POLITECNICO DI MILANO
Master of Science in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

Composable Heuristics for RTL Logic Locking Optimization based on System Dependence Analysis

Supervisor: Prof. Christian Pilato

M.Sc. Thesis by:
Luca Collini, 939607

Academic Year 2020-2021

To the professors who inspired me.

Abstract

The recent advance of Integrated Circuit (IC) technology has brought an increase in both design complexity and manufacturing costs. This leads more and more design houses to become *fabless* [26], that is they are outsourcing the fabrication process to external foundries. The introduction of third-party entities in the IC supply chain brings new security challenges [2]. Reverse engineering is the major threat as it can lead to intellectual property theft and malicious modifications. The total loss from IC counterfeiting was estimated to be around $169 billion already in 2011 [19] and we only expect it to grow in the following years, also considering the increasing number of reported parts [8]. In this scenario, active methods for protecting intellectual property are more effective to protect against counterfeiting than long legal disputes after watermarking identification. *Logic locking* aims at thwarting reverse engineering by adding extra logic to the original design that is controlled by a new set of inputs called *key inputs*. The correct functionality is obtained only if the correct sequence of bits is provided to the key inputs. The extra logic added to lock the design introduces area, power, and timing overheads. In real-world applications, we cannot obfuscate the whole design due to constraints on these overheads. Obfuscating different parts of a design may lead to very different results in terms of security, though saying which parts of a design should be obfuscated to obtain the best possible solution is an open question. A technique for design space exploration for logic locking optimization at HLS [22] showed that carefully selecting the *obfuscation points* can yield much better results than obfuscating the whole design. HLS solutions are not compatible with all design flows. We show that such approach is not always feasible at RTL due to the higher number of possible obfuscation points and the slower simulations. For this reason, we explore more efficient solutions taking into consideration the effects on the chip results when locking an element of the design instead of performing a blind search. We propose four heuristics that analyze different characteristics based on the dependencies between signals. We represent the signal dependencies of a design with a System Dependence Graph. The proposed solution yields higher differential entropy for 92% of the cases when compared to non-optimizing techniques. When compared to state-of-the-art heuristics it shows comparable results while requiring 100× to 400× less computational time.

Sommario

Il progresso della tecnologia dei circuiti integrati negli ultimi decenni ha portato a un aumento sia della complessità della progettazione che dei costi di produzione. Questo porta sempre più aziende di progettazione a diventare *fabless* [26], cioè a esternalizzare il processo di fabbricazione ad aziende esterne. L'introduzione di terze parti nella catena di fornitura dei circuiti integrati porta nuove sfide in termini di sicurezza [2]. Il reverse engineering è la principale minaccia in quanto può portare a furto di proptietà intellettuale e modifiche dannose. La perdita economica causata dalla contraffazione di circuiti integrati è stata stimata in circa 169 miliardi di dollari nel 2011 [19] e ci aspettiamo che possa solo crescere nei prossimi anni, considerando anche il crescente numero di parti segnalate [8]. In questo scenario, i metodi attivi per la protezione della proprietà intellettuale sono più efficaci per la protezione dalla contraffazione rispetto alle lunghe controversie legali dopo l'identificazione del watermark. Il *logic locking* mira a contrastare il reverse engineering aggiungendo ulteriore logica al progetto originale controllata da un nuovo gruppo di input chiamato *chiave*. La corretta funzionalità si ottiene solo se viene fornita come chiave la corretta sequenza di bit. La logica aggiuntiva aggiunta per bloccare il design introduce costi generali di area, potenza e ritardo. Nelle applicazioni del mondo reale, non possiamo offuscare l'intero progetto a causa dei vincoli su questi costi generali. Offuscare parti diverse di un progetto può portare a risultati molto diversi in termini di sicurezza, anche se dire quali parti di un progetto dovrebbero essere offuscate per ottenere la migliore soluzione possibile è un problema aperto. Una tecnica per l'esplorazione dello spazio di progettazione per l'ottimizzazione del logic locking in HLS [22] ha mostrato che scegliere con cura i *punti di offuscamento* può produrre risultati molto migliori rispetto all'offuscamento dell'intero progetto. Le soluzioni HLS non sono compatibili con tutti i flussi di progettazione. Mostriamo che tale approccio non è sempre fattibile a RTL a causa del maggior numero di possibili punti di offuscamento e delle simulazioni più lente. Per questo motivo esploriamo soluzioni più efficienti prendendo in considerazione l'effetto sul risultato del circuito di offuscare un elemento del circuito invece di eseguire una ricerca alla cieca. Proponiamo quattro euristiche che analizzano diverse caratteristiche, basando la nostra analisi sulle dipendenze tra i segnali. Rappresentiamo le dipendenze del segnale di un circuito con un grafo delle dipendenze del sistema. La soluzione proposta produce entropia differenziale maggiore per il 92% dei casi rispetto alle tecniche non ottimizzanti. Se confrontato con l'euristica allo stato dell'arte, mostra risultati comparabili richiedendo da 100× a 400× in meno in termini di tempo di computazione.

Acknowledgements

I would like to thank countless people that helped me, directly and/or indirectly. My mum and dad, that always encouraged and supported me in following my passions. Christian for his guidance and support. The friends on which I can always count on. Milena and Steven, that helped me discover my passion for learning and for Computer Science.

Contents

List of Figures

List of Tables

Acronyms

ASIC - Application Specific Integrated Circuit.

AST - Abstract Syntax Tree.

DSE - Design Space Exploration.

HDL - Hardware Description Language.

HLS - High-Level Synthesys.

I/O - Input/Output.

IC - Integrated Circuit.

IP - Intellectual Property.

LHS - Left Hand Side.

PDG - Program Dependence Graph.

RHS - Right Hand Side.

RTL - Register-Transfer Level.

SDG - System Dependence Graph.

SoC - System on Chip.

Chapter 1

Introduction to Hardware IP Protection

1.1 Context

1.1.1 Background

In the last decades integrated circuits (IC) have seen a rise in both design complexity and manufacturing cost. This phenomenon has limited the number of companies that can afford the billion-dollar manufacturing foundries [7] forcing more and more companies to outsource IC fabrication to third-party foundries [23]. Nowadays a typical business model for a design house is to source pre-designed and pre-verified hardware IPs from different sources like third party IP vendors, integrate them into a system-on-chip (SoC) and send the final layout to an external foundry for fabrication. This trend is also known as the *globalization of the IC supply chain* [29]. However, the introduction of third parties in the IC supply chain introduces new security issues [2]. The first major concern is IP theft. As the market for domain-specific hardware components is becoming more and more lucrative, the high-value effort put into hardware design increases and makes it crucial to prevent IP theft [34]. In fact, the estimated loss due to IP violations alone was \$4 billions in 2008 [27], while the total loss from IC counterfeiting was estimated to be about \$169 billions in 2011 [19].

Figure 1.1 shows how in the last decade the problem of counterfeited parts has affected the semiconductor market. The high number of newly reported counterfeited parts in Figure 1.2 leads to think that there may be a huge number of counterfeited parts that has yet to be detected.

The second major concern is malicious modifications of the design. In

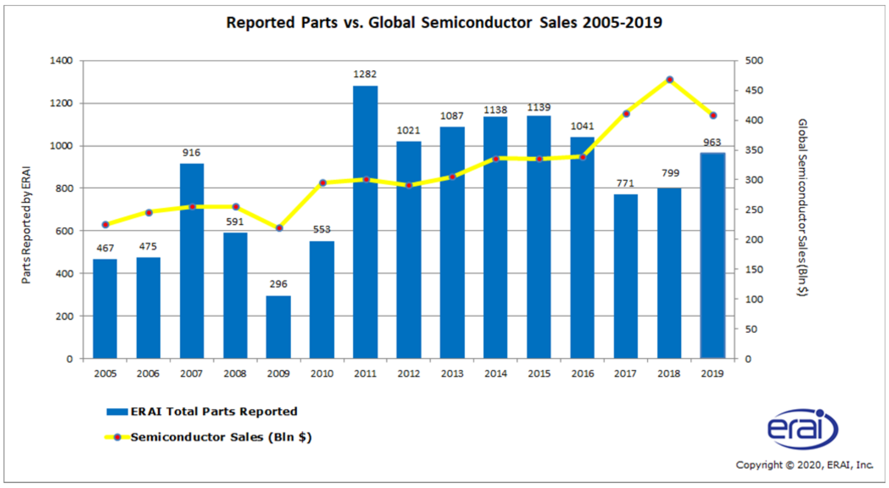


Figure 1.1: Reported counterfeited parts from 2005 to 2019 [8]

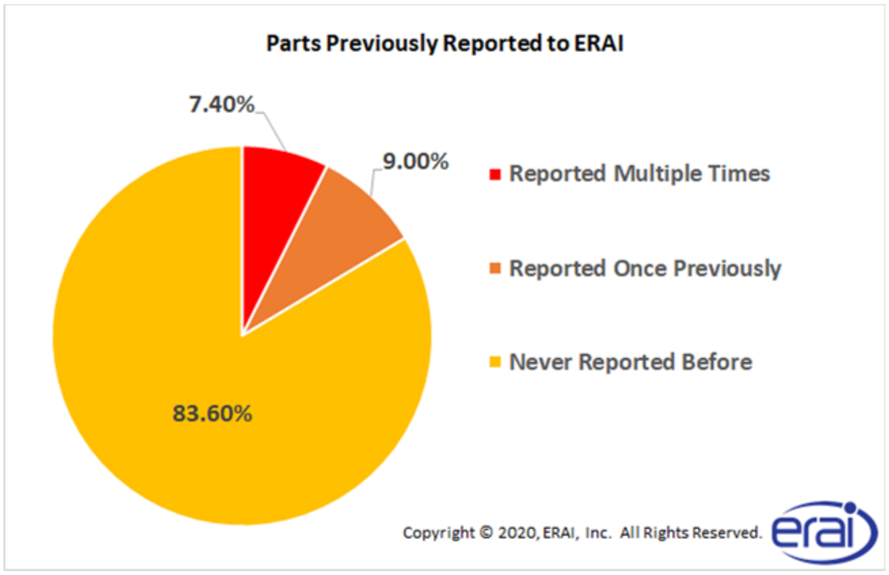


Figure 1.2: Previously reported parts [8]

fact a third party with malicious intentions may want to introduce a hardware Trojan or a backdoor into a design. The threat can range from disclosing secret technologies to compromising valuable network infrastructures, an example is the attempted trade of counterfeited Cisco equipment to the US Department of Defense [35].

Both issues require reverse engineering of the design under attack to extract and copy the functionality. Hardware obfuscation aims at hiding and disabling the functionality of a chip to thwart reverse engineering of the design. They can be divided in two classes: key-less obfuscation, such as split manufacturing, camouflaging, watermarking and fingerprinting, and key-based obfuscation, such as logic locking. Split manufacturing divides the manufacturing process between different untrusted foundries [20]. IC camouflaging prevents netlist extraction by introducing subtle cell design changes [6]. Watermarking and fingerprinting aim at simplifying detection and tracking of illegal copies of the design [1]. Logic locking idea is to apply modifications to the design that make it functional only when the correct key, unknown to the foundry, is applied [3]. It is important to understand that in addition to disabling a design, logic locking must also hide the functionality. An example of a bad logic locking technique would be one that outputs zero for each incorrect key by simply adding a multiplexer on the output. Such technique would still allow reverse engineering and would not protect effectively the design. Obfuscation techniques can be applied at different levels of the design flow. The main distinction of logic locking techniques is done with regard to the logic synthesis step. Logic synthesis is the step with which a netlist is extracted from a hardware design in the form of a formal description written with a hardware description language (HDL) such as Verilog or VHDL. Techniques applied after the synthesis step are called post-synthesis whereas the others are called pre-synthesis. Post-synthesis techniques work either at transistor [30] or netlist level [39, 28]. Pre-synthesis techniques can be applied at either RTL [21] or HLS [23, 40]. In figure 1.3 is reported the design flow of an integrated circuit.

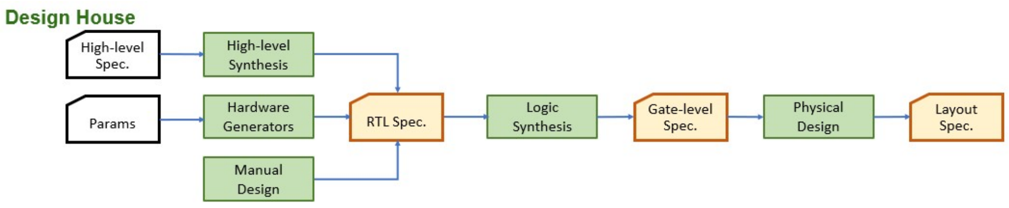


Figure 1.3: Design flow of an integrated circuit

As reported in the scheme of figure 1.3, there are different ways in which the deign flow can start. All of them converge into a Register-Transfer Level (RTL) specification before going through the synthesis step. Working at RTL allows us to develop techniques that are applicable to all the spectrum of designs since every design is in the form of an RTL specification at one point. Our work is focused on register-transfer level logic locking techniques.

1.1.2 Threat Model

In any security field it is important to state the threat model that is being considered. The threat model defines capabilities and intentions of the attacker. Threat models for logic obfuscation primarily rely on the concept of *Oracle* and *Ambiguity*. An Oracle is a chip that performs the correct computation, whereas Ambiguity refers to the ability of the attacker to distinguish between key inputs and normal inputs.
In an *oracle-less* scenario we suppose that the attacker does not have access to a functional unit. This setting is plausible for all those application that are not mass produced for retail market or where the chip is produced for the first time.
In a *oracle-guided* scenario we suppose that the attacker has access to a functional chip that is treated as a black-box unit. The attacker can only query the chip with input patterns and observe their output values.
Distinct Ambiguity is used to describe a situation in which the attacker is able to distinguish between key inputs and normal inputs.
Ubiquitous Ambiguity is used to describe a situation in which the attacker is not able to distinguish key inputs and normal inputs [29]. In our work we considere an Oracle-less scenario with Distinct Ambiguity. This is plausible scenario for low-volume markets where the attackers have strong methods to distinguish between functional and key signals.

1.2 Scenario and Problem Statement

Logic locking requires to add extra logic to the design, introducing overheads in terms of area, power and timing. A hardware designer that is trying to protect his work will be willing to spend up to a certain amount in terms of these overheads. This means that in real-world cases we cannot obfuscate the whole design due to area, power or timing constraints. A simple, yet still open, question arises: which parts should be obfuscated and which parts should be not, to obtain the best design from the security viewpoint? This question immediately arises at least another one:

> How can we say if an obfuscated design is better than another from the security viewpoint?

The ultimate answer to the latter question is that you should try to break both designs and the one that is broken first is less secure than the other. That would be very time consuming and error prone. Luckily there are some proposed metrics in the literature that have been shown to have correlations with resiliency towards certain kind of attacks. We will get into those in Chapter 2.

The first question though remains open. A novel approach to design space exploration of logic locking solutions at High-Level Synthesis (HLS) was proposed in [22] and shows that great improvements can be obtained by exploring the design space. The technique employs a genetic algorithm to perform design space exploration, which is computationally intensive, especially at RTL where the design space is larger and the RTL simulations to evaluate the security metrics are slower. At this level it can be used only on small designs since it only compares a huge number of alternatives with a "blind" search, without reasoning on the properties of the design.

With this work we want to find ways to select parts to obfuscate by reasoning on the effects of obfuscation.

1.3 Methodology

Our work followed a typical research process. We first conducted a systematic literature review on logic locking techniques and evaluation metrics. Then we followed an iterative development plan running two iterations of software prototyping and evaluation using the results and experience from the first round as a feedback for the second one. We used Synopsis vcs to run behavioral simulations of Verilog designs in order to compute the evaluation metrics. We performed synthesis to evaluate area overheads of our benchmark designs using Synopsys Design Compiler R-2020.09-SP1 targeting the Nan-gate 15nm ASIC technology at standard operating conditions (25°C). We implemented a prototype framework leveraging Pyverilog [33], a Python-based Hardware Design Processing Toolkit for Verilog HDL. We used the DEAP framework [9] to implement the design space exploration algorithm.

