Optimizing the Use of Behavioral Locking for High-Level Synthesis

Christian Pilato\textsuperscript{\textdegree}, Senior Member, IEEE, Luca Collini, Luca Cassano\textsuperscript{\textdegree}, Member, IEEE, Donatella Sciuto\textsuperscript{\textdegree}, Fellow, IEEE, Siddharth Garg\textsuperscript{\textdegree}, Member, IEEE, and Ramesh Karri\textsuperscript{\textdegree}, Fellow, IEEE

Abstract—The globalization of the electronics supply chain requires effective methods to thwart reverse engineering and intellectual property (IP) theft. Logic locking is a promising solution, but there are many open concerns. First, even when applied at a higher level of abstraction, locking may result in significant overhead without improving the security metric. Second, optimizing a security metric is application-dependent and designers must evaluate and compare alternative solutions. We propose a metaframework to optimize the use of behavioral locking during the high-level synthesis (HLS) of IP cores. Our method operates on chip's specification (before HLS) and it is compatible with all HLS tools, complementing industrial EDA flows. Our metaframework supports different strategies to explore the design space and to select points to be locked automatically. We evaluated our method on the optimization of differential entropy, achieving better results than random or topological locking: 1) we always identify a valid solution that optimizes the security metric, while topological and random locking can generate unfeasible solutions; 2) we minimize the number of bits used for locking up to more than 90% (requiring smaller tamper-proof memories); and 3) we make better use of hardware resources since we obtain similar overheads but with higher security metric.

Index Terms—Hardware security, high-level synthesis (HLS), intellectual property (IP) protection, logic locking.

I. INTRODUCTION

Due to the end of Dennard scaling, modern system-on-chip (SoC) architectures are increasingly complex and heterogeneous, integrating several processor cores, memories, and specialized hardware accelerators [1]. Such complexity is pushing the design of integrated circuits (ICs) toward system-level methods based on high-level synthesis (HLS) [2]. Fig. 1(a) shows an example of HLS-based IC design flow, where the designers use HLS tools to automatically translate high-level, C-based specifications into register-transfer level (RTL) descriptions. Logic and physical synthesis generate the layout files ready for fabrication. HLS allows designers to raise the abstraction level, focusing on the behavior rather than hardware details and significantly improving design productivity.

At the same time, IC's manufacturing costs are growing. For example, the equipment becomes 5× more expensive when scaling from 90 to 7 nm [3]. Many semiconductor companies cannot afford these costs and are becoming fab-less, outsourcing the IC fabrication to third-party foundries. This process creates security concerns [4]. Since the foundry has access to the design files, a rogue employee can analyze them to steal the intellectual property (IP) and create illegal IC copies [5]. Design companies are using several techniques to thwart reverse engineering and IP counterfeiting [6].

Logic locking is a well-known technique for IP protection [7]. A high-level view of locking-aware design flow is depicted in Fig. 1(b). At design time, gates are added to hide the correct function. These gates are controlled through an additional input signal (locking key) that is known to the design house but not to the foundry. After fabrication, the design house can activate the correct IC function by placing
the locking key in a tamper-proof key-storage element [8].
This process can apply at different abstractions and assumes
the attacker does not have and cannot guess the locking key.
An IC with an incorrect key produces wrong results. On the
other hand, the attackers should not be able to determine
which results are clearly wrong to rule out incorrect keys
and reduce the search space. While locking has been widely
studied, many open issues remain [9]. First, it must provide
sufficient security protection from structural and functional
viewpoints without suggesting to the attacker which keys are
clearly wrong. Second, the cost should be minimized [10].
Third, the technology of the key-storage can limit the number
of key bits that can be used. However, the effects of locking
depend on the chip function and are difficult to be predicted.

Behavioral locking addresses these concerns by locking a
design at a higher level of abstraction [10]. Behavioral lock-
ing methods operate at or above RTL and allow designers to
protect semantic information before it is optimized and embed-
ded into the netlist by logic synthesis. These methods scale
to larger designs by reasoning about the design behavior instead
of netlist structure. Their industrial adoption is limited since
they require custom HLS tools. A valid alternative is to operate
at the specification level (e.g., the input C code) [11], assuming
that HLS preserves the behavior, including the locking effects.
However, in both cases, designers miss a method to select
which elements to lock, incurring overheads and producing
weak or infeasible solutions [10].

This work follows the key idea that locking all elements
of a design does not necessarily provide maximum security.
The effects of some locking transformations may have lim-
ited visibility on the outputs or can be partially canceled out
by other transformations. The optimization process is applica-
tion dependent and requires design space exploration. The
selection should be guided by the analysis of the effects on
the security metric. A designer must explore the application
of locking transformations to identify the combination that maxi-
mizes the security metric while limiting the resource overhead.
So, optimizing a security metric requires a complex design
space exploration that depends on the effects of the locking
transformations on the design function.

We propose a design framework to explore the functional
effects of existing locking techniques at the C level and
optimize their use. Our main contributions are as follows.

1) A metamodel that integrates state-of-the-art C-level
locking, which allows the use of HLS tools to generate
locked RTL, enabling integration into IC design flows.
2) A design-space exploration strategy (solution encoding
and metaheuristics) to select the best combination of
locking points to optimize given security metrics.
3) A proof-of-concept implementation for the optimization
of different entropy with a standard genetic
algorithm (GA).

Since our framework uses a standard integer-based encoding
of the solutions, the designer has the possibility to integrate
any state-of-the-art exploration algorithms.

The remainder of this article continues as follows. After intro-
ducing the threat model and motivating the work (Section II),
we present our design framework to apply behavioral locking with
the support of commercial HLS (Section III). In this section,
we also detail the different components: solution representa-
tion and analysis (Section III-A), design space exploration
(Section III-B), and solution evaluation (Section III-C). Finally,
we present a proof-of-concept implementation and evaluation
of our approach (Section IV).

II. PROBLEM DEFINITION

In the following, we show that identifying the points to
be lock is a complex and application-dependent problem that
requires to explore the design space.

