A Composable Design Space Exploration Framework to Optimize Behavioral Locking

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

Ramesh Karri
New York University, USA
rkarri@nyu.edu

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

***Abstract*—Globalization of the integrated circuit (IC) supply chain exposes designs to security threats such as reverse engineering and intellectual property (IP) theft. Designers may want to protect specific high-level synthesis (HLS) optimizations or micro-architectural solutions of their designs. Hence, protecting the IP of ICs is essential. Behavioral locking is an approach to thwart these threats by operating at high levels of abstraction instead of reasoning on the circuit structure. Like any security protection, behavioral locking requires additional area. Existing locking techniques have a different impact on security and overhead, but they do not explore the effects of alternatives when making locking decisions. We develop a design-space exploration (DSE) framework to optimize behavioral locking for a given security metric. For instance, we optimize differential entropy under area or key-bit constraints. We define a set of heuristics to score each locking point by analyzing the system dependence graph of the design. The solution yields better results for 92% of the cases when compared to baseline, state-of-the-art (SOTA) techniques. The approach has results comparable to evolutionary DSE while requiring 100× to 400× less computational time.**

I. Introduction

Owning IC foundries has high costs, so design houses are outsourcing IC manufacturing [24]. IP protection is critical since foundries have access to the IC design files, exposing them to reverse engineering that enables IP theft and malicious modifications. An example is the attempted trade of counterfeit Cisco equipment to the US Department of Defense [31]. In 2011, the estimated losses due to counterfeits were around $169 billions [15]. Since over 80% of counterfeited parts reported in 2019 by ERAI were not reported before [7], the real numbers are likely higher. Security protections must be introduced since the early stages of the design (e.g., during high-level synthesis - HLS) for IP protection [3, 17, 20].

Logic Locking is a family of hardware locking techniques that aim to thwart reverse engineering. They add extra logic to the original design that is controlled by a new set of inputs called *key inputs* [30]. The correct functionality is obtained only if the correct sequence of bits is provided to the key inputs. Since logic locking adds extra logic into the design, it introduces area, power and time overheads. The overheads are proportional to the number of key bits used [20, 18]. Designers need to protect their IP while curbing the overheads. This means that in the real-world one cannot lock the whole design.

Locking different parts of a design will yield different security guarantees. Designers must explore the design space to understand how to effectively use the budgeted key bits. Effects of locking techniques are difficult to predict, especially in large designs. Locking at a high level (i.e., before logic synthesis) allows reasoning about the semantics of the design, which in turn helps protect sensitive information before it is synthesized and embedded into the netlist. TAO [20] and ASSURE [18] apply high-level locking during and after HLS, respectively. However, these high-level locking methods analyze and lock the design using a greedy, topological exploration of the design. The order in which high-level statements appear in the design impacts the security and overhead.

Researchers developed a design space exploration (DSE) framework to optimize locking during HLS [19]. While this DSE supports trade offs, it requires access to proprietary code-bases of commercial HLS tools, limiting application to existing IPs and HLS-generated components. This DSE does a "blind" search using a genetic algorithm. It does not reason about the properties of the design, making the optimization computationally expensive or even infeasible.

This study proposes a DSE framework to optimize behavioral locking by selecting the parts to be locked. This behavior-level DSE extracts and analyzes the *System Dependence Graph* of the design. We formulate heuristics that analyze properties extracted from signal dependencies in the design and score the locking points. The higher the score of a locking point, the more likely it is used in the design. Our scoring heuristics are *composable*, i.e., one can add new heuristics to evaluate other aspects. The contributions of this study are threefold:

1) A modular DSE framework to apply behavioral locking using RTL simulations and synthesis estimators.
2) Composable scoring heuristics to analyze the System Dependence Graph of a design.
3) A prototype DSE framework and its evaluation.

Our composable DSE framework yields results that are better than topologically-ordered locking in 92% of the cases while requiring up to 400× less time than "blind", genetic DSE.

