for cloud-based HLS. We show how traditional software obfuscation techniques can be modified to become key-based and transient [Bad+21a]. Finally, we present practical implementations of these techniques as well as experimental results: some techniques, such as bogus code insertion, presented at DATE2019 [Bad+19], are particularly well suited, while others are only partially transient.

2.1 Threat Model

2.1.1 HLS-as-a-Service

With the recent surge in cloud computing capabilities, and the growing trend of complex computations being outsourced to dedicated Software-as-a-Service (SaaS) platforms, EDA industry leaders are increasingly moving to the cloud. For example, Synopsys now offers a solution[1] to use their EDA tools for IC design and verification online. Intel, with its DevCloud [Dev], also offers a platform to develop, compile, test and run programs and IPs fully in the cloud. Moreover, their solution promises the possibility to implement and test solutions on FPGA boards. Similarly, the French cloud computing and hardware acceleration companies OVH and Accelize have recently proposed[2] what they call *Acceleration-as-a-Service*: by leveraging cloud computing, FPGA-based acceleration becomes accessible to a larger audience without hardware specific expert knowledge. Hastlayer [Has], a solution developed by Lombiq Technologies company, offers .NET software developers the means to accelerate their programs by automatically generating FPGA-based hardware implementations of parts of their application.

[1] Synopsys Announces Collaboration with Samsung Foundry to Offer Secure and Scalable Environment on the Cloud for IC Design and Verification. *https://www.design-reuse.com/news/45852/synopsys-samsung-foundry-cloud-ic-design-verification.html*

[2] OVH launches Acceleration-as-a-Service Leveraging the New Intel Programmable Acceleration Card and App Store from FPGA Acceleration Partner Accelize. *https://www.design-reuse.com/news/42929/ovh-acceleration-as-a-service-intel-accelize.html*

Given these successful recent examples, an HLS-as-a-Service scenario becomes more likely [DRA15]. A cloud-based HLS service offers several advantages:

- Contrary to the existing, often expensive licensing schemes for most commercial HLS tool, cloud-based HLS could offer a better economic model with flexible pricing and a pay-per-use system. This would also allow better accessibility for small design companies or individual designers.
- For design companies, HLS-as-a-Service requires no complex installation or time-consuming maintenance. It allows easier integration in existing complex and distributed industrial design flows.
- For the HLS tool providers, a cloud-based service could act as a showcase for their product, allowing users to easily perform tests and comparisons between different tools.
- The large computational power available in the cloud could be used to propose advanced HLS tools using resource-expensive approaches such as Machine Learning or Design Space Exploration.

2.1.2 Threat Model

Broad adoption of cloud-based HLS is slowed down by legitimate security concerns. While encryption is widely used to ensure data security in the cloud, a complex operation such as HLS cannot be performed on encrypted code without decrypting the code at some point during the operation.

How to protect behavioral IPs against theft has been a growing concern. In this work, we focus on the scenario, illustrated on Figure 2.1, of a design house using an external cloud-based HLS tool for their BIPs. We assume that the attacker is either an insider of the HLS service, or an outsider who has been able to gain access to the service, and is able to steal the BIP code. We envision three types of risks in this scenario:

1. **Espionage**: the attacker, working for example for a competing company or an adverse government, gains valuable information about what type of algorithms or applications the design house is working on.

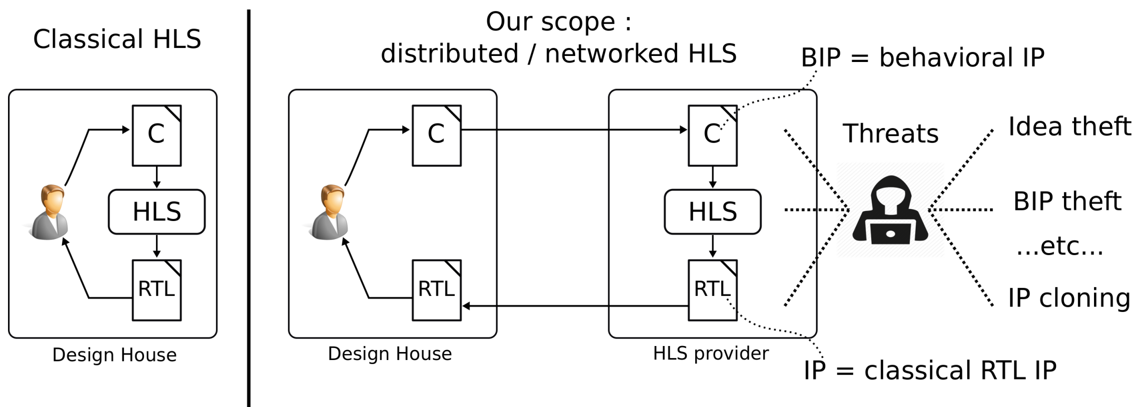


Figure 2.1 – Vulnerable HLS in the cloud design flow.

2. **BIP theft**: the attacker is able to understand the stolen BIP and can modify or directly counterfeit it.

3. **Black-box usage**: even without full insight into how the stolen BIP works, the attacker can still reuse it as a black-box.

Because we cannot prevent theft on the HLS provider's side, our approach aims at minimizing the previously enumerated risks. Using software obfuscation can reduce the ability of an attacker to understand and modify a stolen design. However, on its own, software obfuscation does not modify functionality and thus does not prevent black-box usage. By adding hardware IP protection principles such as logic locking, we can provide a transient, key-based obfuscation scheme that protects BIPs against all three risks *during* HLS.

2.2 Related Work and Background

2.2.1 Cloud Computing Challenge: Ensuring Data Security

How to secure data in the cloud has been an ongoing concern for both industry players and researchers. Some topics such as ensuring strict separation of different user's data, and avoiding accidental loss of data already have many existing mature solutions [Rya13]. One challenge in particular remains: any data in the cloud can

be accessed by the cloud provider, or potentially an outside attacker in case of a security breach. In cases of a simple storage service, data can be encrypted to limit security risks. However, in the case of Software-as-a-Service (SaaS), where the cloud provider is expected to perform calculations on the data, conventional encryption methods cannot be used.

Among the technical solutions that have been proposed on how to guarantee data confidentiality while performing calculations in an untrusted cloud environment, homomorphic encryption offers the highest level of security. Introduced as a concept in 1978 [R+78], homomorphic encryption was first applied to cloud computing security in 2012 [TEE12]. The idea is that operations can be performed on encrypted data, and the resulting data, once decrypted by the client, is exactly the same as if the operations had been performed on the raw, unencrypted data. While ongoing research on homomorphic encryption is producing increasingly efficient and scalable encryption schemes, the current state of the art systems still lead to gigantic overhead both for data size and computation speed. However, the general principle of homomorphic encryption, i.e. performing computations, in our case HLS, on unreadable data, has served as a basis for our idea of *transient obfuscation*, as shown in Section 2.3.1.

2.2.2 Algorithm Level Obfuscation

Obfuscation has been widely studied to protect IPs at RTL or lower design levels as shown in Section 1.2.3. So far, and to the best of our knowledge, only a few works focus on using obfuscation at algorithmic level and/or for BIP protection.

In [VS17a], a study of the impact of commercial and free software obfuscators on HLS quality of results (QoR) is presented. The authors note that the different (often in-house) front-ends used in HLS tools often fail to perfectly deal with obfuscated source code. This results in missing optimizations and thus varying, sometimes strong, impact on the quality of results, in particular a significant increase in area, delay and latency of the resulting circuits. The authors then propose two methods for minimizing QoR degradation while also maximizing obfuscation level: a Genetic Algorithm based approach, and a fast iterative-greedy method. With both methods, the goal is to determine which lines of the source

code should be obfuscated and which lines should stay untouched. The overall focus of the paper is on optimizing obfuscation with an existing, basic software obfuscation tool, and not on proposing any new or modified obfuscation techniques adapted to a hardware context. The obfuscator used in the experiments, Stunnix C/C++, only applies layout obfuscation techniques such as modifying identifiers or replacing numbers with mathematical expressions. While these techniques have a strong impact on code readability, they do not provide reliable or quantifiable safety, especially against automated reverse-engineering tools. Moreover, these kind of techniques do not affect functionality of the obfuscated code, and thus do not provide any protection against black-box usage of the resulting IPs.

Other works such as [Pil+18b] propose adding obfuscation during HLS. In this work, an academic HLS tool, Bambu, is extended with obfuscating transformations that aim at protecting ICs later in the design cycle. In particular, the threat model focuses on untrusted foundries as the main threat for IP theft and reverse-engineering. HLS itself, on the other hand, is considered fully trustworthy, contrary to our threat model. The advantage of this method, in contrast to obfuscation applied on the source code *before* HLS, is that obfuscation is fully integrated in the HLS process and applied at the IR level. This means that there are no optimization issues due to the parser used, as underlined in [VS17a], and the overall process is more efficient and results in less overhead. However, it does require full trust in the security of the HLS tool and thus cannot be applied in our context of cloud-based, untrusted HLS. In a similar approach, the authors in [IK18] extend an in-house HLS tool with an obfuscation pass performed during scheduling and datapath generation phases. Here as well, the result is an obfuscated RTL design which is protected during fabrication. HLS is considered a fully trusted design step.

2.3 Proposed Approach: Transient Obfuscation

2.3.1 Transient Obfuscation

As presented in our threat model in Section 2.1.2, in this work we focus only on protecting a BIP *during* HLS. We assume that both *before* HLS, at source code level, and *after* HLS, at RTL, the BIP is secure. This is quite different from

traditional threat scenarios, where the attacker is usually located at foundry level.

Since the BIP we are trying to protect is initially in the form of high-level source code (written in C/C++), but the whole design flow is still situated in a hardware context, the protection method we propose is situated at the intersection of traditional software and hardware protection methods.

On one hand, as suggested in [VS17a], we could simply apply software code obfuscation to our BIP. However, this leads to several issues. First, software obfuscation, by definition, is used to prevent reverse-engineering, while *maintaining full functionality* of the code. This means that it cannot prevent illegal reuse of the BIP in a black-box manner. Moreover, traditional software obfuscation does not usually take into account any overhead except runtime. In a hardware context where performance is critical, using software obfuscation techniques can result in designs with unacceptable area and timing overhead. Finally, applying strong obfuscation to the source code can perturb the HLS tool and the frontend parser.

On the other hand, most existing hardware obfuscation techniques are meant for lower level IPs. Logic locking, which is successfully used to prevent black-box usage, has for example been applied to RTL IPs, but never, to the best of our knowledge, at behavioral level before HLS. Due to the better readability of high level code compared to HDL or even a netlist, protection methods that are normally added later in the design cycle could be trivial to remove at the behavioral level. Moreover, existing locking techniques work under the constraint that unlocking can only be done after fabrication. In our case, we have the possibility of applying a protection that is only temporary, during HLS, and can be removed immediately afterwards, at RTL.

We thus introduce the concept of *transient obfuscation*, which combines software obfuscation techniques with key-based hardware locking concepts to form a temporary protection during HLS. This method takes into account hardware performance issues, provides protection both against reverse-engineering of the source code and black-box usage of the resulting IP, and is unlocked after HLS to ensure a low design overhead and correct functionality.

This concept is inspired by the idea of homomorphic encryption introduced in Section 2.2.1:

1. The source code is *obfuscated*, instead of *encrypted*, with a series of secret keys

2. HLS is run on the obfuscated code, similarly to how calculations are run on homomorphically encrypted data

3. The resulting RTL is *de-obfuscated* with the correct secret keys. The recovered design has the exact same functionality as the original design would have after HLS without obfuscation.

To be fully *transient*, an obfuscation technique has to respect two criteria:

- After de-obfuscation, original functionality should be fully restored. This means that transient obfuscation should have no impact on functionality of the design.

- After de-obfuscation and synthesis, the design should not have higher area or timing than the original, unobfuscated design. The cost of transient obfuscation is thus zero.

2.3.2 Complete Flow

The goal of this work is to protect a design house against BIP theft and black-box usage during HLS. We propose a transient protection method by adding two steps to the conventional design flow, before and after HLS:

1. Obfuscation: the original C code is obfuscated, making it harder to understand for an attacker. We use key-based obfuscation which modifies the design's functionality by tying the correct behaviour to a set of obfuscation keys. This prevents black-box usage of the design.

2. De-obfuscation: after HLS, the resulting RTL code still relies on keys to function correctly and contains all the obfuscated code. Directly synthesizing the IP at this point would result in a malfunctioning, oversized design. By de-obfuscating with the correct keys, we obtain a *cleaned-up* design which no longer relies on keys to function correctly and where any unnecessary logic

has been removed. Original functionality is thus recovered and the design overhead is strongly reduced.

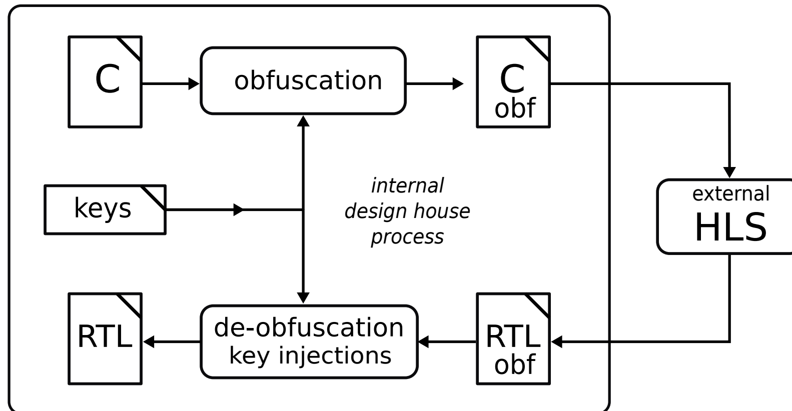


Figure 2.2 – HLS in the cloud design flow secured by KaOTHIC, where the external HLS is considered untrusted.

By combining these two steps, as shown on Figure 2.2, our protection method becomes transient. When using carefully selected and adapted obfuscation techniques, this method has no significant lasting impact on the design. Both steps have been implemented in a fully automated way in an open-source tool, KaOTHIC (Key-based Obfuscating Tool for HLS in the Cloud), and are detailed in the next subsections.

2.4 Proposed Obfuscation Techniques

2.4.1 Bogus Code Insertion

This set of techniques is based on control flow splitting, as presented in [CTL97]. It relies on a simple principle: bogus code is added to the design. A set of predicates, in the form of If/Else statements, is then added to split the control flow between the correct, original code and the bogus code. The goal is to make it near impossible for the attacker to guess which part of the code is fake. This means that the bogus code should be stealthy by strongly resembling the original code, but still provide enough differences to have a real impact on the circuit's behavior.

Key-based Predicates

In conventional control flow splitting implementations, *opaque predicates* [CTL98] are used to split the control flow: for example based on complex mathematical expressions, the value of opaque predicates is known at obfuscation time, but hard to evaluate for an attacker. In this chapter, we propose instead to use *key-based* predicates, where an input value is tested against a constant. Similar to logic locking techniques used for hardware obfuscation, only the correct combination of inputs will allow the code to execute correctly. A simplified example in form of a Data-Flow Graph (DFG) is given in Figure 2.3. The input variable "key" is tested against the value "42". If the test is True, the original code branch is executed. Else, a bogus code branch is executed. It should be noted that while on this figure, the original code is assigned to the True branch and the bogus code is assigned to the False branch, this is not always the case. We chose to randomly assign the original and one or several bogus code branches to the keys test branches. This means that an attacker has no means of guessing the correct key value simply by reading the test values in the code.

Bogus Expressions

This obfuscation technique, illustrated on Figure 2.4, can be directly performed at AST level. Candidate AST nodes for obfuscation are binary expressions of the form:

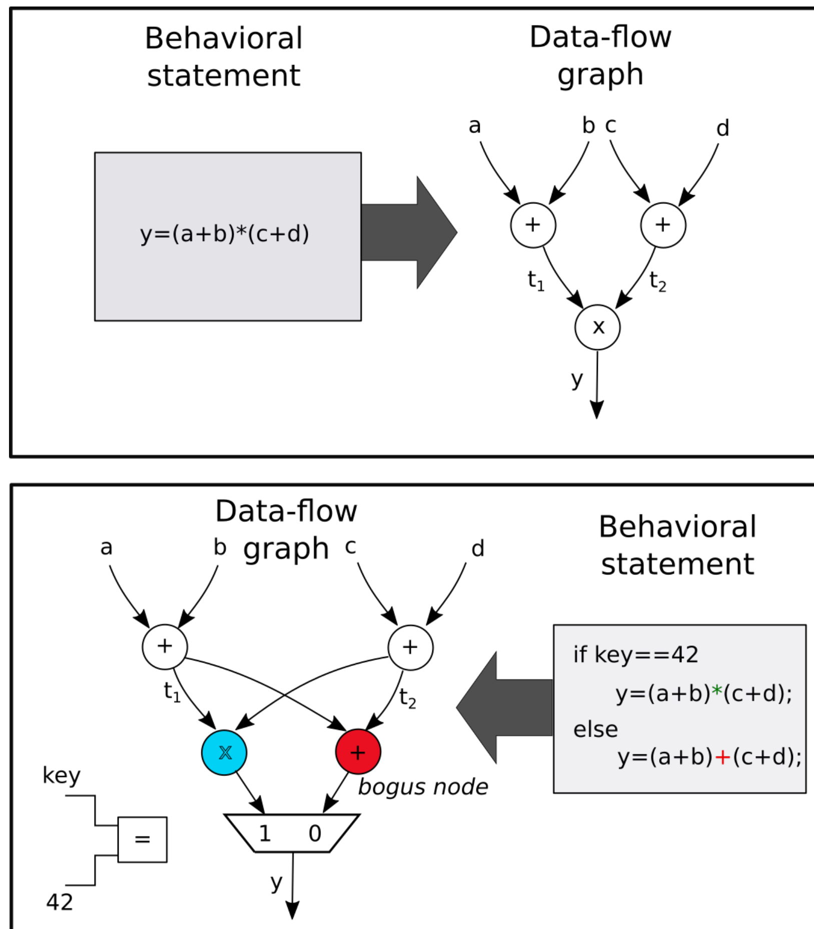


Figure 2.3 – Behavioral and dataflow representation of a code with bogus code insertion.

`<binaryExpression>::=<expression><op><expression>` , where *op* is a bitwise or arithmetic operator. For each selected binary expression, several similar bogus expressions are created with slight variations, see Figure 2.5(b). The order of the operands in the original expression is randomly shuffled. Furthermore, the operators are randomly replaced by other, similar operators. For example:

$$a = (b + c) * 2 \quad \Longrightarrow \quad a = (2 - c) + b$$

The choice of the operators added in the bogus expressions has some importance in improving security but also in reducing overhead. The chosen operators have to be computationally similar to increase stealth (a bitwise operator should not be

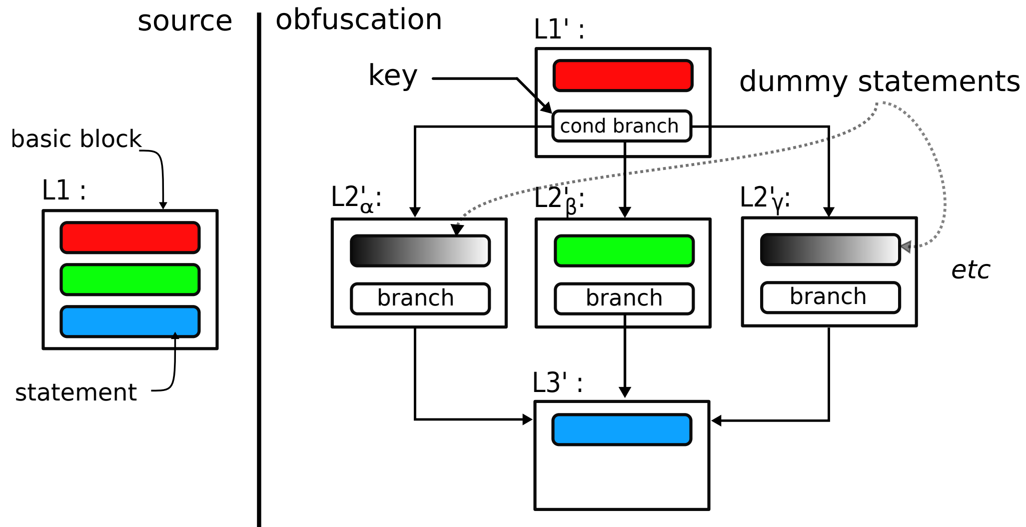


Figure 2.4 – Basic block example before and after obfuscation: insertion of two bogus expressions.

added for an operation between two integers for example) and of equal or lower complexity. Special care has to be given to the fact that the added bogus code, which is generated in a completely automated way by our obfuscation tool, should still be compilable. Applying careful, small modifications to existing expressions significantly eases this process and decreases the risk of adding code leading to compilation errors, while still maintaining a high level of stealth.

Bogus Basic Blocks

In order to diversify and improve upon the previously presented technique, we also propose to insert complete bogus basic blocks, instead of just individual bogus expressions. For this technique, the first step builds the CFG of the source program. A list of candidate basic blocks for obfuscation is then established. For each selected basic block, one or several bogus basic blocks are created, as illustrated on Figure 2.6. The creation of these basic blocks relies on the same principles as for bogus expressions, presented in Section 2.4.1:

- In individual expressions, order of the operands and operations are shuffled.

- Operators are randomly replaced in each expression.

```
int gcd(int m, int n){
    while(m != n){
        if (m > n){
            m = m - n;
        } else {
            n = n - m;
        }
    }
    return m;
}
```

(a) No obfuscation

```
int gcd_obf(int m, int n, int k1, int k2){
    while (m != n){
        if (m > n){
            if (k1 == 42){ // true
                m = m - n;
            } else if (k1 == 25){ // false
                m = n - m;
            } else { // false
                m = m + n;
            }
        } else {
            if (k2 == 54){ // false
                n = n + m;
            } else if (k2 == 12){ // false
                n = m - n;
            } else { // true
                n = n - m;
            }
        }
    }
    return m;
}
```

(b) Bogus Expression Insertion

```
int gcd_obf(int m, int n){
    int control = 1;
    while (control != 0){
        switch(control){
            case 1: {
                if (m != n){
                    control = 2;
                } else {
                    control = 0;
                }
                break;
            }
            case 2: {
                if (m > n){
                    control = 3;
                } else {
                    control = 4;
                }
                break;
            }
            case 3: {
                m = m - n;
                control = 1;
                break;
            }
            case 4: {
                n = n - m;
                control = 1;
                break;
            }
        }
    }
    return m;
}
```

(c) Control Flow Flattening

```
int gcd_obf(int m, int n, int k1, int k2, int k3,
            int k4, int k5, int k6, int k7){
    int control = k1;
    while (control != 0){
        switch(control){
            case 1: {
                if (m != n){
                    control = k2;
                } else {
                    control = k3;
                }
                break;
            }
            case 2: {
                if (m > n){
                    control = k4;
                } else {
                    control = k5;
                }
                break;
            }
            case 3: {
                m = m - n;
                control = k6;
                break;
            }
            case 4: {
                n = n - m;
                control = k7;
                break;
            }
        }
    }
    return m;
}
```

(d) Key-based Control Flow Flattening

Figure 2.5 – Different obfuscation techniques applied by KaOTHIC to a function calculating the gcd of two numbers.

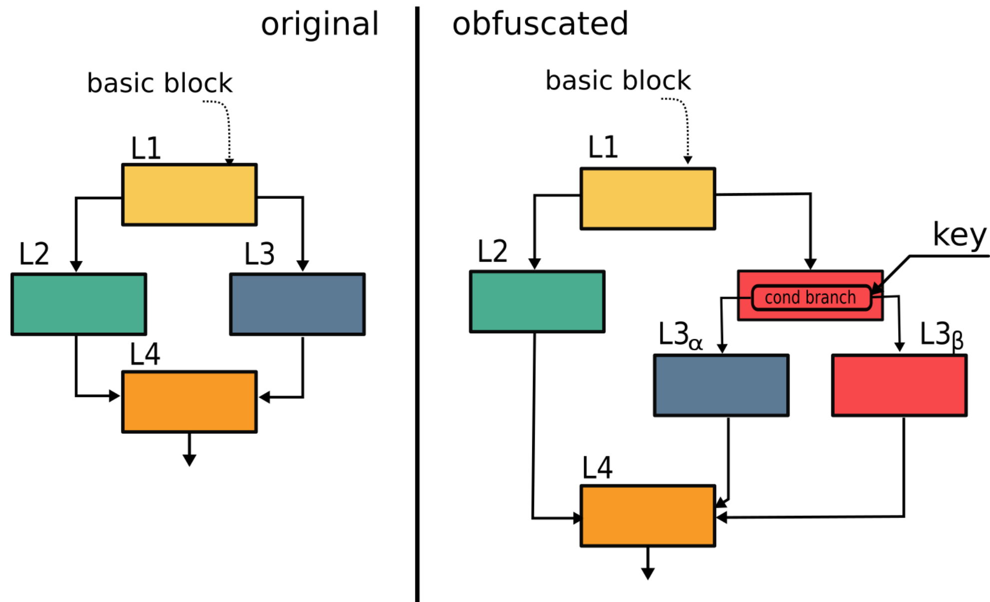


Figure 2.6 – Example CFG before and after obfuscation by adding bogus basic blocks.

- The order of all the expressions in the block is shuffled.

As before, key-based predicates are then added to split the control flow. Finally, an AST is rebuilt from the modified CFG and C-level source code is returned. The success of this technique relies on correctly identifying in an automated way which basic blocks *can* be obfuscated without preventing the correct rebuilding of the AST, and carefully selecting which basic blocks *should* be obfuscated without hindering HLS optimizations. For example, the increment instruction of a For-loop is usually represented as a basic block when a CFG is generated. However, duplicating and modifying this block during obfuscation would prevent correct rebuilding of the loop when printing the obfuscated C code.

2.4.2 Control Flow Flattening

We also studied another technique for hiding the control flow. Control flow flattening is a technique first introduced in [Wan+00] for software obfuscation. Control flow structures that are easily identifiable such as loops or if/else statements are replaced by one global switch statement. The goal is to hinder static analysis of

the program by hiding the targets of the code branches. This technique is usually used in a pure software context and does not take any hardware and EDA specific constraints into account.