Problem Formulation: Given a C specification and a locking
key $K$, select the design points to be locked along with the
corresponding parameters, such that the corresponding RTL
solution has two properties: 1) it is one of the solutions with
the best security metric and 2) it requires the minimal amount
of hardware resources compared to other solutions.

This problem formulation has the optimization of the secu-
ricity metric as the primary goal, determining the design with
minimal resources only afterward. In the rest of this sec-
tion, we define the threat model and the security metric
(along with a motivating example) that we consider for the
proof-of-concept implementation in this work.

A. Threat Model

We base our work on existing solutions for behavioral and
RTL locking [10], [12]. These methods assume an untrusted
foundry that wants to identify the functionality of the given IC,
i.e., the correct RTL implementation of the IP and its behavior
over time, to make illegal IC copies. The untrusted foundry has
access to the layout files of the locked chip. From these files,
the foundry can reverse engineer the types of modules used in
the design (i.e., registers, functional units, and interconnection
elements) and can identify the operations executed by each
functional unit [13]. With this RTL description, the foundry
can perform RTL simulations with different input and locking
key values to extract information from the circuit that can help
reconstruct the functionality. If successful, the foundry has
the possibility of creating illegal copies of the IP.

In this work, we assume the untrusted foundry has neither
access to the correct key nor to a functioning unlocked IC
(oracle). This model is common for low-volume IC customers
where the activated chips are used and available only in sen-
sitive designs (e.g., U.S. DoD). Even in consumer electronics,
when the foundry is fabricating the chip for the first time, we
can assume that an activated chip is not yet available [9].

When no activated chip is available, SAT attacks are not
possible and the attacker can only use random methods and
the defender has to make all possible key-dependent variants
equally plausible without leaking any additional information to
the attacker [14]. Indeed, behavioral locking has been demons-
trated to be able to thwart a wide range of attacks when the
attacker has no access to an unlocked chip [12]. The resulting
solutions can be then combined with scan-based methods to
protect the key against oracle-based attacks [15].

B. Locking Evaluation

Security evaluation is based on the assumption that only
the chip activated with the correct key produces the expected
results, while the other keys introduce errors making the corresponding chips unusable [12]. So, we evaluate the locking of a design \( s \) based on the effects on the output results. Given \( N \) output bits, we compute the average differential entropy [14] as follows:

\[
H_s = \frac{1}{N} \sum_{i=1}^{N} \left( P_i^t \cdot \log \frac{1}{P_i^t} + (1 - P_i^t) \cdot \log \frac{1}{1 - P_i^t} \right)
\]

(1)

where \( P_i^t \) is the probability that the output bit \( i \) of the locked design \( s \) results different from its correct value. The probability \( P_i^t \) is estimated with random simulations, where different test cases (i.e., input sequences) and wrong key values are applied. Let \( T \) and \( W \) be the number of input sequences and wrong keys that have been provided for evaluation, respectively. The probability \( P_i^w \) of each output bit \( i \) is computed as

\[
P_i^w = \frac{\sum_{w=1}^{W} \sum_{t=1}^{T} \text{OUT}[i]_t^w \oplus \text{OUT}[i]_w^t}{W \cdot T}
\]

(2)

where \( \text{OUT}[i]_t^w \) represents the correct value of the output bit \( i \) when the input sequence \( t \) is tested, while \( \text{OUT}[i]_w^t \) represents the actual value of the same output bit when the wrong key \( w \) is provided to the given solution \( s \) together with the same input sequence \( t \).

The differential entropy metric is used to quantify output corruptibility, i.e., how much the locking techniques affect the outputs. This value should be maximized to avoid leaking any information on the correct output values to the attacker. Since \( 0 \leq P_i \leq 1 \), (1) has a maximum value \( H_s \) when \( P_i = 0.5 \) for each output bit \( i \) (see Fig. 2). This corresponds to the case where each output bit assumes value 0 or 1 with equal probability when wrong key values are applied. As a result, the attacker has no information on the correct output values and can only make random guesses. For this reason, our framework aims at maximizing \( H_s \). Although we used differential entropy, our methodology is general and requires a security metric to evaluate each candidate for the given threat model.

C. Behavioral Locking

Behavioral locking hides parts of the function (e.g., constants, control branches, and arithmetic operations) based on the locking key \( K \). It can be applied on C code [11], during HLS [10], or at RTL [12]. The key \( K \) is provided by the designer through an input port and partitioned into subkeys to lock each element, as shown in Fig. 3. The circuit will work correctly only when the correct key is given. This approach is more scalable than gate-level locking, protecting the semantics of the design instead of its structural netlist. We consider the following behavior locking techniques [10]–[12].

Control Branch Locking: Branches in the input behavior can be locked to hide the control flow. Each condition can be locked with one bit key. The condition \( c_p \) is modified as \( c_p \oplus k_j \), where \( k_j \) is a one-bit key. This \( k_j \) is part of the locking key \( K \) and locks this condition checking. The required branch is taken only when the correct \( k_j \) is provided.

Operation Locking: Fake operations are added to hide real RTL operations. Given an operation \( t \) to be locked, the outputs of the two operations (correct and fake) are multiplexed by a key bit \( k_t \). The correct output is connected to 0 or 1 input of this MUX based on the value of \( k_t \). Only with the correct key, the correct operation results are produced.

Constant Locking: We assume a predefined number of bits \( x \) to implement all constants, typically 32 bits (corresponding to an integer value in C), regardless the real bit widths. Each constant \( c_i \) of the behavior is locked as \( c'_i = c_i \oplus k_i \), where \( c'_i \) is the locked value stored in hardware and \( k_i \) is a \( x \)-bit key. The correct constant can be obtained in hardware by reversing the operation, i.e., \( c_i = c'_i \oplus k_i \).

We define a locking point as any of the RTL elements (i.e., a control branch, an operation, or a constant) where it is possible to apply the given locking techniques.