II. Background

A. Threat model: An Oracle-less Attack

A rogue employee at a foundry may reverse engineer an IC function and make unauthorized copies or malicious modifications. The rogue employee has access to the locked layout and can reverse engineer it to recover the RTL [22, 23]. The attacker has access to a testbench to simulate the RTL and infer structure but no access to correct input-output pairs. We

assume the attacker can distinguish between primary inputs and the locking key inputs [26], and between data and control inputs and outputs. We assume the attacker has no access either to the correct key or to a working IC (*oracle*). Therefore the attacker does not know the true I/O behavior of the IC. This oracle-less model is plausible for malicious foundries, especially in low-volume applications [18, 26]. Techniques that prevent oracle-based attacks, such as DisORC [12], can be combined with locking [11]. We consider locking techniques [18, 20] that are resilient to ML-guided structural and functional analysis [5, 21, 28], de-synthesis [13], and redundancy identification [10].

B. Behavioral Locking

Behavioral locking locks a function with a locking key K. We consider three SOTA locking techniques [3, 18, 20]:

- **Operation locking** adds a multiplexer to select between the right operation and a dummy one based on a key bit. For example `a=b+c` is locked as `a=K`$_o$`?(b+c):(b-c)`.
- **Branch locking** XORs the condition with a key bit, inverting it when the key bit is 1. The condition `a<b` is locked as `(a>=b)`$\wedge$`K`$_b$ or `(a<b)`$\wedge$`K`$_b$ depending on `K`$_b$.
- **Constant locking** replaces constants with key bits. For example `a=b+4'b0100` is locked as `a=b+K`$_c$ where `K`$_c$ is the 4-bit constant stored as part of the locking key.

The locking key has two parts. One is randomly generated and locks the control branches and operations, while the other contains the constants extracted from the design. An input port is added to apply the locking keys after fabrication. A *locking point* is a semantic element (i.e. a constant, a branch, or an operation) that can be locked by any of these techniques. We aim at selecting the locking points to improve security and reduce overhead.

C. Security Evaluation

In an oracle-less scenario an attacker can infer the key only by looking at the design files or by observing the design function through simulations. Locking techniques are provably secure [18]. We evaluate security of locked solutions by looking at *output corruptibility*, i.e. how much the lock affects the output values. We use *mean differential entropy* to quantify this effect [2] by measuring the differences between the expected values and the ones obtained when applying a key. Mean differential entropy is defined as follows:

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

where N is the number of output bits. $P_i \in [0, 1]$ is the probability that output i (given an input and a key value) is different from the correct one. H is independent of the number of output bits and is maximized when $P_i = 0.5$. We aim to maximize mean differential entropy making it close to 1. This is the case when $P_i = 0.5, \forall i$. The attacker cannot make any educated guess on the design function, leading to a probability to guess the correct key as 2^{-K} (where K is the number of key bits). In our threat model, the attacker can distinguish between data and control inputs and outputs. We assign $H = 0$ to the solutions with incorrect control signals (i.e. never asserting ready or valid signals). They are avoided because they allow an attacker to identify and discard wrong keys. A similar approach was used for genetic algorithms based DSE, albeit with high computational time [19].

D. Motivation

A fabless design house aims at protecting its IP designs but locking them is costly. To limit costs, a designer can decide the maximum area overhead and pre-define the size of the tamper-proof memory used to store the key. This limits the number of key bits used for locking. Traditional DSE solutions [19] may not scale with the size of the designs, while current heuristic solutions [18, 20] depend on the structure of design (either at C or register-transfer level (RTL)) because they identify the locking points and apply the locking techniques in topological order. Designers need to refactor the design to improve unsatisfactory security results. We propose a DSE framework that allows exploration and selection of locking points, optimizing the design under area or key bit constraints. To "predict" the effect of locking points on the outputs prior to simulation, we consider dependencies between statements.

E. Dependence Graphs

Dependence graphs were proposed to describe dependencies in a program [8, 9]. A Program Dependence Graph (PDG) is a directed graph that represents a single procedure. A System Dependence Graph (SDG) combines all PDG representations of a complex program with edges that model procedure calls. SDGs for hardware descriptions were first proposed for model checking [32]. We analyze the SDG to predict specific effects of the locking points and score them.