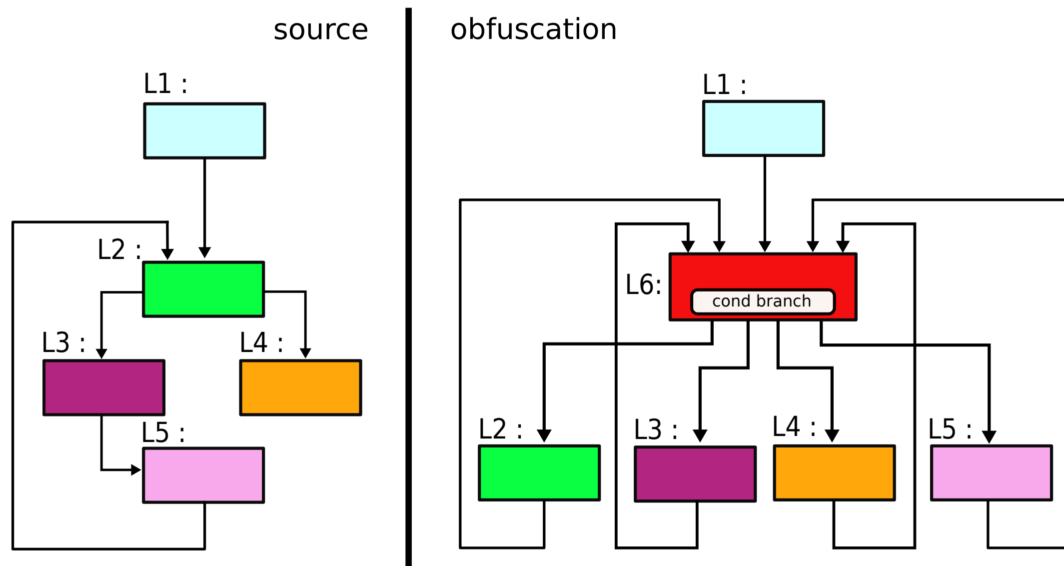


Figure 2.7 – Example CFG before and after obfuscation by control flow flattening.

Key-based Control Flow Flattening

An algorithm to flatten the control flow of a C++ program was presented in [LK09]. The first step generates the control flow graph of the original program by separating it into basic blocks. Next, each basic block is rewritten as a case in one global switch statement. This switch statement is embedded in a loop. A routing variable is used to direct the control flow of the program. At the end of each case, a new value is attributed to the routing variable to indicate the next state. When the value 0 is given, the exit condition for the surrounding loop is reached. An illustration of control flow flattening can be found on Figure 2.7.

One of the vulnerabilities and easiest attack vectors for de-obfuscation of classic control flow flattening is the fact that the routing variable is not hidden, as shown in the example Figure 2.5(c). An attacker can thus follow the code execution and reconstruct the original control flow. Several techniques to hide the values assigned to the routing variable have been proposed, for example by using a computationally

hard problem in the dispatcher [Cho+01], or by using a one-way function to update the routing variable [CP10].

In accordance with our threat model, we only want to secure one design step, HLS, which is why transient obfuscation can be used. To increase security of control flow flattening without further increasing overhead, we propose a key-based control flow flattening method.

We propose to replace the literal values assigned to the routing variable by variables that are inputs to the function, as can be seen on Figure 2.5(d). This means that if the wrong inputs, or keys, are given, the basic blocks will be executed in the wrong order. Only the exact correct sequence of inputs can ensure correct code execution. A code example with classic and key-based control flow flattening can be found in Figure 2.5(d).

Partial Control Flow Flattening

When using control flow flattening, every loop gets broken up and transformed into individual basic blocks. This means that HLS tools are no longer able to perform common optimizations on loops (unrolling, pipelining...). Without these optimizations, the final overhead can be too high for some users.

An additional feature in KaOTHIC was added to enable users to fine-tune which parts of the code should be flattened. By placing pragmas in the code, a user can for example choose to flatten everything except for a particular loop. An other option is to perform what we called "coarse flattening": instead of creating a case for each basic block, several basic blocks can be grouped together in one case, for example all the blocks inside a loop. This selective control flow flattening can help improving the results in terms of design overhead, but comes with a decrease in security.

2.4.3 Further Hiding of the Control Flow

To further improve the security level of the previous obfuscation techniques, several variations as well as new transformations can be proposed. For example, in Section 2.4.2, we used a Switch statement as *dispatch*, i.e. as method for selecting which basic block to execute next in a flattened CFG. In our environment

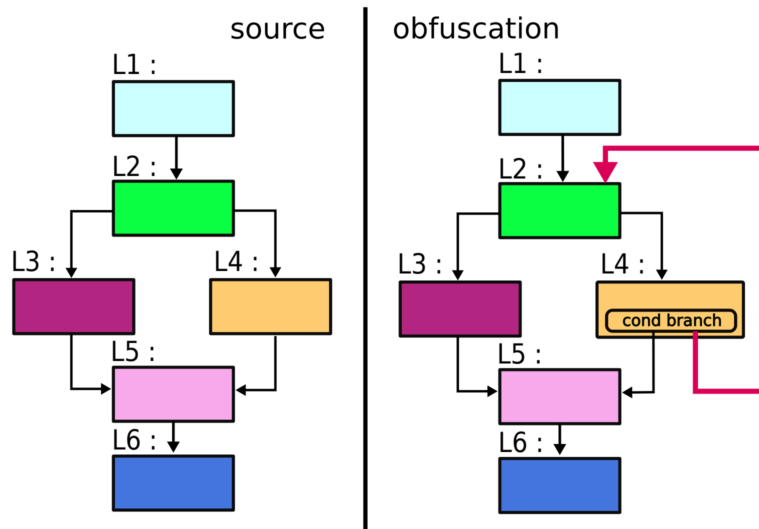


Figure 2.8 – Example CFG before and after obfuscation by adding bogus transitions.

KaOTHIC, we implemented another type of dispatch, *goto dispatch*, where Goto statements are used to direct the control flow. This involves labeling each statement of the original code, then adding Goto statements at the end of each basic block to lead to the next block. Instead of directly pointing to the label of the next block, our Goto statements point to a key input, making it once again hard for an attacker to find the correct control flow. To further increase opaqueness for attackers, we also propose to add bogus transitions (cf. Figure 2.8) between basic blocks, i.e. bogus Goto statements, which can be removed during de-obfuscation with the correct keys. It should furthermore be noted that all of the transformations here can be combined in different orders to increase robustness of the obfuscation.

2.4.4 Data Obfuscation by Literal Replacement

To enhance the previous work in KaOTHIC, mainly focused on hiding the control flow of a program, basic data obfuscation was also added. One of the first steps in data obfuscation is often to hide literals by replacing them with opaque expressions. In our case, we can take advantage of the fact that the obfuscation is temporary. Instead of using opaque expressions, which a motivated attacker might be able to break given enough time, we simply remove literals completely from the code and

replace them with placeholder variables. During de-obfuscation, the key values are added again, ensuring correct functionality of the circuit.

In practice, we implemented a basic transformation that parses the whole code at Abstract Syntax Tree (AST) level and flags all integer literals. Depending on the obfuscation level desired by the user, part or all of the integer literals are replaced by variables. These variables are added as inputs to the main function. The real integer values are stored in a file that stays on the user's machine. During de-obfuscation, the values provided by the user are hard-coded as inputs to the circuit. During logic synthesis, constant propagation will lead to the removal of these now unnecessary inputs and the recovery of the original integer literals. The same process can be applied for other literals such as string literals.

2.5 De-obfuscation: Making the Process Transient

To protect BIPs during cloud-based HLS, we aim at proposing a *transient* protection scheme. To achieve this, we have added a de-obfuscation step after HLS. When the correct obfuscation keys are provided, this step performs two functions: it recovers correct functionality of the IP, and limits design overhead. Most implemented obfuscation techniques rely on adding bogus code, and then using key-based predicates to choose what code to execute, see Section 2.4.1. In that case, de-obfuscation removes all bogus code. For other techniques, such as key-based control flattening, apart from the key logic, there is no bogus code to remove. In that case, the only result of de-obfuscation is recovering the original functionality.

De-obfuscation is performed on the obfuscated RTL code resulting from HLS, usually VHDL or Verilog code. At RT level, the keys that were added as arguments to the main obfuscated function in C are now inputs to the component. For all implemented obfuscating transformations, the resulting RTL code contains the same artifacts: comparisons between key signal and an expected value, and *obfuscation multiplexers* that select what code should be executed based on the results of these comparisons.

In this section, we propose three different approaches for de-obfuscation. The

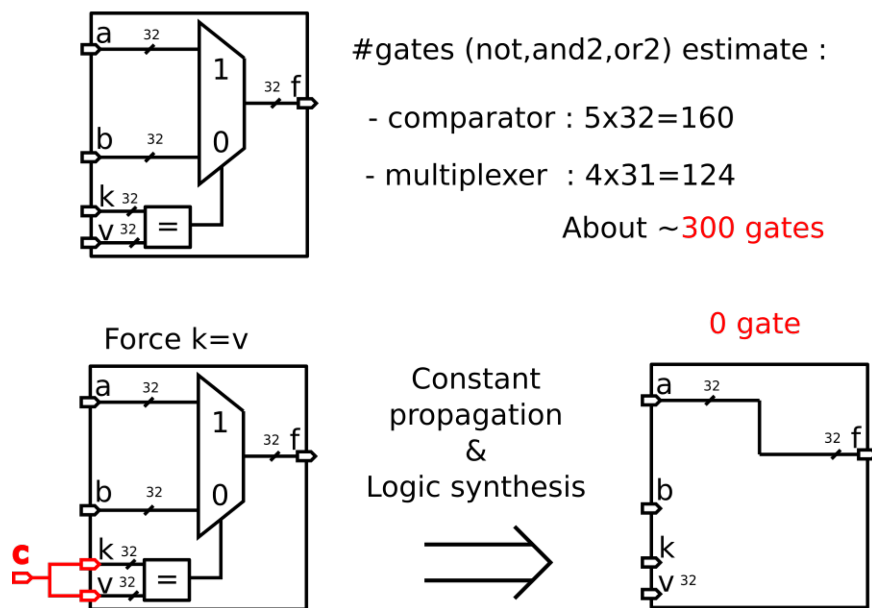


Figure 2.9 – Example of a de-obfuscation by forcing input values.

first two work by injecting the keys in different manners at RTL, and then relying on logic synthesis to remove any dead logic. These two techniques can easily be adapted to different HLS tools and are, thanks to modern powerful logic synthesis techniques, able to remove most, if not all bogus code. The downside is that the de-obfuscated design is only available after synthesis and the designer thus does not have access to the de-obfuscated RTL. With the third approach, de-obfuscation is fully performed on the RTL code, using source-to-source transformations. The designer thus has access to the cleaned-up RTL. However, from a practical standpoint, this technique is harder to implement and requires strong modifications for each different HLS tool used.

2.5.1 Pedagogical Example

On Figure 2.9, we present a simple pedagogical example to illustrate the de-obfuscation process. The figure presents a multiplexer piloted by a comparator. An input key k is compared to a value v. Depending on the result of the comparison, a or b is routed to f. In the case of a simple obfuscation with bogus code insertion (Section 2.4.1), a might for example represent the result of the correct

code branch, to be executed if the correct key value is provided, while b is the result of the bogus code branch.

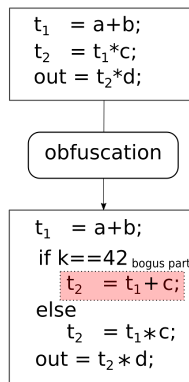
The full obfuscation logic, i.e. key comparison + obfuscation multiplexer, represents approximately 300 logic gates in this example. During de-obfuscation, a constant is used as input for k. If this forced input equals v, then the result of the comparison is always 1. During logic synthesis, due to constant propagation, the whole obfuscation logic gets removed (as shown on the bottom of Figure 2.9), resulting in 0 remaining gates. Furthermore, it should be noted that since b is no longer connected, the added bogus code can also be removed by the synthesis tool.

This simple example shows how, by simply forcing the key input value, the whole obfuscation logic can theoretically be fully and automatically removed during logic synthesis. In this case, our obfuscation process is fully transient and would result in no design overhead. This example works perfectly because the logic is fully combinatorial. In reality, we show in the next section how full constant propagation is prevented by the sequential logic.

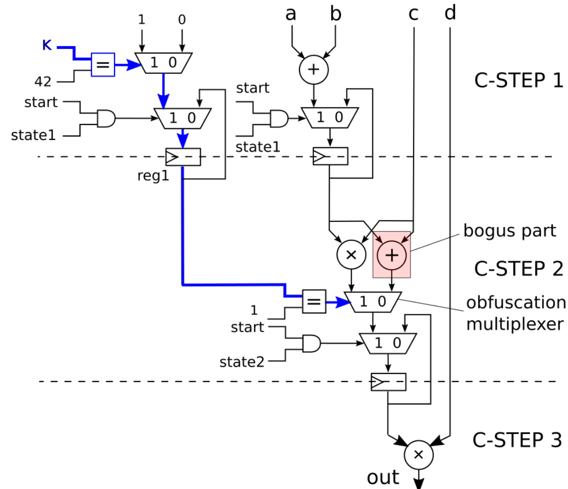
2.5.2 Naive Key Injection

This first approach consists in injecting the correct obfuscation keys at the circuit interfaces in the RTL code. During logic synthesis, the following behaviour is then expected: if the correct keys are injected, through constant propagation the predicates should evaluate correctly, and the bogus code branches should be removed by logic simplification. The circuit would then function correctly and the design overhead compared to the original, unobfuscated circuit should be close to null.

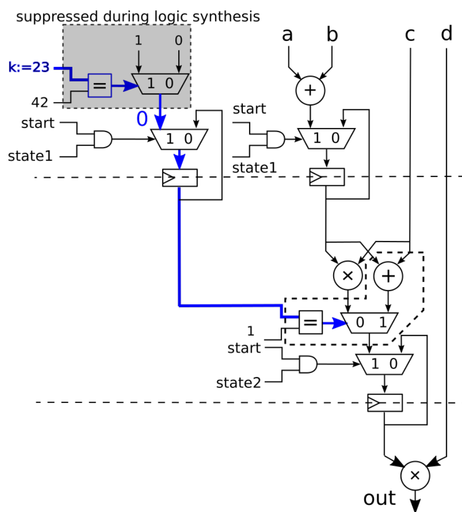
In practice however, this technique does not perform well: the sequential datapath created by the HLS tool scheduling prevents full de-obfuscation. Experiments performed on several benchmarks have shown that the obfuscation multiplexer is often scheduled in a different clock cycle than the corresponding comparison between key input and value, as shown on Figure 2.10(b). This means that a dedicated register is used to store the result of the key comparison ("reg1" on Figure 2.10). Injecting the correct key value as a constant input results in a constant value in the comparison register. However, logic synthesis tools have no way of



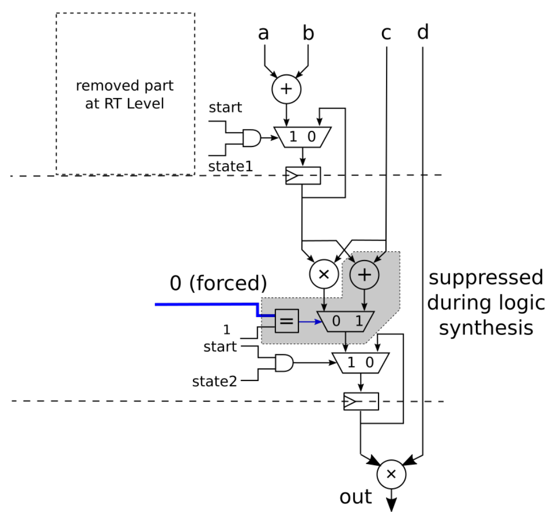
(a) Obfuscation example



(b) Obfuscated RTL datapath generated by Vivado HLS



(c) RTL datapath after naive de-obfuscation



(d) RTL datapath after full de-obfuscation

Figure 2.10 – Example RTL datapath before and after de-obfuscation.

knowing that the register always contains the same value and could thus be removed. As a consequence, the obfuscation multiplexer is not removed either, and most of the bogus code added by obfuscation remains.

While the final circuit is functionally correct, the remaining bogus code causes a significant design overhead.

2.5.3 Targeted RTL Modification

We propose to improve upon this first, *naive* approach, by using a more targeted and in-depth de-obfuscation method, illustrated on Figure 2.10(c). Instead of injecting the keys externally and relying purely on logic collapsing, we locally modify the RTL code itself by performing the following operations for each obfuscation key:

- The given key and its expected value in the code are compared;
- The result of this comparison is stored as `"0"` or `"1"`. This is the value that would during execution be in the aforementioned comparison register;
- The key comparison and the comparison register are completely removed from the RTL code;
- The stored result, `"0"` or `"1"`, is directly injected into the obfuscation multiplexer;
- Logic synthesis is performed as for a normal RTL design.

This more targeted approach enables logic synthesis tools to completely collapse and remove any remaining obfuscation artifacts, resulting in close to no overhead. This guarantees that our obfuscation process is transient.

This second approach was fully automated for VHDL code, and could easily be extended for Verilog as well.

2.5.4 Full RTL De-obfuscation

Our third method is a full de-obfuscation at RTL, before synthesis. We perform a source-to-source modification of the HDL code. This method is more error-prone and difficult to automate, but provides more flexibility for the design house since it now has access to the fully de-obfuscated RTL IP. Moreover, it is less reliant on the code format generated by a specific HLS tool, and should be usable for any HLS tool with no or just small adjustments.

For this approach, we also start by comparing the input keys with the obfuscation key values given in a file and storing the result of this comparison. We then perform a fixed-point analysis on the RTL design. In this phase, we keep track of the signals to which constant values have been assigned, and perform both forward and backward analysis. During forward analysis, all constant values are propagated to existing signals and any now useless elements are removed. For example, a multiplexer with a constant selector can be removed by connecting its output directly to the correct input. During backward analysis, we also remove parts of the circuit that are no longer used. For example, the other input of the previously removed multiplexer becomes unconnected: the logic generating this value can thus be removed, as long as it is not used in other parts of the circuit.

This method was implemented for Verilog code generated by Vivado HLS, using PyVerilog [Tak15] as a parser for HDL manipulation.

2.6 KaOTHIC: Key-based Obfuscating Tool for HLS In the Cloud

All of our proposed obfuscation techniques have been implemented as code transformations in an in-house source-to-source compiler for C, KaOTHIC, written in Python.

2.6.1 Obfuscation Flow

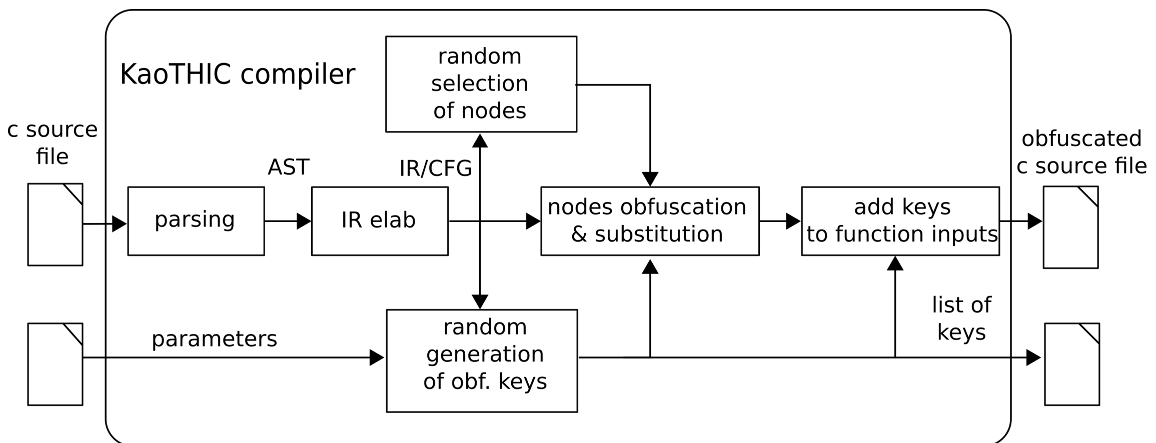


Figure 2.11 – Obfuscation flow in KaOTHIC compiler.

The complete obfuscation flow is illustrated on Figure 2.11. Our tool takes as input the original BIP in form of C source code. It can then be guided with a series of parameters: obfuscation level, applied transformations, order of transformations. The tool outputs the obfuscated BIP again as C code, as well as the list of obfuscation keys. In detail, following steps are performed:

1. The C files are parsed and an Abstract Syntax Tree (AST) is generated;
2. The AST is traversed to build the Control Flow Graph (CFG) of the program;
3. From the CFG, a list of "candidate" basic blocks that can be obfuscated is built;
4. Based on the desired obfuscation level and/or detailed user instructions, basic blocks are randomly selected among the candidates;

5. Each selected candidate is obfuscated and replaced in the original code;
6. For each selected candidate, a key variable is added as input to the obfuscated function and a random value is generated as key;
7. The AST is rebuilt from the modified CFG;
8. From the modified AST, the final, obfuscated C code is recreated.

This generic flow can be adapted to different code transformations. A detailed presentation of all implemented obfuscation techniques can be found in Section 2.4.

2.6.2 Adding Randomness

Since KaOTHIC is open-source, we assume that an attacker also has access to it and thus cannot rely on the secrecy of the process to ensure security. To prevent an attacker from reproducing obfuscation results, randomness is introduced wherever possible in the obfuscation flow:

- Based on user input, a certain number of elements (basic blocks, expressions, etc.) are chosen to be obfuscated among all obfuscation candidates. This choice is usually performed randomly, meaning that running the exact same obfuscation pass on the same code will result in different parts of the code being obfuscated each time. For experimental purposes (reproducibility of results) and to allow better control by the user, it also remains possible to manually choose the obfuscation candidates.
- For each obfuscated item, the correct key value is randomly attributed to either the original or the bogus code. This means that the key-based predicate does not always evaluate to True with the correct, original code branch. The attacker thus has no direct indication in the code for guessing which code branches are real and should be executed, and which are bogus and should be removed.
- The obfuscation keys are numbered in random order and their values are chosen randomly as well.

- The added bogus code is generated according to a set of rules, in order to closely resemble real code and avoid detection, but the exact sequences of instructions, operators and operands is also generated at random.

2.6.3 Obfuscation Parameters

We defined two parameters that can be adjusted by the user to choose the right balance between security level, obfuscated code size and final design overhead.

Obfuscation Level

We call *obfuscation level* the ratio of effectively obfuscated expressions or basic blocks with respect to the total number of such items in the design. For bogus basic block insertion for example, an obfuscation level of 100% means that all possible basic blocks were obfuscated. Increasing the obfuscation level results in an increased number of obfuscation keys to guess for an attacker, and thus in a higher level of security. Ideally, we should always choose an obfuscation level of 100%.

Branching Degree

We call *branching degree* the number of bogus elements added for each obfuscated expression or basic block. A higher branching degree increases the amount of choices an attacker has for each obfuscated item. By combining the maximum obfuscation level, and a high branching degree, security can be significantly increased.

2.7 Experimental Setup and Results

2.7.1 Test Flow

To validate our transient obfuscation approach, we tested the different obfuscation techniques applied individually to 5 benchmarks from MachSuite [Rea+14] and CHStone [Har+08]: advanced encryption standard (`AES`), adaptive differential pulse code modulation encoder (`Adpcm`), optimal sequence alignment algorithm (`Needwun`), mergesort algorithm (`Merge_sort`) and three-dimensional stencil computation (`Stencil3d`).

For each benchmark, we applied HLS followed by logic synthesis once to obtain a baseline report on design size and timing. Next we obfuscated each benchmark with a single obfuscating transformation, followed by HLS with the same parameters as for the original, unobfuscated design. Then de-obfuscation was applied to the resulting obfuscated RTL IP, followed by logic synthesis. The results of these syntheses were used to calculate design overhead of the different approaches.

We also varied the two following parameters for bogus code insertion techniques:

- Obfuscation level: percentage of candidate AST nodes or CFG basic blocks that are obfuscated. 100% means that all possible candidates are obfuscated, leading to the highest level of security.
- Branching degree: amount of bogus expressions or blocks added for each obfuscated item. Increasing it has a strong impact on number of guesses necessary to find the correct key combination.

We used Xilinx Vivado HLS as a HLS tool. Logic synthesis was performed with Synopsys Design Compiler, targeting the Synopsys SAED 90nm educational library for ASIC, as well as with Xilinx Vivado for a Nexys 4 DDR Artix-7 FPGA board.

2.7.2 Overhead - Results and Analysis

In order to analyze the different parameters at play during obfuscation, as well as the validity of our whole process, we performed more detailed tests for one transformation in particular, bogus expression insertion.

Benchmark	No Deobfuscation		Naive Deobfuscation		Full Deobfuscation	
	Area(%)	Delay(%)	Area(%)	Delay(%)	Area(%)	Delay(%)
Adpcm	2.61	-0.06	2.29	-0.08	-0.08	-0.01
AES	0.3	<0.01	0.31	<0.01	0.03	2.72
Merge Sort	0.07	<0.01	0.07	<0.01	0.03	0.01
Mips	1.14	0.5	1.45	0.5	-1.12	-7.47
Needwun	9.01	11.69	8.39	11.69	0.26	<0.01
Stencil3d	9.38	0.73	9.2	0.73	1.54	-0.14

Table 2.1 – Average design overhead on ASIC.

Benchmark	No Deobfuscation			Naive Deobfuscation			Full Deobfuscation		
	LUT(%)	FF(%)	Delay(%)	LUT(%)	FF(%)	Delay(%)	LUT(%)	FF(%)	Delay(%)
Adpcm	5.82	28.9	15.9	-0.08	-0.2	10.85	-0.12	-0.2	11.06
AES	29.0	20.0	1.98	7.05	0.42	3.35	8.97	0.42	4.49
Merge Sort	-1.37	18.7	2.57	-6.18	4.08	-1.27	-5.99	4.08	-3.77
Mips	45.9	210	7.51	1.46	-1.39	4.51	1.78	-1.39	4.61
Needwun	23.9	33.5	-4.22	0.79	0.01	-2.78	-0.6	-0.25	-1.98
Stencil3d	20.7	22.4	-9.44	1.38	4.56	-3.78	-1.25	5.57	-0.86

Table 2.2 – Average design overhead on FPGA.

De-obfuscation

To prove that any additional logic added by obfuscation is removed, and to show the benefit of our de-obfuscation approach, we extended the previously presented test flow by performing in total 4 logic syntheses per test:

- The original, never obfuscated RTL IP, used as baseline.

- The obfuscated RTL IP, with **no** de-obfuscation.
- The obfuscated RTL IP, with **naive** de-obfuscation.
- The obfuscated RTL IP, with **full** de-obfuscation.

The overheads in percent when compared to the original IP are presented for each benchmark and for both ASIC and FPGA on Table 2.1 and Table 2.2. These tables contain the results with a branching degree of 1, meaning that for each obfuscated expression, one bogus expression was added. These results show that with no de-obfuscation, design overhead is usually high, up to 12% on ASIC and up to more than 200% on FPGA. Increasing the branching degree during obfuscation would automatically further increase the area overheads, since adding additional expressions in C code results in additional logic in the final design.

When using naive de-obfuscation, results indicate that overall overhead decreases slightly, but is still significant. On the other hand, de-obfuscating the IPs in a targeted way with our proposed full de-obfuscation method strongly reduces overhead. This proves that we are successfully able to remove the code added during obfuscation.