D. Motivating Example

While behavioral locking is a powerful solution to hide the IC functionality, we argue that locking a large number of locking points may produce a large overhead [10] without necessarily improving the given security metric. Consider locking the cyclic redundancy check (CRC) code IP. For simplicity, we use the Bambu HLS tool [16] targeting a Xilinx Virtex-7 XC7VX690T FPGA at 100 MHz. The algorithm has five operations and seven constants that can be locked with 167 bits. When we constrain behavioral locking to use no more than 50% of these bits and we use the TAO approach [10] (this is also known as topological locking), the RTL has an overhead of 1430 look-up tables (LUTs) and 815 flip-flops (FFs) compared to the unlocked version. Differential entropy of the design is \( \sim50.53 \) (where maximum is 64) and the algorithm locks all operations and two constants. A high differential entropy (63.08) can be achieved by selecting and locking only five operations and one constant. This uses 730 LUTs and 385 FFs with a reduction of overhead by about 50%. Thus, optimizing behavioral locking is important to improve security metric and reduce overhead.

III. PROPOSED EXPLORATION METAFRAMEWORK

We propose a modular and integrated metaframework (see Fig. 4) to optimize the use of behavioral locking during
HLS. The input is a synthesizable C code of the accelerator. Behavioral locking is applied as a source-to-source transformation on such input C code. In this way, we can leverage existing HLS tools for generating the locked RTL description. We assume that the input C code is already synthesizable with the given HLS tool. Since we consider a metric that analyzes only the IC behavior and we assume that HLS-generated designs have the same behavior of the corresponding input C codes, we can perform security assessment directly on the locked C code. This approach is much faster than performing RTL simulations, enabling its use in an exploration framework. To perform locking, we provide the locking key \( K \). The locking key must be independent of the design to avoid that the attacker can infer it. So, it is an input of our methodology. The size of the locking key determines the maximum number of bits that can be used for locking, limiting the locking techniques that can be applied. To compute the differential entropy of each candidate solution (see Section II-B), the framework requires the corresponding C-based test-bench and a set \( T \) of representative inputs to evaluate the behavior of locked circuit versions with correct and incorrect keys. Such input vectors are the same that are used to evaluate the circuit functionality. An additional set \( W \) of wrong keys is also provided. Each wrong key in \( W \) has the same length as the locking key \( K \) and is randomly generated by altering the correct key. By leveraging the given HLS tool, the framework outputs an RTL description of the best locked solution that is ready for the front- and back-end synthesis steps.

Our exploration framework operates as follows. First, we execute the input C code on the set \( T \) of representative inputs to compute the golden outputs. These values will be used to assess the effects of applying the candidate set of locking techniques to the input C code. We parse the input C code to build the corresponding abstract syntax tree (AST) of the functionality to be locked. Each AST node describes a construct occurring in the source code. This representation aids the next steps since it can be analyzed to identify locking points and edited to create alternative locked versions. The rest of the framework has three main steps.

1) **Analysis:** We analyze the input C description to identify the potential locking points. A locking point is an element of the algorithm (i.e., constant, operation, and branch condition) that can be potentially locked based on the available techniques (see Section II-C).

2) **Exploration:** We perform design space exploration to identify the subset of solutions that optimize the given security metric. Each solution represents a combination of decisions concerning how to apply the techniques to each locking point. The corresponding locked C codes are generated by applying the locking techniques specified in the solutions to evaluate the security metric.

3) **Selection:** We apply HLS on the set of locked C codes produced in the previous step and we determine the cost of the resulting RTL designs to select the final solution, which is our best design.

In the exploration phase, we can use several strategies, ranging from random changes (similar to the approach used for logic locking in [17]) to complex metaheuristics, such as GAs and simulated annealing (SA). Metaheuristics are based on the observation of natural behaviors. For example, SA is inspired by annealing in metallurgy to create perturbations and move around the design space and a GA maintains a population of alternative solutions to be recombined. These algorithms perform well in the identification of substructures in the problem [18], [19].

### A. Identification and Representation of Locking Points

During the analysis step, we perform a depth-first analysis of the AST of the input C code to identify the potential locking points in the design region to be protected. The type and number of locking points depend on the algorithm to be implemented and the locking techniques. We identify all potential locking points in the candidate region considering the techniques described in Section II-C as follows.

1) **Constants:** We lock constants with a predefined number \( B_c \) of key bits. The number of key bits is the same for all constants to prevent information leakage about the constant range. Each constant is represented with 0/1 value to specify whether a constant value must be locked. Each constant to be locked (i.e., with the corresponding value set to 1) requires \( B_c \) key bits.

2) **Operations:** We lock logic and arithmetic operations with extra fake operations. For each operation type, we preselect a set of alternative types. Each operation \( o \) is represented in the corresponding vector element with a value that ranges between 0 (no locking) and \( N_o \), where \( N_o \) is the number of alternative operation types predefined for the type of operation \( o \). Each locked operation (i.e., with a value different from 0) requires 1 key bit to multiplex the output of the correct operation with the fake one.

3) **Branches:** We lock the control flow (e.g., if/else statements or ternary operators) with key bits, reordering branches as needed. Each condition evaluation is represented in the solution with 0/1 value to specify whether the corresponding branch is locked or not. Each locked branch (i.e., value set to 1) requires 1 key bit.

When a locking point \( i \) has \( O_i \) alternatives, the decision can be represented with an integer value between 1 and \( O_i \) when
it should be locked and 0 otherwise (see the upper part of Fig. 5). For example, if an addition can be locked with two types of “fake” operations: 1) subtraction and 2) multiplication, the corresponding element can take on: 0 (no locking), 1 (lock with subtraction), and 2 (lock with multiplication). On the contrary, a control branch can assume only two values: 0 (no locking) and 1 (locking). So, the analysis creates a vector of integers that represents decisions for all locking points. Fig. 5 shows a solution encoding for a simple algorithm. The integer vector represents a tor of integers that represents decisions for all locking points.