III. Related Work

The techniques to thwart reverse engineering of hardware designs can be divided in two classes: *key-less locking* (e.g., split manufacturing, camouflaging, watermarking, and fingerprinting) and *key-based locking* (e.g., logic locking). Split manufacturing divides the manufacturing process between different untrusted foundries [16]. IC camouflaging prevents netlist extraction by introducing cell design changes at the GDSII-level [6]. Watermarking and fingerprinting simplify detection and tracking of illegal copies of a design [1]. Logic locking applies design modifications that make it functional only with the correct key, unknown to the foundry [4]. Locking techniques can be applied at all steps of the IC design flow. Post-synthesis techniques work at transistor [27] and netlist level [33]. Pre-synthesis techniques work at RTL [18] or HLS level [20, 34]. Locking techniques thwart reverse engineering, according to the threat model. If the attacker has access to a working IC (oracle) the technique must be resilient to SAT attacks [25]. Without an oracle IC, attacks can rely only on design files [5, 35] which do not reveal information about the function. This research proposes techniques to prevent new attacks or new attacks against existing techniques. Optimising

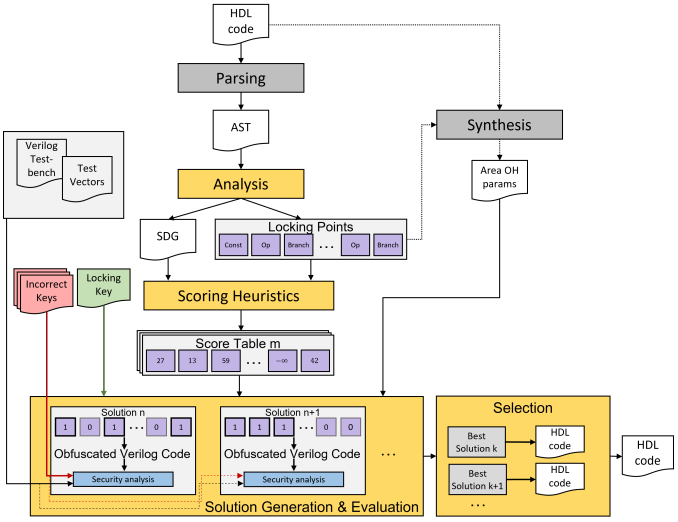


Fig. 1: Overview of the proposed locking framework.

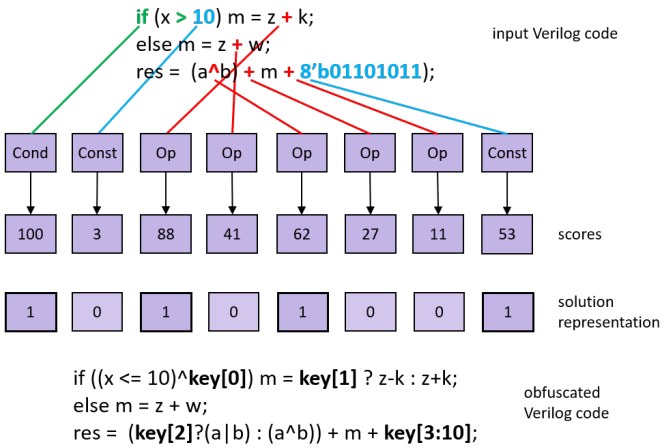


Fig. 2: Example of solution representation.

locking overhead is a recent direction [19]. This work seeks to improve RTL locking from a security vs overhead viewpoint.

IV. Behavioral Locking DSE Framework

We propose a locking DSE framework to evaluate security and overheads of locked designs and optimize them under area or key budgets. Fig. 1 describes the DSE framework and operates on HDL generated by HLS or by designers.

A. Overview

The workflow parses the input HDL to extract an Abstract Syntax Tree (AST), which is an abstract representation of the circuit specification. It then analyzes the AST to extract the SDG and identify the locking points. The SDG has a node for each statement so each SDG node is associated with an arbitrary number of locking points. We evaluate a combination of locking heuristics, i.e., methods to evaluate the locking points based on different characteristics. Each heuristic gives a **score** to each locking point, either a *reward* or a *punishment*. A positive reward means that the locking point yields good locking results. To combine heuristics, we build a *score table* where each locking point has a score obtained by summing scores of all heuristics.

Solution generation and evaluation phases are decoupled to test different locking methods under different area/key constraints. Solutions are evaluated by measuring differential entropy, key size, and area overhead. The DSE framework represents a locking solution as a binary string where each element is a locking point and its value is 1 if and only if the corresponding locking point is locked. Figure 2 shows an example locking solution, locking points, their scores, and depiction of the solution.