1.4 Contributions

In this thesis we present the following main contributions:

- A modular and composable design framework to apply logic locking with the support of commercial RTL synthesis tools (see Chapter 4).

- A procedure to extract System Dependence Graphs from a Verilog design (see Chapter 5).
- A set of scoring heuristics based on the analysis of the System Dependence Graph of a design (see Chapter 6).
- A prototype implementation and evaluation of the proposed approach (see Chapter 7).

1.5 Structure of Thesis

The rest of our thesis is organized as follows:

- Chapter 2 introduces the state of the art presenting the IC supply chain, logic locking techniques and the metrics used to evaluate them. It presents a design space exploration technique to optimize the use of logic locking at High-Level Synthesis. It concludes highlighting how our work contributes to the state of the art.
- Chapter 3 provides an in-depth description of the scenario and the ideas behind our solution. It highlights the necessary components that build-up the solution and introduces the remaining background needed to understand it.
- Chapter 4 presents the approach of our solution. In this chapter we show the architecture of the solution and our design decisions.
- Chapter 5 presents the first of the two main parts of our solution, the System Dependence Graph extraction procedure.
- Chapter 6 presents the second of the two main parts of our solution, the Scoring Heuristics for RTL logic locking optimization.
- Chapter 7 provides the implementation details together with the evaluation of the solution.
- Chapter 8 concludes our thesis. In this chapter we provide a summary of our work, we illustrate the contributions, we discuss the limitations of the solution and possible future research directions.
- Appendix A presents a test bench template for mean differential entropy estimation.

Chapter 2

State of the Art

This chapter discusses more in depth the main topics discussed in this thesis:

- Section 2.1 presents an in depth view on the IC supply chain, highlighting the the steps at which logic locking can be applied and the phases subjected to possible threats.
- Section 2.2 presents the metrics that are used to evaluate logic locking techniques.
- Section 2.3 presents the state of the art of logic locking techniques and how they score after the evaluation metrics presented in section two.
- Section 2.4 presents an optimization technique for logic locking at High Level Synthesis.
- Section 2.5 is a summary of the above bringing together the topics of the sections above underlining the importance of our work.

2.1 IC Design Flow

The IC design flow starts from a set of specifications that can be implemented in different ways. In case of a High-Level Synthesis (HLS) design, the implementation is done with a high level programming language like C or System C and a RTL description is obtained by means of a tool like Vivado HLS. Hardware generators can also be used to obtain a RTL specification from a set of specifications. A RTL description can also be designed manually with a HDL language such as VHDL or Verilog. The RTL description is given as input to a logic synthesis tool, such as Synopsis vcs, that produces a netlist. To obtain a physical layout the netlist goes through a process called place and routing that outpus a GDSII file. In between all these steps verification of the design functionality and compliance with constraints (area and timing) is performed. The verification steps are crucial since the further a bug goes, the more expensive it is to fix it. The final GDSII file is sent to a foundry for fabrication. After being manufactured, the chips are sent to a test facility and eventually they are sent back to the design house for distribution to the clients. In the ASIC supply chain the IP is exposed to different potential adversaries such as SoC integrators, foundries, test facilities and end-users. The major threat is reverse engineering that can lead to IP theft and overbuilding, or hardware Trojan insertion. Figure 2.1 shows illustrates the IC design flow up to the external foundry.

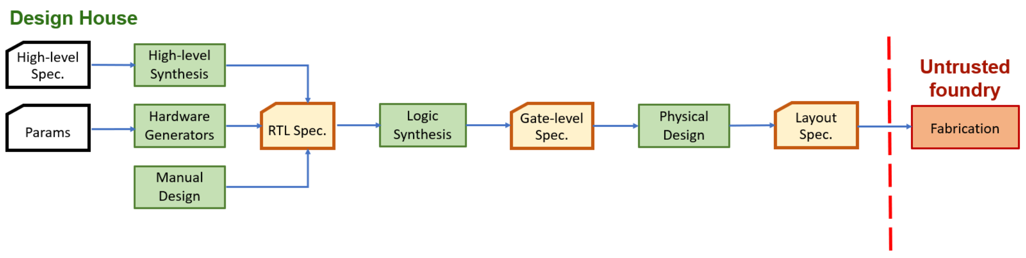


Figure 2.1: Complete design flow of an integrated circuit underlining the division between the design house and the third party foundry

2.2 Evaluation Metrics

Many metrics have been proposed to evaluate obfuscated designs. Some of them directly describe a physical property of the design, such as area or timing overhead, whereas others are values that have been shown to correlate with physical properties [2]:

- **Verification failure metric:** experimental metric that measures how many, and to which extent, outputs are affected by the obfuscation technique. To evaluate this metric an equivalence checking tool such as Synopsys Formality is needed to evaluate the functional difference between the original and the locked designs.

- **Entropy** (also known as Shannon Entropy): experimental metric that measures the amount of information in a data source. In the case of a combinational circuit it relates to the number of distinct outputs of the circuit. It tells us about two properties of the design:

  - *Power:* a design with high entropy must have many different possible outputs and therefore many transitions between logic-0 and logic-1, increasing the dynamic power of the circuit.
  - *Implications for obfuscation:* An obfuscated design with maximum entropy most resembles a random function.

- **Differential entropy:** experimental metric that is calculated with a miter circuit obtained by XORing each bit of the output of the locked circuit with the corresponding of the unlocked circuit. The entropy of the miter circuit is the evaluated to obtain the differential entropy. This metric represents the proportion of bits that differ between the obfuscated and the plain design, quantifying the output corruptability induced by the technique. Experiments found a close relation between differential entropy and power overhead.

- **Reconvergence:** structural metric that represents the rate of internal signals converging in other nodes. Experiments showed that more resilient circuits to the key sensitization attacks have higher reconvergence.

- **Key structure metric:** normalized metric that indicates the structural interconnection between the key gates. A high value for this metric indicates a high resiliency to the key sensitization attacks.

2.3 Logic Locking Techniques

Logic obfuscation techniques can be applied at different steps of the design phase. In case of high level descriptions, a High Level Synthesis (HLS) tool may apply algorithmic-level obfuscation [23]. Post-synthesis obfuscation is the most widespread category at the moment, post-synthesis techniques can be applied by modifying the netlist, for example adding extra logic ports.

Obfuscation at higher levels of abstraction brings advantages as it allows designers to hide the semantic information of the chip design.

On the contrary, post-synthesis techniques cannot protect information that is already embedded in the design by logic synthesis optimizations [16]. On the other hand, obfuscation at the HLS level requires to adapt the design flow. For these reasons, an RTL approach is highly attractive as it is placed in between the existing techniques. A first approach at register-transfer level was proposed in [5] while a promising approach is presented in [21].

Tables 2.1 and 2.2 summarize the classification of the main pre-synthesis and post-synthesis logic techniques.

For completeness, a brief overview of the main logic locking techniques (listed in Tables 2.2 and 2.1) is provided:

- **TAO** [23]: is an extension for HLS tools to produce obfuscated RTL descriptions. TAO obfuscates the algorithm via obfuscation of constants, control branches and adding variants in the flows of data.

- **ASSURE** [21]: is a pre-synthesis tool that works at RT level. Assure applies obfuscation to branches, operations and constants, hiding the semantic of the design.

- **CDFG** [5]: is a RT level technique that obfuscates the data flow graph of a design.

- **BDD** [16]: is a pre-synthesis technique that works on Binary Decision Diagrams. Key bits are added in the original BDD by randomly taking a node and adding two child nodes to it controlled by the key bit.

- **RLL** [25]: is the first proposed logic locking technique. It randomly inserts logic gates (tipically XOR or XNOR gates) controlled by a key bit.

- **SLL** [38]: Strong Logic Locking is a technique that strengthens the insertion of logic gates by inserting key-gates with complex interference among them.

- **Anti-SAT** [37]: is a technique that aims at making unfeasible SAT attacks by increasing the number of iterations to the exponential of number of primary inputs used to implement the AntiSAT block.

- **Cone size** [2]: is a heuristic technique that integrates the key gates with other gates that have the largest fanin or fanout cone or both.

We define an **obfuscation point** as any element that can be obfuscated by a given technique.

Table 2.1: Summary of the main logic locking techniques and scoring for the proposed metrics [2]

		Area overhead	Power overhead	Timing overhead	SAT attack resiliency	Key sens. attack resiliency	Verification failure metric	Entropy	Differential entropy	Reconvergence	Key structure metric
Pre-synthesis	TAO [23]	High	n.a.	Low	n.a.	n.a.	n.a.	n.a.	High [a]	n.a.	n.a.
	ASSURE [21]	Low	n.a.	Low	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
	CDFG [5]	Low	Low	n.a	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
	BDD Random [16]	High	High	Medium	Low	High	High	Low	Medium	Medium	High
	BDD AntiSAT [16, 37]	High	High	Medium	Medium	High	Low	Medium	Low	High	High
	BDD Entropy [2]	Medium	Medium	Low	Low	High	High	High	Medium	Low	High

[a]Calculated as Hamming distance

Table 2.2: *Summary of the main logic locking techniques and scoring for the proposed metrics [2]*

		Area overhead	Power overhead	Timing overhead	SAT attack resiliency	Key sens. attack resiliency	Verification failure metric	Entropy	Differential entropy	Reconvergence	Key structure metric
Post-synthesis	RLL [25]	Low	Low	Low	Low	Low	High	Medium	High	Medium	Low
	SLL [38]	Low	Low	Low	Low	Medium	medium	Medium	High	Medium	Low
	Cone size [2]	Low	Low	Medium	Low	Medium	Low	Medium	High	Medium	Medium
	AntiSAT Random [25, 37]	Medium	Medium	Low	High	Low	High	Medium	High	Low	Medium
	AntiSAT SLL [38, 37]	Medium	Medium	Low	High	Medium	Medium	Medium	High	Low	Medium
	Anti SAT Cone size [2]	Medium	Medium	Medium	High	Medium	Low	Medium	High	Low	Medium

2.4 Design Space Exploration for Logic Locking Optimization

Logic locking introduces overheads in the design. For this reason, researchers are putting effort into optimizing the use of logic locking to get good security results while keeping the overheads low. The approach proposed in [22] uses a genetic algorithm to perform Design Space Exploration (DSE) at HLS. Genetic algorithms are based on the concept of *individual* and *evolution*. An individual represents a solution to the problem in the form of a list of integers. An evolution is a function that generates a new individual starting from an existing one (ancestor). A *generation* is a set of individuals that have the same number of ancestors. Starting from a set of randomly generated individuals (generation 0), each individual is evaluated after a fitness function that determines the ”goodness” or fitness of the individual. Individuals with a low fitness are discarded while the others evolve to create a new generation. The process is repeated until the best fitness do not improve for a certain amount of generations or when the maximum number of generations is reached. Genetic algorithms are computationally intensive and using them for DSE at HLS is feasible only because evaluating an individual requires to run the C files of the design. At RTL the computational effort is much higher because the number of obfuscation points is larger and RTL simulations are more complex. This approach showed that obfuscating the whole design yields worse results than obfuscating only parts of it, highlighting the necessity of carefully selecting obfuscation points.

2.5 Summary

When applying logic locking, one must accept a certain area overhead. Logic locking can be applied at different abstraction levels, leading to different results in terms of physical overheads and robustness towards attacks. Some metrics have been proposed to compare different obfuscated designs but there is not a unified view on the matter yet.

Having metrics to compare different designs is not only crucial to compare different techniques but most importantly to compare different applications of the same technique.

In case the key is limited to a fixed amount of bits (to limit area overhead for example), it is important to understand where it is best to spend this limited amount of bits. Logic locking is a very open field at the moment and RTL techniques deserve further exploration. The logic locking state of the art is also lacking optimization approaches to maximize the efficiency of the obfuscation from a security viewpoint. The approach in [22] shows that it is possible to obtain good result in this direction and it is worth to explore it at RTL. With this work we show that such approach is not feasible at every level of abstraction as getting to lower levels highers the computational load required to evaluate the designs. For this reason we focus on more efficient solutions, taking into consideration the effect of locking on the chip results instead of performing a blind search.

Chapter 3

Problem Definition and Background

In this chapter we dive deeper into the problem statement, breaking it down into specific sub-problems.

- Section 3.1 restates the problem statement with more details.
- Section 3.2 presents the obfuscation techniques used in our work.
- Section 3.3 presents differential entropy with more details, explaining why we picked it as security metric.
- Section 3.4 presents Dependence Graphs.
- Section 3.5 summarizes the above sections.

3.1 Problem Statement

A fabless design house is working on a new IC design at RTL and is willing to protect it since sending the design files for fabrication will expose them to reverse engineering. However, obfuscating the design will introduce overheads, increasing its cost. The designers can fix the maximum cost overhead by deciding a maximum area overhead and the size of the tamper-proof memory that will store the key. HLS solutions such as [23, 22] cannot be applied since the design is already at RTL. They may consider using ASSURE [21], but the ASSURE results are dependent on the structure of the design because of its topological visit of the design. If the results are not good enough, the designers will need to refactor the design to enable the exploration of alternative solutions.

With this work we want to build a framework that allows to optimize logic locking for differential entropy under area and key bits constraint by analyzing the effects of obfuscation.

To analyze the effects of obfuscation points we look at the dependencies between signals via a Dependence Graph. The idea is that applying an obfuscation technique on an obfuscation point will affect all statements that depend on it. By analyzing the dependencies we can find the points that will have major impact on the design or that will build a *chain of obfuscation points* that will be amplify the effects.

We consider a scenario where the attacker does not have access to a working chip (*oracle*).

Moreover we assume that the attacker can distinguish between primary and key inputs (*distinct ambiguity*). In addition we assume that the attacker can distinguish between control and data inputs and outputs. This is plausible for low volume designs, where an attacker cannot buy a working chip from the market. It has been recently shown that oracle-less techniques [21] can be combined with techniques that prevent oracle-based attacks [14] to complement the protection [13]. The attacker may still perform netlist-based attacks such as machine learning-guided structural and functional analysis [4, 24, 31], desynthesis [17], and redundancy identification [12] to unlock the design and perform reverse engineering. For this reason we considered the obfuscation techniques proposed in ASSURE [21], as they have been proved to be resilient towards these attacks.

Our framework should be capable of:

- evaluating the differential entropy of obfuscated solutions (to optimize it as security metric),
- estimating the area of obfuscated solutions (to analyze the impact on the overhead),
- evaluating the number of key bits required by obfuscated solutions (to take the key budget into account),

to obfuscate a design optimizing differential entropy under area or key-bit constraints.

3.2 Semantic Obfuscation Techniques

In our work we consider the obfuscation techniques proposed in ASSURE [21]:

- **Constant obfuscation**: constants are completely replaced by key bits. For example $a = b + 4'b0100$ is obfuscated as $a = b + K_c$ where K_c is a the 4-bit constant stored in the key (see Fig. 3.1).

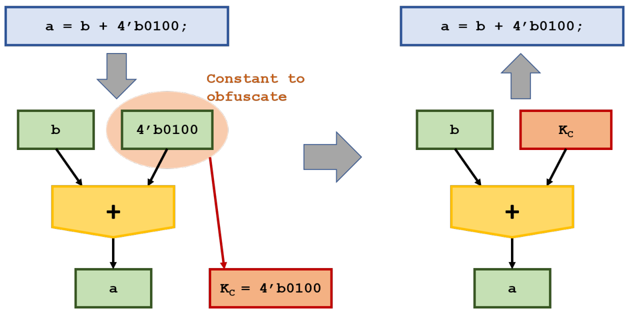


Figure 3.1: Example of constant obfuscation

- **Operation obfuscation**: a multiplexer is added to pick between the right operation and a dummy one based on the value of a key bit. For example $a = b+c$ is obfuscated as $a = K_o?(b+c) : (b-c)$ (see Fig. 3.2).