B. Exploration Phase and Security Assessment

In the exploration phase, we can use and compare different search methods to identify the best combination of locking techniques for the input C code. Our optimization framework can use any exploration method that is able to manipulate a vector of integers like in Fig. 6.

As a proof-of-concept, we implemented a standard GA with integer encoding. GAs maintain a population of $N$ alternative solutions (initialized randomly) and recombine them to identify the best subset of locking techniques. Classic GA operators, such as random mutation and single-point crossover, are applied with probability $P_m$ and $P_c$, respectively, to generate offspring solutions. At the end of each generation, all individuals are ranked based on the given security metric, checking for the best solution and passing the best individuals to the next generation. The procedure terminates when the solution is not improved for some generations or we reach the limit on the number of generations.

Each solution $s$ encodes locking transformations to be applied. First, we compute the number of key bits [see (3)] to determine if the solution is feasible. We then proceed with security assessment. We consider a threat model without an oracle. So we optimize the differential entropy (i.e., effects on the output values), which is a behavioral metric. We perform security evaluation on the locked C code by computing differential entropy in (1). We apply locking techniques to the C code in the solution vector. This new code is compiled with the testbench and executed on test vectors for each alternative wrong key $w \in W$. The outputs are compared with golden outputs to compute differential entropy $H_s$. The exploration phase maximizes this value. If the designers want to
tradeoff more (security) metrics, they can use a linear combination of the corresponding values or perform multiobjective optimization [20].

C. Resource Evaluation and Selection

Applying different locking techniques can lead to the same security level but with different overheads. Once we identify the solutions that optimize the given security metric, we evaluate their resource consumption. First, to increase the number of solutions to evaluate, we pass to the selection phase solutions whose security metric is within a predefined range from the best ones. We obtain more solutions with a minimal degradation (predefined by the designer) on the security metric. We perform commercial HLS on the locked C codes to obtain the corresponding locked RTLs. We rank these RTL designs according to the use of hardware resources, selecting the best as the final solution.

IV. EXPERIMENTAL RESULTS

To validate our solution, we implemented a prototype in Python. We used pycparser parser (ver. 2.19) for C manipulation (analysis and locking) and the DEAP framework (ver. 1.30) [21] for the GA-based DSE. Due to the stochastic nature of the GA, we averaged the results over 30 runs.

We selected eight benchmarks from the Bambu [22], MiBench [23], and CHStone [24] suites. The benchmarks have been selected because used for HLS-based locking [10] and already supported by the given HLS tool. Table I characterizes benchmarks in terms of locking points (branches, operations, and constants) and the total number of key bits required for complete locking. Benchmarks are ordered by increasing the number of total key bits.

We configured the GA as follows. GA population has 300 individuals evolved for 1000 generations or until the best fitness value does not improve for ten consecutive generations. Crossover and mutation probabilities are set to $P_c = 0.5$ and $P_m = 0.2$. Single-element mutation probability is set to $P_l = 0.05$. The initial population is randomly created. For each benchmark, we consider 100 input sets to evaluate the differential entropy. For each benchmark, we generate four keys of different length, namely, $c_1$, $c_2$, $c_3$, and $c_4$, to evaluate effect of key size (25%, 50%, 75% and 100% of the required key bits). For each key, we generate 100 random variants that represent wrong keys for security evaluation [14]. We compare our solution with TAO [10], a state-of-the-art behavioral locking technique that locks the elements depth first (i.e., topological locking), and a random locking. For fair comparison, we reimplemented TAO in our framework. In TAO, the security metric is evaluated on the final solution, without security optimization and is identified as TAO in our experiments. Random locking corresponds to the best individual in the initial GA population, i.e., the best solution among 300 alternatives.

A. Security Metric Optimization

This article does not aim at evaluating the security of the locking techniques, which is given (see [12] for more information on the security guarantees), but aims at optimizing their use for a given security metric. For each benchmark $s$, we computed the theoretical maximum $H^*_s$ of the security metric $H_s$ and we normalized the values obtained with DSE and TAO. The perfect differential entropy is thus equal to 1 but it can be impossible to be achieved for some benchmarks due to the nature of their algorithms and operations.

Fig. 7 compares the differential entropy (normalized with respect to the maximum value) of the state-of-the-art topological locking (TAO), the random solutions (RND), and our method (DSE). The results clearly show that topological locking fails to optimize the security metric. The analysis on the AST is not able to predict the effects on the outputs, leading in most of the cases to solutions with differential entropy equal to zero. These cases happen when the locked solutions invalidate the algorithms with fixed outputs (e.g., always equal to zero) or leading to timeouts (e.g., in case of infinite loops). Also, the points where these solutions are invalidated depends on how the algorithm is written. For example, in the crc benchmark, the topological locking invalidates the design only when the locking impacts the second half of the locking points. These invalid solutions are instead discarded by our exploration method that is always able to find solutions that are close to the optimal value. Random locking can achieve good solutions, but it lacks scalability. Indeed, when increasing the key budget (i.e., $c_4$), it is more probable to select invalidating locking points. Our method is instead able to discard those points, thanks to the exploration and recombination of alternatives.

Fig. 8 shows the number of bits used for locking the solutions and the breakdown for the three techniques. First, results show that topological locking uses predefined number of bits proportional to the budget, while our DSE method number of bits independent of the budget (in many cases less than the limit). The number of bits depends on optimizing the security metric rather than the budget. Constant locking has the most impact on differential entropy as it is selected most and uses most of the key bits. Branches are less used since manipulating control flow is likely to produce invalid designs.