In some cases, the overhead is already close to 0% (e.g. AES on ASIC) without any de-obfuscation. With de-obfuscation, overhead can even be negative. This may happen when the obfuscated code, greatly increased in size due to added bogus code, forces the HLS tool into different, sometimes better optimization choices. In some cases, the different choices thus result in overall smaller final designs. In other cases, the small remaining overhead after full de-obfuscation (e.g. Stencil3D) is in our opinion also due to the choices made by the HLS tool: adding code at behavioral level will for example cause the tool to handle resource sharing differently. Furthermore, the syntactic variances caused by the obfuscation process can also have a significant impact on the HLS tool, see [CGR93].

Obfuscation Level

We studied the impact of obfuscation level, as defined in Section 2.6.3, on overhead using the bogus expression insertion technique as an example. By doing a series of tests with the obfuscation level varying from 5% to 100%, we established whether

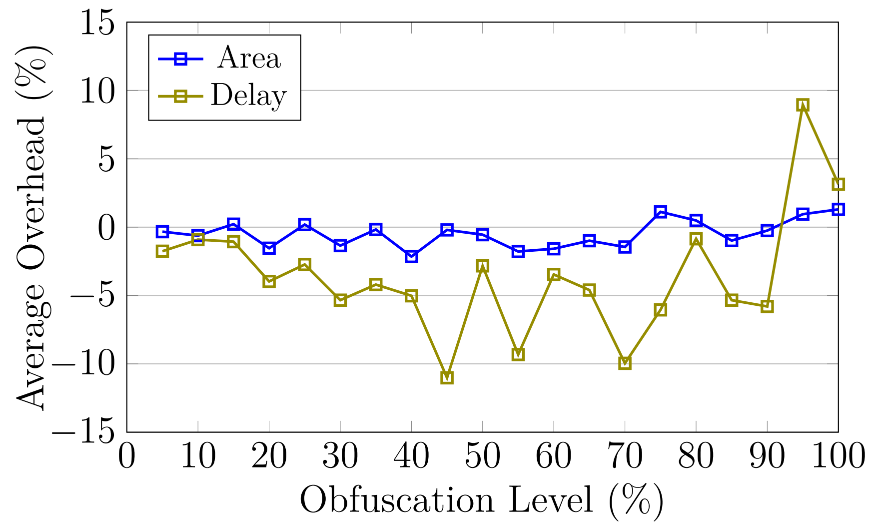


Figure 2.12 – Average design overhead per obfuscation level, for bogus expression insertion - ASIC.

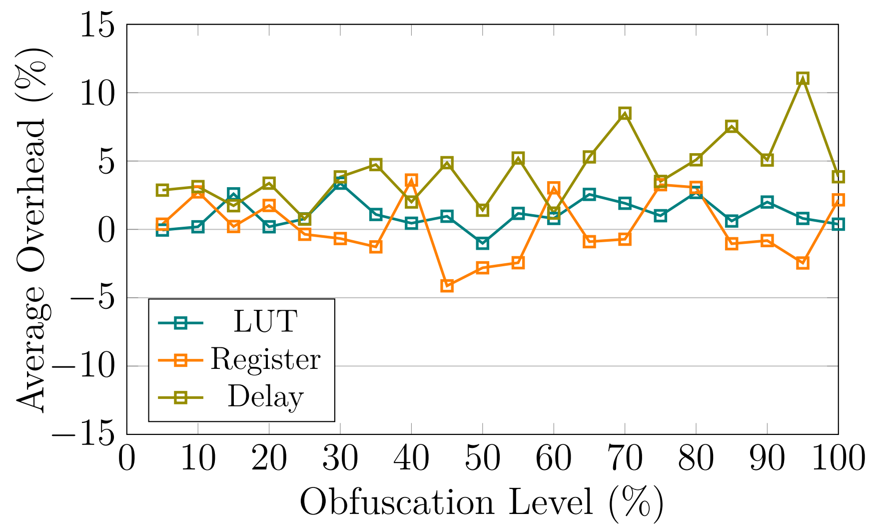


Figure 2.13 – Average design overhead per obfuscation level, for bogus expression insertion - FPGA.

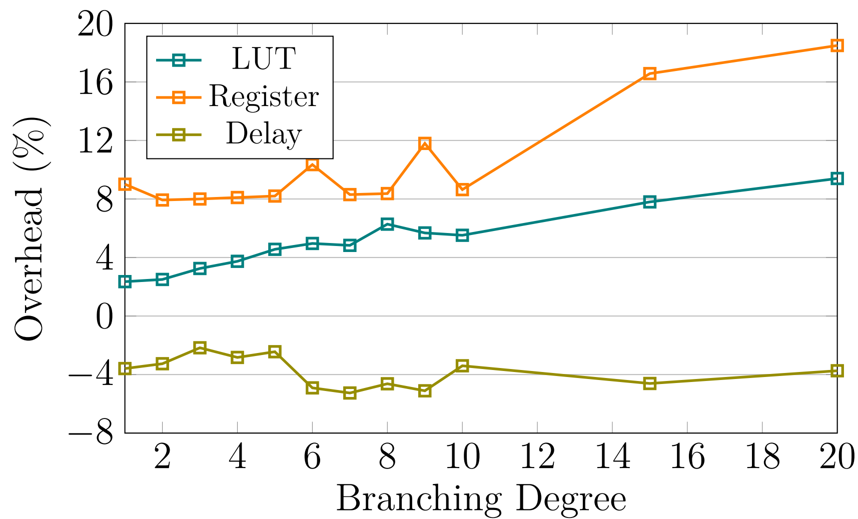


Figure 2.14 – Design overhead per branching degree, for bogus basic block insertion - FPGA - **obfuscated**.

there is a correlation between obfuscation level and overhead. The results, shown on Figure 2.12 and Figure 2.13 for ASIC and FPGA respectively, indicate that there is no such correlation, and in particular that area overhead does not increase with obfuscation level. This positive result demonstrates that our de-obfuscation process is effective no matter what amount of code is obfuscated. We conclude that the maximum level of obfuscation and thus of security, i.e. 100%, can always be chosen.

Branching Degree

We also studied the impact of branching degree, as defined in Section 2.6.3. Preliminary tests were made for one obfuscation technique in particular, bogus basic block insertion. On Figure 2.14 and Figure 2.15, results for Adpcm benchmark, which are representative of results found with other designs, are given. They show that with a higher obfuscation degree, meaning a higher amount of bogus basic blocks added, the design size increases for the obfuscated design. This is to be expected since the number of basic blocks in the obfuscated design is much higher. For the de-obfuscated design however, while the number of registers slightly increases with a high branching degree, overall overhead remains below a threshold

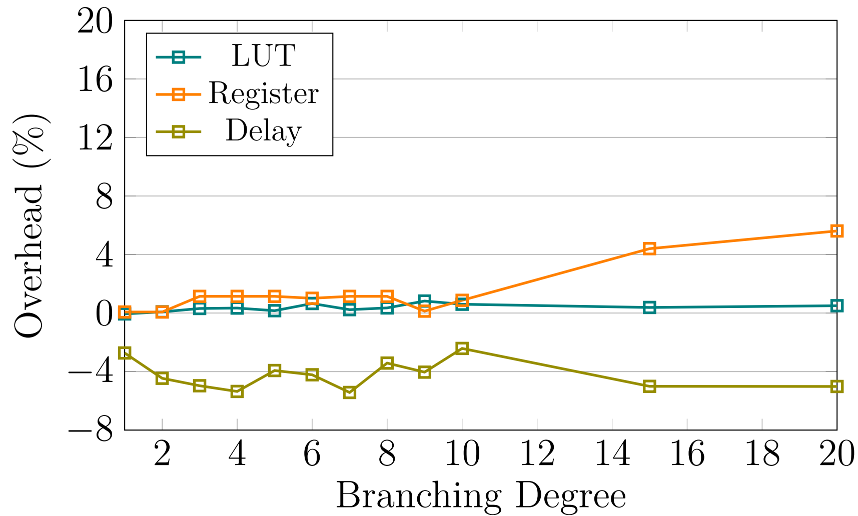


Figure 2.15 – Design overhead per branching degree, for bogus basic block insertion - FPGA - **de-obfuscated**.

of around 5%. These results show that for each obfuscated basic blocks, up to 20 bogus blocks can be added without any significant impact on design performances and size.

Obfuscation Techniques

We have so far performed in-depth testing mainly for bogus code insertion and control flow flattening techniques. For all of these tests, average runtime overhead for HLS is negligible, at under 5%. For bogus code insertion techniques (bogus expressions and bogus basic blocks), the design overhead after logic synthesis is overall satisfactory. In the majority of tests, area and delay overhead are below 5%, which we consider an acceptable threshold. For some benchmarks, overhead can be higher than tolerable. However, this can be mitigated by choosing carefully which obfuscation technique to apply, and which values to use for obfuscation level and branching degree.

For key-based control flow flattening, the results are not as encouraging. When used for software obfuscation, control flow flattening has been shown to result in a high overhead: in [LK09], the authors explain that on average, runtime and size of a program double. In an HLS context, this problem becomes even more

predominant. Not only does control flow flattening hinder compiler front-end optimizations, but it can also severely impact later stages of the HLS process. For example, since loops are no longer easily identifiable in the code, any loop-related optimizations such as loop pipelining or unrolling cannot be performed anymore. This issue can be mitigated by using coarser control flow flattening and leaving some control flow structures intact instead of flattening everything. This does however decrease the level of security. There is thus a trade-off between overhead and security, that has to be balanced for each individual design.

We have also noticed that the dispatch structure used for control flow flattening, either in the form of a global Switch-case statement, or in the form of a series of Goto and labeled statements, has a severe impact on overhead. Flattening the control flow at behavioral source code and rewriting the BIP with such a dispatch structure is effectively equivalent to creating a Finite State Machine (FSM). However this FSM-like structure is not recognized as such by HLS tools, which results in an additional FSM being created on top of the first one. The final overhead due to this behavior is significant (up to 35% on FPGA and 45% on ASIC) and can, in our opinion, not be avoided. It should be noted however that switching from classic control flow obfuscation to key-based control flow obfuscation, which highly improves the security level, does not have any further negative effect on overhead.

Key-based control flow flattening technique, especially when used in conjunction with bogus code insertion techniques, can strongly decrease readability of the code and thwart static code analysis tools. From a security point of view, this is a powerful technique that could be used to secure cloud-based HLS flows. However, it can lead to significant design overhead when used in a hardware context, and should thus be applied only on select designs.

2.7.3 Transience and Security: Discussion

In order for our obfuscation process to be fully transient, two criteria have to be met. First of, the full process, i.e. obfuscation + HLS + de-obfuscation, should ideally not have any impact in terms of design overhead. The results presented in the previous section demonstrate that while the final design is never fully identical to the original one without protection, the difference between both

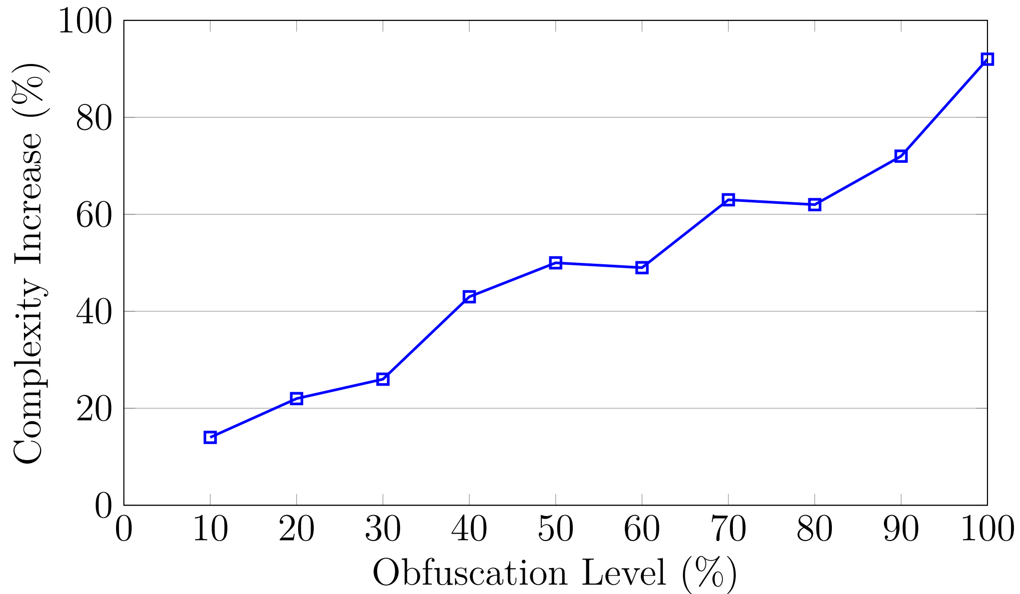


Figure 2.16 – Cyclomatic complexity increase of Needwun source code after obfuscation per obfuscation level.

is usually negligible. For most transformations, the obfuscation overhead is very low. This is however not the case for control flow flattening, which is not well adapted to a hardware context. Second, our process is only transient if the final design has the exact same functionality as the initial design. To verify this, we simulated each circuit before and after de-obfuscation using GHDL [Gin], an open-source simulator for VHDL language, as well as the co-simulation tool provided by Vivado HLS. These simulations show that before de-obfuscation, the obfuscated circuits only perform correctly when the exact right key sequence is applied as inputs. After de-obfuscation, the keys are no longer necessary and the circuit always behave as expected. This proves that our obfuscation process is transient where functionality is concerned, and that the automated de-obfuscation tool does not remove any needed logic.

To evaluate the security of transient obfuscation, we calculated the cyclomatic complexity. While not a precise indicator of the difficulty of attack, a higher complexity does usually imply that the code is more difficult to understand. Our results for bogus expression insertion show that after obfuscation, the complexity increases on average by 109%. Furthermore, the increase in complexity is

positively correlated with the level of obfuscation, see Figure 2.16 for an example with Needwun benchmark, which is representative of the results obtained for other benchmarks. Increasing the level of obfuscation can thus be a simple way to improve security without incurring a significant increase in overhead.

2.8 Conclusion

In this chapter, we explained why HLS-as-a-Service seems a likely scenario. We presented the security concerns, in particular BIP theft and reuse, that can slow down adoption. To secure such a cloud-based HLS flow during design time, we introduced *transient obfuscation*, a novel obfuscation concept which combines software obfuscation techniques and key-based hardware protection methods. By deobfuscating designs after HLS, at RTL or during logic synthesis, we ensure that this protection is transient and does not cause too high overhead. We implemented this concept using an in-house tool, KaOTHIC, which allowed us to run test campaigns with Vivado HLS. Experimental results show that some techniques are better suited to a hardware design context than others, and that simply reusing known software obfuscation techniques is not enough to ensure satisfactory results. In the following chapters, we will base our experiments on bogus code insertion, since this technique is fully transient and does not present any performance issues.

Chapter 3

Transient Obfuscation for BIP Birthmarking

Watermarking can be used to prove ownership by embedding and then extracting a signature in a design. At the behavioral level, existing solutions require modifying commercial or in-house HLS tools by adding a specific watermark embedding step. In this chapter, we propose a solution that is HLS-tool agnostic and relies on side effects of the transient obfuscation techniques presented in the previous chapter. Our solution is based on a software identification technique called birthmarking: by extracting intrinsic properties of a design, instead of a previously embedded signature used in classic watermarking approaches, we are able to prove ownership of a stolen design.

We base our approach on the assumption that our transient obfuscation process creates a form of signature: the sequence of obfuscation, HLS, and then de-obfuscation with specific keys and specific obfuscation parameters results in designs that are unique from a micro-architectural point of view. Two designs, originating from the same C level source code, that have gone through this transient obfuscation process, have identical functionality but present noticeable structural differences. We propose to exploit these differences to identify stolen designs.

After presenting the threat model used for this work and giving background information about watermarking, we explore what effects code transformations such as transient obfuscation can have on HLS results. Next we present related

work on both BIP and software watermarking. We then introduce our approach, based on hardware birthmarking concepts, to uniquely identify IPs after HLS. We study two identifying metrics, based on analysis of a design's scheduling and dataflow. Finally, we provide a detailed experimental setup and in-depth results to validate the approach. The positive results obtained during this work have been published in [Bad+21b].

3.1 Threat Model and Background

3.1.1 Threat Model

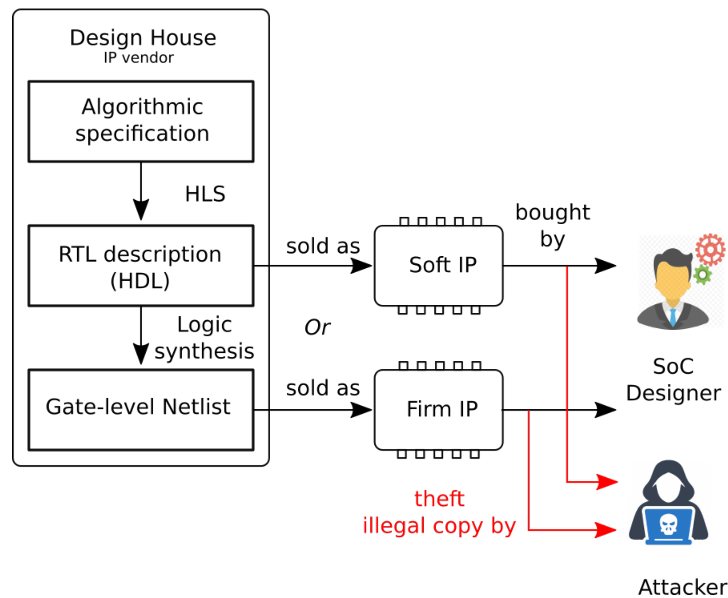


Figure 3.1 – Watermarking Threat Model.

In this work, we focus on the case of a design house using an HLS-based design flow to create IPs that will be sold directly at RTL, or at lower level. A potential attacker, as shown on Figure 3.1, is trying to steal this IP with the goal of making illegal copies and/or reselling the design. The design house on the other hand is trying to protect its IP and wants to be able to claim ownership of a potentially stolen design.

We work with following assumptions:

- The **attacker** has access to the RTL design (soft IP) or can reverse-engineer the gate-level netlist (firm IP) to extract the corresponding RTL description.
- The **attacker** can apply some functionality-preserving modifications to the stolen IP to thwart identification.
- The **defender** (IP vendor/designer) can add proprietary information in form of a *watermark* to the design, with the goal of later identifying and claiming ownership of the design.
- The **defender** is using a commercial HLS tool, which cannot be modified to add a watermark.

3.1.2 Watermarking

Hardware watermarking is a technique used to protect hardware designs from theft and illegal reuse by embedding a unique identifier, or *signature*, that can later be extracted to claim ownership of the stolen IP. The watermarking process is usually comprised of two steps, or functions:

1. *Embed*: this functions takes a design IP as input and adds a watermark w, using a secret key key: $embed(IP, w, key) \implies IP_w$
2. *Extract*: this function extracts the watermark from a design when given the correct key: $extract(IP_w, key) \implies w$

The IP watermark added as defense by the design house should respect following properties [ATA03][Pil+19]:

- **Ownership Credibility**: the IP ownership is easy to verify, with a minimal probability of collision (low false-positive rate)
- **Resilience** against following attacks:
  - **New Watermark Insertion**: the attacker adds his own watermark
  - **Transformations**: the attacker applies some transformations to the design, with the goal of hindering watermark extraction

  - **Removal**: the attacker finds and removes the watermark
  - **Re-synthesis**: the watermark is removed by a synthesis tool

- **Low Design Degradation**: the watermark does not affect functionality of the design and has a minimal impact on design performance

3.1.3 Effects of Code Transformations on High-Level Synthesis

Even without introducing deep transformations in the source code, expressing the same algorithm using an alternative syntax can result in considerable differences in the RTL design: the HLS process is notably sensitive to so-called “syntactic variance”. This phenomenon has been studied since the early 90s [CGR93], where forms of canonical representation to minimize this effect were soon proposed. However, to the best of our knowledge, none of these representations has proven to be fully effective in practice, despite continuous interest from the HLS community. Even intermediate representations based on static single assignment (SSA) form, commonly employed in modern HLS front-ends, and which tend to reinforce the dataflow exposition of the whole program, do not fully suppress this effect.

More recently, some authors [Hua+13] have studied the impact of optimizations performed by the compiler front-end on HLS quality of results. They demonstrate that the choice of optimization passes and of pass ordering can significantly affect the area and latency of the resulting circuits. However, they also note again that this impact is strongly influenced by the input design: an identical set of optimization passes can lead to very different results depending on the input design.

In [VS17a], the authors focus on how software obfuscation techniques, when applied to behavioral source code, can negatively affect HLS quality of results. They show that by obfuscating behavioral IPs using conventional software obfuscation tools, the compiler front-ends used in HLS tools fail to correctly optimize the code. This in turn results in a degradation of the synthesis results.

These different works confirm our intuition that by applying transformations on behavioral level source code, transient obfuscation has a direct effect on the HLS process. Even after de-obfuscation, once original functionality is recovered,

the design keeps traces of the obfuscation process in its structure. These traces are unique for each design and each individual sequence of obfuscating transformations.

3.2 Related Work

3.2.1 BIP Watermarking Techniques

While many publications exist on how to watermark IPs at lower design levels, in recent years there has been an increased focus on watermarking IPs at the behavioral level. Most techniques rely on modifying a HLS tool by adding specific security passes, with the goal of inserting a watermark in a design during HLS. In [KHP05], the authors propose to insert a watermark during register allocation and/or binding: watermark constraints, in the form of fictional edges, are added to the interval graph used to represent the variable lifetimes. These additional constraints modify the final design by leading to a different solution for register allocation. In [SB16], watermarks are also embedded as additional constraints during register allocation. However, the authors propose to use design space exploration (DSE) to find an optimal watermark that still respects user constraints for latency and area. Several similar constraint-based approaches have been proposed, both at lower and at behavioral level. In [Cui+11], the watermark is embedded directly into the FSM, specifically in the state transition graph (STG), by modifying some of the STG edges without changing functionality.

In [Pil+19], the authors introduce the idea of *benevolent Trojans*: watermarks are hidden in designs during HLS by reusing Hardware Trojan (HT) concepts. Just as HTs, the watermarks are composed of a trigger and a payload, and are added before the backend phase of HLS.

3.2.2 Software Watermarking Techniques

Software watermarking was first introduced in 1996 as a patent [DM96], with a technique based on basic block reordering. Several other methods based on reordering, applied for example to operations [SS08], operand coefficients [SJX09],

function declarations or constants [Gon+08], have also been proposed. However, reordering-based watermarking techniques have very low resilience to attacks where semantics preserving transformations are applied [DBC19]: since the watermark extraction step relies on checking the order of items (e.g. basic blocks, operations) in the program, an attacker simply has to reorder the items to thwart correct watermark extraction.

In [QP98], the authors introduce *QP*, an algorithm that embeds a watermark by modifying the register allocation of the program: by adding additional constraints representing a watermark to the interference graph, the registers assigned to variables are changed. Several improvements on the initial algorithm, with stronger credibility and resilience, have been proposed in later years, such as *QPS* [MC03] or *QPI* [ZT05].

Other graph-based approaches, where a watermark in form of an integer is encoded in a graph structure, have also been proposed. In [VVS01] for example, bogus control flow is added to hide the watermark in the CFG.

Several other approaches can also be noted. In [Arb02], the constants in opaque predicates are used to hide a watermark. In [Zen+10], watermarking based on obfuscated interpretation is proposed.

More detailed overviews and surveys of existing software watermarking techniques can be found for instance in [HD11] or [DBC19].

3.3 Proposed Approach

3.3.1 Complete Flow and Notations

Let C be the original design in C. By applying KaOTHIC obfuscation with a key K^i and a series of parameters P^i we get an obfuscated design C^i_{obf}.

Then we apply HLS to the original and the obfuscated design to get RTL designs. Next de-obfuscation is applied.

$$C^i_{obf} = Obf(C, P^i, K^i) \tag{3.1}$$

$$RTL^i_{obf} = HLS(C^i_{obf}) \tag{3.2}$$

$$RTL^i_{deobf} = Obf^{-1}(RTL^i_{obf}, K^i) \tag{3.3}$$

The resulting design, RTL^i_{deobf}, has the same functionality as the original design RTL. However, the micro-architecture of this design is different due to different HLS design choices. On the other hand, both RTL^i_{obf} and RTL^i_{deobf} have the same architecture: the only differences between these two designs is the logic removed during de-obfuscation.

We postulate that our 3 step process (obfuscation + HLS + de-obfuscation) creates a design that is unique and could not have been created by someone else. These architectural differences can thus be used as an identifying watermark.

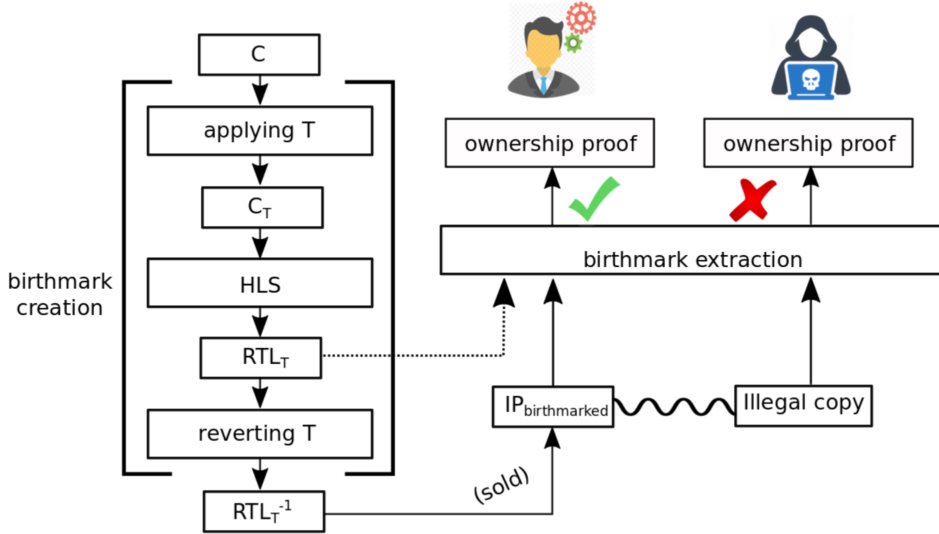


Figure 3.2 – Watermarking Flow.

In our updated threat scenario, shown on Figure 3.2, the potential IP thief only has access to RTL^i_{deobf}, while the design house has sole access to RTL^i_{obf}. Verifying the watermark can be reduced to following problem: proving a relationship between RTL^i_{obf} and RTL^i_{deobf}. More precisely, **by proving that RTL^i_{deobf} *originates* from RTL^i_{obf}, and by being the only entity with access to**

RTL^i_{obf}**, the design house can prove ownership of the design.**

3.3.2 Watermark Insertion

Our watermark insertion process is made of three steps: obfuscation of the source code (behavioral level), HLS, de-obfuscation of the design (RTL). For HLS, our method does not rely on a specific, modified tool: while the de-obfuscation process might require some adjustments depending on the HLS tool used, the overall method stays the same. Our process thus becomes an extension of a classic HLS flow with two extra steps before and after HLS.

