Composable Heuristics for Register-Transfer Level Logic Locking Optimization

Luca Collini
DEIB
Politecnico di Milano
Milano, Italy
luca.collini@mail.polimi.it

Christian Pilato
DEIB
Politecnico di Milano
Milano, Italy
christian.pilato@polimi.it

Abstract—**Protecting the intellectual property (IP) of integrated circuits is more and more important as the globalization of the electronics supply chain exposes designs to new security threats such as reverse engineering and IP theft. Logic Locking is a promising approach to thwart these threats. Existing logic locking techniques take in consideration overheads only in the evaluation phase and not during obfuscation. We propose a framework to optimize the use of register-transfer level locking for a given metric. In this case, we optimize differential entropy under area or key bits constraints. We define a set of heuristics to give a score based on the analysis of the System Dependence Graph. The proposed solution yields better results for 94% 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.**

I. Introduction

An increasing number of design houses are outsourcing Integrated Circuit (IC) manufacturing due to the high cost of foundries [1]. Intellectual Property (IP) protection is becoming a crucial topic in hardware design. In fact, foundries have access to the IC design files exposing them to reverse engineering and consequently enabling IP theft and malicious modifications. The threats can range from overbuilding and disclosing secret technologies to compromising valuable network infrastructures. An example is the attempted trade of counterfeited Cisco equipment to the US Department of Defense [2]. The total loss from IC counterfeiting was estimated to be about $169 billions in 2011 [3]. Since over 80% of the reported counterfeited parts in the 2019 ERAI reported parts statistics were never reported before [4], we expect the real number of counterfeited parts could be much higher.

Hardware obfuscation aims at hiding and disabling the functionality of a chip to thwart reverse engineering of the IC functionality. *Logic Locking* is a well-known family of hardware obfuscation techniques. 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. Since logic locking requires to add extra logic to the design, it introduces overheads in terms of area, power and timing. The overheads are proportional to the cost of the design and the number of key bits used, limiting their use. Hardware designers trying to protect their work while keeping these overheads under a certain limit. This means that in real-world cases we cannot obfuscate the whole design.

Obfuscating different parts of a design will yield different results in terms of security. It is important to spend the key bits as efficiently as possible. However, it is not trivial to predict the effects of locking techniques especially in large designs. Applying logic locking before logic synthesis allows us to reason on the semantic aspects of the design and protect information before it gets embedded into the netlist. ASSURE [5] proposes a set of provably-secure RTL locking techniques but it applies obfuscation following the topological order of the design. The order in which the design is written changes the obfuscation result.

A design space exploration technique for optimizing the use of logic locking during high-level synthesis (HLS) is proposed in [6]. This technique shows interesting results, but the approach requires to use HLS in the design flow, limiting the application only to new and HLS-generated components. Also, the technique employs a genetic algorithm to perform design space exploration, making it computationally intensive, especially at RTL where the number of potential obfuscation points is larger and the RTL simulations are slower. At this level it can be used only on small designs. That is due to the fact that it performs a "blind" search, without reasoning on the properties of the design.

We propose a framework to optimize semantic obfuscation techniques by carefully selecting the parts to obfuscate. We formulated a set of heuristics to give a score to each obfuscation point. The higher the score of an obfuscation point, the more likely it will be used for the obfuscation. These heuristics are all based on properties extracted from signal dependencies in the design. We extracted and analysed the System Dependence Graph of the design. Heuristics are strongly design dependent and it is necessary to combine them. Our scoring heuristics are composable and new scoring heuristics can be easily added to the evaluation chain in a transparent way.

After defining the threat model and motivating the work we present our main contributions:

- a modular and composable design framework to apply logic locking with the support of RTL simulations and

synthesis estimators;
- a set of scoring heuristics based on the analysis of the System Dependence Graph of an input RTL design;
- a prototype implementation and evaluation of the proposed approach.

The proposed framework obtains results that are better than topological obfuscation for 94% of the tested cases while keeping computation time 100 to 400 times lower than existing design space exploration approaches.