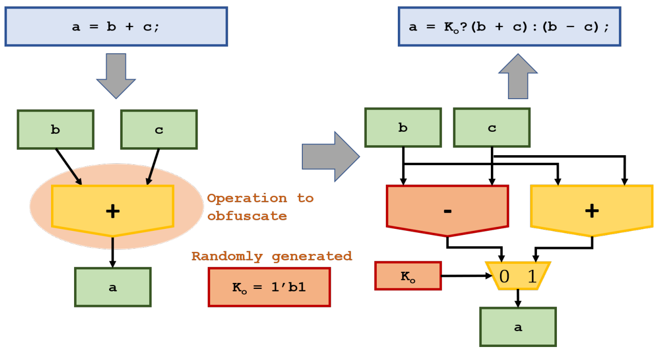


Figure 3.2: Example of operation obfuscation

- **Branch obfuscation**: the condition is XOR-ed with a key bit and it is inverted if the key bit is 1. For example, the condition $(a < b)$ can be obfuscated as $(a >= b) \bigoplus K_b$ or as $(a < b) \bigoplus K_b$ depending on the value of K_b (see Fig. 3.3).

The locking key is composed of two parts. The first part is randomly generated and is used to control the obfuscation of control branches and

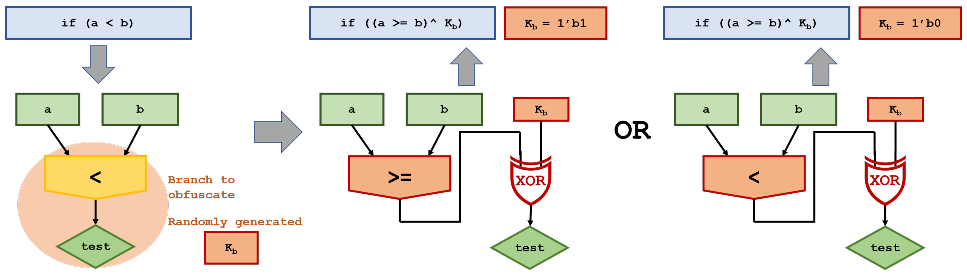


Figure 3.3: Example of branch obfuscation

operations. The second part is used to extract constants from the design embedding them in the key. An new input port is added through which the locking key is provided, the key then gets partitioned into sub-keys to be distributed to all locked elements. This approach protects the semantics of the designs rather than its structural netlist. An *obfuscation point* is an RTL element that can be obfuscated (i.e. a constant value, a conditional branch, or an operation) using a given locking technique. In ASSURE [21], obfuscation points are selected in a topological order (i.e. in the order in which they are found in a depth first search of the AST of the design). The order in which modules are defined affects the obfuscation outcome. We aim at carefully selecting the obfuscation points to achieve better results both in terms of security metrics and area overhead. We used these techniques to optimize because they have been proved to be resilient towards state-of-the-art, structure-based oracle-less attacks such as [4, 24]. Moreover it has been shown that they can be combined with other techniques that prevent oracle-guided attacks [14] to complement the protection [13].

3.3 Differential Entropy

In an oracle-less scenario an attacker will have to infer information either by looking at the design files or by observing the design functionality through simulations. The obfuscation techniques that we consider reveal no information about the design [21]. For this reason we evaluate the security of obfuscated solutions looking at the output corruptibility, i.e. how much the obfuscation techniques change the output values with respect to the expected one. We use the *mean differential entropy* as our security metric as it measures *output corruptibility* [2]. The differential entropy of a design is the entropy measured on a miter circuit obtained by XOR-ing the locked circuit with the original one. Entropy is measured on each output. The entropy of a output bit i is calculated as follows:

$$H_i = P_i \cdot log\frac{1}{P_i} + (1 - P_i) \cdot log\frac{1}{1 - P_i}$$

Where P_i is the probability of output i being equal to 1. In case of differential entropy, Pi is measured as follows:

$$P_i = \frac{\sum_{w=1}^{N} \sum_{t=1}^{M} OUT[i]_t \bigoplus OUT[i]_{t,w}}{N \cdot M}$$

Where $OUT[i]_t$ is the correct value of the output bit i when the input t is given to the unlocked circuit, and $OUT[i]_{t,w}$ is the value of the output bit i when the input t is given together with the wrong key w to the locked circuit. N and M are the number of possible input and key combinations, respectively. Since N and M grow exponentially with the number of input and key bits, the value of P_i is often estimated. Entropy has a maximum value $max(H_i) = 1$ when $P_i = 0.5$. The entropy of a design is the sum of the entropy of each output bit:

$$H = \sum_{i=1}^{N} H_i$$

Where N is the number of output bits. It follows that the maximum value of H is equal to N. In order to say if an entropy value is good (i.e. close to the maximum) one must know the number of output bits of the considered design. To avoid this problem and ease the comparison between different designs we used the mean differential entropy:

$$\overline{H} = \frac{1}{N} \cdot \sum_{i=1}^{N} H_i = \frac{1}{N} \cdot \sum_{i=1}^{N} \left(P_i \cdot log\frac{1}{P_i} + (1 - P_i) \cdot log\frac{1}{1 - P_i} \right)$$

In the threat model that we consider, we suppose that the attacker is able to distinguish between data and control inputs and outputs. For this reason we assigned zero as differential entropy value (worst case value) to those solutions that induced the design to manage control signals incorrectly (i.e. never asserting ready or valid signals). We must avoid these solutions because they would allow an attacker to easily discard wrong keys.

We aim at maximizing the mean differential entropy making it as close as possible to 1. This is the case where $P_i = 0.5, \forall i$. In this situation the attacker cannot make any educated guess on the design functionality, leading to a probability of 2^{-K} (where K is the number of key bits) to guess the correct key.

3.4 Dependence Graphs

Dependence graphs were first proposed in [11] in 1972 to represent dependencies that occur within a program. In our work we are interested in Program Dependence Graphs (PDG) and System Dependence Graphs (SDG). A PDG is a directed graph that represents a program that consists of a single procedure. SDGs are an extension of PDGs that allow to represent programs with multiple procedures and procedure calls, they were first proposed in [10]. The SDG of a program is built by first building a PDG for each procedure of the program and then connecting the PDGs with edges that model procedure calls. The use of SDGs for hardware descriptions was first proposed in [36] for Model Checking. In order to use SDGs with hardware description languages, we need some considerations since the HDL computational paradigm differs fundamentally from the one of software languages [36]. In our work we use System Dependence Graphs to analyse the impact of obfuscation prior to simulation.

3.5 Summary

We propose a framework to optimize logic locking for a given security metric under area and key bits constraints. We operate at RTL as it allows to hide semantic information before it gets embedded into the netlist by logic synthesis optimization, while allowing our framework to be compatible with all design flows. We optimise the application of the semantic obfuscation techniques proposed in ASSURE [21] as they have been proved to be resilient towards structural attacks and allows us to concentrate on the output corruptability. We picked mean differential entropy as our security metrics as it quantifies the output corruptability. To pick the obfuscation points we reason on their impact on the obfuscated solution analyzing the System Dependence Graph. We structure our framework to have a set of scoring heuristics that can be combined to evaluate the obfuscation points. The obfuscation points with a higher score will be more likely to be selected for obfuscation.

Chapter 4

Approach: Heuristics for RTL Locking

In this chapter we describe the approach that we propose in this thesis. We discuss the main design decisions and the architecture of our solution, while the next chapters will go into more details about the core components of our solution.

4.1 Architecture & Design Decisions

We propose an RTL obfuscation framework (see Figure 4.1) to easily evaluate overheads and metrics of obfuscated designs and perform optimizations under area and key budget constraints.

The workflow begins by parsing the HDL code of the input design to be obfuscated in order to extract its Abstract Syntax Tree (AST). We then analyze the AST to extract the SDG and identify the obfuscation points. We apply a set of heuristics based on the SDG analysis to determine the *scores* of each obfuscation point, i.e., the likelihood to be a *obfuscation point* for our design. Each heuristic gives a score to each obfuscation point that can either be a positive (rewarding) or negative (punishing) value. We store the scores of each heuristic into a corresponding *score table*, i.e., a representation of the scores for all obfuscation points. Then, we can combine these score tables in a modular way. Given a combination of heuristics we build a *global score table* that is obtained by aggregating the values of the single score tables of specific heuristics. This global score table associates each

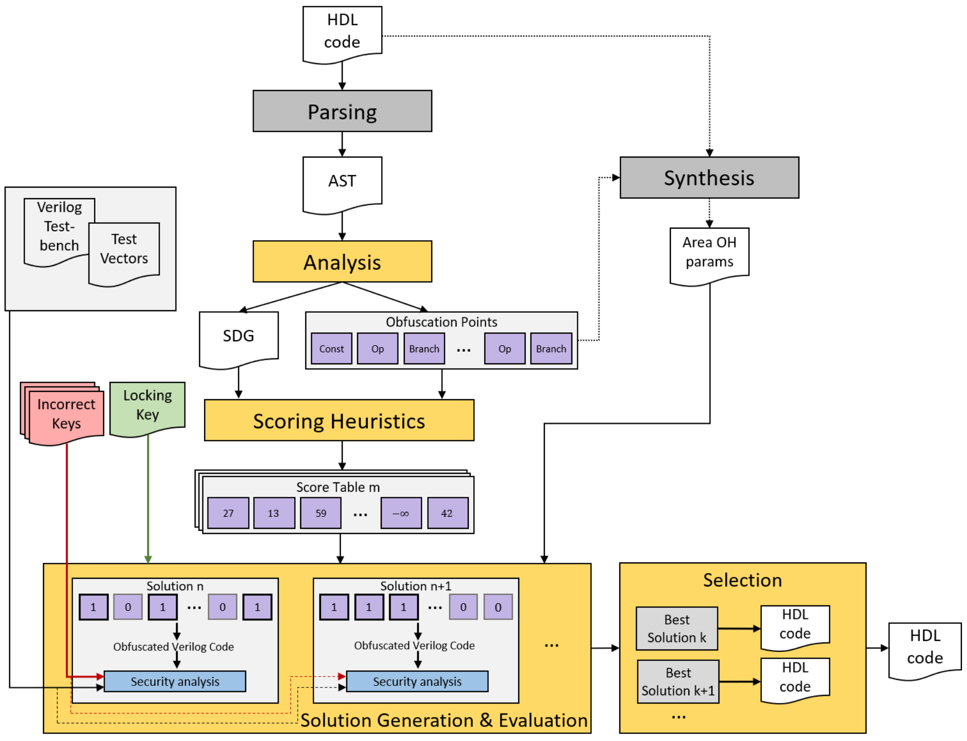
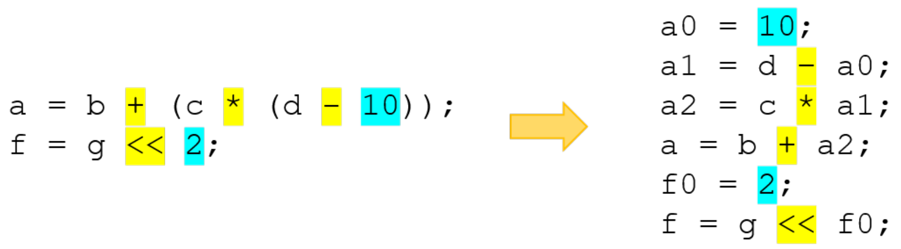


Figure 4.1: Framework flow



```
a = b + (c * (d - 10));
f = g << 2;
```

```
a0 = 10;
a1 = d - a0;
a2 = c * a1;
a = b + a2;
f0 = 2;
f = g << f0;
```

Figure 4.2: Statement transformation to get a single obfuscation point per statement

obfuscation point with its final score that is later used for selecting the ones with higher values.

The SDG has a node for each statement. A statement may have zero, one, or many obfuscation points. So, each obfuscation point is associated with a unique SDG node, while each SDG node is associated with an arbitrary number of obfuscation points. A possible alternative would have been to transform the AST to obtain statements with only one obfuscation point by introducing intermediate signals like illustrated in Fig. 4.2. This would prevent having multiple obfuscation points in the same SDG node, and would allow us to represent the design with better granularity dependencies. However, for associative operations, the order would depend on the order in which the statement is written. We believe that the benefits of this finer approach are not justified due to the higher complexity of the implementation and of the higher resulting number of SDG nodes. With our solution, all the obfuscation points that correspond to the same SDG node have the same score. To avoid obfuscating all the obfuscation points of a statement before moving to the next one, we scale the score of all the obfuscation points that share the same SDG node as follows:

$$S_{OPx} = \frac{x \cdot S_{OPx}}{n}, \quad x = 1, \ldots, n$$

Where $OP1$, ..., OPn are obfuscation points that share the same SDG node. The scaling is applied in a random order to avoid design dependent solutions. A possible alternative would have been to apply the score scaling following the topological order of the obfuscation points but then the solution would depend on the design structure, introducing an additional and undesired degree of freedom.

Given a score table, we can generate a solution in two ways. The first approach selects the obfuscation points with the highest scores until we reach the constraints. The second approach is probabilistic. We map the scores in the range [0.25, 0.75] and use these values as the probabilities of selecting

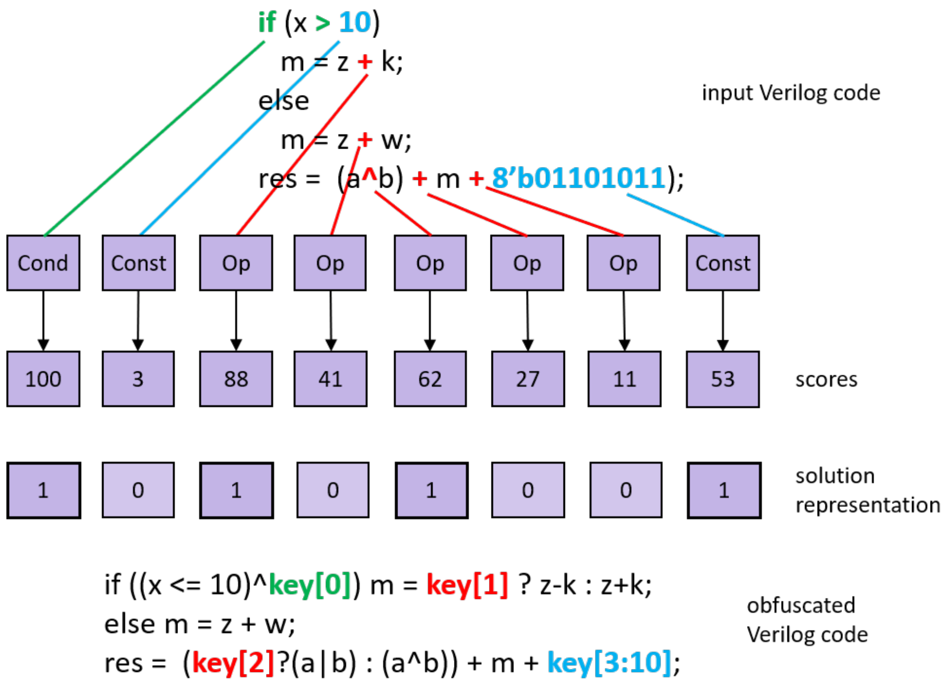


Figure 4.3: Obfuscation example

each obfuscation point. So the obfuscation point with the lowest score will be obfuscated with a probability of 25% while the obfuscation point with the highest score will be obfuscated with a probability of 75%.

The framework represents an obfuscation solution as a binary string where each element represents an obfuscation point, if the i^{th} element of the string is 1, the i^{th} obfuscation point is locked, otherwise it is not locked. The order of the obfuscation points is the one in which they are found in a depth first search on the AST of the design to be obfuscated. This representation allows us to separate the solution generation from the obfuscation and evaluation phase. Figure 4.3 shows an example of obfuscation highlighting obfuscation points, their scores, and the solution representation.

