HLock: Locking IPs at the High-Level Language

Md Rafid Muttaki, Roshanak Mohammadivojdan, Mark Tehranipoor, Farimah Farahmandi
ECE Department, University of Florida
m.muttaki@ufl.edu, r.mohammadivojdan@gmail.com, {tehranipoor, farimah}@ece.ufl.edu

***Abstract*—The introduction of the horizontal business model for the semiconductor industry has introduced trust issues for the integrated circuit supply chain. The most common vulnerabilities related to intellectual properties can be caused by untrusted third-party vendors and malicious foundries. Various techniques have been proposed to lock the design at the gate-level or RTL before sending it to the untrusted foundry for fabrication. However, such techniques have been proven to be easily broken using SAT attacks and machine learning-based attacks. In this paper, we propose HLock, a framework for ensuring hardware protection in the form of locking at the high-level description of the design. Our approach includes a formal analysis of design specifications, assets, and critical operations to determine points in which locking keys are inserted. The locked design is then synthesized using high-level synthesis, which has become an integral part of modern IP design due to its advantages on lesser development and verification efforts. The locking at the higher abstraction with the combination of multiple syntheses shows that HLock delivers superior performance considering attack resiliency (i.e., SAT attack, removal attacks, machine learning-based attacks) and overheads compared to conventional locking techniques. Additionally, HLock provides a dynamic/automatic locking solution for any high-level abstraction design based on performance constraints, attack resiliency, power, and area overheads as well as locking key size, and it is well suited for large-scale designs.**

***Keywords*—High-Level Synthesis; Logic Locking; Obfuscation; IP Protection.**

I. Introduction

With the increasing demand for the latest technology embedded in smart electronic products and the internet of things (IoT) devices, the time allocated to design and verification of the intended product has shortened significantly. However, working on complex designs using smaller technology nodes requires more time dedicated to planning, design, and verification. Moving towards a higher level of abstraction (C/C++) enables designers to address the gaps between the design complexity and the design capabilities allowing them to implement and validate complex designs faster in response to aggressive time-to-market requirements.

High-level synthesis (HLS) automatically translates the high-level language (HLL) description of the design into the corresponding hardware description language (HDL) modules for implementation purposes. Enabling reduced design complexity at HLL such as C/C++, HLS has paved the way for different entities to lower the development cost of products, verification work, and time-to-market. For example, government agencies have relied on third-party intellectual property (IP) blocks for years to address the complexity and shorter time-to-market of modern SoCs, which raises concern as these IPs may have come with malicious functionalities. With the introduction of HLS, the risk of these vulnerabilities has been reduced [1]. HLS is also becoming popular in the industry; in a recent report, Nvidia mentioned that design complexity is growing by 1.4 times per generation, whereas the manpower remains the same [2]. Nvidia's video team was able to simplify the design description by five times and reduced 40% of the design and verification time by adopting HLS.

In the integrated circuit (IC) supply chain, untrusted vendors can pose security concerns by IP piracy [3], counterfeiting [4], inserting hardware Trojans [5], reverse engineering [6], or overproduction [7] for unauthorized usages. Over the last decade, various countermeasures such as IC camouflaging [8], split manufacturing [9], IC metering [10], and logic locking/obfuscation [11] have been proposed to thwart these attacks. Among these techniques, different forms of locking/obfuscation received significant attention by researchers due to their ease of adaptation by design houses.

Key-based locking has been proposed at various phases of the Application-Specific Integrated Circuit (ASIC) design flow to protect the design from unauthorized usages/changes effectively. A widely proposed technique is to introduce locking gates with inputs as secret keys into the synthesized netlist of the design [12], [13]. However, these types of combinational logic locking can be easily broken with the appearance of oracle based SAT and oracle less machine learning (ML)-based attacks [14], [15]. A few works proposed going one level higher in design abstraction to add auxiliary connections among the working units to lock the Register Transfer Level (RTL) of the design [16] and produce the locked netlist by synthesizing the locked RTL. Locking the design at RTL makes it more resilient toward the above-mentioned attacks, as the locked RTL gets transformed and optimized during gate-level synthesis. However, it is very difficult to obfuscate constant values and branches of an already optimized RTL. As a result, protection of IP is not guaranteed by locking at the RTL abstraction.