B. Locking Overhead

We use Bambu, an open-source HLS tool [16], to generate RTL corresponding to plain and locked C codes. Bambu uses gcc 4.8 targeting a Xilinx Virtex-7 FPGA XC7VX690T with a clock period of 10 ns. We targeted FPGAs since the ASIC backend of Bambu is only partially supported [22]. However,

### TABLE I

| Benchmark | Suite  | #Ctrl. | #Op. | #Const. | #Bits |
|-----------|--------|--------|------|---------|-------|
| arf       | Bambu  | 28     | 0    | 0       | 28    |
| patricia  | MiBench| 2      | 9    | 3       | 107   |
| bubblesort| Bambu  | 11     | 4    | 139     |       |
| crc       | MiBench| 5      | 7    | 167     |       |
| sha       | MiBench| 76     | 40   | 1356    |       |
| adpcm     | CHStone| 121    | 69   | 2336    |       |
| aes       | CHStone| 111    | 149  | 4883    |       |
| gsm       | CHStone| 231    | 172  | 5784    |       |

This table characterizes the benchmarks in terms of locking points (branches, operations, and constants) and the total number of key bits required for complete locking.
Fig. 7. Differential entropy comparison between our work (DSE) and topological locking (TAO) for different key budgets. (a) arf. (b) patricia. (c) bubblesort. (d) crc. (e) sha. (f) adpcm. (g) aes. (h) gsm.

Fig. 8. Number of locking bits used by our DSE and TAO topological locking for different key budgets. Each bar reports the number of bits used for locking constants (∗-C), operations (∗-O), and branches (∗-B). (a) arf. (b) patricia. (c) bubblesort. (d) crc. (e) sha. (f) adpcm. (g) aes. (h) gsm.

the results are comparable between the technologies. Logic synthesis is done using Xilinx Vivado 2019.2.

Table II reports the characteristics of the designs obtained from the plain (unlocked) C codes and the overheads of TAO and DSE for different key budgets (c1–c4). We report LUT and FF, along with DSP and BRAM elements. We also report total power consumption (in mW) of the synthesized accelerators. Our DSE has a better use of resources than TAO. There are cases when LUT overhead is more, due to the complex alternative operators. Our method selects fake operators to optimize for security, while TAO uses a predefined alternative. DSP changes are minimal and limited to cases where fake operations are implemented as multipliers. BRAM elements are generally not affected because locking is not applied to memory elements. There are few cases where constant values cannot be converted into BRAM LUTs, reducing the number of these elements and increasing logic. Power consumption is incremented proportionally to the additional logic. Using behavioral locking with different operators affects the HLS scheduling and liveness of temporary values. This impacts the number of registers and number of FFs. On the other hand, performing HLS and logic synthesis after behavioral locking
TABLE II

| Hardware Resources and Corresponding Overheads for Topological Locking (TAO) and Our Work (DSE) |
|--------------------------------------------------|
| **Resources** | **Plain** | **#LUT** | **FF** | **DSP** | **BRAM** | **Power [mW]** |
| c1 | c2 | c3 | c4 | c1 | c2 | c3 | c4 | c1 | c2 | c3 | c4 | c1 | c2 | c3 | c4 | c1 | c2 | c3 | c4 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Plain | | 644 | 185 | 374 | 285 | | 247 | 123 | 233 | 264 | | 181 | 1387 | 1602 | 1462 | 1846 | 1842 | | 0 | 0 | 0 | 0 |
| TAO | | +523 | +817 | +1387 | +1602 | +2162 | +1769 | +1846 | +1842 | +143 | +315 | +498 | +955 | +1552 | +1430 | +1860 | +2149 | | 442 | 377 | 328 | 327 |
| Plain | | +564 | +1841 | +2232 | +1982 | +510 | +144 | +263 | +112 | +673 | +688 | +673 | +267 | +1441 | +730 | +1471 | +1522 | | -11 | -9 | -25 | -17 |
| TAO | | +529 | +2768 | +3825 | +4042 | +5956 | +5784 | +8583 | +9613 | +609 | +2928 | +2665 | +2989 | +8730 | +13206 | +11923 | +12951 |
| Plain | | -15 | -12 | -24 | -30 | - | - | - | - | 67 | 14 | 8 | 49 | 3574 | 9641 | 6954 | 6954 | | 442 | 377 | 328 | 327 |
| TAO | | -15 | -6 | -14 | -14 | +12 | +7 | +1 | +1 | -4 | +26 | +12 | +12 | +13 | +22 | +27 | | -11 | -9 | -25 | -17 |
| DSE | | -15 | -11 | -20 | -23 | - | - | - | - | 67 | 14 | 8 | 49 | 3574 | 9641 | 6954 | 6954 | | 442 | 377 | 328 | 327 |

C. Design Space Exploration Performance

Table III shows the number of alternative solutions that are identified during exploration. Several solutions (up to 100) can reach a similar level of security. Remember that we maintain the solutions that are within an $\epsilon$-distance from the best one. Solutions that are resource hungry are filtered during selection phase. The design exploration that we perform is more expensive than single-run heuristics such as TAO. While TAO completes the locking in few seconds in the worst case, our DSE engine requires many hours to complete (up to one day in the worst case). Table III shows the number of generations required to converge to “stable” solutions. Fig. 9 shows the evolution of the DSE runs for two benchmarks ($\text{sha}$ and $\text{gsm}$) when the key budget is 75%. This is an interesting trade-off between a large design space and the constraint given by the number of key bits. The graphs show how the explorations progress toward better values for the security metrics. For $\text{sha}$, the best solutions are easy to find and the best value of the metric is almost immediate. The exploration terminates quickly. For the $\text{gsm}$ benchmark, the DSE requires more generations to optimize the metric since the design is complex. The average is sometimes decreasing because the exploration phase introduces new individuals that are not necessarily better than...
the previous ones. However, ranking and selection procedures keep the best ones and use the worst ones to explore alternative regions of the design space. These results show that the method is robust and converges in about 20–30 generations. The execution time is affected mostly by time to compile the C code and time to evaluate security. Complex benchmarks require more time even if they have small code. They may lead to longer execution times when invalid solutions time out. The execution times are order of magnitude lower than RTL simulations. However, the improvements justify the extra time required by the designer to explore different solutions. Such exploration is performed only once during the design phase.