Given a score table, we can generate a solution in two ways. The first way selects the locking points with the highest score until we fulfill the constraints. The second way probabilistically maps the scores in the range [0.25, 0.75]. This value is the probability of selecting the locking point. So a locking point with the lowest (highest) score is locked with a probability of 25% (75%). The area overhead is a linear combination of the number of bits used for locking constants C, branches B and operations O, respectively. The values for these parameters are either given by the designer or estimated by the framework using three initial syntheses runs by locking all points of the same type (i.e., all constants, operations, or branches) and dividing the resulting overhead for the total number of locked points.

B. SDG Extraction

We extract SDG representations from Verilog designs[1] by extending the flow in [32] to accommodate changes that fit our locking analysis. To extract the SDG of a Verilog module, we add an input vertex for each input, an output vertex for each output, and an SDG vertex for each continuous assignment. We create a PDG for each `always` block and merge these entities with edges. Let G_{AB} be the PDG of `always` block AB, then G_{AB} is a directed graph whose vertices $v_1, v_2, ..., v_n$ represent the assignment statement and control predicates in AB and the edges represent dependencies between nodes. Verilog has two types of assignments in `always` blocks: blocking (=) and non-blocking (<=). Blocking assignments are sequential. To understand non-blocking assignment, consider the following `always` block sensitive to the positive clock edge:

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

The order of the non-blocking assignments does not affect the `always` block. At clock cycle X `Z` is assigned to `A` but is propagated to `B` only at cycle $X + 1$. Dependencies between non-blocking assignments occur between two activations of the same `always` block. We distinguish between *direct dependencies* and *inter-cycle dependencies*.

[1]Our SDG extraction applies to Verilog designs due to parsing constraints; it can extend to other HDLs.

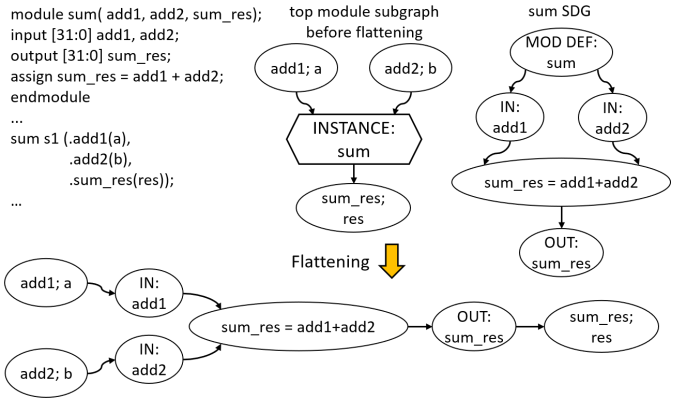


Fig. 3: An example of verilog module flattening.

Definition 1. *There is 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 v_1; or v_1 is a blocking assignment and for some X in the left-hand side in v_2 there exists an execution path from v_1 to v_2 along which there is no assignment to X.*

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 and there is a signal X in the left-hand side that is used in v_2.*

When checking dependency from an assign vertex v_1 to a vertex v_2 within a PDG P_1, we add a direct dependency edge if (1) there is a signal X in the left-hand side of v_1, (2) X is used in v_2, and (3) X is in the sensitivity list of P_1. We add an inter-cycle dependency edge if (1) there is a signal X in the left-hand side of v_1, (2)X is used in v_2, and (3) X is not in the sensitivity list of P_1. We assume only one type of assignment in each `always` block (common practice in hardware design). When a module instantiates a sub-module, we insert an *instance node* connected with *coupling nodes* for input/output ports of the sub-module to represent port mapping. After extracting an SDG from each module, we flatten the design starting from the top module. Each instance node is replaced with the SDG of the sub-module. Coupling nodes are connected with the corresponding inputs and outputs with direct dependency edges (see Fig. 3).

C. Scoring Heuristics

We propose four heuristics to score locking points by analyzing SDG. Scoring functions can be *local* and *global*. Local functions explore SDG up to a distance from a locking point. Global functions do not constrain SDG exploration. Scoring functions are composable; we can use a subset to compute scores to rank locking points. We normalize scores in the [0, 100] so that they affect the solution in same way.

Control Disabling heuristic uses a set of controlling signals divided as input and outputs. It disables locking points (by assigning a value of $-\infty$) in parents of the control outputs and conditions that depend on a controlling input. *EFFECT:* Avoids locking points that cause simulation failures.