Obfuscation

The first step, obfuscation, applies code transformations to the original source code, with the goal of influencing the design choices made by the HLS tool. We propose to reuse transient obfuscation methods presented in the previous chapter (Chapter 2). In particular, we focus on Bogus Basic Block Insertion, as introduced in Section 2.4.1: a list of all eligible basic blocks in the original code is established. Some or all of these blocks are selected to be obfuscated. Before each obfuscated block, an If/Else statement with an input key as condition is inserted. Then, one or several bogus basic blocks are added next to each existing obfuscated block. These bogus blocks are copies of the original blocks, with following modifications: change of operators, switching of operands, switching of instruction order. These modifications are light enough to make the bogus blocks hard to distinguish from real blocks (thus preventing an attack by deobfuscation), but still have a strong impact on the HLS design choices (thus creating a strong watermark). The overall important amount of bogus code added ensures that the design differences are significant.

De-obfuscation

After HLS has been performed on the obfuscated source code, the resulting RTL design is still obfuscated, meaning that it still contains large amounts of bogus code and has an altered functionality. By de-obfuscating, we ensure that original

functionality is recovered and that any bogus logic is removed to minimize resource overhead. The resulting de-obfuscated design has the same functionality as the original design and a similar size, but has a different micro-architecture, which we exploit as a watermark. Since the watermark should be recoverable directly at RTL, we cannot rely on logic synthesis to remove any dead code. Instead, we use the full RTL de-obfuscation method presented in Section 2.5.4: the key values are propagated as constants throughout the design, and any dead code is removed.

3.3.3 Watermark Verification: Birthmarking Concepts

As explained in Section 3.3.1, our watermark verification relies on following idea: proving that one design (RTL^i_{deobf}) was obtained from another design (RTL^i_{obf}) by removing part of the logic (de-obfuscation step). Proving this relationship is closely related to problems of circuit *similarity* and *containment*. A direct link with *software birthmarking*, where the goal is to determine if two programs share a common origin, can also be established. In this subsection, we propose to explore what *software birthmarking* is, how it translates to the hardware domain, and how its principles can be applied to our watermark verification problem.

Software Birthmarking

Software birthmarking was first mentioned in [Gro89], where it was used for program identification. In [Tam+04], the authors proposed different birthmarking techniques to detect the theft of JAVA programs. The idea was further detailed and studied in [MC04].

In these works, a birthmark is defined as a set of characteristics that can be extracted from a program. These characteristics are chosen with the goal of being used as identification: by definition, if two programs have the same birthmark, then one has to be a copy of the other.

Two properties should be verified by birthmarking techniques:

1. **Credibility:** The fact that two programs have the same function does not imply that one is a copy of the other. A birthmarking technique is considered credible if two independently writing programs with the same functionality result in two different birthmarks.

2. **Resilience to transformations:** Birthmarks should be resistant to semantics-preserving transformations such as obfuscation.

It should be noted that birthmarking can only prove that two programs share the same origin. It does not provide a direct proof of authorship, contrary to watermarking. Moreover, whereas watermarking relies on embedding and then extracting a specific mark in a program, birthmarking is based solely on extracting intrinsic properties of a program and thus does not have an embedding step.

Hardware Birthmarking

Software birthmarking principles can be applied to hardware designs, as demonstrated in [ZA15]. In this work, birthmarks are used to assess the similarity of circuits, with the goal of improving productivity by suggesting existing and reusable similar designs during the hardware design process. A hardware birthmark is defined as a set of characteristics that give both functional and structural information about a circuit. Just as software birthmarks, hardware birthmarks can be defined in the following way:

Let a and b be two circuits, and bm a birthmark extraction function. $bm(a)$ is a birthmark of circuit a if and only if:

1. $bm(a)$ is obtained only from a itself
2. a and b are copies of each other $\implies bm(a) = bm(b)$

Extraction Principles

We propose to apply these hardware birthmarking principles to our watermark verification process: as explained above, ownership proof is established by matching a de-obfuscated RTL design with the obfuscated design it originates from. These two designs are not direct copies of each other as is the case when using birthmarking, but they are closely tied together, share the same origin and are highly similar from a structural point of view. Moreover, since the transformation from obfuscated to de-obfuscated design (*de-obfuscation*) is well known, we can state that the de-obfuscated design is *contained* in the obfuscated design (barring some additional transformations done by the attacker).

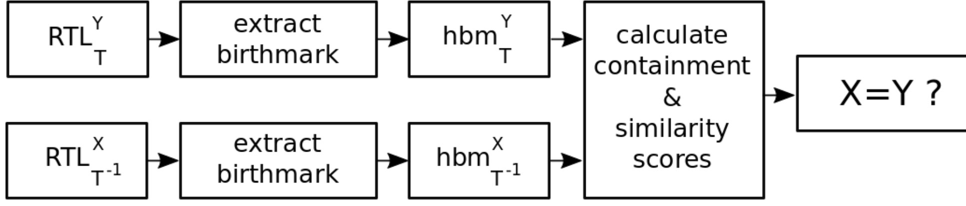


Figure 3.3 – Birthmark Extraction Flow.

Proving this specific type of similarity, or this specific relationship between two designs, can be done by using the same process as for traditional birthmarking: by extracting a set of design characteristics and comparing them, see Figure 3.3.

We propose to find a set of metrics that can be used, individually or combined, as birthmarks. For each of these metrics, it is unlikely that the birthmarks of two designs will be identical. Therefore a *similarity*, or *containment* measure has to be defined and computed for each metric. These measures should have following properties:

$$
\begin{aligned}
&\forall i \in \mathbb{N}, containment(bm(RTL^i_{deobf}), bm(RTL^i_{obf})) = 100\% \\
&\forall i, j \in \mathbb{N}, similarity(bm(RTL^i_{deobf}), bm(RTL^i_{obf})) \\
&\qquad \gg similarity(bm(RTL^i_{deobf}), bm(RTL^j_{obf}))
\end{aligned}
\tag{3.4}
$$

We note that the two properties used to evaluate birthmarking techniques can also be adapted to our watermarking context:

1. **Credibility:** Our watermark extraction technique should only detect direct obfuscated-deobfuscated pairs. If two designs originate from the same original C code, but were obfuscated differently, then they should not be matched by our technique.

2. **Resilience to transformations:** Our technique, by definition, matches two designs where one was transformed by de-obfuscation, i.e. removing part of the logic. However it should be resilient to additional transformations: it should also match if the attacker performs transformations such as inserting

his own watermark or renaming all signals in the design.

3.4 Metrics for Watermark Verification

Our goal is to find several *metrics* that can be extracted from design pairs and used as birthmarks. Some of these metrics will result in birthmarks with a perfect similarity score as long as no modification is made to the de-obfuscated design, for example the number of states in the FSM. However, they might not resist against modifications of the design (attacker applying his own watermark, etc.). This means that the robustness of each metric has to be studied individually, and also explains the need for a combination of these metrics instead of relying on just one of them.

3.4.1 Scheduling

Definitions and Assertions

An algorithmic level design contains a given set of operations, which are spread throughout the basic blocks. During HLS, these operations are *scheduled* in different clock cycles.

Let $OP = \{o_i \mid 1 \leq i \leq n\}$ be the set of operations in an original, un-obfuscated C design. When obfuscating a design, several bogus operations are added. We note $OP_{obf} = \{o_i \mid 1 \leq i \leq m\}$ the set of operations in an obfuscated design, where $m \geq n$. We also define OP_{bogus} as the set of bogus operations added to a design during obfuscation. Following relations can be noted:

$$OP_{obf} = OP \cup OP_{bogus} \tag{3.5}$$

$$OP \subset OP_{obf} \tag{3.6}$$

When de-obfuscating a design, most, if not all operations added during obfuscation are removed. However, all operations present in the original design remain in the de-obfuscated design (by definition of our de-obfuscation method, which preserves design functionality). It should also be noted that no new operations are added during de-obfuscation. We can thus add:

$$OP \subset OP_{deobf} \subset OP_{obf} \tag{3.7}$$

During HLS, operations are scheduled in a set of control steps, or states, $S = \{s_j \mid 1 \leq j \leq r\}$. Each state thus contains a subset of operations $OP(s_j) \subset OP$.

The bogus operations and control flow introduced by obfuscation have a direct impact on the scheduling choices made by any HLS tool. This means that an obfuscated design has a different scheduling in regards to the original design. These differences are also reflected in the deobfuscated design: our de-obfuscation method removes bogus logic, but does not modify the scheduling.

$$\forall s_j \in S_{deobf},\ OP_{deobf}(s_j) \subseteq OP_{obf}(s_j) \tag{3.8}$$

In other words, a de-obfuscated design's scheduling is *contained* in the obfuscated design's scheduling. This assertion is only always true for an obfuscated-deobfuscated design pair, i.e. with two different obfuscated versions of the same original design, the scheduling differs. For example, starting from one design obfuscated in two different versions with parameters 1 and 2, we have:

$$\begin{aligned}
&\forall s_j \in S^1_{deobf},\ OP^1_{deobf}(s_j) \subseteq OP^1_{obf}(s_j) \\
&\forall s_j \in S^2_{deobf},\ OP^2_{deobf}(s_j) \subseteq OP^2_{obf}(s_j) \\
&\forall s_j \in S^1_{deobf},\ OP^1_{deobf}(s_j) \not\subset OP^2_{obf}(s_j) \\
&\forall s_j \in S^2_{deobf},\ OP^2_{deobf}(s_j) \not\subset OP^1_{obf}(s_j)
\end{aligned} \tag{3.9}$$

Scheduling Similarity and Containment

In order to find a measure that allows to verify the properties given in Equation (3.9), we study two potential metrics:

1. The *similarity* between two designs' schedulings.

2. The *containment* of one design's scheduling in another.

These metrics are designed to have following properties:

- Both metrics are scores normalized between 0 and 1.

- Similarity is symmetric, i.e. $similarity(A, B) = similarity(B, A)$.

- Both metrics are defined as the average of that metric applied to individual states:

$$similarity(S^A, S^B) = \frac{\sum_{j=1}^{r_{max}} similarity(s_j^A, s_j^B)}{r_{max}}$$
$$containment(S^A, S^B) = \frac{\sum_{j=1}^{r_{max}} containment(s_j^A, s_j^B)}{r_{max}}$$
$$where\ r_{max} = \max(r^A, r^B) \tag{3.10}$$

It should be noted that S^A and S^B can have a different number of states r^A and r^B. For ease of calculation, we pad the design with less states by adding empty states.

In our threat model, the attacker can rename any signal in the design. We thus cannot rely on variable names to recognize, which makes it hard to match individual operations between two designs. We simplify this problem by focusing only on operation *types*. This means that two states are considered identical if they contain the same amount of each operation type.

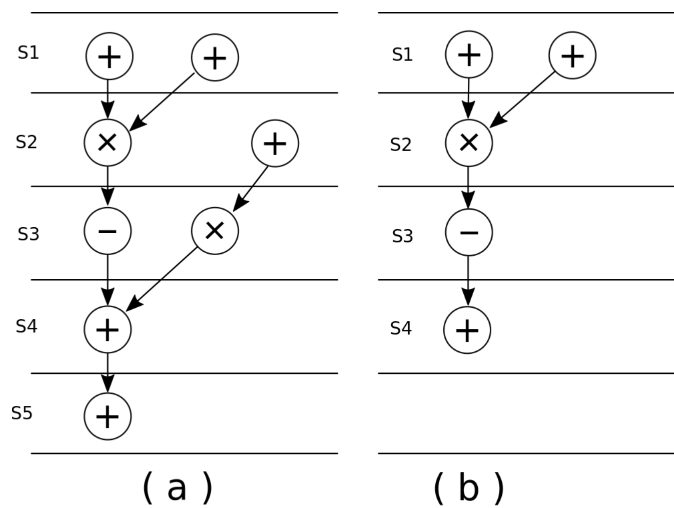


Figure 3.4 – Two scheduling examples.

On Figure 3.4, two simple scheduling examples are given. State s_1 and s_4 are

identical in both examples. s_2 and s_3 of design (b) are contained in design (a). Following properties should be verified by the chosen similarity and containment measures:

$$
\begin{aligned}
similarity(s_1^a, s_1^b) &= 1 \\
similarity(s_3^a, s_3^b) &= 0.5 \\
similarity(s_5^a, s_5^b) &= 0 \\
containment(s_2^a, s_2^b) &= 0.5 \\
containment(s_2^b, s_2^a) &= 1 \\
containment(s_5^a, s_5^b) &= 0 \\
containment(s_5^b, s_5^a) &= 1
\end{aligned}
$$

By applying the similarity and containment measures of two documents given in [Bro97] to sets of operators in each state, we obtain the following formulas:

$$similarity(s^a, s^b) = \frac{|OP(s^a) \cap OP(s^b)|}{|OP(s^a) \cup OP(s^b)|} \tag{3.11}$$

$$containment(s^a, s^b) = \frac{|OP(s^a) \cap OP(s^b)|}{|OP(s^a)|} \tag{3.12}$$

This similarity measure is commonly known as Jaccard similarity coefficient.

It should be noted that, since we only focus on operation types and not individual operations, $OP(s^x)$ is actually a *multiset*, i.e. a set that can contain multiple instances of each element, in this case multiple instances of each operation type. To illustrate, applying these definitions to the examples in Figure 3.4 gives following results:

$$
\begin{aligned}
similarity(s_3^a, s_3^b) &= \frac{|\{-,*\} \cap \{-\}|}{|\{-,*\} \cup \{-\}|} = \frac{|\{-\}|}{|\{-,*\}|} = \frac{1}{2} \\
containment(s_2^a, s_2^b) &= \frac{|\{*,+\} \cap \{*\}|}{|\{*,+\}|} = \frac{|\{*\}|}{|\{*,+\}|} = \frac{1}{2} \\
containment(s_2^b, s_2^a) &= \frac{|\{*\} \cap \{*,+\}|}{|\{*\}|} = \frac{|\{*\}|}{|\{*\}|} = 1
\end{aligned}
$$

3.4.2 Dataflow

When adding bogus code to a C level design, the resulting RTL design's dataflow graph (DFG) also contains an amount of bogus dataflow. However, even after removing any bogus logic, the dataflow can still be different from the original, unobfuscated one: the added bogus code results in different compiler frontend optimizations, as well as different design choices by the HLS tool. Our goal is to leverage this difference whenever possible.

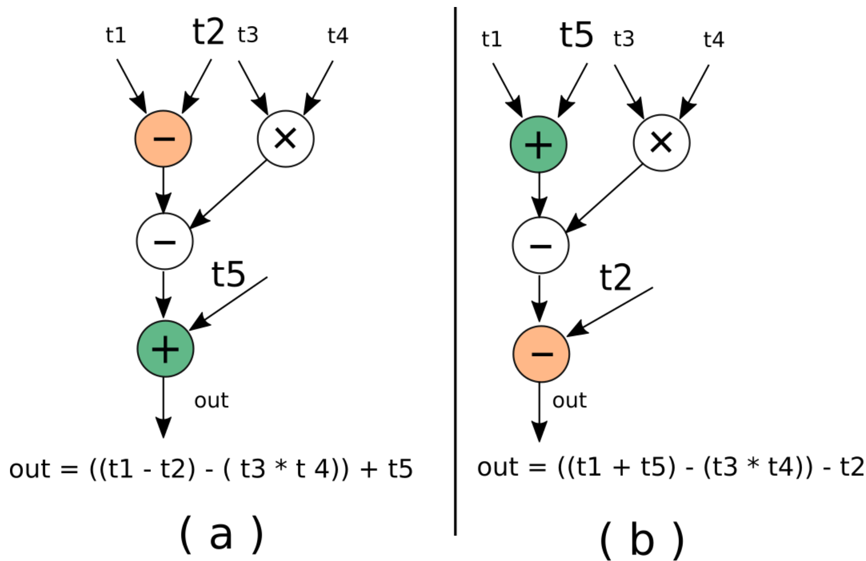


Figure 3.5 – Partial DFG of original and obfuscated code.

The example in Figure 3.5 demonstrates how, by obfuscating the code, the dataflow was modified: while the overall behavior remains the same, the operators are chained in a different order.

Let $DFG = (V, E)$ be the dataflow graph of a design, with V as set of vertices and E a set of edges. In particular, DFG^i_{obf} is the DFG of a design obfuscated with parameters i, and DFG^i_{deobf} is the DFG obtained after de-obfuscating the previous design. Our de-obfuscating process removes signals and operators of the design, in other words it removes part of the vertices and edges of the DFG. There are no new vertices added. However, there can be some new edges, which are used to link two vertices that were previously separated by one or several now deleted vertices.

$$
\begin{aligned}
&V^i_{deobf} \subseteq V^i_{obf} \\
&DFG^i_{deobf} \subseteq DFG^i_{obf}
\end{aligned} \tag{3.13}
$$

The result is that the de-obfuscated DFG should be overall *contained* in the obfuscated DFG. In order to extract a watermark, we thus aim at calculating similarity and/or containment between two dataflow graphs.

Graph Edit Distance

To calculate the similarity of two graphs, the Graph Edit Distance (GED) can be used. This measure is the minimum of graph edit operations necessary to transform one graph into another. Among these operations are vertex and edge insertion, deletion and substitution. In our case, there are two drawbacks with using GED as a similarity measure between two DFGs.

- GED is a NP-complete problem, and even approximate algorithms are computationally heavy for more complex graphs.
- During de-obfuscation of a heavily obfuscated design, large amounts of logic are removed. This results in a design that is much smaller, and thus has a DFG that is highly different. In that case, the GED is large and does not adequately reflect the fact the containment of the smaller, de-obfuscated design in the obfuscated one.

Datapath Extraction

Instead of considering the complete DFG of a design, we propose to focus on individual datapaths leading from circuit inputs to outputs. As proposed in [ZA15], we represent each datapath as a sequence. Each node in this datapath becomes a letter in the sequence, with different letters assigned for each node type: operators, register, signal, etc.. Comparing two datapaths extracted from two designs thus becomes comparing two sequences of letters. Various techniques can be used to evaluate containment and similarity of sequences, such as Levensthein distance, or

sequence alignment techniques, often used in bioinformatics or natural language processing.

Let $I = \{i_1, \ldots, i_m\}; O = \{o_1, \ldots, o_n\}$ be the set of inputs and outputs of a design. For each pair of input and output, we extract a datapath $D(i_x, o_y) = \{n_1, ..., n_p\}$ where each n_i is a node in the datapath, n_1 is the input node i_x, and n_p is the output node o_y. For each design, we thus obtain a set of possible datapaths from inputs to outputs: $D = \{D(i_x, o_y) \mid \forall x \in I, \forall y \in O\}$. There are $m \times n$ possible datapaths in D, where m is the number of inputs, and n is the number of outputs.

Next, for each node n_x in a datapath, we assign a letter depending on the node type. The datapaths D thus become sequences of letters S. The result for each design is a set of sequences $S = \{S_i \mid 1 \leq i \leq p\}$, where $p = m \times n$ is the number of input/output pairs.

Datapath Comparison

On Figure 3.6, two designs with each two inputs and one output are illustrated as an example: the figure shows the shortest path from each input to each output for both designs. Below, using a simple alphabet, each path is converted into a sequence of letters. We cannot rely on naming of inputs and outputs for similarity analysis. The sequences are hence arbitrarily numbered and the names of the I/Os (here i_1, i_2, o_1) are discarded. Following sequences are obtained:

$$\begin{aligned} S_1^A &= \text{``IAFCO''} \\ S_2^A &= \text{``IGCMHO''} \\ S_1^B &= \text{``IGCMO''} \\ S_2^B &= \text{``IAHCO''} \end{aligned} \tag{3.14}$$

Our goal is to compare two DFGs by calculating different metrics such as similarity, distance, or containment. We start with the set of datapath sequences extracted for each design: S^A, S^B. For each sequence S_i^A of design A and each

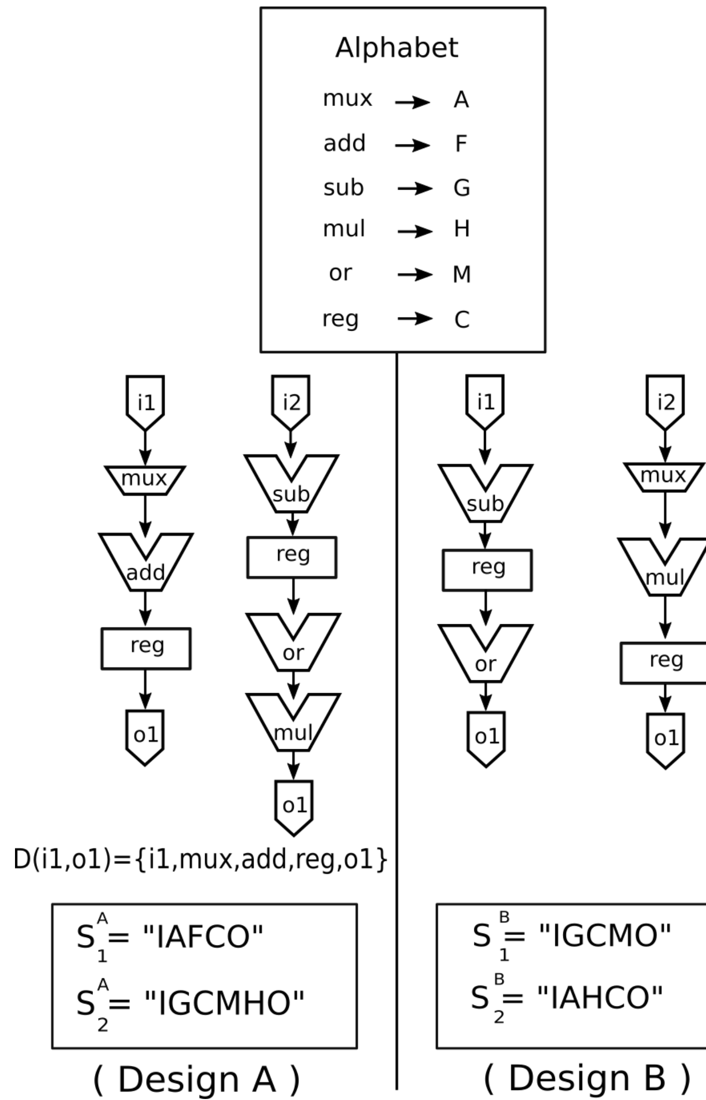


Figure 3.6 – Example of dataflow sequences for two designs.

sequence S_j^B of design B, we calculate a score for each metric, e.g. similarity: $sim(S_i^A, S_j^B)$. We thus obtain a total of $p^A \times p^B$ scores, where p is the number of inputs times the number of outputs of a design.

For the example in Figure 3.6, we obtain 4 similarity scores:

$$
\begin{aligned}
sim(S_1^A, S_1^B) &= 60\% \\
sim(S_1^A, S_2^B) &= 80\% \\
sim(S_2^A, S_1^B) &= 83\% \\
sim(S_2^A, S_2^B) &= 50\%
\end{aligned}
\tag{3.15}
$$

To obtain a global similarity score, we choose to keep the sequence pairs with the highest similarity score. In the case of the example studied here, we keep the pairs (S_1^A, S_2^B) and (S_2^A, S_1^B). The overall similarity is then calculated by taking the average of these pairs similarities:

$$
\begin{aligned}
sim(S^A, S^B) &= avg(sim(S_1^A, S_2^B), sim(S_2^A, S_1^B)) \\
&= avg(80\%, 83\%) \\
&= 81.5\%
\end{aligned}
\tag{3.16}
$$

Datapath Comparison Metrics

We propose to calculate three different metrics to compare two datapath sequences:

1. **Levenshtein distance / similarity**: this is a metric commonly used to measure the distance between two sequences. It can be defined as the number of editing operations (insertion, substitution, deletion) necessary to obtain one sequence from the other. Similarity between two sequences p and q can be defined as follows:

$$similarity(p, q) = 1 - \frac{distance(p, q)}{max(|p|, |q|)} \tag{3.17}$$

2. **Longest Common Subsequence** (LCS): this similarity metric is based on finding the longest common subsequence between two sequences, i.e. finding

the length of the longest subsequence present in both sequences. A similarity score is then calculated by calculating the ratio of the sum of all found subsequences and the total number of letters in both sequences.

3. **Containment**: for scheduling containment, we focused on the containment of a *set* in another. Here, we calculate containment for *sequences*: the order of items in the sequences has to be taken into account. We say that a sequence p is fully contained in a sequence q if p can be obtained from q only by removing zero or more items: no items can be added, and the order of the items cannot be changed. We defined containment of a sequence p in a sequence q as the ratio of letters in p that can be found in the correct order in q divided by the total number of letters in p. For example:

$$\begin{aligned} containment(\text{``}abc'', \text{``}abcd'') &= \frac{3}{3} = 1 \\ containment(\text{``}abcd'', \text{``}abc'') &= \frac{3}{4} = 0.75 \\ containment(\text{``}acb'', \text{``}abcd'') &= \frac{2}{3} = 0.67 \end{aligned} \tag{3.18}$$

3.5 Implementation and Experimental Setup

The goal of this experiment is to verify if the aforementioned metrics can be used as birthmarks, i.e. if their credibility and resilience are satisfactory.

3.5.1 Experimental Setup and Dataset

We start with a set of C designs. These designs are then obfuscated with varying parameters, resulting in several different obfuscated C designs for each original design. HLS is performed on these N obfuscated designs, resulting in N still obfuscated RTL designs. Finally de-obfuscation is applied, leading to N de-obfuscated designs. Both the obfuscated and the de-obfuscated designs form a pool of $2N$ RTL designs, for which each metric is calculated individually. All of the obfuscated

and de-obfuscated designs are combined pair-wise to form a dataset of $N * N$ design pairs. These pairs are labeled *positive* if both designs share a common origin, i.e. one was obtained by de-obfuscating the other, and else *negative*. The resulting dataset contains N design pairs labeled positive, and $N * (N - 1)$ pairs labeled negative.

In order to have a sufficiently sized dataset, we use C programs automatically generated with *Crokus* (presented in Section 3.5.4). The same methodology is then applied to benchmarks from CHStone and MachSuite in order to validate our approach.

For now, the testing pool is composed of 5 original C codes, each obfuscated with 20 different parameters. These different parameter sets are obtained by varying both obfuscation level and branching degree. There are thus 100 obfuscated and 100 corresponding de-obfuscated designs. The dataset counts 100 positive design pairs, and 9900 negative design pairs. This strong imbalance of the dataset is taken into account when evaluating each metric.