II. Background

A. Threat model

We assume that a rogue employee at an external foundry wants to reverse engineer a given IC functionality to make illegal copies. The employee has access to the (obfuscated) layout files that were sent to the foundry for fabrication. The rogue employee can reverse engineer these files and obtain an RTL description [7]. The attacker can then run simulations on the obtained RTL description to infer knowledge on the design. In our work we consider that the attacker is capable of distinguishing between primary inputs and key inputs (*distinct ambiguity*) [8]. Moreover we assume that the attacker can distinguish between data and control inputs and outputs. We also assume that the attacker does not have access to the correct key neither to a working chip (*oracle*) therefore the attacker does not have any information on the true I/O behaviour of the design. This is plausible for low volume applications where it is not possible for an attacker to buy a working chip from the market. It has been recently shown that techniques that prevent oracle-based attacks, such as DisORC [9], can be combined with RTL logic locking techniques to complement the protection [10].

The attacker may still perform netlist-based attacks such as machine learning-guided structural and functional analysis [11]–[13], desynthesis [14], and redundancy identification [15] to unlock the design and perform reverse engineering. However, the techniques that we are optimising have been proved to be resilient towards these attacks without revealing information about the design [5].

B. RTL Locking

RTL locking hides the functionality of a given RTL description based on a locking key K. In our work we consider the obfuscation techniques proposed in ASSURE [5]:

- **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.
- **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 $a = K_o?(b + c) : (b - c)$.
- **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.

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 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. We aim at carefully picking the obfuscation points to achieve better results both in terms of security metrics and area overhead.

C. Security Evaluation

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 [5]. 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* [16], i.e. the differences between the expected output values and the ones obtained when applying a given key. The differential entropy of a design is the sum of the differential entropy measured on each output bit. We used the mean differential entropy as it is independent of the number of output bits of the design. We measured mean differential entropy using the following formula:

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

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.

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

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.

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 not manage control signals properly (i.e. never asserting ready or valid signals). Those solutions allow an attacker to easily discard wrong keys and must be avoided.

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.

D. Motivation

A fabless design house wants to protect its new RTL design to avoid competitors to steal its IP. However, obfuscating the design will inevitably increase its cost. To limit the extra cost, designers decide a maximum area overhead and pick a tamper-proof memory in which the key will be stored, fixing a maximum number of key bits that can be used for obfuscation. HLS solutions such as [6] are not applicable since the design is already at RTL. They may consider using ASSURE [10] but such technique is dependent on the structure of RTL files and they would need to refactor the design if the results are not satisfactory and they want to explore alternative solutions. With this work we propose a framework that provides different procedures to select obfuscation points, allowing us to perform optimizations under area or key bits constraints. To analyze the effect of obfuscation points prior to simulation we look at dependencies between statements. We represent such dependencies with a Dependence Graph.

E. Dependence Graphs

Dependence graphs were first proposed in 1972 to represent dependencies that occur within a program [17]. 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 single procedure. An SDG is an extension that allows us to represent programs with multiple procedures and calls among them. They were first proposed in [18]. To build the SDG of a program, the PDGs of the procedures are connected with each other through edges that model procedure calls. The use of SDGs for hardware descriptions was first proposed for Model Checking [19]. To use SDGs with hardware descriptions, we need some additional considerations due to the different computational paradigm between hardware and software [19]. SDGs are at the base of the heuristics that we propose to select the obfuscation points. In particular, we use System Dependence Graphs to analyze the impact of the obfuscation points prior to actual RTL simulation.

III. RTL Locking Framework

We propose an RTL obfuscation framework (see Figure 1) to easily evaluate overheads and metrics of obfuscated designs and perform optimizations under area or key budget constraints. The workflow begins with the parsing of the HDL code to extract the Abstract Syntax Tree (AST) of the design. The AST is analyzed to extract the SDG and to identify the obfuscation points. 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.

We evaluate a set of heuristics combinations by analyzing the SDG. Heuristics give a score to each obfuscation point that

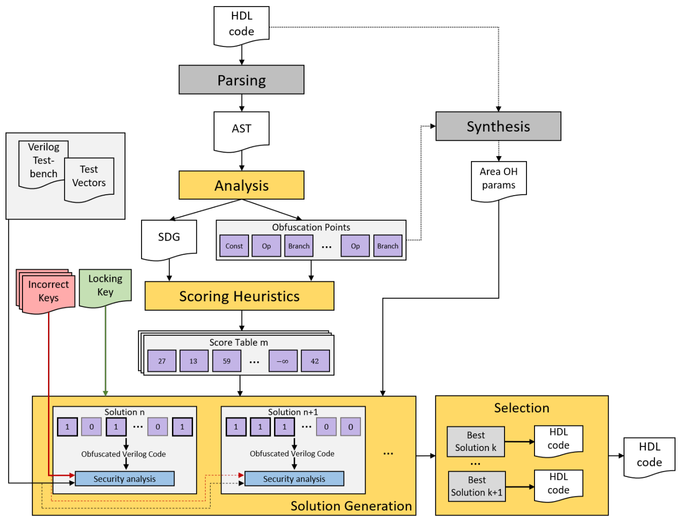


Fig. 1: Framework flow caption

can either be a positive (rewarding) or negative (punishing) number. For each combination of heuristics we build a *score table* that associates each obfuscation point with its final score, obtained by summing the scores of the heuristics. 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:

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

Where $OP1$, ..., OPn are obfuscation points that share the same SDG node.

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

The solution generation and the solution evaluation phases are decoupled to allow us to test different methods under different constraints.

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

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

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

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

IV. SDG Extraction

We built a tool for SDG extraction from Verilog[1] designs. We took inspiration from the considerations made in [19] 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.

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 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:

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

```
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.*

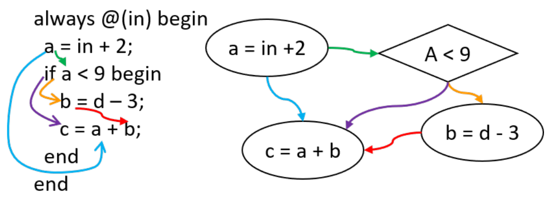


Fig. 2: Example of direct dependencies

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.*

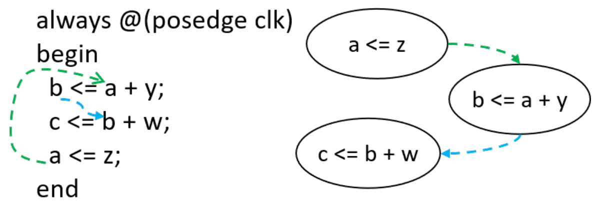


Fig. 3: Example of inter-cycle dependencies

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

Figure 4 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. When checking the dependency 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.

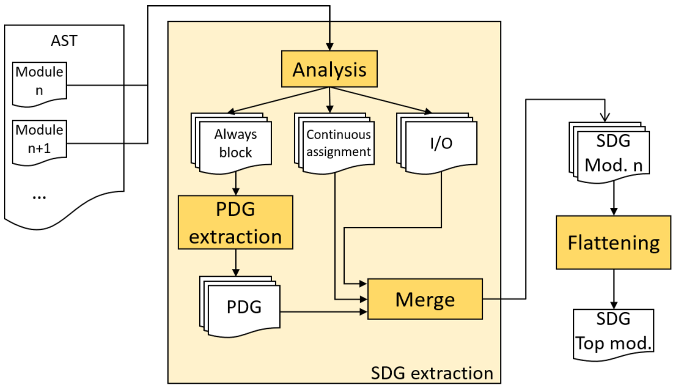
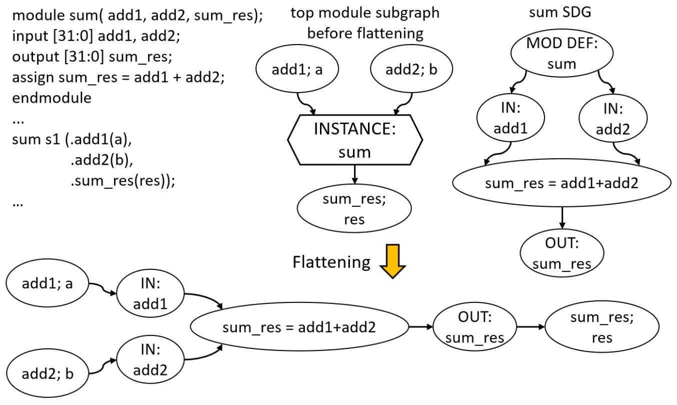


Fig. 4: SDG extraction flow

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 the SDG of the corresponding module and coupling nodes are connected with the corresponding inputs and outputs with direct dependency edges (Fig. 5).