The solutions are evaluated measuring the mean differential entropy, the key size, and the estimated area overhead.

The mean differential differential entropy is estimated by running behavioural simulations of the obfuscated design. Before the evaluation phase begins, *golden outputs* are measured by simulating the original circuit with a set of input vectors. Then the same input vectors and a set of key vectors are passed to a test bench that compares the output of the obfuscated design with the golden outputs to estimate P_i. From the estimated values of P_i, the mean differential entropy is obtained using the following formula, first introduced in Section 3.3:

$$\overline{H} = \frac{1}{N} \cdot \sum_{i=1}^{N} \left(P_i \cdot log \frac{1}{P_i} + (1 - P_i) \cdot log \frac{1}{1 - P_i} \right)$$

Finding a method to estimate the area overhead is important because measuring it would require to perform the complete synthesis of the design, increasing the computational load.

In this way we can generate different solutions and evaluate them with our estimation method. Then only the best ones can be synthesized to check the results. The area overhead is estimated as follows:

$$\text{AreaOverhead} = \alpha \cdot C + \beta \cdot B + \gamma \cdot O$$

Where C, B and O are the number of bits used for obfuscating constants, branches, and operations, respectively. α, β and γ are parameters that can be either given by the designer or estimated by the framework. To estimate the overheads parameters, the framework measures the mean percentage overhead for each type of obfuscation point. To do so, it synthesizes and measures the area of the plain design and of three obfuscated designs, each of them obtained by obfuscating all the obfuscation points of the specific category. Let D_p be the plain design, D_c be the design obtained by obfuscating all and only the constants, D_b be the design obtained by obfuscating all and only the conditional branches, and D_o be the design obtained by obfuscating all and only the operations. Then α, β and γ are obtained as follows:

$$\alpha = \left(\frac{Area(D_c)}{Area(D_p)} - 1 \right) \cdot \frac{1}{\#key\_bits(D_c)}$$

$$\beta = \left(\frac{Area(D_b)}{Area(D_p)} - 1 \right) \cdot \frac{1}{\#key\_bits(D_b)}$$

$$\gamma = \left(\frac{Area(D_o)}{Area(D_p)} - 1 \right) \cdot \frac{1}{\#key\_bits(D_o)}$$

4.2 Summary

In this chapter we presented the architecture of our solution together with the design decisions made during our work. We extract the AST of the given design from its HDL description. Then we analyze the AST to build the SDG and identify the obfuscation points. The SDG is then analyzed by our heuristics, obtaining a score table that pairs each obfuscation point with its score. Different heuristics can be combined by summing their score tables together. Given a score table we select the obfuscation points using one of the two proposed methods. *In-order selection* picks the obfuscation points following the score order. *Probabilistic selection* maps the scores into a probability of selecting the obfuscation points and then randomly picks the points with the corresponding probability. In order to optimize the security metric under area constraint we propose a method to estimate the area overhead. In this way we do not need to perform the synthesis of each possible solution, reducing the computational requirements.

The area overhead estimation is based on a linear interpolation of area overhead introduced by applying the single obfuscation techniques.

Chapter 5

SDG Extraction

In this chapter we present a detailed overview of the System Dependence Graph (SDG) extraction procedure. We first present Program Dependence Graphs (PDG), giving the definition of *direct* and *inter-cycle* dependencies then we present the SDG extraction flow. An introduction to dependence graphs can be found in Section 3.4.

We built a tool for SDG extraction from Verilog[1] designs. We took inspiration from the considerations made in [36] to build SDGs for HDL, with some changes to adapt them to obfuscation analysis. Below is reported a brief description of how we extracted System Dependence Graphs from Verilog designs, the source code of this part of the implementation is Open-Source and is available at [15].

An SDG is obtained by integrating different PDGs that represent the procedures of a program. In case of Verilog, we can see `always` blocks as procedures that are called when a signal in the sensitivity list changes value. Continuous assignment statements can be seen as an `always` block that only contains the assignment and has all right hand-side signals in the sensitivity list.

5.1 PDG Extraction

Let G_{AB} be the PDG of an `always` block AB, then G_{AB} is a directed graph with several types of edges. The vertices $v_1, v_2, ..., v_n$ represent the assignment statement and control predicates that are present in AB. The edges represent dependencies between the nodes with an edge $e = (v_1, v_2)$ meaning

[1]Our SDG extraction procedure is not limited to Verilog designs, even though implementation details may vary due to language specific features.

that v_2 is dependent on v_1. Verilog presents two kind of assignments that can occur in an `always` block: blocking ($=$) and non-blocking ($<=$) assignments. Blocking assignments behave in a sequential way, like assignments in software languages, while non-blocking assignments present a more complex behaviour. When the `always` block is activated at a specific time-step, all the right hand sides (RHS) of non-blocking assignments are captured. Only at the end of the time-step the captured RHS values are assigned to the respective left-hand sides. The behaviour of non-blocking assignments makes it impossible to have a dependency between two non-blocking assignments at a given time step. It is possible though to have dependency occur between two different activations of the same `always` block. It is a common practice to use non-blocking assignments within clocked `always` blocks to model registers.

Let us consider the lines below within an always block sensitive on the positive clock edge:

```
A <= Z;
B <= A + Y;
C <= B + W;
```

The order of the non-blocking assignments does not affect the behaviour of the `always` block. At clock cycle X the value of `Z` is assigned to `A` but is not propagated to `B`. It is only at cycle $X+1$ that the value of `Z` is propagated to `B`. For this reason we distinguish between *direct dependencies* and *inter-cycle dependencies.*

Definition 1. *We have a direct dependency from v_1 to v_2 if and only if v_1 is a predicate vertex, and the execution of v_2 depends on the truth of v_1; or v_1 is a blocking assignment vertex with some signal X in the left-hand side that is used in v_2, and there exists an execution path from v_1 to v_2 along which there is no assignment to X.*

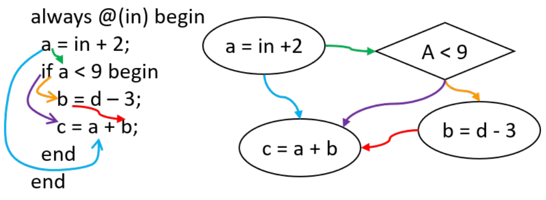
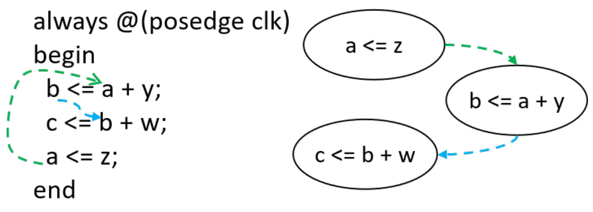


Figure 5.1: Example of direct dependencies

Definition 2. *We have an inter-cycle dependency from v_1 to v_2 if and only if v_1 is a non-blocking assignment with some signal X in the left-hand side that is used in v_2.*



```
always @(posedge clk)
begin
  b <= a + y;
  c <= b + w;
  a <= z;
end
```

Figure 5.2: Example of inter-cycle dependencies

We assume to have only one kind of assignment in each `always` block. This is a general practice in hardware design.

5.2 SDG Extraction Flow

We combine all PDGs of a module to obtain a representation with a single graph, the SDG. Then we combine all SDGs to obtain a single representation of the entire design.

Our goal is to obtain a single graph that represent the whole design, from the inputs to the outputs of the top module.

We first extract the SDGs of all the modules leaving behind the of instantiation of sub-modules. We then perform a *flattening* step to put together the SDGs of all modules. If a module is instantiated multiple times, we make a clone of its SDG for each of its instances. In this way the final SDG can accurately represent the dependencies of a sub-module taking into consideration its actual position in the design.

Figure 5.3 illustrates the workflow to extract the SDG of a Verilog module. We analyse each module extracting a PDG from each `always` block, a continuous assignment vertex for each continuous assignment, an input vertex for each input, and an output vertex for each output. Then we proceed adding edges between all these entities merging them into an SDG:

- For each input i we need to check if any assign or condition node n depends on i, if so we add a direct dependency from the input node of i to n.

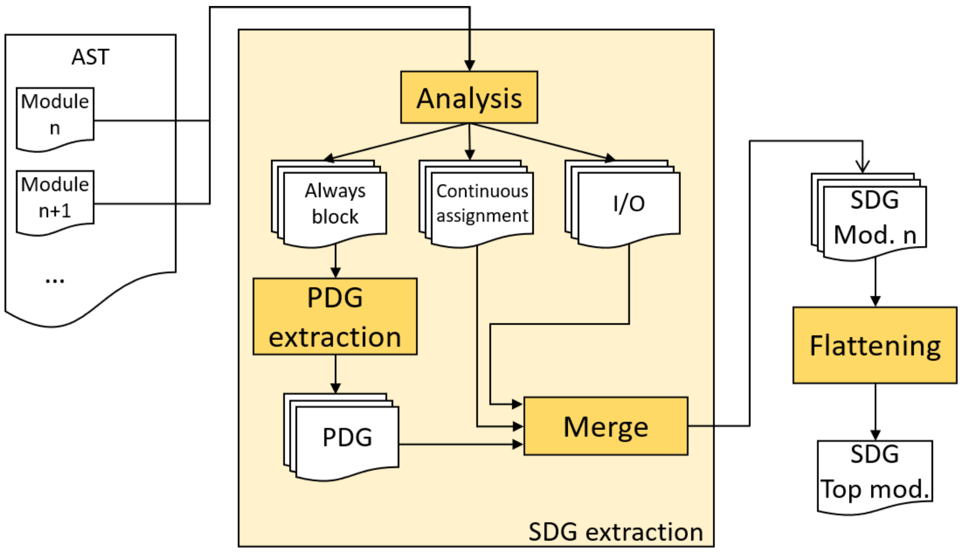
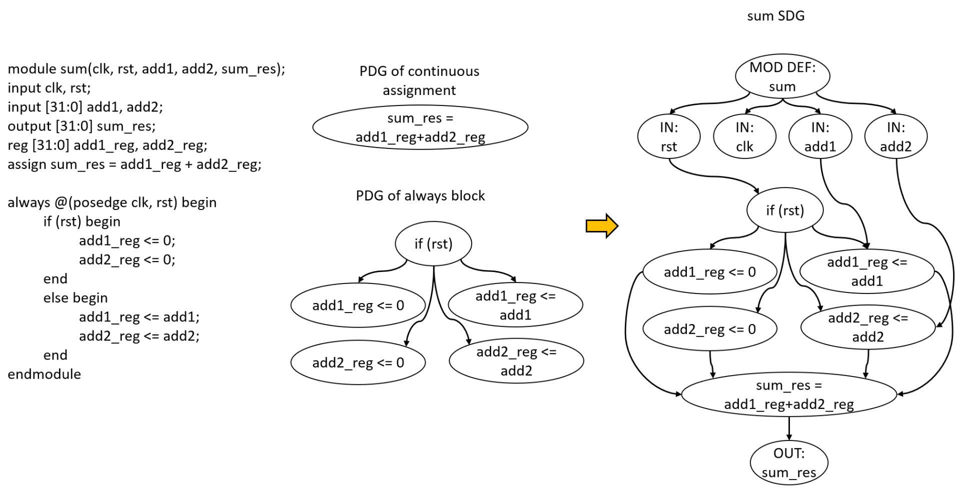


Figure 5.3: SDG extraction flow

- For each continuous assignment a we need to check if any other assign, condition, or output node n depends on the rhs of a, if so we need to add a dependency edge from a to n.
- For each PDG P for each statement s in P we need to check if any assign, condition (that do not belong to P), or output node n depends on s, if so we add a dependency edge from a to n.

When adding a dependency edge from a given assign vertex (of any kind) v_1 to a vertex v_2 within a PDG P_1, we add a direct dependency edge if there is some signal X in the lhs of v_1 and X is used in v_2 and X is in the sensitivity list of P_1. We add an inter-cycle dependency edge if there is some signal X in the lhs of v_1 and X is used in v_2 and X is not in the sensitivity list of P_1. Figure 5.4 shows an example of merge between two simple PDGs to obtain the SDG.

When a module instantiates a sub-module we insert a "placeholder vertex" connected with a *coupling vertex* for each input/output of the sub-module. Coupling vertices represent the port mapping for the sub-module. After extracting a SDG from each module we perform flattening starting from the top module. Each instance vertex is substituted with a clone of the SDG of the corresponding module and coupling nodes are connected with the corresponding inputs and outputs nodes with direct dependency edges (Fig. 5.5). The flattening procedure is recursive, for each instance node that is found we check in a computed table (or software cache) if we

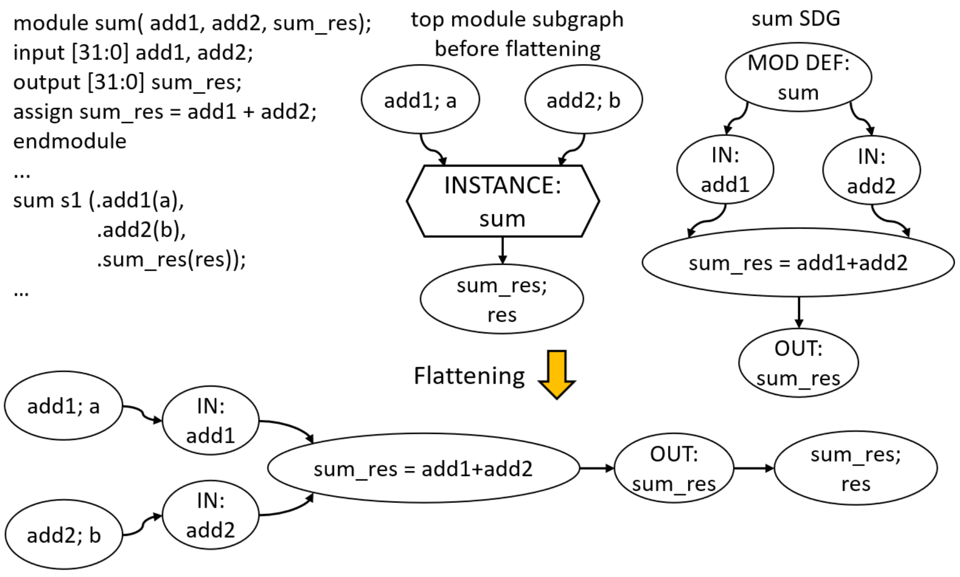


```
module sum(clk, rst, add1, add2, sum_res);
input clk, rst;
input [31:0] add1, add2;
output [31:0] sum_res;
reg [31:0] add1_reg, add2_reg;
assign sum_res = add1_reg + add2_reg;

always @(posedge clk, rst) begin
        if (rst) begin
                add1_reg <= 0;
                add2_reg <= 0;
        end
        else begin
                add1_reg <= add1;
                add2_reg <= add2;
        end
endmodule
```

Figure 5.4: Example of PDGs merging to obtain the SDG

have a flattened representation of the needed module. If the module has not been flattened yet, we proceed by flattening it, otherwise we return a clone of the flattened module SDG. By returning a clone of the SDG we are sure not to have problems with those designs that present multiple instantiations of the same module, moreover at the end we have a list with the flattened version of each module that may be interesting in some applications.



```
module sum( add1, add2, sum_res);
input [31:0] add1, add2;
output [31:0] sum_res;
assign sum_res = add1 + add2;
endmodule
...
sum s1 (.add1(a),
        .add2(b),
        .sum_res(res));
...
```

Figure 5.5: Example of module flattening

5.3 Summary

In this chapter we have seen how we can build a complete representation of the dependencies in a design following a divide-and-conquer approach and starting from smaller representations. We first build a representation for each `always` block, the Program Dependence Graph (PDG). We combine the PDGs of a module together with its continuous assigns and input/output to obtain the System Dependence Graph (SDG) of the module. In this step we represent sub-modules with a placeholder node. Once we have the SDGs of all modules, we combine them with a flattening step to obtain the complete SDG of the top module.