TABLE I: Characteristics of MIT LL CEP RTL benchmarks.

		# Locking points			Total	SDG
Design	Modules	Const	Ops	Branches	# bits	# 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

Bounded (Direct) Children heuristic takes a locking point O and returns number of assignments and conditions up to distance D that depend on O. *EFFECT:* Favors locking points that propagate more and influence on the outputs. A node influences parts of a design if it has many children in SDG.

Bounded Parents heuristic takes a locking point O and returns number of locking points up to a distance D converging in O. *EFFECT:* Favors locking points with high convergence. They build long sequences of lock points which are harder to break.

Max I/O Path Length heuristic assigns a value to each locking point O equal to the maximum number of locking points in input to output path passing through O. *EFFECT:* Favors locking points that can build longer lock sequences.

V. Experimental Results

We implemented a prototype DSE framework that embodies our methodology by using Pyverilog [29] to parse input Verilog and create its abstract syntax tree (AST). Once locked, which is applied as AST transformations, Pyverilog re-generates the locked Verilog description for logic synthesis. We selected five modules from the MIT-LL Common Evaluation Platform (CEP) [14]. The benchmarks have representative control branches and overhead among those used in [18]. FIR and IIR modules represent designs generated by HLS or hardware generators. Table I reports locking points for each category, maximum number of key bits, and number of SDG nodes for each benchmark. The DSE framework runs synthesis using Synopsys Design Compiler (R-2020.09-SP1) targeting Nangate 15nm technology at 25C standard operating conditions. We use Synopsys VCS for RTL simulation. To calculate the mean differential entropy we estimate the output probability by running 10,000 simulations (100 random keys $\times$ 100 random inputs) [2].

Preliminary results show the heuristics are design- and constraint-dependent. Since they are computationally light, we run several combinations of them, reporting the best result. We compare our results against DSE based on a genetic algorithm [19], topological-order locking [18, 20], and random locking. We use the following labels: CONTR DIS for *Control Disabling*, NCHILD for *Bounded children*, DCHILD for *Bounded direct children*, NPAR for *Bounded parents*, and LPATH for *Max I/O path*. We use CONTR DIS in all combinations, so it is omitted. We added the prefix PROB if we use the probabilistic approach for solution generation.

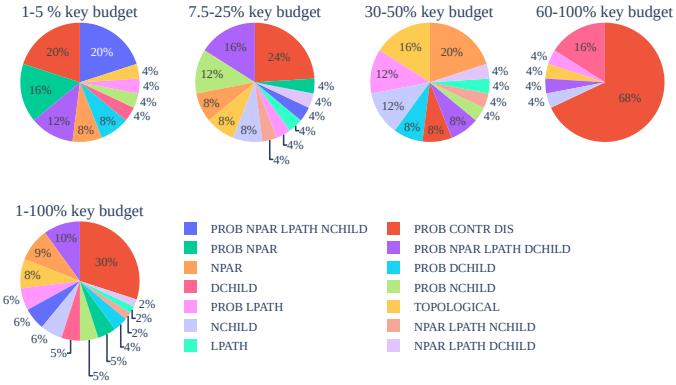


Fig. 4: Frequency of techniques for key budgets for all designs.

Effect of Heuristic Parameters: Increasing the distance parameter for the bounded heuristics flattens the locking points, reducing the performance. We set a distance of 3 for the NCHILD/DCHILD and 2 for NPAR. For each benchmark, we ran DSE with 20 constraints on the key budget: 1-5, 7.5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 60, 70, 80, 90, 100% of the maximum number of key bits. We configured DSE to optimize the mean differential entropy of the design and evaluated area overhead. The best solution was selected as the one with mean differential entropy within 0.001 from the best and with the lowest estimated area overhead, similar to [19]. We evaluated mean differential entropy with respect to topological order locking and random locking. We identified the technique which yields the best results more often for the key budgets (1-5%, 7.5-25%, 30-50%, 60-100%, 1-100%). We ran the following heuristics by generating in-order and probabilistic solutions: 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.