```
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));
...
```

Fig. 5: Example of module flattening

V. Scoring Heuristics

We propose a set of heuristics that give a score to the obfuscation points based on an analysis of the SDG. The higher the score of an obfuscation point, the higher is the probability to pick it. A scoring function can either be rewarding, i.e. increasing the score of an obfuscation point, or punishing, i.e. decreasing the score of an obfuscation point. 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 can be used in a composable way, 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. To avoid having one scoring heuristic dominating all the others, we normalized all the scores for each scoring heuristic in the range [0, 100].

A. Control Disabling

This heuristic takes as argument a set of controlling signals divided as input and output signals. It disable the obfuscation points that would compromise the control outputs by assigning them a value of $-\infty$. It disable the parents of the control outputs and conditions that have a controlling input as parent. This allows us to avoid obfuscating those points that would cause simulation failures, yielding entropy 0 by definition.

B. Bounded (Direct) Children

The bounded children function takes as arguments an obfuscation point O and a distance D. It returns the number of (direct) assignments and conditions up to a distance D from O that are dependent from O. This function favors obfuscation points that have a higher propagation in the design. These points have a higher probability of having a wider influence on the outputs. 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.

C. Bounded Parents

The bounded parents function takes as arguments an obfuscation point O and a distance D. It 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 build longer sequences of obfuscated points. Longer sequences of obfuscated points are harder to brake.

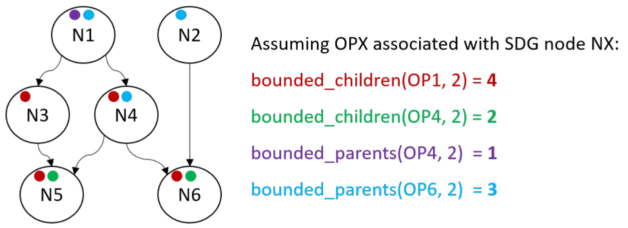

N1
N2
N3
N4
N5
N6
Assuming OPX associated with SDG node NX:
bounded_children(OP1, 2) = 4
bounded_children(OP4, 2) = 2
bounded_parents(OP4, 2) = 1
bounded_parents(OP6, 2) = 3

Fig. 6: Highlighting nodes that contribute to the score of an obfuscation point for a given heuristic

D. Max I/O Path Length

The maximum input output path length is a function that assigns a value to each obfuscation point O. The value is 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 function aims at favoring obfuscation points that can build longer obfuscation sequences. It has the same goal of bounded parents, it trades off higher computational complexity for more accurate results.

VI. Experimental Setup

We implemented a prototype framework leveraging Pyverilog [20], a Python-based Hardware Design Processing Toolkit for Verilog HDL. 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. Our framework works on the AST and, once the obfuscation step is finished, Pyverilog generates the obfuscated Verilog description ready for logic synthesis. We picked five designs from the MIT-LL Common Evaluation Platform (CEP) [21] to be evaluated with our framework. Two of these benchmarks (FIR and IIR) were generated using SPIRAL [22], a hardware generator. The selected benchmarks are a subset of those used in [5] as we had to build a test-bench to measure the metrics 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 I: 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 I 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.

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. 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 on the strictness of 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 [6] for HLS and against topological-order obfuscation like ASSURE [5]. To do so, we implemented a genetic algorithm that generates solutions for our framework. The evaluation is performed in the same way.