Our SDG representation allows us to represent dependencies of the whole design in a single graph. We leverage the SDG representation to build our scoring heuristics.

Chapter 6

Scoring and Selection Heuristics

We propose four heuristics that give a score to the obfuscation points based on an analysis of the SDG. The higher is the score of an obfuscation point, the higher is the probability to select it.

A scoring function can be rewarding, i.e. increasing the score of an obfuscation point, or punishing, i.e. decreasing its score.

We identified two main categories of scoring functions: local and global functions. Local functions explore the SDG up to a certain distance from each obfuscation point whereas global functions do not bound the exploration of the SDG.

The scoring functions are composable, i.e. any subset of the proposed scoring functions can be used to compute the scores that are in turn used to rank the obfuscation points.

When combining different functions, the final score table is obtained by summing together all the score tables obtained by the given heuristics. To avoid having one scoring heuristic dominating all the others, we normalized all the scores for each scoring heuristic in the range [0, 100]. Figure 6.1 shows an example of heuristic combination.

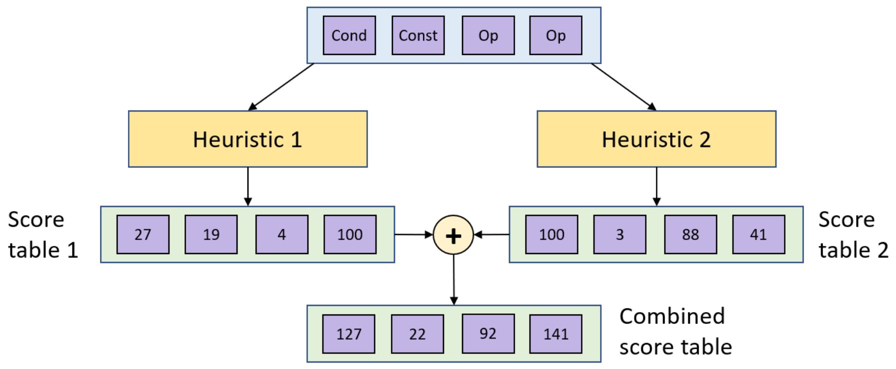


Figure 6.1: Example of composed heuristic

6.1 Scoring Heuristics

6.1.1 Control Disabling

Obfuscating points that influence control signals yields weaker solutions. In fact it would be easy for an attacker to discard invalid keys if the circuit never reaches a ready state or signals a valid output. This heuristic has the goal of finding these points that weaken the solution and disabling them. To disable the points we assign them a value of $-\infty$. We take as argument a set of controlling signals classified as input and output signals. These arguments are defined by the designer in a setting file passed to the framework as the name of controlling signals can vary across designs.

For the controlling inputs we identify and disable all the conditions that are directly dependent on any control signal. In fact disabling obfuscation points at any distance from a point dependent on a controlling input may disable the entire design. In the cases that we encountered was sufficient to disable those conditions that control the first transition of the finite state machine of the design.

For controlling outputs we identify each controlling output, then we explore the SDG going backwards and we disable all the obfuscation points that we find. Those are the points that influence the controlling outputs.

This allows us to avoid obfuscating those points that would cause simulation failures, yielding entropy 0 by definition. Figure 6.2 shows an example of Control Disabling with an input control signal `valid`.

6.1.2 Bounded (Direct) Children

The idea at the base of this heuristic is that an obfuscation point that influences a wide portion of the design has a greater impact on the obfuscation

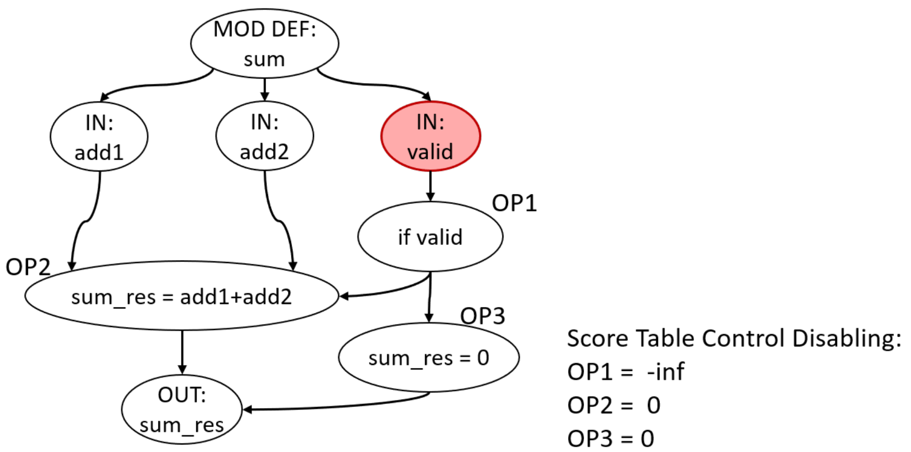


Figure 6.2: Example of Control Disabling with an input control signal `valid`

result than an obfuscation point that influences only a small design portion. We take into consideration two variants. One only considering direct dependencies and one considering direct and inter-cycle dependencies. Considering inter-cycle dependencies introduces the problem of cycles in the search phase, so we need to keep track of the visited nodes. This is not the case when the visit is limited to direct dependencies if the design is well written and does not present combinational loops. This heuristic is bounded to a maximum distance from the obfuscation point under evaluation as on big designs a global evaluation may be too complex. Given a distance D, for each obfuscation point O, the bounded children function returns the number of (direct) assignments and conditions up to a distance D from O that are dependent from O. This scoring process favors obfuscation points that have a higher propagation in the design. In fact, a node with a high number of children in the dependence graph, is a node that influences a relevant portion of the design. These points have a higher probability of having a wider influence on the outputs. Figure 6.3 shows an example oh application of the bounded children heuristic with both direct and inter-cycle variants. We highlighted the nodes that contribute to the score of the different evaluations.

6.1.3 Bounded Parents

The Bounded Partents heuristic is based on the idea that if a big design portion converges in an obfuscation point, by selecting it we can hide all the information behind that point. This heuristic aims at selecting the points that will corrupt a signal that has already been trough some work in order to

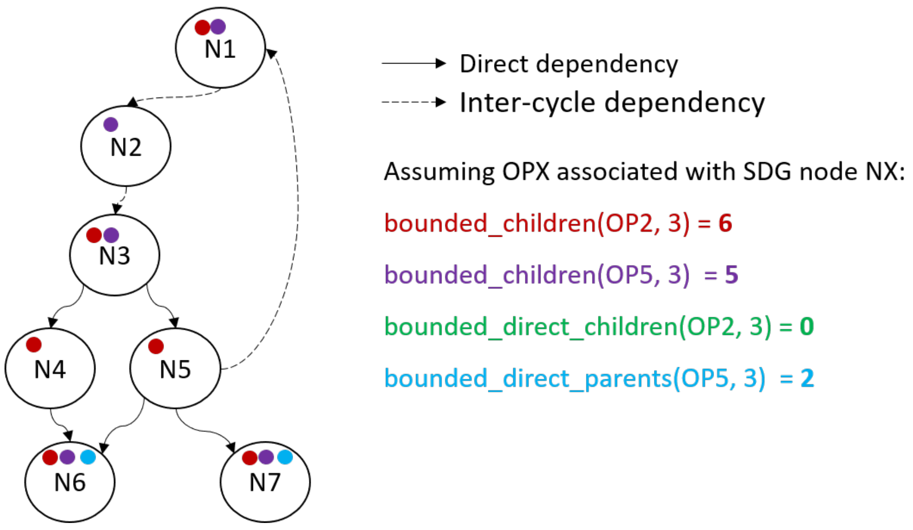

N1
N2
N3
N4
N5
N6
N7
Direct dependency
Inter-cycle dependency
Assuming OPX associated with SDG node NX:
bounded_children(OP2, 3) = 6
bounded_children(OP5, 3) = 5
bounded_direct_children(OP2, 3) = 0
bounded_direct_parents(OP5, 3) = 2

Figure 6.3: Example of bounded children heuristic

nullify such work. Also, selecting a point that depends on a high number of obfuscation points highers the probability of creating a chain of obfuscation points (a series of dependent obfuscation points). Creating a chain of obfuscation points amplifies the effects on output results. In fact to obtain the correct value on the output bits influenced by a chain of obfuscated points, the whole key section controlling the chain must be correct. This heuristic is bounded to a maximum distance from the obfuscation point on which it is evaluated. This limits the complexity of the visits as in big designs a global evaluation may be too complex. This heuristic only considers direct dependencies as after our first evaluation round the inter-cycle dependency showed poor results. That is due to the fact that registered signals usually have at least two different assignments, one for reset and one for normal operation. This alters the parents counts when considering the inter-cycle dependencies. Given a distance D, for each obfuscation point O, the bounded parents heuristic returns the number of obfuscation points up to a distance D from O that converge in O. This function favors the obfuscation points that have a high convergence. These points are the most convenient to take in order to corrupt a signal dependent on a big design portion and build longer sequences of obfuscated points. Figure 6.4 shows an example of the bounded parents heuristic highlighting the nodes that contribute to the score of the different evaluations.

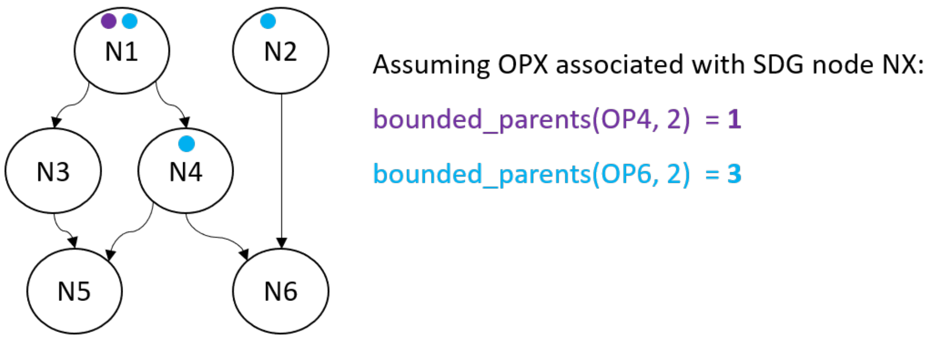

N1
N2
N3
N4
N5
N6
Assuming OPX associated with SDG node NX:
bounded_parents(OP4, 2) = 1
bounded_parents(OP6, 2) = 3

Figure 6.4: Example of bounded parents heuristic

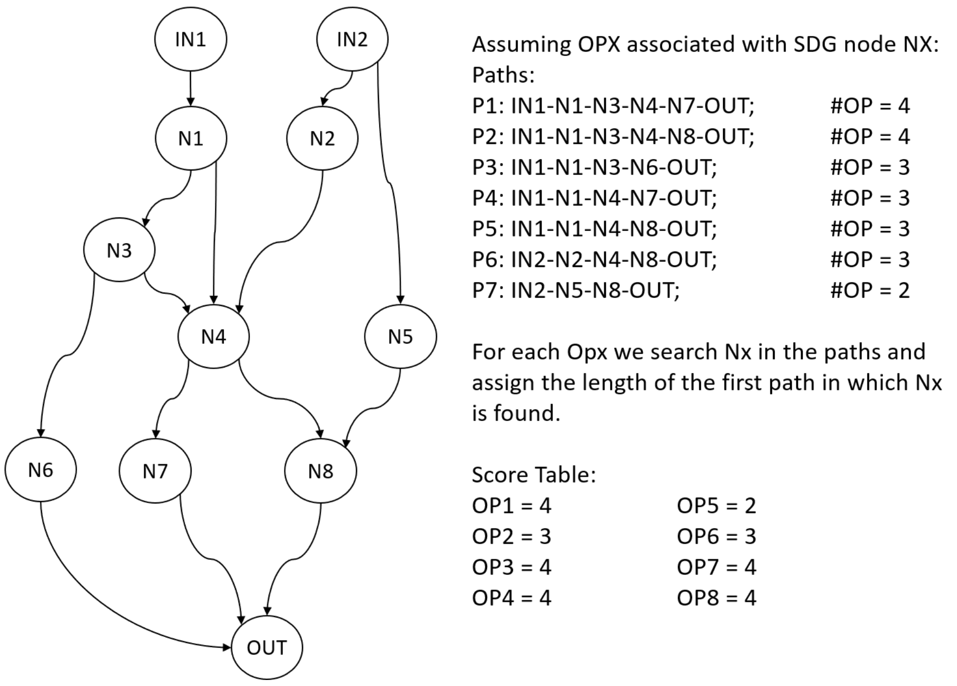
6.1.4 Max I/O Path Length

The idea behind the Max I/O Path Length heuristic looks for chain of obfuscation points in an exhaustive way. To identify long obfuscation point chains, this heuristics visits all the paths from each input to each output counting the number of obfuscation points in each path. This heuristic is not bounded and follows both direct and inter-cycles dependencies, for this reason it needs to keep track of visited nodes to avoid loops. The heuristic assigns to each obfuscation point O a value equal to the maximum number of obfuscation points that can be found in a path from an input to an output that passes through O. This heuristic performs an accurate search focused on the length of obfuscation chains. It performs visits across the whole graph so the complexity is higher than bounded heuristics. Figure 6.5 shows an example of application of Max I/O Path Length.

6.2 Selection Methods

6.2.1 In-order

The idea behind this selection method is simple: we order the obfuscation points from highest to lowest scores, then we start obfuscating from the first obfuscation points until we reach our constraints. Given a score table, the generated solution using this method is deterministic. However, the stochastic nature of the score scaling of obfuscation points belonging to the same SDG node makes the whole process from SDG to obfuscated solution non deterministic.



Assuming OPX associated with SDG node NX:
Paths:

P1: IN1-N1-N3-N4-N7-OUT;	#OP = 4
P2: IN1-N1-N3-N4-N8-OUT;	#OP = 4
P3: IN1-N1-N3-N6-OUT;	#OP = 3
P4: IN1-N1-N4-N7-OUT;	#OP = 3
P5: IN1-N1-N4-N8-OUT;	#OP = 3
P6: IN2-N2-N4-N8-OUT;	#OP = 3
P7: IN2-N5-N8-OUT;	#OP = 2

For each Opx we search Nx in the paths and assign the length of the first path in which Nx is found.

Score Table:

OP1 = 4	OP5 = 2
OP2 = 3	OP6 = 3
OP3 = 4	OP7 = 4
OP4 = 4	OP8 = 4

Figure 6.5: Example of Max I/O Path Length application

6.2.2 Probabilistic

This selection method selects obfuscation points in a probabilistic way. The scores, where positive, are mapped in the range $[0.25, 0.75]$. Negative score are put equal to 0. These new values correspond to the probability of selecting the corresponding obfuscation point. We then scan the obfuscation points in topological order and for each obfuscation point p we generate a random number n in the range $[0, 1]$, if n is less then the score of p we use the point. We continue to iterate over the obfuscation points until we meet the constraints or we do not update the solution for a given number of iterations. This selection methods increases the probability of selecting obfuscation points that are close to each other if they have a good enough score. Obfuscation points close to each other are more likely to introduce smaller area overhead because the extra logic can be optimized more easily by the logic synthesis tools.

Figure 6.6 shows an example of obfucation points selection starting from a score table.