Heuristic Behavior: Fig. 4 shows how many times each heuristic yields a best solution. All heuristic combinations yielded a best solution at least twice. Certain heuristics are more likely to give a best result within a key budget. 60-100% constraint interval is predominated by CONTR DIS with probabilistic generation. Fig. 5 shows the entropy results for each key budget, highlighting the technique with the best solution. Different techniques are better on different designs and key budgets. Fig. 6 compares the approach with topological and random locking. Topological locking has a higher variability in entropy and yields a best solution in 8% of the cases. In designs with control signals, random locking selects points that invalidate the solution.

DSE Comparison: Table II compares the entropy results of our work with the ones obtained with topological ordering [18, 20] and the ones obtained with DSE [19] (i.e., standard genetic algorithm with uniform crossover and mutation operators). We also report the execution times of all methods. In case of topological locking, solutions may invalidate the designs, leading to null entropy. Our approach systematically achieves much better differential entropy, closer to the values obtained with evolutionary DSE but 100 to 400 times faster. The reported time for our method includes the time to evaluate the full set of 14 heuristic combinations. All methods include also the time to evaluate the area of the best solution with logic synthesis.

Acknowledgments

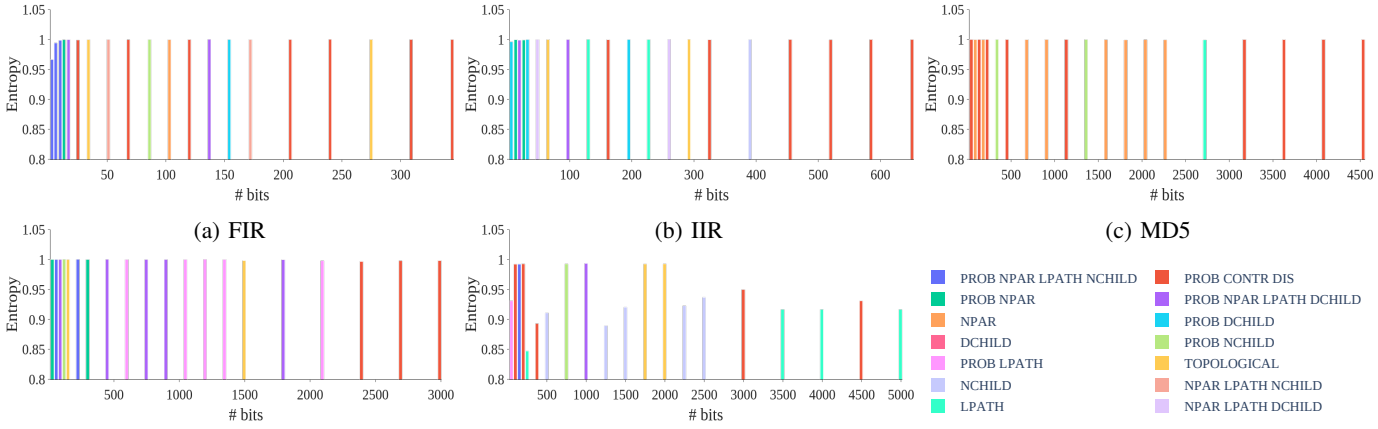
Research supported in part by NSF (# 1526405), ONR (# N00014-18-1-2058), NYU Center for Cybersecurity, and NYUAD Center for Cybersecurity.

VI. Conclusion

We propose a locking DSE framework that optimizes a security metric subject to area or key size constraints. The entropy results are better than the ones obtained by locking in a topological order for 92% cases. In our DSE method the security metric results do not depend on the topological order of design exploration. To improve performance in less constrained scenarios we may add heuristics to lower the scores of locking points that have less impact on the locked result. Future research includes new analyses on the SDG and design of new estimators for overheads.