3.5.2 Scheduling

For each RTL design, we start by extracting the list of states and operations. Then a script automatically lists which operation is scheduled in which state. Similarity and containment are computed, as explained in Section 3.4.1, by comparing the operations scheduled state by state.

To test resilience of this metric against transformations such as obfuscation or the attacker adding his own watermark, we randomly add noise in the form of bogus operators to the de-obfuscated designs. In this case, the de-obfuscated design's scheduling is no longer 100% contained in the obfuscated design. However, the overall containment score, combined with the similarity score, should still be high enough to have a significant probability of correct detection.

3.5.3 Dataflow

We use Yosys [WGK13], an open-source synthesis tool, to obtain the DFG of each tested design in the form of a DOT file. Then for each input-output pair, when possible, we extract the shortest dataflow path using Networkx, a Python library

for graph manipulation. These paths are converted into letter sequences using a pre-defined alphabet, with one letter assigned to every node type. We thus store for each design a list of input-output pairs and a sequence standing for their shortest path.

Next we compare two designs by calculating several metrics pairwise for all possible input-output pairs. Finally, for each design pair and each metric, we retrieve both the average and the maximum value across all input-output pair combinations. We thus obtain two possible values for each design pair and each metric.

The three metrics studied are calculated as follows:

- Containment: an in-house script is used to calculate the containment of one sequence inside another.
- Similarity: a Python library is used to calculate the Levenshtein distance between two sequences. The similarity is directly obtained from this distance, as explained in Section 3.4.2
- LCS (Longest Common Subsequence): a Python function calculates the similarity of two sequences based on their longest common subsequence.

3.5.4 Crokus

Crokus is a parser and source-to-source code manipulation tool for a large subset of C language. It was developed in Ruby at ENSTA Bretagne by JC Le Lann with the goal of easing code manipulation for various experiments, at AST, IR and CFG level. For the purpose of this thesis, we added several specific additional passes. For this chapter, we added the ability to automatically generate random C programs, starting from a series of parameters such as the number of basic blocks, the average number of instructions per basic block, the presence of loops, etc. This functionality enables us to quickly generate large amounts of C programs, to be used for statistical experiments relying on large datasets. Crokus also has the ability to automatically insert hardware Trojans into C programs. A more detailed presentation of this functionality is provided in Section 4.4.1. Crokus is fully open-source and available for reuse on GitHub [Le].

3.6 Results and Analysis

In this section, we present results for different containment and similarity measures for both proposed birthmarks: scheduling and dataflow. We show how these birthmarks can be used to classify design pairs to verify whether a watermark is present. Furthermore, we demonstrate how initial results can be improved by changing obfuscation parameters, and we discuss the credibility and resilience to attacks of our approach.

3.6.1 Scheduling

Initial Results

	Similarity			Containment		
	Min	Max	Avg	Min	Max	Avg
All pairs (10000 pairs)	0.0%	99.8%	4.7%	8.3%	100%	51.1%
All correct pairs (100 pairs)	**89.9%**	**99.8%**	**97.8%**	**100%**	**100%**	**100%**
All wrong pairs (9900 pairs)	0.0%	99.6%	3.8%	8.3%	100%	50.6%
Wrong pairs but same original C code (1900 pairs)	0.9%	99.6%	11.2%	14.2%	100%	54.1%
Different original C code (8000 pairs)	0.0%	18.8%	2.0%	8.3%	91.4%	49.7%

Table 3.1 – Similarity and containment of scheduling: experimental results.

In Table 3.1, we present experimental results for similarity and containment of different combinations of obfuscated - deobfuscated designs. Similarity values vary between 0% and 99.8%, with a low average of 4.7%. Containment values vary between 8.3% and 100%, but with a higher average of 51.1%. As expected, all deobfuscated schedulings are fully contained in their obfuscated origin's scheduled (correct pairs: $obf_X - deobf_X$). Their similarity is also high, with an average of 97.8%, but can be as low as 89.9%. For incorrect pairs ($obf_X - deobf_Y$), the minimum and average similarity and containment values are much lower. However,

the maximum values, at respectively 99.6% and 100%, are close to the results obtained for correct pairs. This means that there is a risk of having some false positives when using these metrics for watermark detection.

We also calculated results for pairs of designs that do not have the same obfuscation ($obf_X - deobf_Y$), but share the same original C design before obfuscation. These pairs have a higher likelihood of having close schedulings, but should not be detected as correct pairs, i.e. they should not have the same birthmark. This concept was defined in Section 3.3.3 as *credibility* of a birthmark. Here we note that both similarity and containment are on average quite low for these specific pairs, at 11.2% and 54.1%. This means that both metrics have a high credibility. However, the maximum value (99.6% and 100%) show that the aforementioned risk of false positives is due to these same origin pairs in particular, which slightly diminishes the overall credibility.

Similarity and Containment as Classifiers

To complete the experimental results, we tested how scheduling similarity and containment perform as binary classifiers: all design pairs with a similarity or containment above a chosen threshold are classified as positive (correct pairs), while all pairs with results below this same threshold are classified as negative. We counted correct (true positive (TP), true negative (TN)) and incorrect (false positive (FP), false negative (FN)) results for varying thresholds. For each classifier version, we then calculated following metrics:

- *Recall*, also known as true positive rate or sensitivity, is the fraction of correct pairs that were actually identified as positive: $Recall = TP/(TP + FN)$.
- *Precision*, also known as positive predictive value, is the fraction of correct pairs among all pairs identified as positive: $Precision = TP/(TP + FP)$.
- *Accuracy* is the proportion of correctly classified pairs among all pairs: $Accuracy = (TP + TN)/(TP + TN + FP + FN)$
- *Balanced accuracy* is used for imbalanced datasets, and takes into account the fact that one class (in our case negative) has more items than the other class (positive).

	Containment		
	threshold = 0.8	threshold = 0.9	threshold = 1
True Positive	100	100	100
False Positive	452	22	3
True Negative	9448	9878	9897
False Negative	0	0	0
Precision	**18.1%**	**82.0%**	**97.1%**
Recall	**100%**	**100%**	**100%**
Accuracy	95.5%	99.8%	100%
Balanced accuracy	**97.72%**	**99.89%**	**99.99%**

Table 3.2 – Using **scheduling containment** as classifier: experimental results.

For containment, the results given in Table 3.1 show that the minimum value for correct pairs is 100%. This means that 100% can directly be used as a threshold: only pairs with a containment of 100% are presumed correct. The results given in Table 3.2 confirm this choice: all correct pairs are classified as positive, resulting in 0 false negatives. The *recall* is thus at 100%. However, the small amount of false positives means that the precision, and overall accuracy, are slightly lower. A close examination of the testing dataset shows that all false positives are pairs where both designs originate from the same original C design, and were obfuscated with a low *obfuscation level* (25%). By forcing the use of a higher obfuscation level, the differences between obfuscated designs are heightened. In this manner, the previous results can be significantly improved, and the risk of false positives strongly decreased.

For similarity, experimental results show that the minimum value for correct pairs is 89.9%. By setting this value as threshold, we can ensure that the classifier has no false negatives, and a recall of 100%. However, this relatively low threshold means that some pairs are inaccurately classified as positive, resulting in lower precision and accuracy. On the other hand, experimental results show that the maximum similarity value for incorrect pairs is 99.6%. Setting this value as threshold can guarantee that there are no false positives, as shown in Table 3.3. However, this higher threshold results in almost all correct pairs being incorrectly classified as negative. While the overall accuracy is still high at 99% due to the imbalance of the dataset, the balanced accuracy, at 50.5%, shows that this clas-

	Similarity		
	threshold = 0.8	threshold = 0.89	threshold = 0.997
True Positive	100	100	1
False Positive	8	3	0
True Negative	9892	9897	9900
False Negative	0	0	99
Precision	**92.6%**	**97.1%**	**100%**
Recall	**100%**	**100%**	**1%**
Accuracy	99.9%	99.97%	99.0%
Balanced accuracy	**99.96%**	**99.99%**	**50.5%**

Table 3.3 – Using **scheduling similarity** as classifier: experimental results.

sifier does not perform well. Finding the best performing classifier, by choosing the best threshold value, is thus a trade-off between recall and sensitivity. In this case, the same conclusion can be reached as for containment: the results can be improved by adding a constraint on obfuscation level, which has to be at minimum 50%.

Similarity and containment perform well as classifiers for a scheduling-based birthmark, with both reaching a balanced accuracy of 99.99% when using well-chosen threshold values: 0.89 for similarity, and 1 for containment. In both cases, the precision is slightly lower (97.1%), due to a small amount of false positives. These false positives have been identified as designs with a low level of obfuscation.

Results with Higher Obfuscation Level

As explained above, the previous results can be improved by mandating a minimum obfuscation level of 50%. This minimizes the amount of false positives. By removing all tests with an obfuscation level below 50%, the maximum similarity value for incorrect pairs is 35.1%, and the maximum containment value for incorrect pairs is 91%, as shown in Table 3.4. By choosing thresholds higher than these two values, we can ensure that there are no false positives for our testing dataset.

	Similarity			Containment		
	Min	Max	Avg	Min	Max	Avg
All pairs (5625 pairs)	0.0%	99.7%	4.8%	8.7%	100%	51.1%
All correct pairs (75 pairs)	**91.3%**	**99.7%**	**97.7%**	**100%**	**100%**	**100%**
All wrong pairs (5550 pairs)	0.0%	35.1%	3.5%	8.7%	91.0%	50.4%
Wrong pairs but same original C code (1050 pairs)	0.9%	35.1%	10.1%	16.9%	86.4%	53.4%
Different original C code (4500 pairs)	0.0%	18.8%	2.0%	8.7%	91.0%	49.7%

Table 3.4 – Similarity and containment of scheduling: experimental results with **obfuscation level $\geq$ 50%**.

	Similarity threshold = 0.36	Containment threshold = 0.92
True Positive	75	75
False Positive	0	0
True Negative	5550	5550
False Negative	0	0
Precision	**100%**	**100%**
Recall	**100%**	**100%**
Accuracy	100%	100%
Balanced accuracy	**100%**	**100%**

Table 3.5 – Using **scheduling similarity and containment** as classifier: experimental results with **obfuscation level $\geq$ 50%**.

For both metrics, any threshold value chosen between the maximum value for incorrect pairs and the minimum value for correct pairs will lead to a classification with 100% accuracy. In both cases, we propose to use the lowest possible value. This means that in degraded conditions where the de-obfuscated design was altered and its similarity or containment to the obfuscated design is lower than expected, our classification is still highly likely to detect it. In other words, we aim above all to minimize the amount of false negatives. In Table 3.5, we show the classification

results for similarity and containment with a threshold of respectively **0.36** and **0.92**: for both metrics, the classification is 100% accurate.

By forcing the design house to use a higher obfuscation level (minimum 50%), previously found false positives can be avoided and a 100% classification accuracy can be reached for both similarity and containment.

Testing Resilience: Results with Added Noise

	Similarity			Containment		
	Min	Max	Avg	Min	Max	Avg
All correct pairs (75 pairs) - 5% noise	**46.4%**	**83.1%**	**65.7%**	**69.4%**	**94.4%**	**83.3%**
All correct pairs (75 pairs) - no noise	91.3%	99.7%	97.7%	100%	100%	100%
Wrong pairs (5550 pairs) - no noise	0.0%	35.1%	3.5%	8.7%	91.0%	50.4%

Table 3.6 – Similarity and containment of scheduling: experimental results with **5% bogus operators**.

To test resilience of scheduling as a birthmark, we add noise in the form of 5% random bogus operators to each de-obfuscated design. In Table 3.6, we give similarity and containment values for correct pairs, formed by an obfuscated design and its corresponding de-obfuscated design with and without added noise, as well as values for wrong pairs, without noise. The idea is that correct pairs, even with noise added, should have higher containment and similarity values than incorrect, no noise, pairs. As explained above, we restrict results to designs with an obfuscation level of at least 50%. Experimental results indicate that both containment and similarity of correct pairs are lower when noise is added to the de-obfuscated design. However, both are still on average significantly higher than for wrong pairs.

To test the resilience of similarity and containment as classifiers, as shown in Table 3.7, we reuse the threshold values obtained previously. For similarity,

	Similarity threshold = 0.36	Containment threshold = 0.75	Containment threshold = 0.92
True Positive	75	74	1
False Positive	0	302	0
True Negative	5550	5248	5550
False Negative	0	1	74
Precision	**100%**	**19.7%**	**100%**
Recall	**100%**	**98.7%**	**1.3%**
Accuracy	100%	94.6 %	98.7%
Balanced accuracy	**100%**	**96.61%**	**50.67%**

Table 3.7 – Using scheduling similarity and containment as classifier: experimental results with **5% bogus operators**.

with the previous threshold at **36%**, the classification of designs with noise in the form of 5% bogus operators results in perfect accuracy, with no false positive or false negative. However, for containment, since the values for correct pairs are now between 69.4% and 94.4%, with an average of 83.3%, reusing the previous threshold of 92% results in a large amount of false negatives: in fact, as shown in Table 3.7, only one of 75 positive pair is classified correctly. Lowering the threshold value can decrease the number of false negatives, but will also increase the number of false positives. The best overall accuracy is obtained by choosing a threshold value of **75%**.

> By adding noise in the form of 5% bogus operators, we tested resilience of the two birthmarks. While the results with containment as a classifier are slightly less good, overall both metrics still perform well. We can thus conclude that both containment and similarity are highly resilient. Future work to improve these results could include combining similarity and containment into a single metric.

3.6.2 Dataflow

To test whether our dataflow birthmarking approach is valid, we use the same analysis as for scheduling in the previous subsection. Each of the three metrics

	Containment		
	Min	Max	Average
All pairs (3136 pairs)	20.0%	100%	73.5%
All correct pairs (56 pairs)	**67.0%**	**100%**	**93.0%**
All wrong pairs (3080 pairs)	20.0%	100%	73.2%
Wrong pairs but same original C code (740 pairs)	64.0%	100%	88.7%
Different original C code (2340 pairs)	20.0%	100%	68.3%

Table 3.8 – **Containment** of dataflow: experimental results with obfuscation level $\geq 50\%$.

	Levenshtein		
	Min	Max	Average
All pairs (3136 pairs)	15.0%	100%	54.7%
All correct pairs (56 pairs)	**72.0%**	**100%**	**93.2%**
All wrong pairs (3080 pairs)	15.0%	100%	54.0%
Wrong pairs but same original C code (740 pairs)	62.0%	100%	88.9%
Different original C code (2340 pairs)	15.0%	80.0%	43.0%

Table 3.9 – **Similarity (Levenshtein)** of dataflow: experimental results with obfuscation level $\geq 50\%$.

(containment, Levensthein, subsequence similarity) is tested individually: for every correct and incorrect design pair, the three metrics are calculated. Then the minimum, maximum and average value for every metric and every class (correct pairs, incorrect pairs) are obtained. Finally, the accuracy of these metrics as binary classifiers is tested with different threshold values. Given the results obtained for scheduling, we opt to restrict the test set to designs with an obfuscation level of at least 50%.

In Table 3.8, Table 3.9 and Table 3.10, we present experimental results for the three metrics. In all three cases, the average metric value for correct pairs is significantly higher than the average for incorrect pairs. This indicates that the metrics could all potentially be used to distinguish between correct and incorrect pairs. However, contrary to the results obtained for scheduling, the overlap of values for incorrect and correct pairs is much more important. This means that there is a high risk of getting incorrect classifications, with either false positives or

	Subsequence		
	Min	Max	Average
All pairs (3136 pairs)	27.0%	100%	63.7%
All correct pairs (56 pairs)	**79.0%**	**100%**	**94.9%**
All wrong pairs (3080 pairs)	27.0%	100%	63.2%
Wrong pairs but same original C code (740 pairs)	70.0%	100%	91.8%
Different original C code (2340 pairs)	27.0%	88.0%	54.1%

Table 3.10 – **Similarity (subsequence)** of dataflow: experimental results with obfuscation level $\geq 50\%$.

false negatives, depending on the chosen threshold value.

	Containment threshold = 0.94	Similarity (Levensthein) threshold = 0.81	Similarity (subsequence) threshold = 0.82
True Positive	44	50	55
False Positive	867	573	828
True Negative	2213	2507	2252
False Negative	12	6	1
Precision	**4.8%**	**8.0%**	**6.2%**
Recall	**78.6%**	**89.3%**	**98.2%**
Accuracy	72%	81.5%	73.6%
Balanced accuracy	**75.21%**	**85.34%**	**85.67%**

Table 3.11 – Using dataflow containment and similarities as classifiers: experimental results.

In Table 3.11, containing the results of using each metric as a binary classifier, we display only the ideal threshold value for each metric, chosen after performing tests with a range of values with the goal of maximizing balanced accuracy.

The results given in Table 3.11 show that all three metrics perform reasonably well as classifiers. The balanced accuracy for containment is lowest, at 75.2%, while it is slightly higher for the two similarity metrics, at respectively 85.3% and 85.7%. In all three cases, the main issue is a high number of false positives, i.e. incorrect pairs that are falsely classified as correct. This is likely not due to the choice of metrics but rather to the abstractions we used to analyze the dataflow. In

particular, we chose to focus only on the shortest path between each input-output pair, which was the most straightforward to calculate practically. By restricting our study to shortest paths, it is likely that significant information contained in each dataflow was not included in the resulting paths. To improve these results, we thus suggest to extract other paths from the global dataflow to complete the information.

In conclusion, the results obtained here demonstrate that dataflow can potentially be used as a birthmark. However, more study is needed to find a better abstraction containing all the dataflow information.

3.6.3 Benchmark Results

To further validate the previous results, obtained for purely artificial designs generated with Crokus, we also tested our approach using scheduling as a metric on well-known benchmarks: `adpcm` and `AES`.

	Similarity			Containment		
	Min	Max	Avg	Min	Max	Avg
All pairs (225 pairs)	5.8%	98.8%	31.7%	19.7%	100%	57.8%
Correct pairs (15 pairs)	**89.5%**	**98.8%**	**95.9%**	**100%**	**100%**	**100%**
Wrong pairs (210 pairs)	5.8%	94.5%	27.1%	19.7%	99.5%	54.8%

Table 3.12 – Similarity and containment of scheduling: experimental results with obfuscation level $\geq 50\%$ for **adpcm**.

	Similarity			Containment		
	Min	Max	Avg	Min	Max	Avg
All pairs (225 pairs)	4.2%	91.6%	15.7%	20.4%	100%	51.9%
Correct pairs (15 pairs)	**85.5%**	**91.6%**	**89.3%**	**100%**	**100%**	**100%**
Wrong pairs (210 pairs)	4.2%	25.8%	10.4%	20.4%	79.5%	48.5%

Table 3.13 – Similarity and containment of scheduling: experimental results with obfuscation level $\geq 50\%$ for **AES**.

For each benchmark, we created several different obfuscated versions, by varying obfuscation level and branching degree. Given the results obtained on artificial benchmarks, we set the minimum required obfuscation level to 50%. Then we applied HLS, followed by de-obfuscation, to each new design. The goal of our tests was to determine whether our birthmarking approach is able to distinguish between different obfuscated versions of a same initial benchmark, and correctly identify pairs of obfuscated-deobfuscated RTL designs.

Experimental results presented in Table 3.12 and Table 3.13 show that for both similarity and containment, there is a clear distinction between correct and incorrect pairs for the two benchmarks: both values are close to 100% for all correct pairs, and on average much lower for incorrect pairs. This mirrors the results obtained for artificial benchmarks.

	Similarity threshold = 0.36	Containment threshold = 0.75
True Positive	15	15
False Positive	32	30
True Negative	178	180
False Negative	0	0
Precision	**31.9%**	**33.3%**
Recall	**100%**	**100%**
Accuracy	85.8%	86.7%
Balanced accuracy	**92.38%**	**92.86%**

Table 3.14 – Using scheduling similarity and containment as classifier: experimental results for **adpcm**.

We test if `adpcm` pairs can be correctly classified by reusing the threshold values found for Crokus-generated benchmarks with noise (cf. Table 3.7): 0.36 for similarity, and 0.75 for containment. The results given in Table 3.14 show that most pairs are classified correctly, with a small amount of false positives. The overall balanced accuracy is high for both metrics: 92.4% for similarity and 92.9% for containment.

The same approach is used for `AES`: the results in Table 3.15 show that in this case, all pairs are correctly classified for similarity. For containment, a few false

	Similarity threshold = 0.36	Containment threshold = 0.75
True Positive	15	15
False Positive	0	4
True Negative	210	206
False Negative	0	0
Precision	**100%**	**78.9%**
Recall	**100%**	**100%**
Accuracy	100%	98.2%
Balanced accuracy	**100%**	**99.05%**

Table 3.15 – Using scheduling similarity and containment as classifier: experimental results for **AES**.

positives can be noted, but the balanced accuracy is still at over 99%.

These test results show that our approach, validated on fictional designs, can also be applied to real, well-known benchmarks. Further study is necessary to find the ideal threshold values for classification with each metric.

3.7 Conclusion

In this chapter, we introduced a novel way to watermark IPs at the behavioral level, *without* requiring any modifications to existing HLS tools. We show how software birthmarking and existing hardware birthmarking concepts can be adapted to our context. We leverage the side effects of previously studied transient obfuscation techniques on the HLS input code. These side effects lead to different RTL designs that are hard to replicate without knowing the exact sequence and parameters of applied obfuscating transformations. We demonstrate that by extracting birthmarks based on scheduling and datapath information of the RTL designs, we are able to analyze the similarity and containment of designs. These metrics can then be used to prove a relationship between two designs, and thus identify stolen BIPs with an accuracy of over 96%, with an overall low resource overhead (below 6%).

Chapter 4

Transient Obfuscation against Hardware Trojan Insertion

With the ongoing globalization and outsourcing of hardware circuit manufacturing, concerns about an emerging security threat are being raised: Hardware Trojans (HT), which can take the form of malicious modifications or insertions in a circuit, have been garnering increasing attention. While research has mainly focused on HT insertion during manufacturing in untrusted foundries, the risks of attacks at earlier design stages have also been investigated. In particular, [Pil+18a] demonstrates how a malicious High-Level Synthesis tool can inject HTs into behavioral IPs. These Trojans, which can take diverse forms and affect different parts of the IP, are particularly difficult to detect, and pose a clear security threat.

Defense against HT insertion has already been studied by the scientific community at different design stages. Countermeasures can broadly be divided into 3 categories, as presented in [Bhu+14]: detection approaches, where the goal is to detect a HT *after* it was inserted, run-time monitoring, which aims at providing surveillance of running circuits after manufacturing with potentially active Trojans, and Design For Security (DFS), where IPs are designed specifically to facilitate Trojan detection or prevent Trojan insertion.

To the best of our knowledge, no work has yet presented any countermeasures against HTs inserted by a malicious HLS tool. In this chapter, we aim at providing a design methodology to prevent HT insertion during HLS with a high degree of

confidence and minimal impact on HLS Quality of Results (QoR) and overall IP performance. Obfuscation has successfully been used on hardware designs to thwart HT insertion during manufacturing and at lower design levels, as shown for example in [YDZ17]. Our goal is to verify whether transient obfuscation, which already provides protection against reverse-engineering and theft, can also be used to protect behavioral IPs against Trojan attacks during HLS. To this end, we explore two different approaches, for HT detection and for prevention of HT insertion.

4.1 Threat Model and Definitions

4.1.1 Hardware Trojans

The goal of HT attacks is to maliciously modify a circuit. This modification can have several purposes:

- leak sensitive information (e.g. a secret key) through a back-door or side-channel
- decrease performance of the circuit (denial of service)
- create unwanted behavior in the circuit (change of functionality)

A HT should not be detected during tests after its insertion, but it should be possible to activate it by a trigger during normal execution. HTs can be inserted by untrusted actors at several stages of the design process, or during manufacturing. The increasing decentralization of the different steps of IC life cycle has strongly increased the risk of HT insertion. For example, delegating manufacturing to external, untrusted foundries has exposed design companies to the threat of HT insertion by these foundries. The reliance on third party IPs and several complex EDA software tools has also added to the risk of HT attacks. Contrary to software Trojans, an inserted HT cannot be removed after manufacturing, which makes it difficult to counter its effects during normal operation.

Traditional post-manufacturing testing cannot be relied upon to detect HTs inserted during manufacturing or earlier design steps. Hardware verification techniques normally rely on using a golden model. In the case of third-party IPs,

golden models are often not available. Simulation or emulation, which is used to validate a design, can only prove the correct behavior of a design according to given functional specifications. It does not prove that there is no additional unwanted behavior due to an added HT.

To avoid detection, the area and power overhead caused by a Trojan insertion are usually very low. To ensure this low overhead, the HT can take advantage of unused FSM states and reuse existing logic.

A HT is usually composed of two parts: an activation mechanism, called *trigger*, and a *payload*, which modifies the circuit behaviour. HTs are often classified based on their trigger condition:

- Analog HTs: activated by on-chip sensors, for example by temperature or delay
- Digital HTs: activated by logic function
  - Combinational HTs: simultaneous occurrence of several conditions, for example a specific user input
  - Sequential HTs: sequence of state transitions, or when an internal counter reaches a certain value

As shown in the taxonomy presented in Figure 4.1, HTs can also be classified based on their insertion phase, abstraction level, function, location or physical characteristics.

4.1.2 Threat Model

There are several different threat models for HT attacks, as presented most recently in [Xue+20]. The most commonly studied attack scenarios are at the fabrication level, with an untrusted manufacturer, or at the design level, with an in-house attacker. However, CAD tools are also a possible attack vector. Maliciously modifying a CAD tool to insert HTs in design files is considered more difficult [Xue+20] than an attack during design or fabrication. On the other hand, it also offers the advantage of inserting a HT that is stealthier and harder to remove.