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.

TABLE II: Naming scheme for heuristics.

Heuristic	Abbreviation
Bounded children	NCHILD
Bounded direct children	DCHILD
Bounded parents	NPAR
Max I/O path	LPATH

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 and estimated area overhead with respect to topological order obfuscation. 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 II.

If the solution was generated with the probabilistic approach, we also added ”PROB” as prefix.

VII. Experimental Results

From the first runs where we ran the heuristic combinations alone, there is not a single heuristics that always performs better than the others. Figure 7 shows that certain heuristics are more likely to yield a best result when used within a certain key budget interval. Only NPAR LPATH DCHILD never generated a best solution, while all other heuristic combinations yielded a best solution at least twice. The probabilistic solution generation is the best technique to build solutions when the key budget is close to 100% since it yields solutions using less or equal bits than the given budget, whereas in-order generation forces the use of the full budget. For this reason probabilistic generation finds better solutions when we reach the point in which obfuscating more will decrease the differential entropy.

Figure 8 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

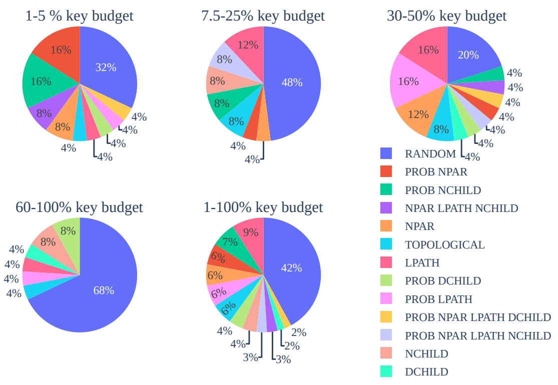


Fig. 7: Technique frequencies for key budget intervals across all designs

different key budget intervals. Figure 9 shows the comparison with topological obfuscation. The results show that topological obfuscation presents a higher variability in the entropy results and yields a best solution only in 6% of the cases. Figure 10 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 11 shows that the estimated area of the solutions found by the heuristics is either very close to or lower than topological obfuscation. Figure 12 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 big discrepancy between the estimated area and the measured one. Mean values are reported in Table III. 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 IV shows a comparison with the genetic algorithm for design space exploration. The two approaches obtain values that are very close with the proposed combined heuristic approach being 100 to 400 times faster, where applicable. The reported time for composable heuristics is the total time to evaluate the full set of heuristic combinations that we considered 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. This result shows how the proposed methodology scales much better for large designs.

TABLE III: Area overhead mean relative error (m.r.e.)

Design	Area Overhead m.r.e. [%]
FIR	13.40
IIR	19.93
MD5	100.72
DES3	55.16
SHA256	5.63
ALL	38.97

VIII. Related Work

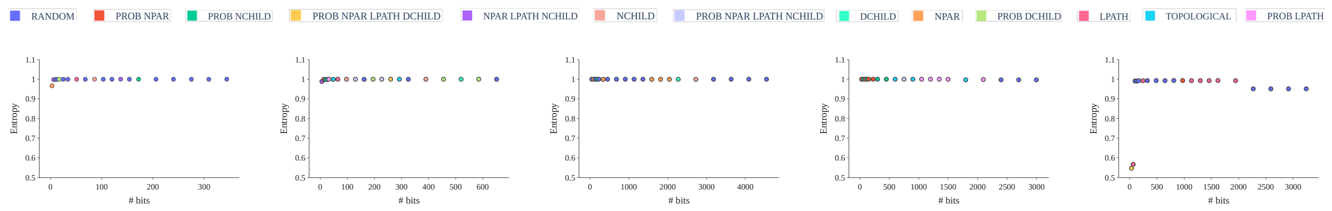
Numerous techniques have been proposed to thwart reverse engineering of hardware designs. They can be divided in two classes: key-less obfuscation, such as split manufacturing, camouflaging, watermarking and fingerprinting, and key-based obfuscation, such as logic locking. Split manufacturing divides the manufacturing process between different untrusted foundries [23]. IC camouflaging prevents netlist extraction by introducing subtle cell design changes at the GDS level [24]. Watermarking and fingerprinting aim at simplifying detection and tracking of illegal copies of the design [25]. 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 [26]. Obfuscation techniques can be applied at all steps of the IC design flow, post-synthesis techniques work either at transistor [27] or netlist level [28], [29]. Pre-synthesis techniques can be applied at either RTL [5] or HLS [30], [31].