References

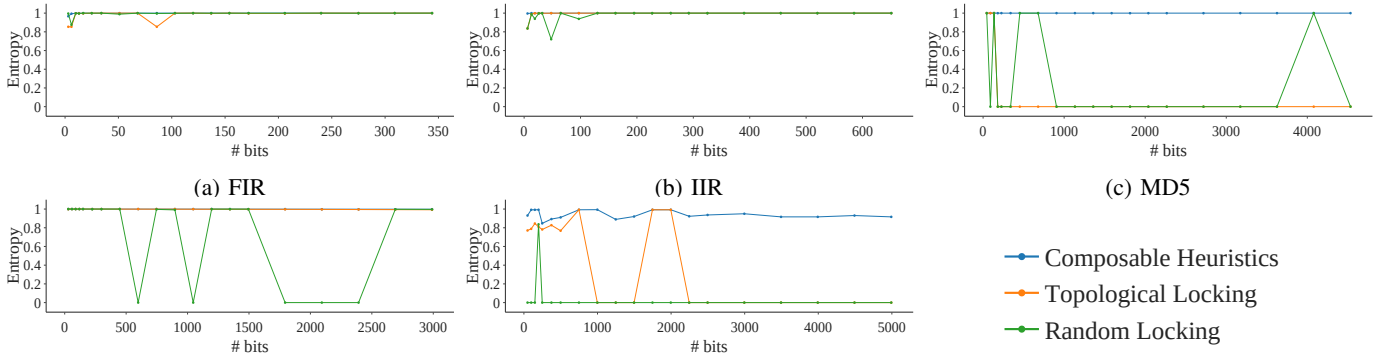
[1] A. Abdel-Hamid et al. “A Survey on IP Watermarking Techniques”. In: *Design Autom. for Emb. Sys.* 9 (2004), pp. 211–227.
[2] S. Amir et al. “Development and Evaluation of Hardware Obfuscation Benchmarks”. In: *Journal of Hardware and Systems Security* 2 (2018).
[3] H. Badier et al. “Transient Key-based Obfuscation for HLS in an Untrusted Cloud Environment”. In: *DATE*. 2019, pp. 1118–1123.
[4] A. Chakraborty et al. “Keynote: A Disquisition on Logic Locking”. In: *IEEE Transactions on CAD* 39 (2020).
[5] P. Chakraborty et al. “SAIL: Machine Learning Guided Structural Analysis Attack on Hardware Obfuscation”. In: *AsianHOST*. 2018.
[6] R. P. Cocchi et al. “Circuit Camouflage Integration for Hardware IP Protection”. In: *DAC*. 2014, pp. 1–5.
[7] ERAI. *ERAI Reported Parts Statistics*. Available at: https://tinyurl.com/3jrxe8ww (Last access: Apr. 1, 2020). 2019.
[8] S. Horwitz et al. “Interprocedural Slicing Using Dependence Graphs”. In: *PLDI*. 1988, pp. 35–46.
[9] D. Kuck et al. “On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup”. In: *IEEE Transactions on Computers* C-21.12 (1972), pp. 1293–1310.
[10] L. Li et al. “Piercing Logic Locking Keys through Redundancy Identification”. In: *DATE*. 2019, pp. 540–545.
[11] N. Limaye et al. “Fortifying RTL Locking Against Oracle-Less (Untrusted Foundry) and Oracle-Guided Attacks”. In: *DAC* (2021).
[12] N. Limaye et al. “Thwarting All Logic Locking Attacks: Dishonest Oracle with Truly Random Logic Locking”. In: *IEEE Transactions on CAD* (2020).
[13] M. E. Massad et al. *Logic Locking for Secure Outsourced Chip Fabrication: A New Attack and Provably Secure Defense Mechanism*. arXiv:1703.10187. 2017.
[14] MIT Lincoln Laboratory. *Common Evaluation Platform (CEP)*. Available at: https://github.com/mit-ll/CEP.
[15] Omdia. *Top 5 Most Counterfeited Parts Represent a $169 Billion Potential Challenge for Global Semiconductor Market*. Available at: https://tinyurl.com/2v64mcbu (Last access: Nov. 1, 2020). 2012.
[16] T. D. Perez et al. “A Survey on Split Manufacturing: Attacks, Defenses, and Challenges”. In: *IEEE Access* (2020), pp. 1–23.
[17] C. Pilato et al. “Securing Hardware Accelerators: A New Challenge for High-Level Synthesis”. In: *IEEE Embedded Systems Letters* 10.3 (2018), pp. 77–80.
[18] C. Pilato et al. “ASSURE: RTL Locking Against an Untrusted Foundry”. In: *IEEE Transactions on VLSI Systems* (2021), pp. 1–13.



(a) FIR (b) IIR (c) MD5

(d) DES3 (e) SHA256

Fig. 5: Differential entropy results, highlighting the heuristic yielding the best solution.



(a) FIR (b) IIR (c) MD5

(d) DES3 (e) SHA256

Fig. 6: Differential entropy comparison with topological and random locking.