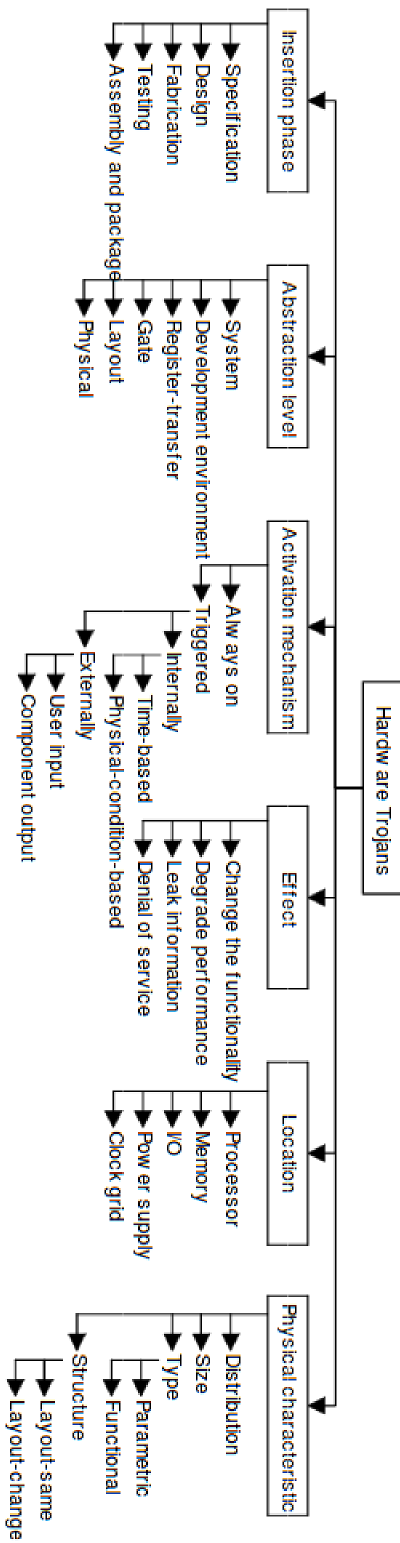


Figure 4.1 – HT Taxonomy - Trust-Hub.org

While it is unlikely that a CAD tool vendor himself would try to inject HTs, given the high risk of eventual detection and subsequent loss of customers, there is a real risk of outside attacks due to the complexity and modularity of modern CAD tools and their configuration. A motivated attacker could for example provide a modified version of one of the tool's modules, or change the tool's settings to encourage HT insertion [PBR16]. A theoretical explanation of how a generic CAD tool could automatically insert HTs is given in [PBR16]. A more practical implementation of these ideas in an HLS tool specifically is proposed in [Pil+18a], where 3 realistic examples of HT insertions are given (see Section 4.1.3).

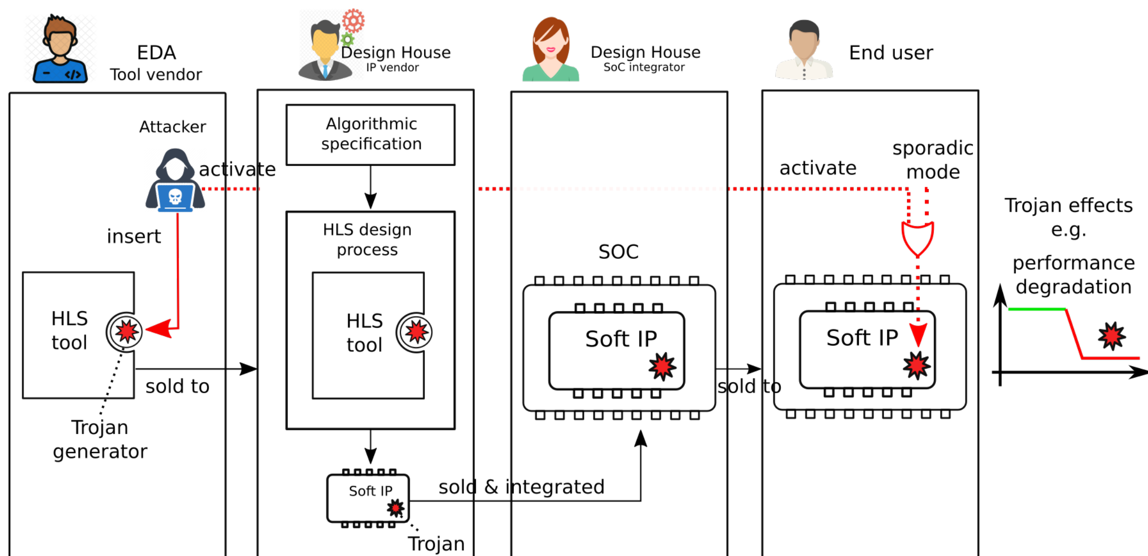


Figure 4.2 – HT insertion by HLS tool - threat model.

In this work, we focus on the risk of HT insertion by a maliciously modified HLS tool. We assume that this tool is able to insert a HT in a fully automated way in any conventional design, without the need for insider knowledge. The design house has access to the design files both before and after HLS, and uses an external, untrusted HLS tool. While the original, behavioral level design is well-known and verified, the design house has no way of verifying with complete certainty that no HT has been inserted during HLS.

The complete affected design flow, with all involved actors, is represented on Figure 4.2. The first step involves an HLS tool modified by an attacker at the EDA tool vendor: a HT generator is inserted into the tool. When this compromised

tool is then used by a design house to create an IP, a HT is automatically inserted during HLS into the resulting IP. This modified IP is sold to a SoC integrator. The resulting SoC, containing the compromised IP, is finally fabricated and sold to the end user. At this point, the HT can be activated either by the attacker or by a trigger mechanism (internal counter, random input, ...) and its payload is executed. All these steps take place without any of the actors being aware of the HT insertion.

Following the taxonomy given by Trust-Hub [Tru] (cf. Figure 4.1), we focus on a digital HT that is:

- Inserted at design time.
- At algorithmic or register-transfer level.
- Triggered externally by a specific input or triggered internally, for example by a counter.

4.1.3 Examples of HTs in BIPs

In this section, we present a few practical implementations of HTs inserted in BIPs. In [Pil+18a], the authors propose 3 different HT attacks added stealthily to a design by a maliciously modified HLS tool. The attacks are implemented as additional passes during HLS optimizations in an open-source HLS tool, Bambu [PF13]. These Trojans do not modify the functional behavior of the IP, and add a very low area overhead, which means that they are hard to detect by the designer:

- **Degradation attack**: The goal of this Trojan is to degrade the performance of the IP component. The authors propose to add bogus paths containing empty states (so-called *bubbles*) to the design's FSM. When the HT is activated, these new paths are executed. This leads to a clear degradation of performance since the number of cycles necessary to complete the computation increases.
- **Battery exhaustion attack**: This Trojan strongly increases power consumption, leading to faster battery usage. After scheduling and resource

binding steps, idle functional units are identified for each clock cycle. Bogus computations are then added to be performed by these functional units whenever the Trojan is activated. This can strongly increase the overall power consumption of the design.

- **Downgrade attack**: The goal of this Trojan is to reduce the security level of round-based cryptographic algorithms. In the presented example, when the HT is activated, the number of executed rounds for a SHA-256 algorithm is reduced. The security of the algorithm is thus severely compromised. This attack however requires that a malicious actor from the design house cooperates with the HLS developer to indicated precisely which modifications are necessary for the HT.

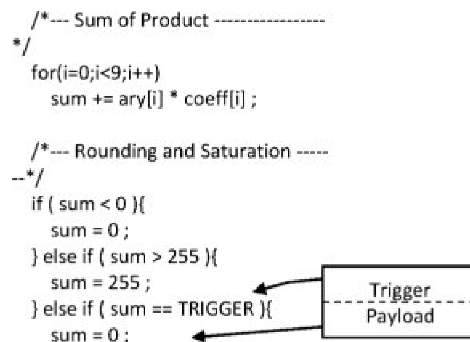


Figure 4.3 – Simple example of a HT inserted in a BIP. (from [VS16])

In [VS16], the authors introduce several HTs inserted in BIPs. An example of a simplified HT inserted in a FIR filter and triggered with an If/Else statement can be found on Figure 4.3. The authors give several practical examples using benchmarks such as AES, UART and sobel filter. For each type of HT, they provide several trigger mechanisms (combinational or sequential), as well as different payloads (with memory or without).

- **Change in functionality**: used on the sobel filter, this HT modifies it so that when activated, the filter output is overwritten with bogus values. It is activated either when certain input values are given (combinational trigger), or when a certain number of inputs has been given (sequential trigger based on a counter).

- **Information leakage**: applied on an AES circuit, this HT aims at leaking the secret key used in the encryption algorithm.

- **Denial of service**: this HT is inserted in a UART circuit, specifically in the transmitter block. It is activated once an internal counter value is reached and degrades performance by delaying the transmission, thus leading to data loss.

These HTs are all easy to implement in BIPs, requiring only a few lines of code. Furthermore, they only cause a small increase in area of the design and are thus hard to detect after synthesis. The authors extend their work in [VS17b], where they present S3CBench, a new benchmark suite of SystemC designs containing different variations of HTs.

4.2 Hardware Trojan Countermeasures: Related Work

Countermeasures are usually divided into three broad categories [Bhu+14] [LLZ16]: detection, run-time monitoring, and design-for-security (DFS). In this section, we focus particularly on techniques that can be used to thwart HTs at earlier design stages, before manufacturing.

4.2.1 Detection and Run-time Monitoring

At low abstraction levels, several pre-silicon HT detection approaches based on **formal methods** have been proposed. For example, in [Raj+16] and [RVK15], model checking is used to detect if there is any leakage of critical information or malicious modification of critical registers. These techniques rely on the IP vendor and user agreeing upon a set of security properties. A discussion of methods based on theorem proving and equivalence checking can be found in [Guo+15].

Instead of formal methods, post-silicon methods such as **functional testing** or **side-channel analysis** can also be used. Testing approaches usually focus on generating specific test patterns that can detect inserted HTs ([Ban+08] [Dup+15]),

whereas side-channel analysis rely on measuring parameters such as delay [CG13], power [Aar+10] or temperature to detect the side-effects induced by Trojans in infected ICs. Side-channel analysis approaches are most effective for large HTs, but are vulnerable to process noise and can fail to detect smaller HTs. They also rely on access to a Trojan-free golden IC for comparison. On the other hand, functional testing approaches are more efficient for small HTs, but the generation of useful test vectors is much more challenging [Cha+09].

While HT detection becomes harder the more abstract the design level is [PBR16], several methods for HT detection in RTL IPs have nevertheless been proposed in recent years. For example, in [PMP17], the authors show how a control flow subgraph matching algorithm can be used to detect various HTs in RTL designs, based on their structure. Their work is based on a library containing basic RTL HTs implementations as well as several modifications of the implementations to form more complex variants. In [Cho+20], the authors use **machine learning** to detect HTs in RTL designs. They extract all branching statements from the design and form a feature vector based on branching probability for each branch. Their assumption is that HT statements have a low chance of execution and thus a low branching probability since they are usually activated by rare conditions. This feature vector is then used to classify all branches using a decision tree classifier. The authors are thus able to accurately identify the majority of branches containing a HT trigger. Formal methods can also be used at high abstraction levels: in [VS16], the authors use **property checking** to detect HTs in BIPs. They identify potential HTs by using software profiling to identify unexecuted parts of the code. Then specific test vectors are created using formal verification methods, with the goal of forcing the testbench to execute the code identified earlier as potential HT.

Most detection methods are only designed to work for a small set of known HTs, for example those in benchmarks such as Trust-Hub [Tru]. This means that there is no proof that these methods are also effective at detecting unknown HTs. Furthermore, detection methods do not prove with full certainty that there are no Trojans present in a design [Bhu+14]. **Run-time monitoring** can be used as an additional countermeasure after fabrication. These approaches rely on monitoring the computations of an IC and potentially disable the chip if a HT is detected. Among the solutions explored are for example adding a software module outside

the processor to detect HT activation [BNS09], or adding reconfigurable logic to monitor security in real time [AB09].

4.2.2 Design for Security

To avoid the many challenges faced by Trojan detection methods and to provide countermeasures at earlier design stages, several design approaches have been created specifically for security against HTs. Design-for-Security (DFS) methods usually have one of two objectives: preventing HT insertion, or facilitating HT detection.

To **facilitate detection**, [Zho+14] and [STP11] propose to insert test points in the circuit which increase observability and controllability of nodes in the design. This in turn increases the probability of activating a Trojan during testing. Other methods aim at facilitating side-channel analysis, for example by minimizing background side-channel signals [ST11], or by adding structures [Raj+11] or sensors [Nar+12] that lead to higher sensitivity of the circuit to Trojans .

To **prevent insertion**, many techniques rely on removing any unused space on the design. In [BDA19], the authors propose to identify unused resources after routing. Dummy logic is then inserted to fill empty space in the design. Finally the design is encrypted using AES to prevent the attacker from replacing the non-functional part of the design with a Trojan. In [XT13], filler cells in the design are replaced with functional standard cells. Additionally, these cells are connected with a self-authenticating mechanism in order to prevent tampering. The other main approach relies on logic obfuscation: by hiding functionality and structure of designs, the difficulty of Trojan insertion is greatly increased for attackers. [YDZ17] gives an overview of several published obfuscation methods against Trojans, with a focus on HTs inserted at low design levels. In [LW14], the authors propose to further obfuscate the design by adding reconfigurable logic: this embedded logic is unknown to the actors of the supply chain and only the end user knows the right configuration to establish normal circuit functionality.

Several papers also propose methods that combine both objectives: facilitating detection *and* preventing insertion. In [CB11], a key-based obfuscation technique is proposed against Trojans triggered by rare internal events, based on two mod-

ifications of the state transition graph: extra states are added to exponentially increase the size of the reachable state space, and an obfuscated mode is added. The extra states prevent the attacker from finding rare events in the circuit to use as trigger for the Trojan. The Trojan is thus inserted in non-ideal locations and easier to detect by post-silicon testing methods. Additionally, some inserted Trojans will only be activated when the circuit is in obfuscated mode. During normal mode, they are thus benign and have no effect on the circuit. This technique is applied on gate-level netlists.

4.3 Proposed Countermeasures for Hardware Trojan Insertion during HLS

Our main goal is to prevent HT insertion by a malicious HLS tool. We propose to study how transient obfuscation can be used as countermeasure. This obfuscation method relies on adding a significant amount of bogus code to a C-level design. After HLS, the designer can remove the bogus code, either by directly manipulating the RTL code, or by using constant propagation during logic synthesis. In the end, while a small overhead due to different design choices made by the HLS tool remains, all of the bogus logic is removed. We postulate that this process of adding and then removing bogus code can be used in different ways to thwart HTs.

4.3.1 Payload: Removal by De-obfuscation

The first proposed method has the goal of removing or mitigating the effect of a HT inserted during HLS, at RTL, by using transient obfuscation. It relies on following idea, illustrated on Figure 4.4: during obfuscation, large amounts of bogus code are added to the design. If the HT payload is inserted in a bogus basic block, then it is highly likely that the payload will be automatically removed alongside the bogus logic during de-obfuscation. Increasing the proportion of bogus code directly improves the likelihood of HT payload removal. This idea presents similarities to the approach presented in [CB11], where some inserted Trojans become benign by only being activated in an obfuscated mode, and not in the normal mode.

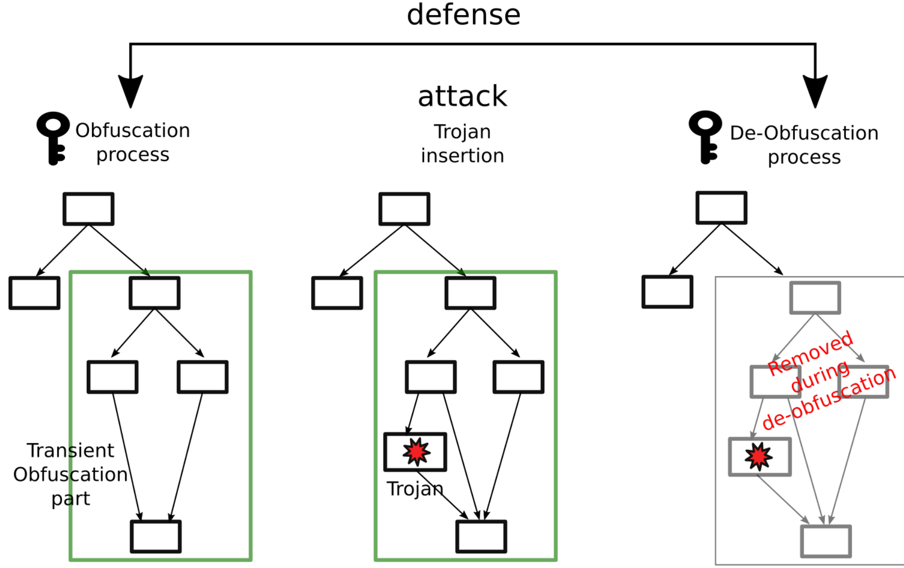


Figure 4.4 – HT removal by de-obfuscation flow.

Proposed Flow and Notations

Let C be the original design at behavioral level. By applying obfuscation using KaOTHIC's Bogus Basic Block Insertion (Section 2.4.1), we obtain an obfuscated design C_{obf}, containing both real and bogus basic blocks. Next HLS is applied. If the tool has been compromised or maliciously modified, a HT is inserted in the design, resulting in: RTL_{obf}^{HT}. Finally de-obfuscation is applied. If the HT was inserted in a bogus basic block, it gets deleted during de-obfuscation and we obtain RTL_{deobf}, a design that is functionally fully identical to the original design. If on the other hand the HT was inserted in a real basic block, the resulting design RTL_{deobf}^{HT} still contains the HT and our process has failed to protect the design.

To improve the likelihood of successful HT removal, it is thus crucial to increase the proportion of bogus code. While our process will never be able to guarantee a 100% success rate, we aim at reaching an acceptable, satisfyingly high HT removal rate.

Probability of HT removal

In this section, we aim at calculating a theoretical probability of HT removal. This probability can then be compared to the real removal rate obtained during an experimental phase.

We define following additional notations:

- b basic blocks in the original code
- b_{obf} basic blocks in the obfuscated code
- $b_{bogus} = b_{obf} - b$ bogus basic blocks in the obfuscated code
- Obfuscation level λ: ratio of effectively obfuscated basic blocks with respect to the total number of basic blocks in the design.
- Branching degree δ: number of bogus basic blocks per obfuscated basic block.
- $P(\lambda, \delta)$ probability of HT removal for a given obfuscation level and branching degree

For each basic block that is obfuscated, several new blocks are added: a conditional block based on an input key, as well as one or several bogus blocks, depending on the branching degree δ. This means that after obfuscation, each obfuscated block adds $1 + \delta$ blocks to the code. The total amount of bogus blocks can thus be calculated as follows:

$$b_{bogus} = b \times \lambda \times (1 + \delta) \tag{4.1}$$

We assume here that whenever a HT is inserted in a bogus code block, it will be removed during de-obfuscation. The probability P that a HT is removed can thus be defined as the ratio of bogus code against the total obfuscated code:

$$P = \frac{b_{bogus}}{b_{obf}} \tag{4.2}$$

Using these two equations, we can thus calculate:

$$\begin{aligned} P(\lambda, \delta) &= \frac{b_{bogus}}{b_{obf}} \\ &= \frac{b_{bogus}}{b + b_{bogus}} \\ &= \frac{b \times \lambda \times (1+\delta)}{b + b \times \lambda \times (1+\delta)} \\ &= \frac{\lambda(1+\delta)}{1+\lambda(1+\delta)} \\ &= 1 - \frac{1}{1+\lambda(1+\delta)} \end{aligned} \tag{4.3}$$

This resulting formula confirms the intuition that to maximize probability of HT removal, branching degree and/or obfuscation level should be as high as possible. Per definition, the maximum possible obfuscation level is 100%. Previous experimental results (see Section 2.7.2) have demonstrated that this maximal value can always be used without incurring any significant design overhead. The previous formula can thus be simplified by replacing λ with 1:

$$P(\delta) = 1 - \frac{1}{\delta + 2} \tag{4.4}$$

With a branching degree of 8, a probability of removal of 90% can theoretically be reached. To reach a 95% chance of removal, a branching degree of 18 is necessary.

Contrary to obfuscation level, increasing the branching degree can quickly lead to a strong design overhead. This means there is a trade-off between cost and security.

4.3.2 Combinational Trigger: Detection during De-obfuscation

In the previous section, we showed how HTs can be removed by our de-obfuscation process. However, during our experiments, we have also noticed that in some cases, the inserted HT *prevents* normal de-obfuscation. In particular, if the circuit inputs

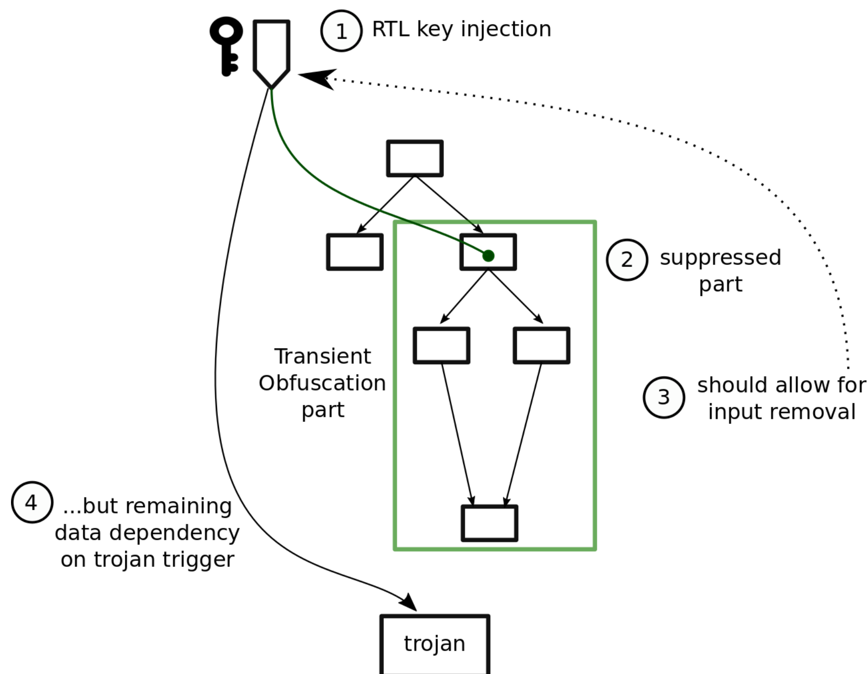


Figure 4.5 – HT detection during de-obfuscation.

that are used as obfuscation keys are also used as part of the HT trigger, then our automated de-obfuscation tool cannot fully remove the obfuscation keys.

For this approach, which is intended to be complementary to the first method we presented, we thus focus on HTs with combinational triggers based on design inputs: the Trojan is activated when the correct input combination is entered. The approach is illustrated on Figure 4.5: during key injection while de-obfuscating the RTL (1), the logic added during obfuscation is removed (2). Normally, through constant propagation and dead code elimination, this should enable our tool to remove the obfuscation key input (3). However, if this key input is also used as part of a combinational Trojan trigger (4), then the remaining data dependency prevents the input key removal. It should be noted that the design is normally not modified between obfuscation and de-obfuscation. This means that the only case where the obfuscation keys are also used in other parts of design is when unwanted behavior was added during HLS. By slightly modifying our de-obfuscation tool to raise an error whenever this scenario occurs, we are thus able to detect that unwanted logic was inserted. By knowing which obfuscation key is involved, we

are then able to follow the trigger and detect the full HT.

```
-- obf, no trojan
tmp_2_fu_233_p2 <= "1" when (key2 = ap_const_lv32_2A3) else "0";
tmp_3_fu_269_p2 <= "1" when (key3 = ap_const_lv32_2C8) else "0";
tmp_4_fu_275_p2 <= "1" when (key4 = ap_const_lv32_1C9) else "0";
tmp_5_fu_281_p2 <= "1" when (key1 = ap_const_lv32_8D) else "0";
tmp_6_fu_287_p2 <= "1" when (key5 = ap_const_lv32_126) else "0";
```

Figure 4.6 – No Trojan, these lines are removed during de-obfuscation.

```
-- obf + trojan
tmp_5_fu_491_p2 <= "1" when (key4 = ap_const_lv32_1C9) else "0";
tmp_6_fu_496_p2 <= "1" when (key1 = ap_const_lv32_8D) else "0";
tmp_2_fu_289_p2 <= "1" when (key2 = ap_const_lv32_2A3) else "0";
tmp_10_fu_501_p2 <= "1" when (key5 = ap_const_lv32_126) else "0";
tmp_8_fu_486_p2 <= "1" when (key3 = ap_const_lv32_2C8) else "0";

tmp_fu_273_p4 <= input1(31 downto 1);
tmp_7_fu_325_p1 <= key1(31 - 1 downto 0);
```

Figure 4.7 – Key1 and Input1 are used as triggers for the Trojan.

Figure 4.6 and Figure 4.7 show two VHDL code snippets as an example. On the first figure, the design has 5 obfuscation keys (*key1* - *key5*), which are design inputs. These inputs are each compared to a constant and the result of the comparison is sent to a functional unit. This result is then used as input to the *obfuscation multiplexer*, which decides if the correct or bogus code block is executed. During de-obfuscation, since the obfuscation key values are known, the result of the comparisons are also fixed and the whole logic on Figure 4.6 can be removed. On the other hand, on Figure 4.7, *key1*, as well as another circuit input *input1*, are used as inputs for a combinational Trojan trigger. In this case, *key1* has an additional data dependency and is thus not removed by our de-obfuscation tool. The tool raises an error, flagging the location of this particular line. The unexpected behavior can then be closely examined by the design house, leading to detection of the HT.

> By applying a light modification to our de-obfuscation tool, we are able to raise a flag whenever a Trojan uses obfuscation keys as trigger. The tool points to the exact variable that is involved, thus allowing a design house to easily retrace the problem, detect the Trojan trigger and remove it.

It should be noted that in this section, we focus only on the case of a combinational HT trigger reusing obfuscation keys as inputs. However, several other types of HT triggers or payloads interwoven with the obfuscation logic could also prevent de-obfuscation. Our method can easily be extended by studying other examples of HTs.

4.4 Experimental Setup and Preliminary Results

For our experiments, we do not have access to a malicious HLS tool. There are two possibilities to imitate insertion of a HT by an HLS tool: simulating a HT inserted by HLS front-end, by using a modified source-to-source compiler for high level source code, or adding the HT after HLS, in the RTL description (e.g. using VHDL or Verilog source-to-source transformations). Because it is simpler to manipulate high level source code, and we already have access to several in-house tools for C code manipulation, we chose to focus on a tool for HT insertion in C code. In particular, we implemented a tool that inserts a HT based on the Collatz conjecture.

4.4.1 "Collatz" Hardware Trojan

We created a HT based on following requirements:

- The HT is inserted by a source-to-source compiler, before HLS.
- The trigger is combinational and uses a combination of circuit inputs.
- The payload induces a performance degradation: the circuit performs unnecessary calculations, leading to higher latency.
- The HT is not removed by the HLS tool (or logic synthesis tools) during optimizations and dead code elimination.
- The HT does not modify circuit functionality and thus cannot be detected during functional testing.

- For experimental purposes, we do not focus on stealth of the HT. However, it should require a minimal amount of code.