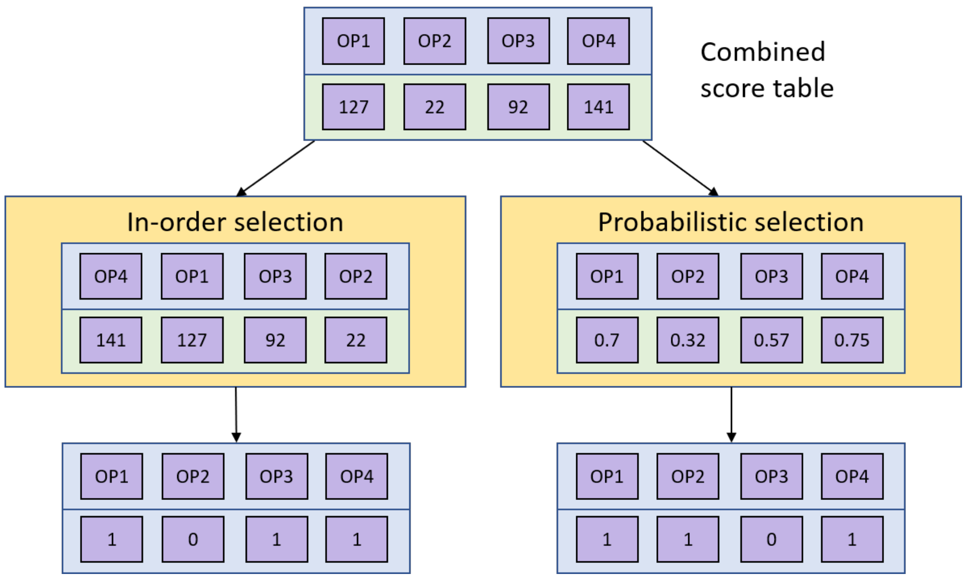


Figure 6.6: Example of obfuscation points selection

6.3 Summary

In this chapter we have seen the base methods that we can use to analyze the System Dependence Graph to select the obfuscation points. We propose four scoring heuristics and two selection methods. The scoring heuristics analyze different aspects of each obfuscation point and assign a score to each of them. These base methods can be combined together to obtain several obfuscation procedures. This framework allows us to generate solutions using different combinations of scoring heuristics and any of the two selection methods. The framework then evaluates all the alternatives and proposes the best one as the final candidate solution.

Chapter 7

Implementation and Evaluation

In this chapter we go through the implementation details of our solution and its experimental evaluation.

- Section 7.1 illustrates the implementation details, showing the technology used, and an overview of the use of the framework.
- Section 7.2 presents the experimental setup used for the experiments and then present and comment the results.

7.1 Implementation Details

We implemented a prototype framework leveraging Pyverilog [33], a Python-based Hardware Design Processing Toolkit for Verilog HDL. We opted to use Python (version 3.6+) as it is ideal for fast prototyping and offered both a parser and an evolutionary computation framework. The downsides of Python are mainly performance related. Better performances can be obtained by implementing more optimized solutions using lower-level languages. We felt that the trade-off between development time and execution time was worth using Python instead of a lower-level language. We used Pyverilog to parse the Verilog design and create its abstract syntax tree (AST). The SDG extraction explores the AST leveraging the `NodeVisitor` class defined in Pyverilog. We defined new visitors to extract the data needed to build the SDG. The SDG extraction procedure was built to be independent of the rest of the framework. In this way it can be used as a library in any project that requires design analyses. In Figure 7.1, we show the UML of our SDG extraction implementation.

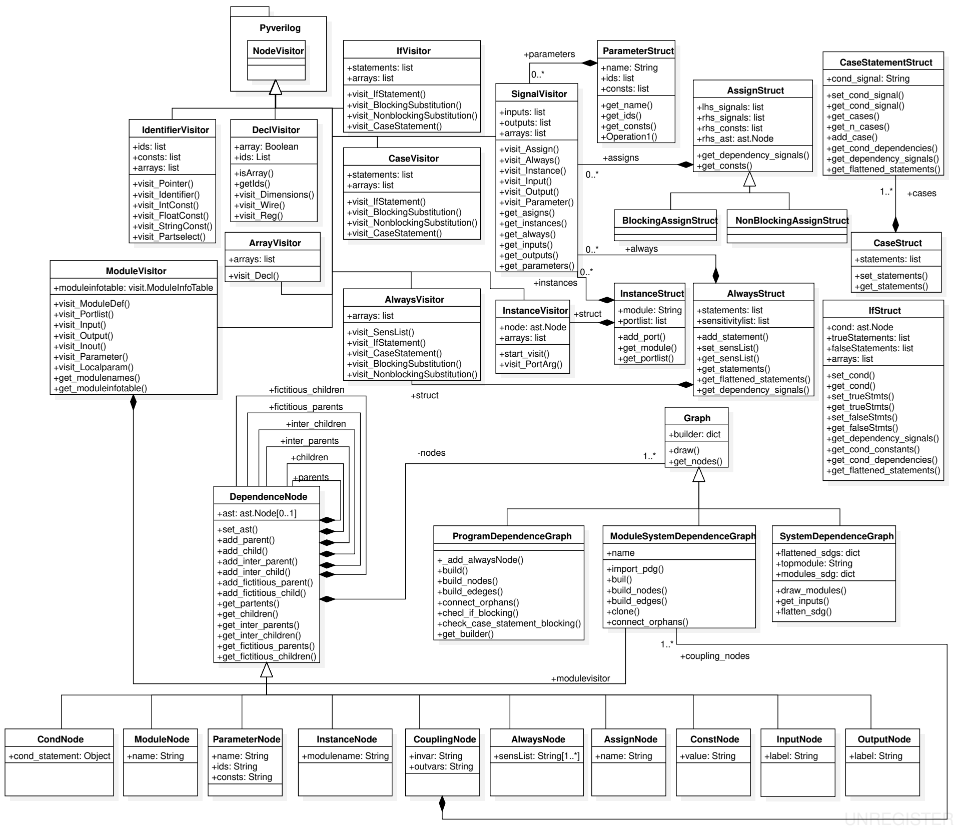


Figure 7.1: UML of the SDG extraction implementation.

As it is showed in the UML diagram, we keep track of different kinds of nodes via inheritance. Our heuristics do not use that information, but we felt it was valuable information for other possible uses of the SDG.

Once the obfuscation step is finished, our framework uses Pyverilog to generate the obfuscated Verilog description ready for logic synthesis and simulation. We then estimate the differential entropy by running behavioral simulations on the solutions. The test bench evaluates the outputs of different primary input and key combinations and outputs the mean differential entropy in a text file that is then read by our framework. Appendix A shows a test bench template, also explaining how to build one for a new design.

To implement the genetic algorithm for design space exploration we used the DEAP [9] framework, a state-of-the-art evolutionary computation framework implemented in Python as well. The framework runs the synthesis of the designs using Synopsys Design Compiler R-2020.09-SP1 targeting the Nangate 15nm ASIC technology at standard operating conditions (25°C). For the behavioral simulations we used Synopsys vcs. The choice of the RTL tools is arbitrary and the framework can be adapted to work with different tools. We decided to use these tools as they are used in industry and provide a real use-case scenario. Figure 7.2 shows the tool flow, highlighting the main technologies used for the components. The following is an output snippet from running the framework with FIR filter with 172 key bits budget:

```
Design parsing completed in 0.0478 seconds
- Num. of obfuscation points = 34
- Constants = 10
- Operations = 24
- Branches = 0

BEST INDIVIDUAL: 1000011110101110101000010110100001
BEST HEURISTIC: PROB NCHILD
BEST BITS: 172
BEST AREA: 0.8700305931940606
BEST ENT: 0.999977
AREA ESTIMATION ERROR: -0.16205798015968464
AREA MEASURED: 0.707972613034376
```

In the output log there is also information about the obfuscation steps and the results of all techniques tried. As showed in the log snippet above, at the end it reports the information about the best solution. In the output folder there are all the files related to the evaluated solutions.

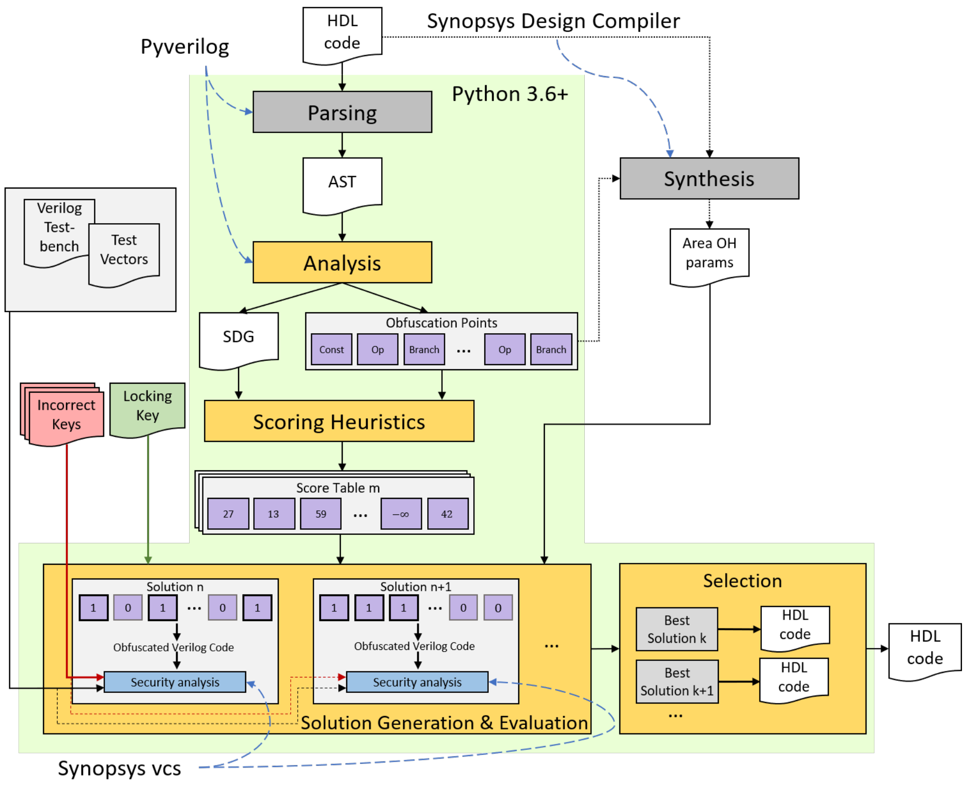


Figure 7.2: Framework flow highlighting the technologies used

7.2 Evaluation

7.2.1 Design of Evaluation

We evaluated our framework by running it to optimize obfuscation for a set of designs with different key budgets. We evaluated both the mean differential entropy and the area estimation. We picked five designs from the MIT-LL Common Evaluation Platform (CEP) [18] to be evaluated with our framework. Two of these benchmarks (FIR and IIR) were generated using SPIRAL [32], a hardware generator. The selected benchmarks are a subset of those used in [21] as we had to build a test-bench to measure the differential entropy for each third-party design. For a design house it should not be a problem to adapt a test bench to work with our framework.

Table 7.1: Characteristics of RTL benchmarks

Design	Modules	Const	Ops	Branches	# Bits	SDG nodes
FIR	5	10	24	0	344	157
IIR	5	19	43	0	651	231
SHA256	3	159	36	2	4,992	619
MD5	2	150	50	1	4,533	829
DES3	11	523	3	775	2,990	3,745

Table 7.1 reports the number of obfuscation points for each category, the maximum number of key bits, and the number of nodes of the SDG for the considered benchmarks.

To calculate the mean differential entropy, we estimated the output probability running 10,000 simulations obtained by combining 100 random keys with 100 random inputs.

The first empirical results showed that the behaviour of the heuristics is dependent on the design and the constraint. It is difficult to predict in advance which heuristic will perform better in a given case. Since the techniques are not computationally intensive we run a set of scoring heuristic combinations, giving as outcome the best result that we obtain.

We compared our results against a design space exploration approach at RTL (like the one presented in [22] for HLS), topological-order obfuscation (like ASSURE [21]), and random obfuscation (where we select obfuscation points completely at random).

From the first tests, increasing the distance parameter for the bounded heuristics has a flattening effect on the obfuscation points, reducing the performances. For this reason, we set a distance of 3 for the bounded children heuristic and 2 for the bounded parents.

For each benchmark, we ran the framework with 20 different constraints on the key budget as follows: 1, 2, 3, 4, 5, 7.5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 60, 70, 80, 90, 100% of the maximum number of key bits. The framework was configured to optimize the mean differential entropy of the design and to evaluate the area overhead. The best solution was selected as the one with mean differential entropy within 0.001 from the best value and with the lowest estimated area overhead. We evaluated mean differential entropy with respect to topological order obfuscation, random obfuscation, and design space exploration with a genetic algorithm. We looked at which technique achieves the best results more often for different key budgets (1-5%, 7.5-25%, 30-50%, 60-100%, 1-100%). We also evaluated the error of the area estimation of the best results by running the synthesis of the obfuscated solutions and comparing them with the estimated value.

We ran the following combinations of scoring heuristics with both in-order and probabilistic solution generation:

- Control disabling and bounded direct children
- Control disabling and bounded children
- Control disabling and bounded parents
- Control disabling and max I/O path length
- Control disabling, bounded direct children, bounded parents and max I/O path length
- Control disabling, bounded children, bounded parents and max I/O path length

We also ran control disabling alone with probabilistic generation, resulting in a random solution excluding controlling points. For the different combinations we used the naming scheme reported in Table 7.2.

If the solution was generated with the probabilistic approach, we also added "PROB" as prefix. We omitted CONTR DIS in the labels since it is used in all combinations.

7.2.2 Metrics

We evaluated our framework by comparing the differential entropy of solutions yielded from our heuristics with the one yielded by topological obfuscation, random obfuscation, and design space exploration. We evaluated the

Table 7.2: Naming scheme for heuristics.

Heuristic	Abbreviation
Control Disabling	CONTR DIS
Bounded children	NCHILD
Bounded direct children	DCHILD
Bounded parents	NPAR
Max I/O path	LPATH

error of area estimations by comparing them with the real area overhead. We estimated differential entropy with the following equation:

$$\overline{H} = \frac{1}{N} \cdot \sum_{i=1}^{N} \left(P_i \cdot log\frac{1}{P_i} + (1 - P_i) \cdot log\frac{1}{1 - P_i} \right)$$

Where P_i is the probability of output i being equal to 1, therefore $P_i \in [0, 1]$ and N is the number of output bits.

$$P_i = \frac{\sum_{w=1}^{W} \sum_{t=1}^{T} OUT[i]_t \bigoplus OUT[i]_{t,w}}{W \cdot T}$$

Where $OUT[i]_t$ is the correct value of the output bit i when the input t is given to the unlocked circuit, and $OUT[i]_{t,w}$ is the value of the output bit i when the input t is given together with the wrong key w to the locked circuit. W and T are the number of considered input and key combinations, respectively. In our tests we picked $W = T = 100$ for a total of 10,000 combinations of randomly-generated inputs and keys.

7.2.3 Results

From the first runs where we ran the heuristic combinations alone, there is not a single heuristic that outperforms the others. Figure 7.3 shows how many times each heuristic yielded a best solution across all designs. All heuristic combinations yielded a best solution at least twice. Certain heuristics are more likely to yield a best result when used within a certain key budget interval. The 60-100% constraint interval is predominated by the sole Control Disabling heuristic with probabilistic generation. This may be due to noise in the lower scoring points.