TABLE II: Comparison with locking in topological order and DSE approach at 4 key budget constraints.

Design	Topological Order [18, 20]					Composable Heuristics (this work)					DSE [19]				
	Mean differential entropy				Time	Mean differential entropy				Time	Mean differential entropy				Time
	25%	50%	70%	100%	[min]	25%	50%	70%	100%	[min]	25%	50%	75%	100%	[min]
FIR	0.85402	0.99977	0.99859	0.99991	0.1	0.99986	0.99997	0.99975	0.99994	2	0.99996	1.00000	1.00000	0.99999	241
IIR	0.99983	0.99995	0.99989	0.99988	0.2	0.99927	0.99958	0.99963	0.99984	3	0.99996	0.99999	0.99999	0.99999	366
MD5	0.00000	0.00000	0.00000	0.00000	0.1	0.99997	0.99936	0.99983	0.99983	3	0.99995	0.99995	0.99995	0.99995	448
DES3	0.99992	0.99820	0.00000	0.00000	0.4	0.99995	0.99951	0.99836	0.99679	4	0.99996	0.99996	0.99996	0.99992	612
SHA256	0.00000	0.00000	0.00000	0.00000	0.3	0.99366	0.99331	0.95100	0.95100	3	0.99954	0.99967	0.99967	0.99967	1325

[19] C. Pilato et al. *On the Optimization of Behavioral Logic Locking for High-Level Synthesis*. arXiv:2105.09666. 2021.

[20] C. Pilato et al. "TAO: Techniques for Algorithm-Level Obfuscation during High-Level Synthesis". In: *DAC*. 2018.

[21] C. Prabuddha et al. "SURF: Joint Structural Functional Attack on Logic Locking". In: *HOST*. 2019, pp. 181–190.

[22] R. S. Rajarathnam et al. "ReGDS: A Reverse Engineering Framework from GDSII to Gate-level Netlist". In: *HOST*. 2020, pp. 154–163.

[23] J. Rajendran et al. "Belling the CAD: Toward Security-Centric Electronic System Design". In: *IEEE Transactions on CAD* 34.11 (2015), pp. 1756–1769.

[24] S. Saha. "Emerging Business Trends in the Microelectronics Industry". In: *Open Journal of Business and Management* 04 (2016).

[25] K. Shamsi et al. "Circuit Obfuscation and Oracle-Guided Attacks: Who Can Prevail?" In: *GLSVLSI*. 2017, pp. 357–362.

[26] K. Shamsi et al. "IP Protection and Supply Chain Security through Logic Obfuscation: A Systematic Overview". In: *ACM Trans. Des. Autom. Electron. Syst.* 24 (2019).

[27] M. M. Shihab et al. "Design Obfuscation through Selective Post-Fabrication Transistor-Level Programming". In: *DATE*. 2019.

[28] D. Sisejkovic et al. "Challenging the Security of Logic Locking Schemes in the Era of Deep Learning: A Neuroevolutionary Approach". In: *J. Emerg. Technol. Comput. Syst.* 17.3 (2021).

[29] S. Takamaeda-Yamazaki. "Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL". In: *ARC*. 2015, pp. 451–460.

[30] B. Tan et al. *Benchmarking at the Frontier of Hardware Security: Lessons from Logic Locking*. arXiv:2006.06806. 2020.

[31] US Dep. of Justice. *Departments of Justice and Homeland Security Announce 30 Convictions in Counterfeit Network Hardware*. Available at: https://tinyurl.com/y9p978ka (Last access: Apr. 1, 2020). 2010.

[32] S. Vasudevan et al. "Efficient Model Checking of Hardware Using Conditioned Slicing". In: *Electron. Notes Theor. Comput. Sci.* 128 (2005), pp. 279–294.

[33] M. Yasin et al. "On Improving the Security of Logic Locking". In: *IEEE Transactions on CAD* 35 (2016), pp. 1411–1424.

[34] M. Yasin et al. "SFLL-HLS: Stripped-Functionality Logic Locking Meets High-Level Synthesis". In: *ICCAD*. 2019, pp. 1–4.

[35] Y. Zhang et al. "TGA: An Oracle-Less and Topology-Guided Attack on Logic Locking". In: *ASHES*. 2019, pp. 75–83.