V. RELATED WORK

Several approaches have been proposed to protect IP cores from reverse engineering and IP theft, including split manufacturing, camouflaging, watermarking, and logic locking. Split manufacturing divides the IC design in two parts that are fabricated in separate foundries [25], [26]. Camouflaging hides the Boolean function of a gate at the layout level [27]. Watermarking allows certifying ownership of IP by embedding a designer’s signature into the design [28]. Logic locking makes the circuit function dependent on a key unknown to the foundry [29]. These techniques can apply at the transistor [30], logic [31], [32], or behavior [13] levels.

Locking modifies the behavior of a circuit, which does not produce the expected outputs until it is “activated” with a secret key. It protects the circuit from illegal copies since the correct execution requires access not only to the IC layout but also the key. This approach protects against attackers with access to the design files [29]. Circuit locking can be performed at logic [17], [33] or behavioral level [34], [35]. Protection techniques depend on the threat model assumed by the designer: if the attacker has access to an activated chip (oracle), locking must resist SAT attacks [36]. If no oracle is available, the attacker can only analyze the design files, which hardly reveal knowledge on the structure or function. The attacker can only apply random guesses. To scale to larger designs approaches raise the abstraction level such as applying locking during HLS [10], [37], even if they require tool modifications. Indeed, DSE at the RTL level has been shown to suffer poor performance because of the high number of locking points [38]. On the contrary, our method can integrate algorithm-level analysis and pruning steps to reduce the number of candidate locking points. They are not compatible with industrial EDA flows. Existing methods propose alternative techniques without explicitly optimizing security metrics [14]. We focus on identifying the best combination of techniques by analyzing the effects on the security metric.

Design space exploration has been recently applied to hardware IP protection [39]–[41]. In [39], the DSE optimizes an area-delay function for IP watermarking. In [40], the designers analyze the effects of restricted design spaces on obfuscated specifications. In [41], locking is applied to selectively protect specific regions of the search space and not the hardware IP core. However, in all cases the security metric was not the primary optimization goal.

VI. POTENTIAL FRAMEWORK EXTENSIONS

We propose a solution to optimize differential entropy for behavioral locking in an oracle-less attack scenario. However, our framework can be extended in several directions.

Locking Techniques: Designers can integrate new locking techniques. They must define the values for the alternatives for the technique and modify the analysis step to generate the elements in the solution vector. They can also develop further analysis and pruning steps to reduce the design space.

DSE Heuristics: Design space exploration metaheuristics are transparent to the locking techniques. Assuming that all techniques are orthogonal with each other, any common operator for design space exploration generates valid solutions by manipulating the vector of integers representing the locking solution, as shown in Fig. 6.

Locking Metrics: Our framework can protect the circuit against oracle-based attacks, where the security metric is resilience to SAT-attacks. In this case, HLS must be performed already during security evaluation to create RTL designs on which SAT attacks are performed [42]. Solutions that are broken (i.e., the key is recovered) can be marked as infeasible and discarded. To limit the execution time of the exploration, the designer can use the number of SAT clauses as a metric that corresponds to the complexity of SAT attacks. Eventually, the designer selects the solution that minimizes overhead among the ones that maximize the resilience to SAT.

VII. CONCLUSION

Although HLS is popular, security constraints are not yet supported by commercial HLS tools. Countermeasures are applied to the code executing on the processors or manually implemented into IP blocks yielding suboptimal and even insecure designs. HLS should consider security side-by-side performance and cost [43]. Recent solutions are adding security awareness into HLS [10], [44], [45]. They are not yet mature for industrial adoption. Our method optimizes IP cores with locking before HLS while limiting the overhead via design space exploration at the C level. Locked RTL is obtained by using any HLS tools. This is a pathway for behavioral locking of industrial designs using commercial design flows. The proposed locking maximizes a given security metric (i.e., differential entropy) by exploring the locking effects with a GA. Results demonstrate that full locking is
not necessary to maximize security. By selecting the locking points one can maximize security while limiting resource overhead. Operating at the C level makes our solution compatible with commercial HLS. Future research will work in two directions. To improve the framework, we will evaluate and compare alternative DSE techniques. To expand its application, we will apply it to new scenarios and security metrics.

REFERENCES

[1] M. Horowitz, “1.1 computing’s energy problem (and what we can do about it),” in *IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers (ISSCC)*, 2014, pp. 10–14.

[2] R. Nane et al., “A survey and evaluation of FPGA high-level synthesis tools,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591–1604, Oct. 2015.

[3] S. W. Jones, *Technology and Cost Trends at Advanced Nodes*, IC Knowl., Georgetown, MA, USA, 2019.

[4] J. Hurtarte, E. Wolsheimer, and L. Tafoya, *Understanding Fabless IC Technology*. Amsterdam, The Netherlands: Elsevier, Aug. 2007.

[5] U. Guin, K. R. DiMase, J. M. Carulli, M. Tehranipoor, and Y. Makris, “Counterfeit integrated circuits: A rising threat in the global semiconductor supply chain,” *IEEE Proc.*, vol. 102, no. 8, pp. 1207–1228, Aug. 2014.

[6] M. M. TehraniPoo, U. Guin, and D. Forte, *Counterfeit Integrated Circuits: Detection and Avoidance*. Cham, Switzerland: Springer, 2015.

[7] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, “Security analysis of logic obfuscation,” in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, Jun. 2012, pp. 83–89.

[8] M. T. Rahman et al., “Defense-in-depth: A recipe for logic locking to prevail,” *Integration*, vol. 72, pp. 39–57, May 2020.

[9] K. Shamsi, M. Li, K. Plaak, S. Fazzari, D. Z. Pan, and Y. Jin, “IP protection and supply chain security through logic obfuscation: A systematic overview,” *ACM Trans. Design Autom. of Electron. Syst.*, vol. 24, no. 6, p. 65, Nov. 2019.

[10] C. Pilato, F. Regazzoni, R. Karri, and S. Garg, “TAO: Techniques for algorithm-level obfuscation during high-level synthesis,” in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2018, pp. 1–6.