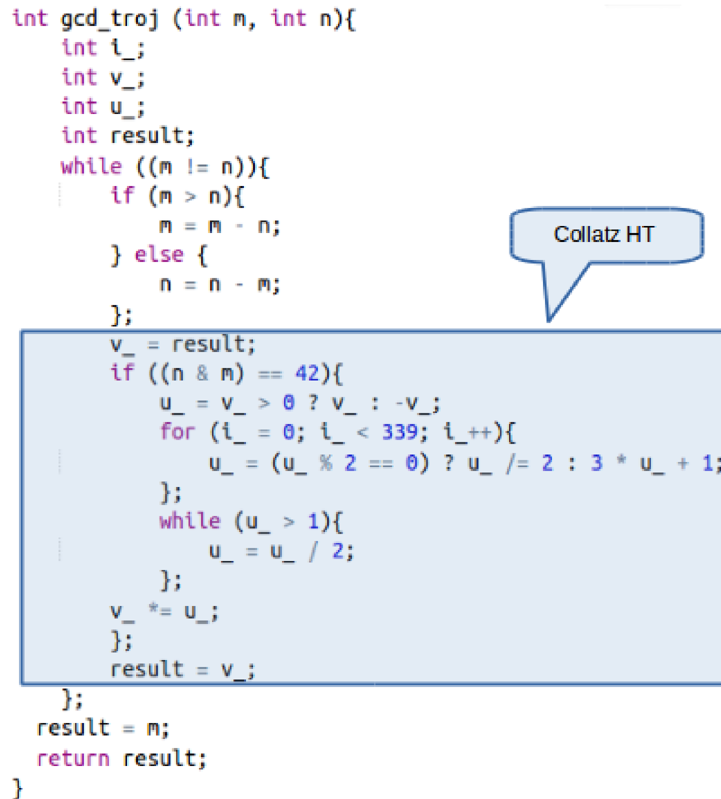


Figure 4.8 – Code example of a Collatz HT inserted in a greatest common divisor calculation.

On Figure 4.8, we give a code sample where a “Collatz” HT was inserted into a function calculating the greatest common divisor of two integers m and n. This HT has a combinational trigger based on an If/Else statement, activated when a combination of two inputs has a certain value. In the case of the example on Figure 4.8, the Trojan is activated only when $m \,\&\, n = 42$. It does not modify the circuit’s functionality, but slows it down by performing an unnecessary calculation when activated. The payload is embedded at a random place in the code. It takes one of the original codes’ variables (*result* in the example Figure 4.8), performs

some calculation involving this variable and then returns it. The result of this calculation is used in the original circuit to ensure that the compiler does not remove the payload as dead code. The result of the calculation is exactly the same as the original value, in order to ensure correct functionality of the circuit. On the other hand the calculation is complex enough to ensure that a dataflow analysis by the compiler is not able to simplify and remove it. In our case, the calculation is based on the Collatz conjecture (also known as Syracuse problem). The result is known at design time, but a compiler will not be able to find and simplify it.

> This HT insertion was fully automated and added as a pass to Crokus (Section 3.5.4). A test performed with the HT inserted at random in AES benchmark shows no increase of C simulation time as long as the HT is not activated. When the Trojan is activated with the trigger inputs, the simulation time increases by 334%. After HLS, when simulating the VHDL design, the number of cycles also increases by 219% only when the HT is activated. Tests with other benchmarks yielded similar results.

4.4.2 HT Removal

Experimental evaluation of our approach focuses on following problem: proving that the previously calculated probability of removal is close to the real removal rate.

Our goal is thus to experimentally calculate, for different obfuscation level and branching degree values, the HT removal rate. We start by generating several C codes with Crokus: `test_X`. For each original C code, we apply obfuscation with varying obfuscation level and branching degree using KaOTHIC: `test_X(`λ`,`δ`)`. Next, for each obfuscated C code, we perform several instances of Trojan insertion with Crokus: `test_X(`λ`,`δ`,troj_Y)`. To avoid having to perform HLS, de-obfuscation and checking for the Trojan at RTL, which are time consuming and more complex to set up, we use a shortcut method that yields the same results: for each C code, we hardcode the correct obfuscation key values as inputs to the function. This is functionally equivalent to performing de-obfuscation by key injection

on the RTL design. Then we use Clang (a compiler front-end for LLVM: [Cla]) and opt (an optimizer for LLVM: [opt]) to parse and optimize the C code. In particular, by using *interprocedural sparse conditional constant propagation* (`-ipsccp` option), the key constants are propagated throughout the obfuscated function and any dead code is removed. This operation is equivalent to what the logic synthesis tool would perform at lower level after de-obfuscation. Finally, we use a regular expression to check if the Trojan is still present in the IR of the optimized code. By using this method, we are able to verify for each code if the Trojan would have been removed during de-obfuscation or not.

test case	obf. level	branch. degree	test A	test B	test C	test D	**average removal rate**	**theoretical removal rate**
1	25%	1	0.11	0.0	0.20	0.10	**0.14**	**0.33**
2	25%	3	0.20	0.0	0.40	0.60	**0.40**	**0.50**
3	25%	5	0.15	0.0	0.05	0.45	**0.22**	**0.60**
4	25%	7	0.50	0.0	0.40	0.60	**0.50**	**0.67**
5	25%	9	0.35	0.0	0.60	0.74	**0.56**	**0.71**
6	50%	1	0.20	0.20	0.50	0.55	**0.42**	**0.50**
7	50%	3	0.50	0.40	0.60	0.80	**0.63**	**0.67**
8	50%	5	0.75	0.55	0.60	0.60	**0.65**	**0.75**
9	50%	7	0.75	0.68	0.75	0.80	**0.77**	**0.80**
10	50%	9	0.85	0.50	0.80	0.70	**0.78**	**0.83**
11	75%	1	0.65	0.30	0.45	0.55	**0.55**	**0.60**
12	75%	3	0.65	0.70	0.70	0.70	**0.68**	**0.75**
13	75%	5	0.50	0.75	0.85	0.65	**0.67**	**0.82**
14	75%	7	0.65	0.55	0.70	0.90	**0.75**	**0.86**
15	75%	9	0.90	0.60	0.80	0.75	**0.82**	**0.88**
16	100%	1	0.70	0.47	0.55	0.70	**0.65**	**0.67**
17	100%	3	0.80	0.50	0.70	0.80	**0.77**	**0.80**
18	100%	5	0.90	0.85	0.85	0.80	**0.85**	**0.86**
19	100%	7	0.90	0.70	0.95	0.85	**0.90**	**0.89**
20	100%	9	0.90	0.80	0.90	0.85	**0.88**	**0.91**

Table 4.1 – HT Removal Rate: Experimental Results.

We do a series of tests with 4 automatically generated C codes of varying sizes and properties: `test A` -> `test D` on Table 4.1. For each of these initial C codes,

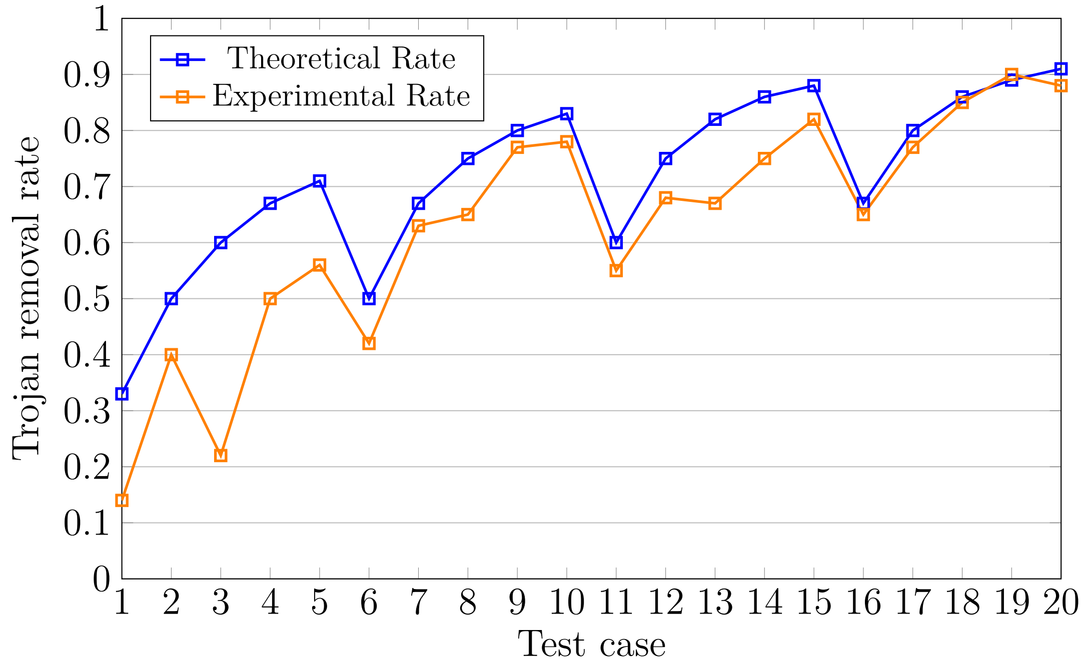


Figure 4.9 – Theoretical and experimental HT removal rate.

we obfuscate with 4 different values of obfuscation level and 5 different values of branching degree, resulting in 20 obfuscated designs per original code and thus 20 test cases. This means tests are done with a total of 80 different obfuscated designs. Next, for each of these 80 obfuscated designs, we try 20 instances of HT insertion. This results in a total of 1600 obfuscated designs with Trojan. Then we verify for each of these designs if the Trojan would be removed during de-obfuscation. For each original design and each test case, we can thus give a HT removal rate calculated over 20 Trojan insertions. These values are given in Table 4.1 in the four columns `test A` -> `test D`. In this table, we also indicate the average removal rate for each test case, i.e. for each combination of obfuscation level and branching degree. For each test case, this average is calculated over 80 (4x20) individual tests. Finally, using the formula established in Equation (4.3), we give the theoretical removal rate for each test case.

To evaluate the correctness of the removal probability (Equation (4.3)), we calculated the correlation between this theoretical rate and the experimental rate obtained during experiments.

The resulting correlation coefficient, taken over a range of 20 test cases with varying obfuscation levels and branching degrees, is **0.93**. This means that there is a strong correlation between the theoretical and real rate. The two rates are represented graphically on Figure 4.9. This same correlation is also visible when examining the graph. It is noticeable that for most test cases, experimental results are slightly less good than the theoretical rate, while the overall tendency remains the same. This strong correlation is very encouraging and indicates that our method is highly likely to successfully remove hardware Trojans of the form studied here.

4.4.3 Discussion

The positive results presented in the previous section demonstrated that our method is a valid technique for removing Trojans inserted during HLS. The close correlation between theoretical and experimental Trojan removal rate validates our approach. However, these results are only preliminary and several further experiments could be designed. First, the tests were only done on synthetic benchmarks generated with a random code generation tool. Tests with real benchmarks should also be performed. Moreover, tests with a wider range of obfuscation levels and branching degrees could be used to further verify the removal rate. Finally, tests so far were all done with one Trojan: “Collatz” HT. For more realistic results, further tests with other behavioral level Trojans should also be done.

The basic approach presented in this work can be further improved: so far, we have assumed that the HT payload is inserted in a block chosen at random. However, many automated HT insertions rely on an algorithm to automatically choose the best point of insertion. For example, in the degradation attack presented in [Pil+18a], bubbles are inserted in the FSM to slow down computation. An algorithm which calculates a cost function for each basic block (BB) is used to determine between which states the bubbles should be inserted. By adding fake BBs and/or fake transitions between existing BBs, it is possible to skew the computation of this cost function, thus resulting in an increased likelihood of the algorithm

inserting the bubbles between bogus states. This in turn means an increased likelihood of HT removal during de-obfuscation. Another potential approach could be based on exploiting the work in [ST13]: this paper focuses on soft IPs in HDL and their vulnerability to HT insertion, by identifying potential HT insertion locations. The authors note that Trojans are usually inserted in statements that are rarely executed or on signals that are hard to observe. In particular, highly nested statements and low observability signals are a likely target for HT insertion. By adding bogus code that fulfills these properties, Trojan insertion algorithms could be lured to insert the HTs there instead of on real code. Overall, this more precise approach requires detailed study of automated HT insertion algorithms, and has no guarantee of satisfying results when applied on yet unknown HTs.

4.5 Conclusion

In this chapter, we explored how transient obfuscation can be used for protection against HLS-based hardware Trojan attacks. We designed two methods which can be used in a complimentary way. The first approach focuses on preventing HT insertion: by adding large amounts of bogus code to a design during obfuscation, the chance that a Trojan is inserted in a bogus code block is increased. This code, including the HT, is then removed during de-obfuscation. A preliminary experimental study showed that we are able to successfully remove HTs in this way. We established a theoretical study the HT removal rate, and confirmed it with high certainty during experimental study. This chance of removal can be increased by using high obfuscation level and branching degree. Our second approach can be used to detect HTs that have not been removed. It focuses in particular on HTs with a combinational trigger that reuses design inputs. These types of trigger can lead to abnormal behavior during de-obfuscation due to unexpected remaining data dependencies. By modifying our de-obfuscation tool, we are able to successfully pinpoint this behavior and raise an alert leading to detection of the HT.

These two approaches with promising results show that transient obfuscation is a good candidate for protection against hardware Trojans. Further work should study these approaches more closely, by exploring other types of Trojans and providing more in-depth experimental studies.

Conclusion

Summary of contributions

In **Chapter 2**, we presented the case for a HLS-as-a-Service scenario, and explained why widespread adoption is slowed down by security concerns: BIP theft and reverse-engineering. We introduced *transient obfuscation*, a novel concept for BIP protection. This idea relies on two separate steps, to be performed respectively before and after HLS. The first step, key-based obfuscation, hides the structure of the IP and locks it to prevent illegal reuse. During the second step, by using the correct obfuscation keys, the IP is de-obfuscated. Our method is considered *transient* because of two factors: after de-obfuscation, original functionality is restored, and the design overhead is negligible. We studied several different obfuscation techniques that can become transient, and implemented them in a new tool, KaOTHIC. During experimental validation, we analyzed the different techniques and their impact on several known benchmarks through HLS and logic synthesis for FPGA and ASIC using industrial tools. The results show that some techniques are better suited than others for transient obfuscation, with *bogus code insertion* techniques being ideal candidates. Using transient obfuscation, we are thus able to present a secure, low overhead design flow for cloud-based HLS.

In **Chapter 3**, we extended the scope of the previous chapter by focusing on the problem of stolen BIP identification. We exploited a side-effect of transient obfuscation: after de-obfuscation, while original design functionality is restored, structural differences in the micro-architecture of the design remain, due to the effects of the obfuscating transformations on HLS results. We propose a flow where these differences are used as *birthmarks* to identify stolen designs. In particular, we

created two metrics that can be extracted from any design at RTL to characterize their dataflow and scheduling. During a test campaign, we successfully used these metrics to identify stolen designs with an accuracy of over 96%. The extensive tests were first done on synthetic benchmarks that we generated for the purpose of this work, and then validated on well-known benchmarks.

In **Chapter 4** finally, we focused on another security threat introduced by untrusted behavioral synthesis tools: the risk of hardware Trojan insertion by a compromised HLS tool. We proposed two methods that use transient obfuscation against this threat, with different purposes. The first method relies on the de-obfuscation step to automatically *remove* hardware Trojans inserted in bogus code blocks. We provided a theoretical approach for estimating the chance of removal of a Trojan depending on the obfuscation parameters. Then, using an in-house tool to insert a Trojan in C level designs, we validated this approach with a test campaign. Experimental results showed that the real removal rate is closely correlated to the theoretical one. These tests demonstrated that, with a high enough level of obfuscation and branching degree, there is a high likelihood of Trojan removal. The second method we presented took advantage of the de-obfuscation step to *detect* inserted hardware Trojans. By modifying our de-obfuscation tool, we were able to detect Trojans with a specific type of trigger: combinational trigger that uses design inputs. This method could likely be extended to other types of triggers and payloads.

Perspectives

We identified several perspectives to extend the different contributions in this thesis.

First of all, the experimental study of transient obfuscation techniques could be extended in several ways. Closer attention could for example be brought to analyzing the resilience of our approach against manual or automated attacks. Several security metrics proposed in other works on obfuscation could thus be adapted and reused for our research. Moreover, de-obfuscation has only been automated for code generated by Xilinx VivadoHLS so far. By extending our method to code generated by other HLS tools such as Bambu, Gaut or Catapult,

the experimental study could be broadened, leading to more extensive results.

Another possible extension of the work on transient obfuscation could focus on exploring more obfuscation techniques, either by reusing and adapting known software and hardware obfuscation methods, or by creating new techniques. This extension could also aim at further improving the transience of existing techniques, for example by better taking into account the effects of control flow transformations on HLS quality of results.

Finally, the key locking mechanism could be improved: in this work, we only paid reduced attention to issues of stealth and resilience. By using a more complex key mechanism, or by focusing on how to better integrate and hide it in the existing design, the risk of detection and removal could be reduced.

Concerning our birthmarking approach, future work needs to focus on improving the two presented birthmarking metrics, or on providing new metrics that would characterize design architectures with more precision. This would allow higher accuracy during birthmark extraction, and also lead to better resistance against attacks. Research efforts could for example concentrate on improving the datapath metric, which in this work has given slightly less satisfying results and does not represent the full datapath information that is available.

During our work on birthmarking, we also tried a machine learning (ML) based approach for birthmark extraction and verification, which did not yield sufficient results and was much more complicated to put into place that initially expected. This approach failed mainly due to one issue: a lack of good RTL representations that can be used as input for a machine learning method. While there have been some recent studies ([Zha+19]) of metrics that can be extracted from RTL design and used as input features for ML algorithms, there appears to be still a lot of research missing on this topic. In particular, graph-based neural networks, which have shown promising results for ML on software codes ([Li+19b]), appear to be a promising research direction for ML on RTL designs.

Finally, while our research on using transient obfuscation against hardware Trojans gave some promising preliminary results, a more thorough study of the subject is necessary: an extension to other types of Trojans, a better study of automated Trojan insertion algorithms, or a focus on attack surface expansion to confuse these algorithms, are all topics that require more work.

List of published contributions

[Bad+19] H. Badier, J.-C. Le Lann, P. Coussy, and G. Gogniat. “Transient key-based obfuscation for HLS in an untrusted cloud environment”. In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 1118–1123 (cit. on p. 40).

[Bad+21a] H. Badier, J.-C. Le Lann, P. Coussy, and G. Gogniat. “Protecting Behavioral IPs during Design Time: Key-Based Obfuscation Techniques for HLS in the Cloud”. In: *Behavioral Synthesis for Hardware Security*. Springer, 2021 (cit. on p. 40).

[Bad+21b] H. Badier, C. Pilato, J.-C. Le Lann, P. Coussy, and G. Gogniat. “Opportunistic IP Birthmarking using Side Effects of Code Transformations on High-Level Synthesis”. In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2021 (cit. on p. 76).

[Ker+19] F. Kermarrec, S. Bourdeauducq, H. Badier, and J.-C. Le Lann. “LiteX: an open-source SoC builder and library based on Migen Python DSL”. In: *OSDA’2019 Open Source Design Automation, colocated with DATE’19*. 2019 (cit. on p. 12).

[LBK20] J.-C. Le Lann, H. Badier, and F. Kermarrec. “Towards a Hardware DSL Ecosystem: RubyRTL and Friends”. In: *OSDA’2020 Open Source Design Automation, colocated with DATE’20*. 2020 (cit. on p. 12).

Bibliography

[Aar+10] J. Aarestad, D. Acharyya, R. Rad, and J. Plusquellic. "Detecting Trojans Through Leakage Current Analysis Using Multiple Supply Pad IDDQS". In: *IEEE Transactions on information forensics and security* 5.4 (2010), pp. 893–904 (cit. on p. 119).

[AB09] M. Abramovici and P. Bradley. "Integrated circuit security: new threats and solutions". In: *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*. 2009, pp. 1–3 (cit. on p. 120).

[Ade08] S. Adee. "The hunt for the kill switch". In: *IEEE Spectrum* 45.5 (2008), pp. 34–39 (cit. on p. 3).

[Akh+15] A. Akhunzada, M. Sookhak, N. B. Anuar, A. Gani, E. Ahmed, M. Shiraz, S. Furnell, A. Hayat, and M. K. Khan. "Man-At-The-End attacks: Analysis, taxonomy, human aspects, motivation and future directions". In: *Journal of Network and Computer Applications* 48 (2015), pp. 44–57 (cit. on p. 31).

[AKP07] Y. Alkabani, F. Koushanfar, and M. Potkonjak. "Remote activation of ICs for piracy prevention and digital right management". In: *2007 IEEE/ACM International Conference on Computer-Aided Design*. IEEE. 2007, pp. 674–677 (cit. on p. 23).

[Arb02] G. Arboit. "A method for watermarking java programs via opaque predicates". In: *The Fifth International Conference on Electronic Commerce Research (ICECR-5)*. 2002, pp. 102–110 (cit. on p. 80).

[ATA03] A. T. Abdel-Hamid, S. Tahar, and E. M. Aboulhamid. "IP watermarking techniques: Survey and comparison". In: *The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, 2003. Proceedings.* IEEE. 2003, pp. 60–65 (cit. on pp. 20, 77).

[ATA04] A. T. Abdel-Hamid, S. Tahar, and E. M. Aboulhamid. "A survey on IP watermarking techniques". In: *Design Automation for Embedded Systems* 9.3 (2004), pp. 211–227 (cit. on pp. 25, 27).

[Ban+08] M. Banga, M. Chandrasekar, L. Fang, and M. S. Hsiao. "Guided test generation for isolation and detection of embedded Trojans in ICs". In: *Proceedings of the 18th ACM Great Lakes symposium on VLSI*. 2008, pp. 363–366 (cit. on p. 118).

[Bar+12] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. "On the (im) possibility of obfuscating programs". In: *Journal of the ACM (JACM)* 59.2 (2012), pp. 1–48 (cit. on p. 30).

[Bas+19] K. Basu, S. M. Saeed, C. Pilato, M. Ashraf, M. T. Nabeel, K. Chakrabarty, and R. Karri. "CAD-Base: an attack vector into the electronics supply chain". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 24.4 (2019), pp. 1–30 (cit. on p. 3).

[BDA19] N. K. Brar, A. Dhindsa, and S. Agrawal. "Impact of Dummy Logic Insertion on Xilinx Family for Hardware Trojan Prevention". In: *International Conference on Advanced Informatics for Computing Research*. Springer. 2019, pp. 64–74 (cit. on p. 120).

[Bhu+14] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan. "Hardware Trojan attacks: threat analysis and countermeasures". In: *Proceedings of the IEEE* 102.8 (2014), pp. 1229–1247 (cit. on pp. 17, 111, 118, 119).

[BNS09] G. Bloom, B. Narahari, and R. Simha. "OS support for detecting Trojan circuit attacks". In: *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*. IEEE. 2009, pp. 100–103 (cit. on p. 120).

[Bro97] A. Z. Broder. "On the resemblance and containment of documents". In: *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*. IEEE. 1997, pp. 21–29 (cit. on p. 90).

[BS05] A. Balakrishnan and C. Schulze. "Code obfuscation literature survey". In: *CS701 Construction of compilers* 19 (2005) (cit. on p. 31).

[BS19] A. Babaei and G. Schiele. "Physical unclonable functions in the internet of things: State of the art and open challenges". In: *Sensors* 19.14 (2019), p. 3208 (cit. on p. 19).

[BY07] M. Brzozowski and V. N. Yarmolik. "Obfuscation as intellectual rights protection in VHDL language". In: *6th International Conference on Computer Information Systems and Industrial Management Applications (CISIM'07)*. IEEE. 2007, pp. 337–340 (cit. on p. 26).

[Cal+04] A. E. Caldwell, H.-J. Choi, A. B. Kahng, S. Mantik, M. Potkonjak, G. Qu, and J. L. Wong. "Effective iterative techniques for fingerprinting design IP". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23.2 (2004), pp. 208–215 (cit. on p. 20).

[Cas+07] E. Castillo, U. Meyer-Baese, A. Garcia, L. Parrilla, and A. Lloris. "IPP@ HDL: efficient intellectual property protection scheme for IP cores". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 15.5 (2007), pp. 578–591 (cit. on p. 25).

[CB09] R. S. Chakraborty and S. Bhunia. "HARPOON: An obfuscation-based SoC design methodology for hardware protection". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.10 (2009), pp. 1493–1502 (cit. on p. 23).

[CB10] R. S. Chakraborty and S. Bhunia. "RTL hardware IP protection using key-based control and data flow obfuscation". In: *2010 23rd International Conference on VLSI Design*. IEEE. 2010, pp. 405–410 (cit. on pp. 25, 26).

[CB11] R. S. Chakraborty and S. Bhunia. "Security against hardware Trojan attacks using key-based design obfuscation". In: *Journal of Electronic Testing* 27.6 (2011), pp. 767–785 (cit. on pp. 120, 121).

[CBC07] L.-w. Chow, J. P. Baukus, and W. M. Clark Jr. *Integrated circuits protected against reverse engineering and method for fabricating the same using an apparent metal contact line terminating on field oxide.* US Patent 7,294,935. 2007 (cit. on p. 21).

[CBH16] B. Colombier, L. Bossuet, and D. Hély. "From secured logic to IP protection". In: *Microprocessors and Microsystems* 47 (2016), pp. 44–54 (cit. on p. 22).

[CD00] R. Chapman and T. S. Durrani. "IP protection of DSP algorithms for system on chip implementation". In: *IEEE Transactions on signal processing* 48.3 (2000), pp. 854–861 (cit. on p. 27).

[Cec+14] M. Ceccato, M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. "A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques". In: *Empirical Software Engineering* 19.4 (2014), pp. 1040–1074 (cit. on p. 36).

[CG13] B. Cha and S. K. Gupta. "Trojan detection via delay measurements: A new approach to select paths and vectors to maximize effectiveness and minimize cost". In: *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2013, pp. 1265–1270 (cit. on p. 119).

[CGR93] V. Chaiyakul, D. D. Gajski, and L. Ramachandran. "High-Level Transformations for Minimizing Syntactic Variances". In: *30th ACM/IEEE Design Automation Conference*. IEEE. 1993, pp. 413–418 (cit. on pp. 67, 78).

[Cha+09] R. S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia. "MERO: A statistical approach for hardware Trojan detection". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2009, pp. 396–410 (cit. on p. 119).

[Cho+01] S. Chow, Y. Gu, H. Johnson, and V. A. Zakharov. "An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs". In: *International Conference on Information Security*. Springer. 2001, pp. 144–155 (cit. on p. 54).

[Cho+12] L. W. Chow, J. P. Baukus, B. J. Wang, and R. P. Cocchi. *Camouflaging a standard cell based integrated circuit*. US Patent 8,151,235. 2012 (cit. on p. 21).

[Cho+20] H. S. Choo, C. Y. Ooi, M. Inoue, N. Ismail, M. Moghbel, and C. H. Kok. "Register-Transfer-Level Features for Machine-Learning-Based Hardware Trojan Detection". In: *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences* 103.2 (2020), pp. 502–509 (cit. on p. 119).

[Cla] Clang. *C language family frontend for LLVM*. `https://clang.llvm.org/`. Accessed: 2021-02-20 (cit. on p. 129).

[Col] C. Collberg. *Tigress*. `https://tigress.wtf/`. Accessed: 2021-01-22 (cit. on p. 35).