Locking techniques aim at thwarting reverse engineering in scenarios where the attacker has access to the design files. Locking techniques depend on the threat model. In case the attacker has access to a working chip (oracle) the technique must be resilient towards SAT attacks [32]. When no oracle is available, attacks can rely only on the design files [11], [33] which should not reveal any information about the real structure of the design.

The main research thread in the area is about proposing new techniques preventing new kind of attacks or new attacks against existing techniques. Optimising the overhead introduced by logic locking has only recently started to be investigated [6]. This work is focused on techniques to improve the efficiency of logic locking at RTL from a security metric vs overheads viewpoint. Working at RTL allows for a wider integration on IC design workflows. The obfuscated netlists are obtained with commercial synthesis tools.

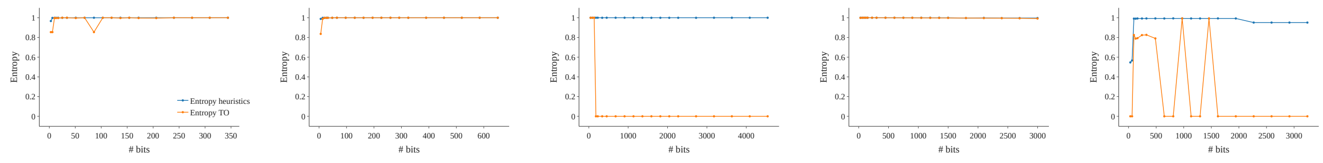
IX. Conclusion

The proposed obfuscation framework aims at optimising a security metric under either area or number of key bits constraints. Operating at RTL makes our solution compatible with all IC design flows. 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 94% of the cases. With our methodology the security metric results do not depend on the topological order of the design. Interesting research directions include the evaluation of new analyses on the SDG, the development of new estimators for overheads, and the possibility of performing multi-constrained optimization.



(a) FIR (b) IIR (c) MD5 (d) DES3 (e) SHA256

Fig. 8: Differential entropy results, highlighting the heuristic yielding the best solution



(a) FIR (b) IIR (c) MD5 (d) DES3 (e) SHA256

Fig. 9: Differential entropy comparison with topological obfuscation

TABLE IV: 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

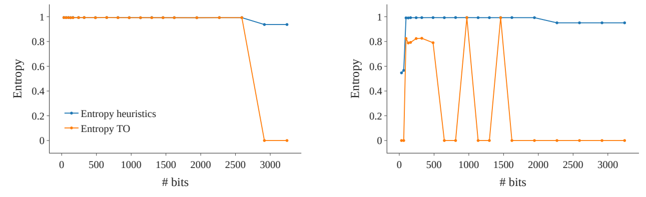


Fig. 10: Impact of inverting sub-module instances in SHA256

References

[1] S. Saha, "Emerging business trends in the microelectronics industry," *Open Journal of Business and Management*, vol. 04, pp. 105–113, 2016.

[2] 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.

[3] 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.

[4] 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.

[5] C. Pilato, A. B. Chowdhury, D. Sciuto, S. Garg, and R. Karri, "ASSURE: RTL locking against an untrusted foundry," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1–13, 2021.

[6] C. Pilato, L. Collini, L. Cassano, D. Sciuto, S. Garg, and R. Karri, "On the Optimization of Behavioral Logic Locking for High-Level Synthesis," arXiv:2105.09666, 2021.

[7] R. S. Rajarathnam, Y. Lin, Y. Jin, and D. Z. Pan, "ReGDS: A Reverse Engineering Framework from GDSII to Gate-level Netlist," in *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020, pp. 154–163.

[8] K. Shamsi, M. Li, K. Plaks, S. Fazzari, D. Z. Pan, and Y. Jin, "IP Protection and Supply Chain Security through Logic Obfuscation: A Systematic Overview," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, 2019.

[9] N. Limaye, E. Kalligeros, N. Karousos, I. G. Karybali, and O. Sinanoglu, "Thwarting All Logic Locking Attacks: Dishonest Oracle with Truly Random Logic Locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2020.

[10] 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.

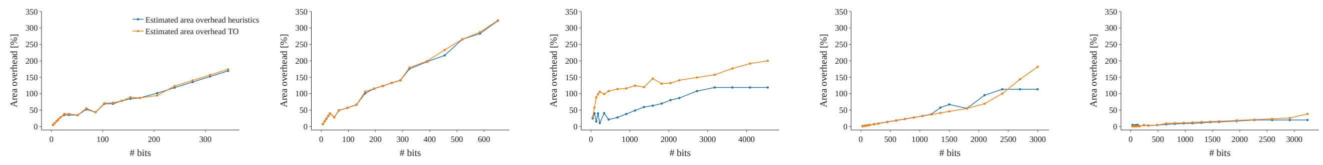
[11] P. Chakraborty, J. Cruz, and S. Bhunia, "SAIL: Machine Learning Guided Structural Analysis Attack on Hardware Obfuscation," in *Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 2018, pp. 56–61.

[12] C. Prabuddha, J. Cruz, and B. Swarup, "SURF: Joint Structural Functional Attack on Logic Locking," in *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019, pp. 181–190.