[11] H. Badier, J. L. Lann, P. Cousoy, and G. Gogniat, “Transient key-based obfuscation for HLS in an untrusted cloud environment,” in *Proc. IEEE Design Autom. Test Europe Conf. (DATE)*, Mar. 2019, pp. 1118–1123.

[12] C. Pilato, A. B. Chowdhury, D. Sciuto, S. Garg, and R. Karri, “ASSURE: RTL locking against an untrusted foundry,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 29, no. 7, pp. 1306–1318, Jul. 2021.

[13] J. Rajendran, A. Ali, O. Sinanoglu, and R. Karri, “Belling the CAD: Toward security-centric electronic system design,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 11, pp. 1756–1769, Nov. 2015.

[14] S. Amir et al., “Development and evaluation of hardware obfuscation benchmarks,” *J. Hardw. Syst. Security*, vol. 2, pp. 142–161, Jun. 2018.

[15] N. Limaye et al., “Fortifying RTL locking against oracle-less (untrusted foundry) and oracle-guided attacks,” in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2021, pp. 91–96.

[16] C. Pilato and F. Ferrandi, “Bambu: A modular framework for the high level synthesis of memory-intensive applications,” in *Proc. IEEE Conf. Field-Programmable Logic Appl.* (FPL), 2013, pp. 1–4.

[17] J. Roy, F. Koushanfar, and I. Markov, “EPIC: Ending piracy of integrated technology,” *IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers (ISSCC)*, vol. 38, no. 10, pp. 211–227, Sep. 2004.

[18] A. Chakraborty et al., “Keynote: A disquisition on logic locking,” in *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 1952–1972, Oct. 2020.

[19] M. M. Shihab et al., “Design obfuscation through selective post-fabrication transistor-level programming,” in *Proc. IEEE Design Autom. Test Europe Conf. (DATE)*, 2019, pp. 528–533.

[20] A. Sengupta, M. Nabeel, M. Asif, and O. Sinanoglu, “Customized locking of IP blocks on a multi-million-gate SoC,” in *Proc. ACM/IEEE Int. Conf. Comput.-Aided Design (ICCAD)*, 2018, pp. 1–7.

[21] S. Patnaik, M. Asif, J. Knechtel, and O. Sinanoglu, “Obfuscating the interconnects: Low-cost and resilient full-chip layout camouflaging,” in *Proc. ACM/IEEE Int. Conf. Comput.-Aided Design (ICCAD)*, 2017, pp. 41–48.

[22] M. Yasin, J. Rajendran, O. Sinanoglu, and R. Karri, “On improving the security of logic locking,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 9, pp. 1411–1424, Sep. 2016.

[23] R. Chakraborty and S. Bhunia, “RTL hardware IP protection using key-based control and data flow obfuscation,” in *Proc. Int. Conf. VLSI Design*, 2010, pp. 405–410.

[24] F. Koushanfar, “Provably secure active IC metering techniques for piracy avoidance and digital rights management,” *IEEE Trans. Inf. Forensics Security*, vol. 7, pp. 51–63, 2012.

[25] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin, “Circuit obfuscation and oracle-guided attacks: Who can prevail?” in *Proc. ACM Great Lakes Symp. VLSI (GLSVLSI)*, 2017, pp. 357–362.

[26] M. Yasin, C. Zhao, and J. R. Rajendran, “SFL-HLS: Stripped-functionality logic locking meets high-level synthesis,” in *Proc. ACM/IEEE Int. Conf. Comput.-Aided Design (ICCAD)*, 2019, pp. 1–4.

[27] L. Collini, R. Karri, and C. Pilato, “A composable design space exploration framework to optimize behavioral locking,” in *Proc. IEEE Design Autom. Test Europe Conf. (DATE)*, 2022, pp. 1–6.

[28] A. Sengupta and S. Bhaduria, “Untrusted third party digital IP cores: Power-delay trade-off driven exploration of hardware trojan secured Datapath during high level synthesis,” in *Proc. Great Lakes Symp. VLSI (GLSVLSI)*, 2015, pp. 167–172.

[29] Z. Wang and B. C. Schaefer, “Partial encryption of behavioral IPs to selectively control the design space in high-level synthesis,” in *Proc. IEEE Design Autom. Test Europe Conf. (DATE)*, 2019, pp. 642–645.

[30] Z. Wang and B. C. Schaefer, “Locking the re-usability of Behavioral IPs: Discriminating the search space through partial encryptions,” in *Proc. IEEE Design Autom. Test Europe Conf. (DATE)*, 2021, pp. 42–45.

[31] C. Karfa, R. Chouksey, C. Pilato, S. Garg, and R. Karri, “Is register-transfer level locking secure?” in *Proc. IEEE Design Autom. Test Europe Conf. (DATE)*, 2020, pp. 550–555.

[32] C. Pilato, S. Garg, K. Wu, R. Karri, and F. Regazzoni, “Securing hardware accelerators: A new challenge for high-level synthesis,” *IEEE Embedded Syst. Lett.*, vol. 10, no. 3, pp. 77–80, Sep. 2018.

[33] C. Pilato, K. Wu, S. Garg, R. Karri, and F. Regazzoni, “TaintHLS: High-level synthesis for dynamic information flow tracking,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 5, pp. 798–808, May 2019.

[34] N. Veeranna and B. C. Schaefer, “Efficient behavioral intellectual properties source code obfuscation for high-level synthesis,” in *Proc. IEEE Latin-American Test Symp. (LATS)*, Mar. 2017, pp. 1–6.
Christian Pilato (Senior Member, IEEE) received the Ph.D. degree in information technology from Politecnico di Milano, Milan, Italy, in 2011. He is a Tenure-Track Assistant Professor with Politecnico di Milano. He was a Postdoctoral Research Scientist with Columbia University, New York, NY, USA, from 2013 to 2016 and Università della Svizzera italiana, Lugano, Switzerland, from 2016 to 2018. He was also a Visiting Researcher with New York University, New York; TU Delft, Delft, Netherlands; and Chalmers University of Technology, Gothenburg, Sweden. His research interests include high-level synthesis, reconfigurable systems, and system-on-chip architectures, with emphasis on memory and security aspects.