Figure 7.4 shows the entropy results for each key budget, highlighting the technique that generated the best solution. Different techniques show to work better on different designs and different key budget intervals. Figure 7.5 shows the comparison with topological obfuscation. The results

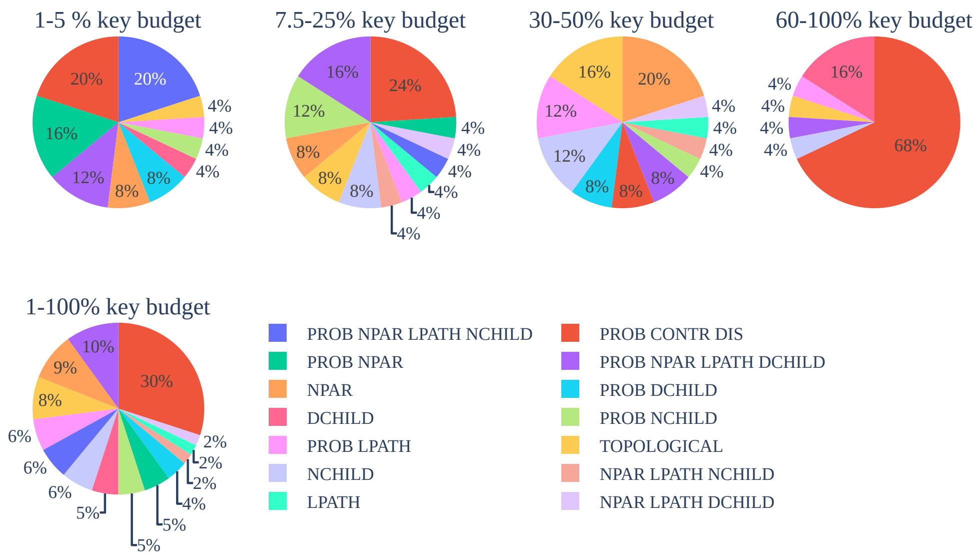
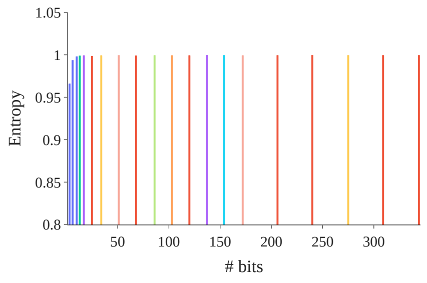


Figure 7.3: Technique frequencies for key budget intervals across all designs

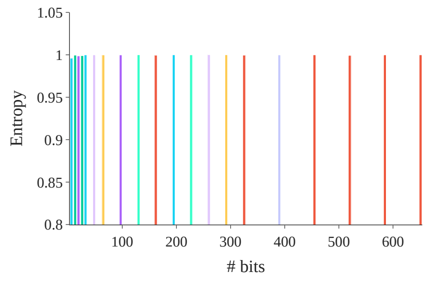
show that topological obfuscation presents a higher variability in the entropy results and yields a best solution only in 8% of the cases. Figure 7.6 shows the comparison with random obfuscation. With designs that present control signals, random obfuscation often selects points that invalidate the solution. Figure 7.7 shows how modifying the order of the sub-modules instances results in a huge difference in the mean differential entropy values of the topological order solution while the composable heuristics results only change slightly where the best solution comes from a probabilistic generation.

Figure 7.8 shows the evaluation of the area estimation. For FIR, IIR, and SHA256 the area estimation is very close to the actual area across all points. On the other hand, DES and MD5 show a large discrepancy between the estimated area and the measured one. Mean values are reported in Table 7.3. Obfuscating sparse points yields unpredictable changes in the synthesis optimization phase sometimes causing larger overheads. The area evaluation method may be improved to take additional features into consideration for designs that present this behaviour.

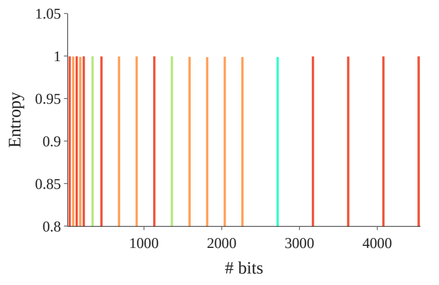
Table 7.4 shows a comparison with the genetic algorithm used for de-



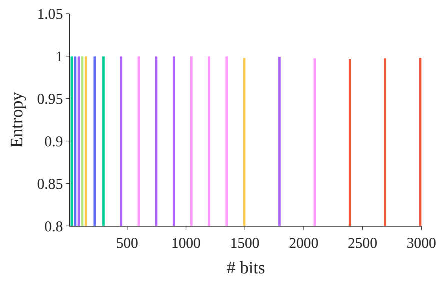
(a) FIR



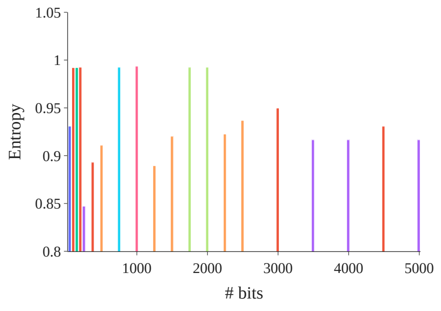
(b) IIR



(c) MD5



(d) DES3



(e) SHA256

🟦 PROB NPAR LPATH NCHILD	🟥 PROB CONTR DIS
🟩 PROB NPAR	🟪 PROB NPAR LPATH DCHILD
🟧 PROB DCHILD	🟦 NPAR
🟥 PROB NCHILD	🟩 PROB LPATH
🟪 DCHILD	🟨 TOPOLOGICAL
🟦 NCHILD	🟧 NPAR LPATH DCHILD
🟩 NPAR LPATH NCHILD	🟪 LPATH

Figure 7.4: Differential entropy results, highlighting the heuristic yielding the best solution

Table 7.3: Area overhead mean relative error (m.r.e.)

Design	Area Overhead m.r.e. [%]
FIR	10.86
IIR	16.33
MD5	75.33
DES3	37.24
SHA256	6.69
ALL	29.29

sign space exploration. The two approaches obtain values that are very close with the proposed combined heuristic approach while being 100 to 400 times faster, where applicable. The reported time for composable heuristics is the total time to evaluate the all of the 14 heuristic combinations that we considered in the evaluation settings for a given key budget, for the design space exploration approach is the time of an exploration for a given key budget. We did not consider the time to calculate the area estimation parameters as it is done only once. However, for the composable heuristics, we also counted the time for the synthesis of the best solution. This result shows how the proposed methodology scales much better for large designs.

7.2.4 Discussion

The proposed obfuscation framework aims at optimizing a security metric under either area or number of key bits constraints. Operating at RTL makes our solution compatible with all IC design flows.

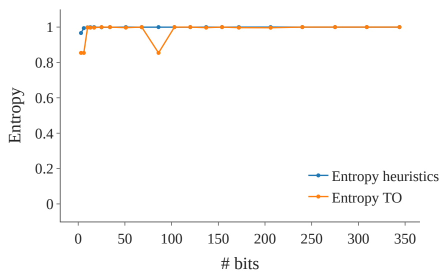
The framework performs its best in highly-constrained scenarios, which are also the most relevant in real-case scenarios.

Our methodology drastically decreases the computational power required compared to existing techniques, enabling us to target larger designs.

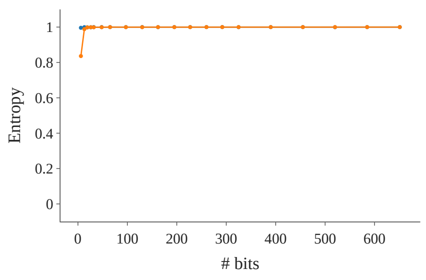
The entropy results are better than the ones obtained by applying obfuscation in topological order for 92% of the cases. In addition, our results do not depend on the design structure, taking away responsibility from the designer.

Table 7.4: Comparison with DSE approach at 4 key budget constraints

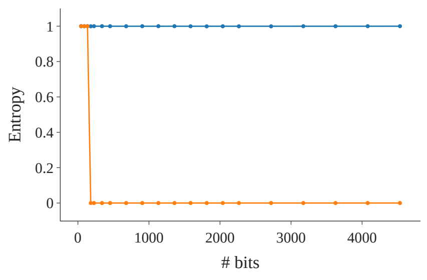
Design	Composable Heuristics					DSE				
	Mean differential entropy				Time [min]	Mean differential entropy				Time [min]
	25%	50%	70%	100%		25%	50%	75%	100%	
FIR	0.999853	0.999967	0.999746	0.999935	2	0.999963	1.000000	1.000000	0.999997	240
IIR	0.999236	0.999582	0.999633	0.999842	3	0.999964	0.999999	0.999999	0.999993	360
MD5	0.999939	0.999364	0.999832	0.999832	3	0.999954	0.999952	0.999952	0.999952	450
DES3	0.999947	0.999513	0.998356	0.996788	4	0.999963	0.999957	0.999960	0.999960	600
SHA256	0.993654	0.993307	0.951003	0.951003	3	0.999540	0.999665	0.999665	0.999665	1300



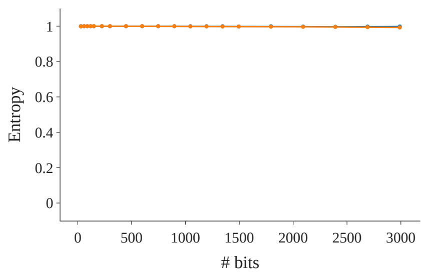
(a) FIR



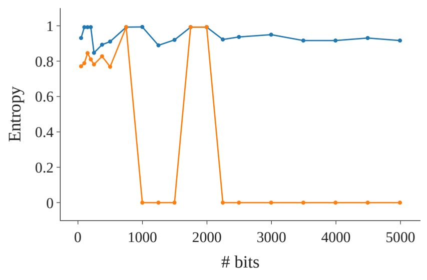
(b) IIR



(c) MD5

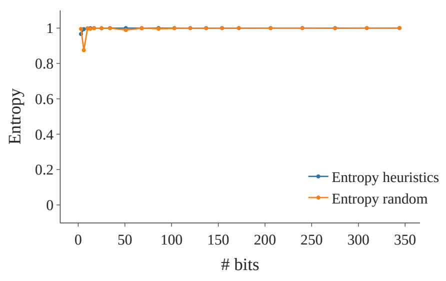


(d) DES3

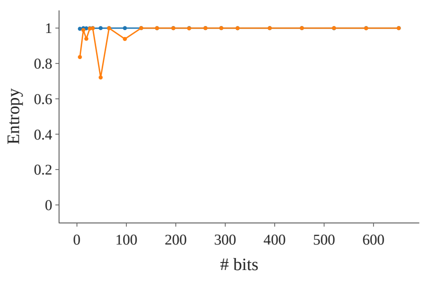


(e) SHA256

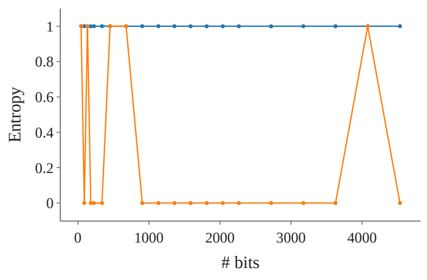
Figure 7.5: Differential entropy comparison with topological obfuscation



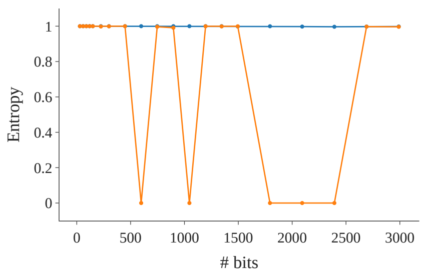
(a) FIR



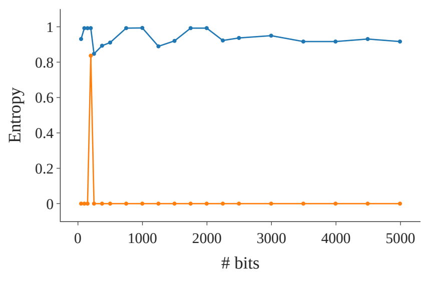
(b) IIR



(c) MD5



(d) DES3



(e) SHA256

Figure 7.6: Differential entropy comparison with random obfuscation

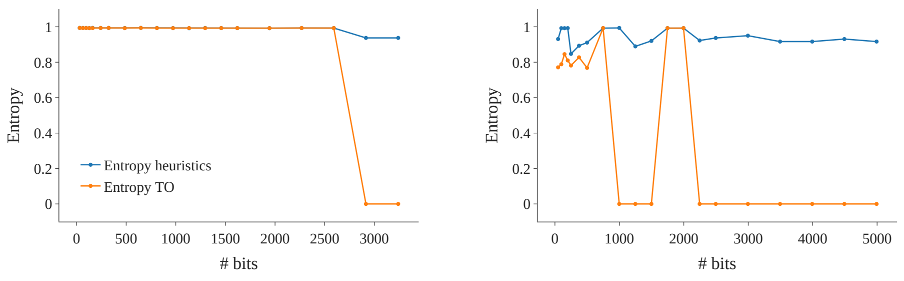
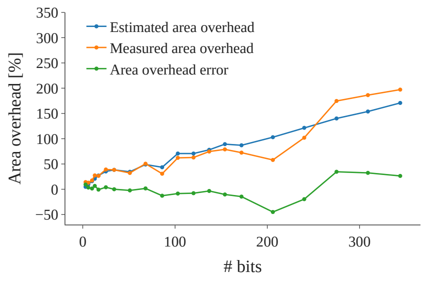
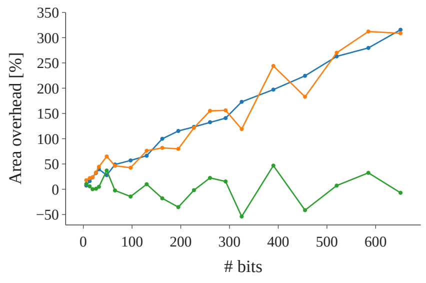


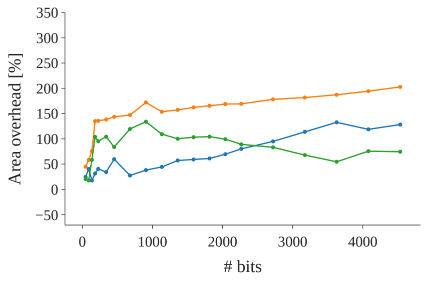
Figure 7.7: Impact of inverting sub-module instances in SHA256



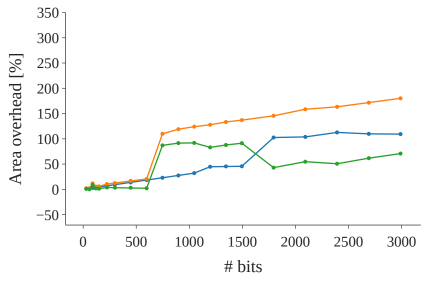
(a) FIR



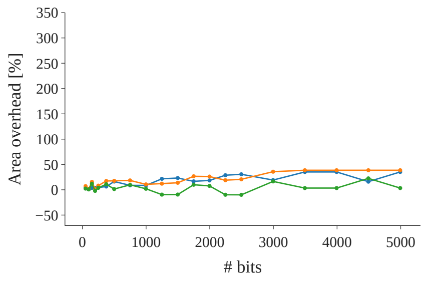
(b) IIR



(c) MD5



(d) DES3



(e) SHA256

Figure 7.8: Area estimation evaluation

Chapter 8

Conclusion and Future Work

8.1 Summary and Lessons Learned

Logic locking introduces area, power, and timing overheads in the design to be protected. In real-world cases we cannot obfuscate the whole design due to area, power, or timing constraints. It is important to carefully select the obfuscation points to maximize the security metrics while satisfying such constraints. The literature is missing optimization techniques to maximize security metrics under overhead constraints. A genetic algorithm for design space exploration [22] was proposed to optimize logic locking at high-level synthesis. The approach showed that a careful selection of obfuscation points yields far better results than obfuscating the whole design. Such approach is limited by the level of abstraction at which it operates, that allows to use it only with high-level synthesis generated designs. At register-transfer level such approach is not feasible due to the higher computational requirements of simulations and the higher number of potential obfuscation points. With our work we propose a framework at register-transfer level to optimize the selection of obfuscation points for a given security metric under area and key-bit constraints. The selection of obfuscation points is based on the analysis of the dependencies between signals. The idea is that selecting an obfuscation point will affect all statements that depend on it. By analyzing the dependencies we can find the obfuscation points that will have a major impact on the design or that will build a chain of obfuscation points that will be harder to break. To represent the dependencies of signals in the design we extract the System Dependence Graph of the design. We then created a set of heuristics that analyze different characteristics of the obfuscation points by analyzing the System Dependence Graph. Each heuristic gives a score to each obfuscation points. Heuristics are composable to evaluate different aspects during the selection phase. We evaluate different heuristic combinations and we output the results by ordering of the security metric score. This approach drastically decreases (from 100 to 400× faster) the computational time compared to design space exploration techniques like [22] while obtaining comparable results. Contrary to topological obfuscation, the results of our solution are independent of the design structure. Our solution performed better than topological obfuscation in 92% of the cases.