[Cou+09] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. "An introduction to high-level synthesis". In: *IEEE Design & Test of Computers* 26.4 (2009), pp. 8–17 (cit. on p. 12).

[CP10] J. Cappaert and B. Preneel. "A General Model for Hiding Control Flow". In: *Proceedings of the tenth annual ACM workshop on Digital rights management*. 2010, pp. 35–42 (cit. on p. 54).

[CTL97] C. Collberg, C. Thomborson, and D. Low. *A Taxonomy of Obfuscating Transformations*. 1997 (cit. on pp. 29, 32, 34, 48).

[CTL98] C. Collberg, C. Thomborson, and D. Low. "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1998, pp. 184–196 (cit. on pp. 36, 48).

[Cui+11] A. Cui, C.-H. Chang, S. Tahar, and A. T. Abdel-Hamid. "A robust FSM watermarking scheme for IP protection of sequential circuit design". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.5 (2011), pp. 678–690 (cit. on p. 79).

[CZZ17] C.-H. Chang, Y. Zheng, and L. Zhang. "A retrospective and a look forward: Fifteen years of physical unclonable function advancement". In: *IEEE Circuits and Systems Magazine* 17.3 (2017), pp. 32–62 (cit. on pp. 19, 23).

[DBC19] A. Dey, S. Bhattacharya, and N. Chaki. "Software watermarking: Progress and challenges". In: *INAE Letters* 4.1 (2019), pp. 65–75 (cit. on p. 80).

[Dev] DevCloud. `https://software.intel.com/content/www/us/en/develop/tools/devcloud.html`. Accessed: 2021-01-13 (cit. on p. 40).

[DF19] S. Dupuis and M.-L. Flottes. "Logic locking: A survey of proposed methods and evaluation metrics". In: *Journal of Electronic Testing* 35.3 (2019), pp. 273–291 (cit. on pp. 22, 23).

[DM96] R. I. Davidson and N. Myhrvold. *Method and system for generating and auditing a signature for a computer program*. US Patent 5,559,884. 1996 (cit. on p. 79).

[DRA15] M. Dashtbani, A. Rajabzadeh, and M. Asghari. "High Level Synthesis as a service". In: *2015 5th International Conference on Computer and Knowledge Engineering (ICCKE)*. IEEE. 2015, pp. 331–336 (cit. on p. 41).

[Dup+15] S. Dupuis, P.-S. Ba, M.-L. Flottes, G. Di Natale, and B. Rouzeyre. "New testing procedure for finding insertion sites of stealthy hardware Trojans". In: *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2015, pp. 776–781 (cit. on p. 118).

[DV13] J. Delvaux and I. Verbauwhede. "Side channel modeling attacks on 65nm arbiter PUFs exploiting CMOS device noise". In: *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE. 2013, pp. 137–142 (cit. on p. 20).

[Fal+11] P. Falcarin, C. Collberg, M. Atallah, and M. Jakubowski. "Guest editors' introduction: Software protection". In: *IEEE Software* 28.2 (2011), pp. 24–27 (cit. on p. 31).

[FT03] Y.-C. Fan and H.-W. Tsao. “Watermarking for intellectual property protection”. In: *Electronics Letters* 39.18 (2003), pp. 1316–1318 (cit. on p. 25).

[Gaj+92] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin. *High-level synthesis: introduction to chip and system design*. 1992 (cit. on p. 14).

[Gar+13] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. “Candidate Indistinguishability Obfuscation and Functional Encryption for all Circuits”. In: *2013 IEEE 54th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society. 2013, pp. 40–49 (cit. on p. 30).

[Gas+02] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas. “Silicon physical random functions”. In: *Proceedings of the 9th ACM Conference on Computer and Communications Security*. 2002, pp. 148–160 (cit. on p. 19).

[Gin] T. Gingold. *GHDL*. `http://ghdl.free.fr/`. Accessed: 2021-01-22 (cit. on p. 72).

[Gon+08] D. Gong, F. Liu, B. Lu, P. Wang, and L. Ding. “Hiding information in java class file”. In: *2008 International Symposium on Computer Science and Computational Technology*. Vol. 2. IEEE. 2008, pp. 160–164 (cit. on p. 80).

[Got+01] H. Goto, M. Mambo, H. Shizuya, and Y. Watanabe. “Evaluation of tamper-resistant software deviating from structured programming rules”. In: *Australasian Conference on Information Security and Privacy*. Springer. 2001, pp. 145–158 (cit. on p. 32).

[Gro89] D. Grover. “Program identification”. In: *The protection of computer software—its technology and applications*. Cambridge University Press. 1989, pp. 119–150 (cit. on p. 83).

[Gui+14] U. Guin, K. Huang, D. DiMase, J. M. Carulli, M. Tehranipoor, and Y. Makris. “Counterfeit integrated circuits: A rising threat in the global semiconductor supply chain”. In: *Proceedings of the IEEE* 102.8 (2014), pp. 1207–1228 (cit. on p. 16).

[Guo+15] X. Guo, R. G. Dutta, Y. Jin, F. Farahmandi, and P. Mishra. “Pre-silicon security verification and validation: A formal perspective”. In: *Proceedings of the 52nd Annual Design Automation Conference*. 2015, pp. 1–6 (cit. on p. 118).

[Har+08] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. "CHStone: A Benchmark Program Suite for Practical C-based High-Level Synthesis". In: *2008 IEEE International Symposium on Circuits and Systems*. IEEE. 2008, pp. 1192–1195 (cit. on p. 65).

[Has] Hastlayer. `https://hastlayer.com/`. Accessed: 2021-01-13 (cit. on p. 40).

[HBF08] D. E. Holcomb, W. P. Burleson, and K. Fu. "Power-up SRAM state as an identifying fingerprint and source of true random numbers". In: *IEEE Transactions on Computers* 58.9 (2008), pp. 1198–1210 (cit. on p. 19).

[HD11] J. Hamilton and S. Danicic. "A survey of static software watermarking". In: *2011 World Congress on Internet Security (WorldCIS-2011)*. IEEE. 2011, pp. 100–107 (cit. on p. 80).

[HE12] M. Hataba and A. El-Mahdy. "Cloud protection by obfuscation: Techniques and metrics". In: *Proceedings of the 2012 Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*. 2012, pp. 369–372 (cit. on p. 31).

[Her+14] C. Herder, M.-D. Yu, F. Koushanfar, and S. Devadas. "Physical unclonable functions and applications: A tutorial". In: *Proceedings of the IEEE* 102.8 (2014), pp. 1126–1141 (cit. on p. 20).

[Hos+18] S. Hosseinzadeh, S. Rauti, S. Laurén, J.-M. Mäkelä, J. Holvitie, S. Hyrynsalmi, and V. Leppänen. "Diversification and obfuscation techniques for software security: A systematic literature review". In: *Information and Software Technology* 104 (2018), pp. 72–93 (cit. on pp. 33, 36).

[Hua+13] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, S. Brown, and J. Anderson. "The effect of compiler optimizations on high-level synthesis for FPGAs". In: *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2013, pp. 89–96 (cit. on p. 78).

[IK18] S. A. Islam and S. Katkoori. "High-level synthesis of key based obfuscated RTL datapaths". In: *2018 19th International Symposium on Quality Electronic Design (ISQED)*. IEEE. 2018, pp. 407–412 (cit. on pp. 27, 44).

[Ird] Irdeto. *Cloakware*. `https://irdeto.com/cloakware-software-protection/`. Accessed: 2021-01-22 (cit. on p. 35).

[Jai+03] A. K. Jain, L. Yuan, P. R. Pari, and G. Qu. "Zero overhead watermarking technique for FPGA designs". In: *Proceedings of the 13th ACM Great Lakes symposium on VLSI*. 2003, pp. 147–152 (cit. on p. 20).

[JM07] R. W. Jarvis and M. G. Mcintyre. *Split manufacturing method for advanced semiconductor circuits*. US Patent 7,195,931. 2007 (cit. on p. 18).

[Jun+15] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. "Obfuscator-LLVM–software protection for the masses". In: *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE. 2015, pp. 3–9 (cit. on p. 35).

[Kah+98] A. B. Kahng, J. Lach, W. H. Mangione-Smith, S. Mantik, I. L. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe. "Watermarking techniques for intellectual property protection". In: *Proceedings of the 35th annual Design Automation Conference*. 1998, pp. 776–781 (cit. on p. 20).

[KHP05] F. Koushanfar, I. Hong, and M. Potkonjak. "Behavioral synthesis techniques for intellectual property protection". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 10.3 (2005), pp. 523–545 (cit. on pp. 26, 79).

[KM14] A. Kulkarni and R. Metta. "A new code obfuscation scheme for software protection". In: *2014 IEEE 8th International Symposium on Service Oriented System Engineering*. IEEE. 2014, pp. 409–414 (cit. on p. 32).

[Kna96] D. W. Knapp. *Behavioral synthesis: digital system design using the synopsys behavioral compiler*. Prentice-Hall, Inc., 1996 (cit. on p. 12).

[KP13] N. Kae-Nune and S. Pesseguier. "Qualification and testing process to implement anti-counterfeiting technologies into IC packages". In: *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2013, pp. 1131–1136 (cit. on pp. 3, 16).

[Le] J.-C. Le Lann. *Crokus*. `https://github.com/JC-LL/crokus`. Accessed: 2021-02-20 (cit. on p. 98).

[Li+19a] H. Li, S. Patnaik, A. Sengupta, H. Yang, J. Knechtel, B. Yu, E. F. Young, and O. Sinanoglu. "Attacking split manufacturing from a deep learning perspective". In: *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2019, pp. 1–6 (cit. on p. 19).

[Li+19b] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli. "Graph matching networks for learning the similarity of graph structured objects". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 3835–3845 (cit. on p. 137).

[Lim+05] D. Lim, J. W. Lee, B. Gassend, G. E. Suh, M. Van Dijk, and S. Devadas. "Extracting secret keys from integrated circuits". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 13.10 (2005), pp. 1200–1205 (cit. on p. 19).

[LK09] T. László and Á. Kiss. "Obfuscating C++ Programs via Control Flow Flattening". In: *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica* 30 (2009), pp. 3–19 (cit. on pp. 53, 70).

[LLZ16] H. Li, Q. Liu, and J. Zhang. "A survey of hardware Trojan threat and defense". In: *Integration* 55 (2016), pp. 426–437 (cit. on p. 118).

[LW14] B. Liu and B. Wang. "Embedded reconfigurable logic for ASIC design obfuscation against supply chain attacks". In: *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2014, pp. 1–6 (cit. on p. 120).

[MC03] G. Myles and C. Collberg. "Software watermarking through register allocation: Implementation, analysis, and attacks". In: *International Conference on Information Security and Cryptology*. Springer. 2003, pp. 274–293 (cit. on p. 80).

[MC04] G. Myles and C. Collberg. "Detecting software theft via whole program path birthmarks". In: *International Conference on Information Security*. Springer. 2004, pp. 404–415 (cit. on p. 83).

[Mey+11] U. Meyer-Bäse, E. Castillo, G. Botella, L. Parrilla, and A. Garcia. "Intellectual property protection (IPP) using obfuscation in C, VHDL, and verilog coding". In: *Independent Component Analyses, Wavelets, Neural Networks, Biosystems, and Nanoengineering IX*. Vol. 8058. International Society for Optics and Photonics. 2011, 80581F (cit. on p. 26).

[MV10] R. Maes and I. Verbauwhede. "Physically unclonable functions: A study on the state of the art and future research directions". In: *Towards Hardware-Intrinsic Security*. Springer, 2010, pp. 3–37 (cit. on p. 19).

[Nar+01] N. Narayan, R. D. Newbould, J. D. Carothers, J. J. Rodriguez, and W. T. Holman. “IP protection for VLSI designs via watermarking of routes”. In: *Proceedings 14th Annual IEEE International ASIC/SOC Conference (IEEE Cat. No. 01TH8558)*. IEEE. 2001, pp. 406–410 (cit. on p. 20).

[Nar+12] S. Narasimhan, W. Yueh, X. Wang, S. Mukhopadhyay, and S. Bhunia. “Improving IC security against Trojan attacks through integration of security monitors”. In: *IEEE Design & Test of Computers* 29.5 (2012), pp. 37–46 (cit. on p. 120).

[Oli01] A. L. Oliveira. “Techniques for the creation of digital watermarks in sequential circuit designs”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20.9 (2001), pp. 1101–1117 (cit. on p. 25).

[opt] opt. *LLVM optimizer*. https://llvm.org/docs/CommandGuide/opt.html. Accessed: 2021-02-20 (cit. on p. 129).

[PBR16] I. Polian, G. T. Becker, and F. Regazzoni. “Trojans in early design steps—an emerging threat”. In: (2016) (cit. on pp. 115, 119).

[PF13] C. Pilato and F. Ferrandi. “Bambu: A modular framework for the high level synthesis of memory-intensive applications”. In: *2013 23rd International Conference on Field programmable Logic and Applications*. IEEE. 2013, pp. 1–4 (cit. on p. 116).

[Pil+18a] C. Pilato, K. Basu, F. Regazzoni, and R. Karri. “Black-Hat High-Level Synthesis: Myth or Reality?” In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.4 (2018), pp. 913–926 (cit. on pp. 3, 111, 115, 116, 132).

[Pil+18b] C. Pilato, F. Regazzoni, R. Karri, and S. Garg. “TAO: techniques for algorithm-level obfuscation during high-level synthesis”. In: *Proceedings of the 55th Annual Design Automation Conference*. 2018, pp. 1–6 (cit. on pp. 27, 44).

[Pil+19] C. Pilato, K. Basu, M. Shayan, F. Regazzoni, and R. Karri. “High-Level Synthesis of Benevolent Trojans”. In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 1124–1129 (cit. on pp. 77, 79).

[PMP17] L. Piccolboni, A. Menon, and G. Pravadelli. “Efficient control-flow subgraph matching for detecting hardware trojans in RTL models”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 16.5s (2017), pp. 1–19 (cit. on p. 119).

[PT06] M. Pecht and S. Tiku. “Bogus: electronic manufacturing and consumers confront a rising tide of counterfeit electronics”. In: *IEEE spectrum* 43.5 (2006), pp. 37–46 (cit. on pp. 3, 16).

[QP98] G. Qu and M. Potkonjak. “Analysis of watermarking techniques for graph coloring problem”. In: *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*. 1998, pp. 190–193 (cit. on p. 80).

[Qua] Quarkslab. *Quarks AppShield*. `https://quarkslab.com/quarks-appshield/`. Accessed: 2021-01-22 (cit. on p. 36).

[R+78] R. L. Rivest, L. Adleman, M. L. Dertouzos, et al. “On data banks and privacy homomorphisms”. In: *Foundations of secure computation* 4.11 (1978), pp. 169–180 (cit. on p. 43).

[Raj+11] J. Rajendran, V. Jyothi, O. Sinanoglu, and R. Karri. “Design and analysis of ring oscillator based Design-for-Trust technique”. In: *29th VLSI Test Symposium*. IEEE. 2011, pp. 105–110 (cit. on p. 120).

[Raj+12] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri. “Security analysis of logic obfuscation”. In: *Proceedings of the 49th Annual Design Automation Conference*. 2012, pp. 83–89 (cit. on p. 22).

[Raj+13] J. Rajendran, M. Sam, O. Sinanoglu, and R. Karri. “Security analysis of integrated circuit camouflaging”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013, pp. 709–720 (cit. on p. 21).

[Raj+16] J. Rajendran, A. M. Dhandayuthapany, V. Vedula, and R. Karri. “Formal security verification of third party intellectual property cores for information leakage”. In: *2016 29th International conference on VLSI design and 2016 15th international conference on embedded systems (VLSID)*. IEEE. 2016, pp. 547–552 (cit. on p. 118).

[Ras+99] A. Rashid, J. Asher, W. H. Mangione-Smith, and M. Potkonjak. “Hierarchical watermarking for protection of DSP filter cores”. In: *Proceedings of the IEEE 1999 Custom Integrated Circuits Conference (Cat. No. 99CH36327)*. IEEE. 1999, pp. 39–42 (cit. on p. 27).

[Rea+14] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks. “MachSuite: Benchmarks for Accelerator Design and Customized Architectures”. In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2014, pp. 110–119 (cit. on p. 65).

[RKM08] J. A. Roy, F. Koushanfar, and I. L. Markov. “EPIC: Ending Piracy of Integrated Circuits”. In: *2008 Design, Automation and Test in Europe*. IEEE. 2008, pp. 1069–1074 (cit. on pp. 16, 22).

[RSK13] J. Rajendran, O. Sinanoglu, and R. Karri. “Is split manufacturing secure?” In: *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2013, pp. 1259–1264 (cit. on pp. 18, 19).

[RVK15] J. Rajendran, V. Vedula, and R. Karri. “Detecting malicious modifications of data in third-party intellectual property cores”. In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE. 2015, pp. 1–6 (cit. on p. 118).

[Rya13] M. D. Ryan. “Cloud computing security: The scientific challenge, and a survey of solutions”. In: *Journal of Systems and Software* 86.9 (2013), pp. 2263–2268 (cit. on p. 42).

[SB16] A. Sengupta and S. Bhadauria. “Exploring low cost optimal watermark for reusable IP cores during high level synthesis”. In: *IEEE Access* 4 (2016), pp. 2198–2215 (cit. on p. 79).

[SBM16] A. Sengupta, S. Bhadauria, and S. P. Mohanty. “Embedding low cost optimal watermark during high level synthesis for reusable IP core protection”. In: *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2016, pp. 974–977 (cit. on p. 26).

[SD07] G. E. Suh and S. Devadas. “Physical unclonable functions for device authentication and secret key generation”. In: *2007 44th ACM/IEEE Design Automation Conference*. IEEE. 2007, pp. 9–14 (cit. on p. 19).

[SJX09] Z. Sha, H. Jiang, and A. Xuan. “Software watermarking algorithm by coefficients of equation”. In: *2009 Third International Conference on Genetic and Evolutionary Computing*. IEEE. 2009, pp. 410–413 (cit. on p. 79).

[SL12] P. Sivadasan and P. S. Lal. “Securing SQLJ source codes from business logic disclosure by data hiding obfuscation”. In: *arXiv preprint arXiv:1205.4813* (2012) (cit. on p. 31).

[SR17] A. Sengupta and D. Roy. “Protecting IP core during architectural synthesis using HLT-based obfuscation”. In: *Electronics Letters* 53.13 (2017), pp. 849–851 (cit. on p. 27).

[SRM15] P. Subramanyan, S. Ray, and S. Malik. “Evaluating the security of logic encryption algorithms”. In: *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE. 2015, pp. 137–143 (cit. on p. 22).

[SS08] M. Shirali-Shahreza and S. Shirali-Shahreza. "Software watermarking by equation reordering". In: *2008 3rd International Conference on Information and Communication Technologies: From Theory to Applications*. IEEE. 2008, pp. 1–4 (cit. on p. 79).

[ST11] H. Salmani and M. Tehranipoor. "Layout-aware switching activity localization to enhance hardware Trojan detection". In: *IEEE Transactions on Information Forensics and Security* 7.1 (2011), pp. 76–87 (cit. on p. 120).

[ST13] H. Salmani and M. Tehranipoor. "Analyzing circuit vulnerability to hardware Trojan insertion at the behavioral level". In: *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*. IEEE. 2013, pp. 190–195 (cit. on p. 133).

[STP11] H. Salmani, M. Tehranipoor, and J. Plusquellic. "A novel technique for improving hardware trojan detection and reducing trojan activation time". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20.1 (2011), pp. 112–125 (cit. on p. 120).

[Stu] Stunnix. `http://stunnix.com/`. Accessed: 2021-01-22 (cit. on p. 35).

[Tak15] S. Takamaeda-Yamazaki. "Pyverilog: A python-based hardware design processing toolkit for verilog hdl". In: *International Symposium on Applied Reconfigurable Computing*. Springer. 2015, pp. 451–460 (cit. on p. 61).

[Tam+04] H. Tamada, M. Nakamura, A. Monden, and K.-i. Matsumoto. "Design and evaluation of birthmarks for detecting theft of java programs." In: *IASTED Conf. on Software Engineering*. 2004, pp. 569–574 (cit. on p. 83).

[Tan+20] B. Tan, R. Karri, N. Limaye, A. Sengupta, O. Sinanoglu, M. M. Rahman, S. Bhunia, D. Duvalsaint, A. Rezaei, Y. Shen, et al. "Benchmarking at the Frontier of Hardware Security: Lessons from Logic Locking". In: *arXiv preprint arXiv:2006.06806* (2020) (cit. on p. 23).

[TC00] I. Torunoglu and E. Charbon. "Watermarking-based copyright protection of sequential functions". In: *IEEE Journal of Solid-State Circuits* 35.3 (2000), pp. 434–440 (cit. on p. 24).

[TEE12] M. Tebaa, S. El Hajji, and A. El Ghazi. "Homomorphic encryption applied to the cloud computing security". In: *Proceedings of the World Congress on Engineering*. Vol. 1. 2012. 2012, pp. 4–6 (cit. on p. 43).

[Tru] Trust-Hub. `https://www.trust-hub.org/`. Accessed: 2021-02-17 (cit. on pp. 116, 119).

[VD01] W. M. Van Fleet and M. R. Dransfield. *Method of recovering a gate-level netlist from a transistor-level*. US Patent 6,190,433. 2001 (cit. on p. 15).

[VS16] N. Veeranna and B. C. Schafer. "Hardware Trojan detection in behavioral intellectual properties (IP's) using property checking techniques". In: *IEEE Transactions on Emerging Topics in Computing* 5.4 (2016), pp. 576–585 (cit. on pp. 117, 119).

[VS17a] N. Veeranna and B. C. Schafer. "Efficient behavioral intellectual properties source code obfuscation for high-level synthesis". In: *2017 18th IEEE Latin American Test Symposium (LATS)*. IEEE. 2017, pp. 1–6 (cit. on pp. 3, 27, 43–45, 78).

[VS17b] N. Veeranna and B. C. Schafer. "S3CBench: Synthesizable security systemC benchmarks for high-level synthesis". In: *Journal of Hardware and Systems Security* 1.2 (2017), pp. 103–113 (cit. on p. 118).

[VVS01] R. Venkatesan, V. Vazirani, and S. Sinha. "A graph theoretic approach to software watermarking". In: *International Workshop on Information Hiding*. Springer. 2001, pp. 157–168 (cit. on p. 80).

[Wan+00] C. Wang, J. Hill, J. Knight, and J. Davidson. *Software Tamper Resistance: Obstructing Static Analysis of Programs*. Tech. rep. Technical Report CS-2000-12, University of Virginia, 12 2000, 2000 (cit. on p. 52).

[WC12] R. A. Walker and R. Camposano. *A survey of high-level synthesis systems*. Vol. 135. Springer Science & Business Media, 2012 (cit. on p. 12).

[WGK13] C. Wolf, J. Glaser, and J. Kepler. "Yosys - a free Verilog synthesis suite". In: *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*. 2013 (cit. on p. 97).

[XT13] K. Xiao and M. Tehranipoor. "BISA: Built-in self-authentication for preventing hardware Trojan insertion". In: *2013 IEEE international symposium on hardware-oriented security and trust (HOST)*. IEEE. 2013, pp. 45–50 (cit. on p. 120).

[Xu+17] H. Xu, Y. Zhou, Y. Kang, and M. R. Lyu. "On secure and usable program obfuscation: A survey". In: *arXiv preprint arXiv:1710.01139* (2017) (cit. on pp. 29, 30).

[Xue+20] M. Xue, C. Gu, W. Liu, S. Yu, and M. O'Neill. "Ten years of hardware Trojans: a survey from the attacker's perspective". In: *IET Computers & Digital Techniques* 14.6 (2020), pp. 231–246 (cit. on p. 113).

[YDZ17] Q. Yu, J. Dofe, and Z. Zhang. "Exploiting hardware obfuscation methods to prevent and detect hardware trojans". In: *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE. 2017, pp. 819–822 (cit. on pp. 112, 120).

[YY10] I. You and K. Yim. "Malware obfuscation techniques: A brief survey". In: *2010 International conference on broadband, wireless computing, communication and applications*. IEEE. 2010, pp. 297–300 (cit. on p. 32).

[ZA15] K. Zeng and P. Athanas. "Discovering reusable hardware using birthmarking techniques". In: *2015 IEEE International Conference on Information Reuse and Integration*. IEEE. 2015, pp. 106–113 (cit. on pp. 84, 92).

[Zen+10] Y. Zeng, F. Liu, X. Luo, and C. Yang. "Robust software watermarking scheme based on obfuscated interpretation". In: *2010 International Conference on Multimedia Information Networking and Security*. IEEE. 2010, pp. 671–675 (cit. on p. 80).

[Zha+19] J. Zhao, T. Liang, S. Sinha, and W. Zhang. "Machine learning based routing congestion prediction in FPGA high-level synthesis". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 1130–1135 (cit. on p. 137).

[Zho+14] B. Zhou, W. Zhang, S. Thambipillai, and J. Teo. "A low cost acceleration method for hardware Trojan detection based on fan-out cone analysis". In: *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*. 2014, pp. 1–10 (cit. on p. 120).

[ZT05] W. Zhu and C. Thomborson. "Algorithms to watermark software through register allocation". In: *International Conference on Digital Rights Management*. Springer. 2005, pp. 180–191 (cit. on p. 80).

DOCTORAT BRETAGNE LOIRE / MATHSTIC

ENSTA Bretagne

Title: Transient Obfuscation for HLS Security: Application to Cloud Security, Birthmarking and Hardware Trojan Defense

Keywords: High-level Synthesis, Hardware Security, Cloud, Obfuscation, Watermarking, Hardware Trojans

Abstract: The growing globalization of the semiconductor supply chain, as well as the increasing complexity and diversity of hardware design flows, have lead to a surge in security threats: risks of intellectual property theft and reselling, reverse-engineering and malicious code insertion in the form of hardware Trojans during manufacturing and at design time have been a growing research focus in the past years. However, threats during high-level synthesis (HLS), where an algorithmic description is transformed into a lower level hardware implementation, have only recently been considered, and few solutions have been given so far.

In this thesis, we focus on how to secure designs during behavioral synthesis using either a cloud-based or an internal but untrusted HLS tool. We introduce a novel design time protection method called transient obfuscation, where the high-level source code is obfuscated using key-based techniques, and deobfuscated after HLS at register-transfer level. This two-step method ensures correct design functionality and low design overhead. We propose three ways to integrate transient obfuscation in different security mechanisms.

First, we show how it can be used to prevent intellectual property theft and illegal reuse in a cloud-based HLS scenario. Then, we extend this work to watermarking, by exploiting the side-effects of transient obfuscation on HLS tools to identify stolen designs. Finally, we show how this method can also be used against hardware Trojans, both by preventing insertion and by facilitating detection.