Dr. Pilato served as the Program Chair of EUC 2014 and is currently the Program Chair of ICCD 2022. He serves in the organizing and program committees of many conferences on EDA, CAD, embedded systems, and reconfigurable architectures, such as DAC, ICCAD, DATE, ASP-DAC, CASES, PPL, FPT, and ICCD. He is a Senior Member of ACM and a member of HiPEAC.

Luca Collini received the M.Sc. degree (summa cum laude) in computer science and engineering from Politecnico di Milano, Milan, Italy, in 2021, along with the Honourable Programme seal, which is a recognition for top-level students in the computer science program. He is currently pursuing the Ph.D. degree with New York University (NYU), New York, NY, USA.

He is an Affiliated Member with the NYU Center for Cybersecurity, NYU. He was also a Research Assistant with Politecnico di Milano until January 2022. His research interests include electronics design automation, intellectual property protection, and System-on-Chip security, with particular emphasis on high-level methods to address these challenges.

Luca Cassano (Member, IEEE) received the B.S. degree in computer engineering, and the M.Sc. and Ph.D. degrees in computer engineering from the University of Pisa, Pisa, Italy, in 2009 and 2013, respectively.

He is a Tenure-Track Assistant Professor with Politecnico di Milano, Milan, Italy. His research activity focuses on the definition of innovative techniques for fault simulation, testing, untestability analysis, diagnosis, and verification of fault-tolerant and secure digital circuits and systems.

Dr. Cassano won the European semifinals of the 2014 TTTC’s E. J. McCluskey Dissertation Award for his Ph.D. thesis, titled “Analysis and Test of the Effects of Single Event Upsets Affecting the Configuration Memory of SRAM-based FPGAs.” He served as the Program Chair for DFTS 2021 and he is currently the Program Chair of DFTS 2022. He serves in the organizing and program committees of several conferences on EDA, CAD, and test, such as ETS, IOLTS, DDECS, and DSD. He is an Associate Editor of Integration, the VLSI Journal, and the Journal of Electronic Testing.

Donatella Sciuto (Fellow, IEEE) received the Laurea (M.S.) degree in electronic engineering from Politecnico di Milano, Milan, Italy, in 1988, the MBA degree from Bocconi University, Milan, in 1992, and the Ph.D. degree in electrical and computer engineering from the University of Colorado, Boulder, CO, USA, in 1988.

She is currently the Executive Vice Rector of the Politecnico di Milano, where she is also a Full Professor of Computer Science and Engineering. More recently, she has been involved in managing projects in the area of smart cities and in the application of new ICT technologies to different application fields. She has published over 300 scientific papers. Her main research interests cover the methodologies for the design of embedded systems and multicore systems considering performance, power, and security metrics.

Prof. Sciuto serves in different capacities in IEEE Awards Committees and scientific boards of IEEE journals and conferences. She has served as the Vice-President of Finance and then the President of the IEEE Council of Electronic Design Automation from 2009 to 2013. She is a Fellow of IEEE for her contributions in embedded system design.

Siddharth Garg (Member, IEEE) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology Madras, Chennai, India, in 2004, and the Ph.D. degree in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, USA, in 2009.

He joined New York University, New York, NY, USA, in Fall 2014, as an Assistant Professor, and prior to that, was an Assistant Professor with the University of Waterloo, Waterloo, ON, Canada, from 2010 to 2014. His general research interests are in computer engineering, and more particularly, in secure, reliable, and energy-efficient computing.

Dr. Garg has received the NSF CAREER Award in 2015, and paper awards at the IEEE Symposium on Security and Privacy in 2016, USENIX Security Symposium 2013, the Semiconductor Research Consortium TECHCON in 2010, and the International Symposium on Quality in Electronic Design in 2009. He also received the Angel G. Jordan Award from the ECE Department, Carnegie Mellon University for outstanding thesis contributions and service to the community. In 2016, he was listed in Popular Science Magazine’s annual list of “Brilliant 10” researchers. He serves on the technical program committees of several top conferences in the area of computer engineering and computer hardware, and has served as a reviewer for several IEEE and ACM journals.

Ramesh Karri (Fellow, IEEE) received the B.E. degree in ECE from Andhra University, Visakhapatnam, India, and the Ph.D. degree in computer science and engineering from the University of California San Diego, San Diego, CA, USA.

He is a Professor of ECE with New York University (NYU), New York, NY, USA. He Co-directs the NYU Center for Cybersecurity, NYU. He founded the Embedded Systems Challenge, the annual red team blue team event, and co-founded Trust-Hub. He published over 250 articles in leading journals and conference proceedings. His research and education activities in hardware cybersecurity include trustworthy ICs, processors and cyber-physical systems, security-aware computer-aided design, test, verification, validation, and reliability, nano meets security, hardware security competitions, benchmarks, and metrics, biochip security, and additive manufacturing security.

Prof. Karri received the Humboldt Fellowship and the NSF CAREER Award. His work on hardware cybersecurity received best paper nominations, such as ICCD 2015 and DFTS 2015, and awards, such as ACM TODAES 2018, ITC 2014, CCS 2013, DFTS 2013, and VLSI Design 2012. He is the Editor-in-chief of ACM Journal on Emerging Technologies in Computing Systems and served(d) as the editorial boards of IEEE Transactions, such as IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, IEEE EMBEDDED SYSTEMS LETTERS, IEEE TRANSACTIONS ON DESIGN & TEST, and IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, and ACM Transactions, such as ACM Transactions on Design Automation of Electronic Systems and ACM Journal on Emerging Technologies in Computing Systems. He was an IEEE Computer Society Distinguished Visitor from 2013 to 2015. He served on the Executive Committee of the IEEE/ACM DAC leading the Security@DAC initiative from 2014 to 2017. He served as the program/general chair of conferences and serves on program committees. He is a Fellow of the IEEE for leadership and contributions to Trustworthy Hardware.