8.2 Outputs and Contributions

With our thesis we provided the following main contributions:

- a modular and composable design framework to apply logic locking with the support of RTL simulations and synthesis estimators as presented in Chapter 4;
- a procedure to extract System Dependence Graphs from a Verilog design as presented in Chapter 5;
- a set of scoring heuristics based on the analysis of the System Dependence Graph of an input RTL design as presented in Chapter 6;

The main tangible output of this thesis is the prototype implementation and evaluation of the proposed approach, presented in Chapter 7.

8.3 Limitations

The number of designs that we tested to evaluate our framework is small. Adding a new design to our test suite requires writing a test bench for the security metric evaluation. For us, writing a test bench for a third party design of which we only have the RTL design, without documentation, is not a trivial task. For a design house, writing a test bench to fit our framework should not present a problem.

We set a fixed value for the bounded heuristics parameters after running some preliminary tests. It would be interesting to expand the optimization task to consider different values for these parameters.

The SDG extraction procedure takes into consideration Verilog features. The concepts presented in Chapter 5 are not restricted to Verilog designs, but they correspond to different language features (i.e. signals and variables in VHDL vs. blocking and non-blocking assignments in Verilog). A VHDL implementation can be extracted using the presented concepts, mapping them to the corresponding language features.

We optimize the security metric under key bits and area overhead constraints. It would be interesting to perform multi-objective optimization and trying to minimize area overhead, maximize the security metric under key-bit constraint.

8.4 Future Work

There are many interesting research directions that we may follow to continue this work. New overhead estimators can be proposed to consider also timing and power overheads in the optimization process. Multi-objective optimization would have a great impact as it would allow to actively reduce the overheads instead of just constraining them. The use of SDGs in logic locking optimization can be further explored with new heuristics and machine-learning algorithms.

Bibliography

[1] Amr Abdel-Hamid, Sofiène Tahar, and El Mostapha Aboulhamid. A Survey on IP Watermarking Techniques. *Design Autom. for Emb. Sys.*, 9:211–227, 2004.

[2] Sarah Amir, Bicky Shakya, Xiaolin Xu, Yier Jin, Swarup Bhunia, Mark Mohammad Tehranipoor, and Domenic Forte. Development and Evaluation of Hardware Obfuscation Benchmarks. *Journal of Hardware and Systems Security*, 2:142–161, 2018.

[3] Abhishek Chakraborty, Nithyashankari Gummidipoondi Jayasankaran, Yuntao Liu, Jeyavijayan Rajendran, Ozgur Sinanoglu, Ankur Srivastava, Yang Xie, Muhammad Yasin, and Michael Zuzak. Keynote: A Disquisition on Logic Locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39:1952–1972, 2020.

[4] Prabuddha Chakraborty, Jonathan Cruz, and Swarup Bhunia. SAIL: Machine Learning Guided Structural Analysis Attack on Hardware Obfuscation. In *Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 56–61, 2018.

[5] Rajat Chakraborty and Swarup Bhunia. RTL Hardware IP Protection Using Key-Based Control and Data Flow Obfuscation. pages 405–410, 01 2010.

[6] Ronald P. Cocchi, James P. Baukus, Lap Wai Chow, and Bryan J. Wang. Circuit Camouflage Integration for Hardware IP Protection. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–5, 2014.

[7] Dean Takahashi . Globalfoundries: Next-generation chip factories will cost at least $10 billion. Available at: https://venturebeat.com/2017/10/01/globalfoundries-next-generation-chip-factories-will-cost-at-least-10-billion/ (Last accessed: April 1, 2020), 2017.

[8] ERAI. ERAI Reported Parts Statistics. Available at: https://www.erai.com/erai_blog/3167/_2019_erai_reported_parts_statistics (Last accessed: April 1, 2020), 2019.

[9] Félix-Antoine Fortin, François-Michel De Rainville, M.A. Gardner, Marc Parizeau, and Christian Gagné. Deap: Evolutionary algorithms made easy. *Journal of Machine Learning Research, Machine Learning Open Source Software*, 13:2171–2175, 2012.

[10] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, 1988.

[11] D.J. Kuck, Y. Muraoka, and Shyh-Ching Chen. On the number of operations simultaneously executable in fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, 1972.

[12] L. Li and A. Orailoglu. Piercing Logic Locking Keys through Redundancy Identification. *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 540–545, 2019.

[13] N. Limaye, A. B. Chowdhury, C. Pilato, M. T. M. Nabeel, O. Sinanoglu, S. Garg, and R. Karri. Fortifying RTL Locking Against Oracle-Less (Untrusted Foundry) and Oracle-Guided Attacks. *ACM/IEEE Design Automation Conference (DAC)*, 2021.

[14] Nimisha Limaye, Emmanouil Kalligeros, Nikolaos Karousos, Irene G. Karybali, and Ozgur Sinanoglu. Thwarting All Logic Locking Attacks: Dishonest Oracle with Truly Random Logic Locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2020.

[15] Luca Collini, Politecnico di Milano. Verilog SDG Extraction. Available at: https://github.com/Lucaz97/Verilog-SDG-Extraction.

[16] Mohamed El Massad, Jun Zhang, Siddharth Garg, and Mahesh V. Tripunitara. Logic Locking for Secure Outsourced Chip Fabrication: A New Attack and Provably Secure Defense Mechanism. arXiv:1703.10187, 2017.

[17] Mohamed El Massad, Jun Zhang, Siddharth Garg, and Mahesh V. Tripunitara. Logic Locking for Secure Outsourced Chip Fabrication: A New Attack and Provably Secure Defense Mechanism. arXiv:1703.10187, 2017.

[18] MIT Lincoln Laboratory. Common Evaluation Platform (CEP). Available at: https://github.com/mit-ll/CEP.

[19] Omdia. Top 5 Most Counterfeited Parts Represent a $169 Billion Potential Challenge for Global Semiconductor Market. Available at: https://www.electronicproducts.com/top-5-most-counterfeited-parts-represent-a-169-billion-potential-challenge-for-global-semiconductor-market/ (Last accessed: November 1, 2020), 2012.

[20] Tiago D. Perez and Samuel Pagliarini. A Survey on Split Manufacturing: Attacks, Defenses, and Challenges. *IEEE Access*, 8:184013–184035, 2020.

[21] Christian Pilato, Animesh Basak Chowdhury, Donatella Sciuto, Siddharth Garg, and Ramesh Karri. ASSURE: RTL locking against an untrusted foundry. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–13, 2021.

[22] Christian Pilato, Luca Collini, Luca Cassano, Donatella Sciuto, Siddharth Garg, and Ramesh Karri. On the Optimization of Behavioral Logic Locking for High-Level Synthesis. arXiv:2105.09666, 2021.

[23] Christian Pilato, Francesco Regazzoni, Ramesh Karri, and Siddharth Garg. TAO: Techniques for Algorithm-Level Obfuscation during High-Level Synthesis. In *Proceedings of the 55th Annual Design Automation Conference*, 2018.

[24] Chakraborty Prabuddha, Jonathan Cruz, and Bhunia Swarup. SURF: Joint Structural Functional Attack on Logic Locking. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 181–190, 2019.

[25] J. A. Roy, F. Koushanfar, and I. L. Markov. EPIC: Ending Piracy of Integrated Circuits. In *Design, Automation and Test in Europe*, pages 1069–1074, 2008.

[26] Samar Saha. Emerging business trends in the microelectronics industry. *Open Journal of Business and Management*, 04:105–113, 2016.

[27] Semi. Innovation is at risk as semiconductor equipment and materials industry loses up to $4 billion annually due to IP infringement. Available at: http://dev7.semi.org/en/white-paper-ip-infringement-causes-4-billion-loss-industry-annually (Last accessed: November 1, 2020), 2008.

[28] Abhrajit Sengupta, Mohammed Ashraf, Mohammed Nabeel, and Ozgur Sinanoglu. Customized Locking of IP Blocks on a Multi-Million-Gate SoC. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7, 2018.

[29] Kaveh Shamsi, Meng Li, Kenneth Plaks, Saverio Fazzari, David Z. Pan, and Yier Jin. IP Protection and Supply Chain Security through Logic Obfuscation: A Systematic Overview. *ACM Trans. Des. Autom. Electron. Syst.*, 24, 2019.

[30] Mustafa M. Shihab, Jingxiang Tian, Gaurav Rajavendra Reddy, Bo Hu, William Swartz, Benjamin Carrion Schaefer, Carl Sechen, and Yiorgos Makris. Design Obfuscation through Selective Post-Fabrication Transistor-Level Programming. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 528–533, 2019.

[31] Dominik Sisejkovic, Farhad Merchant, Lennart M. Reimann, Harshit Srivastava, Ahmed Hallawa, and Rainer Leupers. Challenging the Security of Logic Locking Schemes in the Era of Deep Learning: A Neuroevolutionary Approach. *J. Emerg. Technol. Comput. Syst.*, 17(3), 2021.

[32] SPIRAL team. SPIRAL software/hardware generation for performance. Available at: https://www.spiral.net/index.html.

[33] Shinya Takamaeda-Yamazaki. ”Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL”. In Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C. Diniz, editors, *Applied Reconfigurable Computing*, pages 451–460, 2015.

[34] Benjamin Tan, Ramesh Karri, Nimisha Limaye, Abhrajit Sengupta, Ozgur Sinanoglu, Md Moshiur Rahman, Swarup Bhunia, Danielle Duvalsaint, R. D., Blanton, Amin Rezaei, Yuanqi Shen, Hai Zhou, Leon Li, Alex Orailoglu, Zhaokun Han, Austin Benedetti, Luciano Brignone, Muhammad Yasin, Jeyavijayan Rajendran, Michael Zuzak, Ankur Srivastava, Ujjwal Guin, Chandan Karfa, Kanad Basu, Vivek V. Menon,

Matthew French, Peilin Song, Franco Stellari, Gi-Joon Nam, Peter Gadfort, Alric Althoff, Joseph Tostenrude, Saverio Fazzari, Eric Breckenfeld, and Kenneth Plaks. Benchmarking at the Frontier of Hardware Security: Lessons from Logic Locking. arXiv:2006.06806, 2020.

[35] US Department of Justice. Departments of Justice and Homeland Security Announce 30 Convictions, More Than $143 Million in Seizures from Initiative Targeting Traffickers in Counterfeit Network Hardware. Available at: https://archives.fbi.gov/archives/news/pressrel/press-releases/departments-of-justice-and-homeland-security-announce-30-convictions-more-than-143-million-in-seizures-from-initiative-targeting-traffickers-in-counterfeit-network-hardware (Last accessed: April 1, 2020), 2010.

[36] Shobha Vasudevan, E. Allen Emerson, and Jacob A. Abraham. Efficient Model Checking of Hardware Using Conditioned Slicing. *Electron. Notes Theor. Comput. Sci.*, 128:279–294, 2005.

[37] Y. Xie and A. Srivastava. Anti-SAT: Mitigating SAT Attack on Logic Locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(2):199–207, 2019.

[38] M. Yasin, J. J. Rajendran, O. Sinanoglu, and R. Karri. On Improving the Security of Logic Locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(9):1411–1424, 2016.

[39] Muhammad Yasin, Jeyavijayan JV Rajendran, Ozgur Sinanoglu, and Ramesh Karri. On Improving the Security of Logic Locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35:1411–1424, 2016.

[40] Muhammad Yasin, Chongzhi Zhao, and Jeyavijayan JV Rajendran. SFLL-HLS: Stripped-Functionality Logic Locking Meets High-Level Synthesis. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–4, 2019.

Appendix A

Differential entropy testbench template

To estimate the mean differential entropy, we need to run a behavioural simulation of the obfuscated design. In this appendix we illustrate the structure of a test bench to measure differential entropy in our framework.

The following is a template for a testbench:

```
parameter N_OUT = 256;
parameter DELTA = 0.00000000001;

// this is generated by our framework, contains:
// inputs, keys and golden outputs
‘include "../../in_obf.v"

module entropy_tb;

//keeps track of the miter circuit output
real n_out_miter_logic1[N_OUT-1:0];

real diffEntropy, Pi, mean_diffEntropy;

reg [N_OUT-1:0] out, out_miter;
reg clk, reset;
reg [N_KEY-1:0] locking_key;

// design specific signals
reg [511:0] in;
reg ready, out_valid, start;
reg out_valid;

int failed;

// module instantiation
mod_obf dut_obf (
.clk(clk), .rst(reset), .in(in), .ready(ready), .start(start),
.out_valid(out_valid), .out(out), .locking_key(locking_key));

always #5 clk = ~clk;
initial begin : prog_blk
   int fd_log;
   fd_log = $fopen("../vcs_entropy.log", "w");

   foreach (n_out_miter_logic1[j])
      n_out_miter_logic1[j] = 0;

   failed = 0;
   {start, valid, ready} = 0;
   in = 0;
   locking_key = 0;

   diffEntropy = 0;
   Pi = 0;
   clk = 0;
   reset = 1;

   repeat (5) @(posedge clk);
   reset = 0;
   repeat (30) @(posedge clk);

   foreach (InData[i])
   begin
      foreach (Locking_key[j])
      begin
         locking_key = Locking_key[j];
         reset = 1;
         repeat (5) @(posedge clk);
         reset = 0;
         repeat (30) @(posedge clk);
```

```
// if the orig circuit is ready and the obf one
// is not after 30 cycles
// we consider the test as failed.
repeat (300) @(posedge clk);
if(ready == 1'b0)
begin
    $display("NOT READY ON TIME");
    $fwrite(fd_log,"1");
    $finish;
    break;
end

in = InData[i][511:0];
#10
init = 1'b1;
#20
next = 1'b1;
#10
init = 1'b0;
next = 1'b0;
fork: execute
    // wait for ready_obf and read outputs
    begin
        wait(out_valid == 1'b1);
        out_miter = out ^ golden_output[i];
        foreach ( n_out_miter_logic1[k] )
            n_out_miter_logic1[k] =
            n_out_miter_logic1[k] + out_miter[k];
        disable execute;
    end
    begin
        // works as timeout
        repeat (6000) @(negedge clk);
        failed = 1;
        disable execute;
    end
join
if (failed == 1)
begin
```

```
                $display("NOT VALID ON TIME");
                $fwrite(fd_log,"1");
                $finish;
            end

        end
    end

    foreach (n_out_miter_logic1[j])
    begin
        Pi = n_out_miter_logic1[j]/(N_P_VECT*N_KEY_VECT);
        if (Pi == 0 || Pi == 1) // fixes the problem of Pi = 1 or 0
            diffEntropy = 0;
        else
            diffEntropy = diffEntropy +
            (Pi * $log10(1.0/Pi)/$log10(2) +
            (1.0 - Pi)*$log10(1.0 /(1.0-Pi))/$log10(2));
    end

    mean_diffEntropy = diffEntropy/N_OUT;

    $fwrite(fd_log,"%f", mean_diffEntropy);
    $fclose(fd_log);
    $finish;
end : prog_blk

endmodule
```

We generate an include file with input vectors for both primary inputs and keys, together with golden outputs obtained by simulating the original design with the same primary input vectors. The test bench for the golden outputs is similar, as it is a reduced version of the one reported above.

We collect the output of the miter circuit for each combination of primary input and key input to estimate P_i. Then we evaluate the differential entropy with the following formula, previously described in Section 3.3.

$$\overline{H} = \frac{1}{N} \cdot \sum_{i=1}^{N} \left(P_i \cdot log\frac{1}{P_i} + (1 - P_i) \cdot log\frac{1}{1 - P_i} \right)$$

The designers that are writing the test bench must know at least the control side of the design. The key points of the test bench are the synchronizing ones. Before giving the inputs we must be sure that the design is ready and we must read the output when it is valid. If a design has no control input or output those phases can be omitted but the designers must know the correct cycles in which they should provide the inputs and when to read the corresponding outputs.