[13] D. Sisejkovic, F. Merchant, L. M. Reimann, H. Srivastava, A. Hallawa, and R. Leupers, "Challenging the Security of Logic Locking Schemes in the Era of Deep Learning: A Neuroevolutionary Approach," *J. Emerg. Technol. Comput. Syst.*, vol. 17, no. 3, 2021.

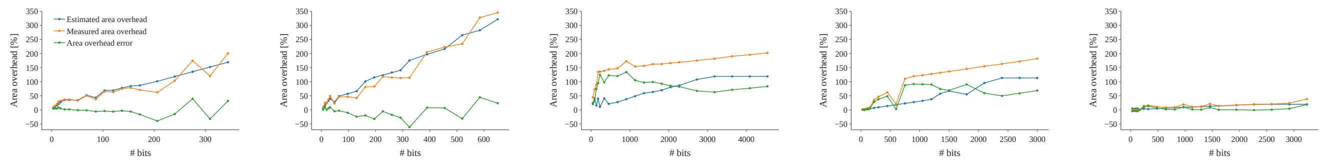
[14] M. E. Massad, J. Zhang, S. Garg, and M. V. Tripunitara, "Logic Locking for Secure Outsourced Chip Fabrication: A New Attack and Provably Secure Defense Mechanism," arXiv:1703.10187, 2017.

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



(a) FIR (b) IIR (c) MD5 (d) DES3 (e) SHA256

Fig. 11: Area estimation comparison with topological obfuscation



(a) FIR (b) IIR (c) MD5 (d) DES3 (e) SHA256

Fig. 12: Area estimation evaluation

[16] S. Amir, B. Shakya, X. Xu, Y. Jin, S. Bhunia, M. M. Tehranipoor, and D. Forte, “Development and Evaluation of Hardware Obfuscation Benchmarks,” *Journal of Hardware and Systems Security*, vol. 2, pp. 142–161, 2018.

[17] D. Kuck, Y. Muraoka, and S.-C. Chen, “On the number of operations simultaneously executable in fortran-like programs and their resulting speedup,” *IEEE Transactions on Computers*, vol. C-21, no. 12, pp. 1293–1310, 1972.

[18] 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*, 1988, p. 35–46.

[19] S. Vasudevan, E. A. Emerson, and J. A. Abraham, “Efficient Model Checking of Hardware Using Conditioned Slicing,” *Electron. Notes Theor. Comput. Sci.*, vol. 128, p. 279–294, 2005.

[20] S. Takamaeda-Yamazaki, “”Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL”,” in *Applied Reconfigurable Computing*, K. Sano, D. Soudris, M. Hübner, and P. C. Diniz, Eds., 2015, pp. 451–460.

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

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

[23] T. D. Perez and S. Pagliarini, “A Survey on Split Manufacturing: Attacks, Defenses, and Challenges,” *IEEE Access*, vol. 8, pp. 184 013–184 035, 2020.

[24] R. P. Cocchi, J. P. Baukus, L. W. Chow, and B. J. Wang, “Circuit Camouflage Integration for Hardware IP Protection,” in *Proceedings of the 51st Annual Design Automation Conference*, 2014, p. 1–5.

[25] A. Abdel-Hamid, S. Tahar, and E. M. Aboulhamid, “A Survey on IP Watermarking Techniques,” *Design Autom. for Emb. Sys.*, vol. 9, pp. 211–227, 2004.

[26] A. Chakraborty, N. G. Jayasankaran, Y. Liu, J. Rajendran, O. Sinanoglu, A. Srivastava, Y. Xie, M. Yasin, and M. Zuzak, “Keynote: A Disquisition on Logic Locking,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, pp. 1952–1972, 2020.

[27] M. M. Shihab, J. Tian, G. R. Reddy, B. Hu, W. Swartz, B. Carrion Schaefer, C. Sechen, and Y. Makris, “Design Obfuscation through Selective Post-Fabrication Transistor-Level Programming,” in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 528–533.

[28] 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*, vol. 35, pp. 1411–1424, 2016.

[29] A. Sengupta, M. Ashraf, M. Nabeel, and O. Sinanoglu, “Customized Locking of IP Blocks on a Multi-Million-Gate SoC,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–7.

[30] C. Pilato, F. Regazzoni, R. Karri, and S. Garg, “TAO: Techniques for Algorithm-Level Obfuscation during High-Level Synthesis,” in *Proceedings of the 55th Annual Design Automation Conference*, 2018.

[31] M. Yasin, C. Zhao, and J. J. Rajendran, “SFLL-HLS: Stripped-Functionality Logic Locking Meets High-Level Synthesis,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–4.

[32] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin, “Circuit Obfuscation and Oracle-Guided Attacks: Who Can Prevail?” in *Proceedings of the on Great Lakes Symposium on VLSI*, 2017, p. 357–362.

[33] Y. Zhang, P. Cui, Z. Zhou, and U. Guin, “TGA: An Oracle-Less and Topology-Guided Attack on Logic Locking,” in *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop*, 2019, p. 75–83.