Locking/obfuscation at the HLL (e.g., C/SystemC/C++) can be defined as selecting the critical assets, operations, branches, and functions of the HLL design by a formal analysis and hiding the functionality by inserting secret keys at selected points. This type of obfuscation increases the protection level of hardware IPs since the designer has outright access to functionality, different operations, and the reliance between assets in the design. As a result, locking can be performed more effectively by meeting design specifications such as overhead (area and power), attack resiliency, and size of the key. Recently, [17], [18] proposed locking at the high-level abstraction (C/C++). Although they have shown comparisons based on overheads due to additional logic/functionality, they lack performance analysis based on attack resiliency. Moreover, these approaches suffer from high overheads.

In this paper, we propose HLock, a comprehensive framework for locking design at the C/C++ level and applying HLS on the locked design to generate automated RTL. HLock starts with a set of specifications including the design itself. The goal is to protect critical assets and functionalities of the design that adversaries may target to exploit them using reverse engineering. Then, we analyze the intermediate representation (IR) and control data flow graph (CDFG) to determine the possible locking points that provide us the highest protection and lock candidates at the high-level (C/C++). Next, we select the key size by taking into account all the selected locking points and the provided specification and lock the design with locking candidates. HLS is applied to the locked design involving steps from constructing CDFG, allocating functions to working modules, scheduling operations to different clock cycles, optimizing the functional units to producing the RTL.

This work was supported in part by Semiconductor Research Corporation (SRC).

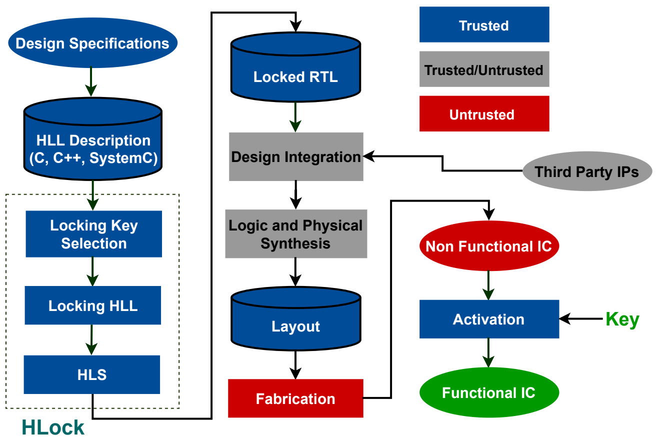


Fig. 1: The overview of HLock.

Moreover, the generated RTL goes through a second synthesis from RTL to gate-level netlist using the technology library. This approach facilitates uniform distribution of locking keys rather than inserting keys in an already optimized netlist. Also, optimum output corruptibility is ensured based on specification constraints to maximize resiliency against the SAT attack. Additionally, automated steps involving the HLS with optimizations during two levels of transformations ensure the design is resilient against reverse engineering and machine learning attacks. The overview of HLock is shown in Figure 1 involving both trusted and untrusted elements from design specification to the functional IC. The main contributions of this paper are as follows:

- We propose a comprehensive locking framework at HLL in terms of the desired attack resiliency (e.g., SAT attacks), overhead, and locking key size involving HLS and based on a formal analysis of the IR and CDFG of the design.
- HLock provides a wide array of locking nodes, candidates, and key sizes to meet the specifications.
- HLock addresses locking as an optimization problem and propose a technique that uses Integer Linear Programming (ILP) to provide the best possible solution within given constraints.
- Finally, HLock provides much better SAT and ML attack resiliency at lower overheads when compared with the results of previously proposed locking techniques.

The remainder of the paper is organized as follows. We discuss the threat model in Section II. Section III presents background on obfuscation and HLS. In Section IV, we present HLock for HLS-based logic locking, and Section V presents our experimental results. Finally, Section VI concludes the paper.

II. Threat Model

There are various attacks against locked/obfuscated hardware designs to recover their secret keys such as [12], [19]. However, a generalized attack model was presented based on the Boolean satisfiability problem that can break the conventional locking techniques in a short time [14]. In our threat model, the adversary can be (i) third-party vendors (such as design-for-test or design-for-debug insertion that could steal locked IP) or (ii) the untrusted foundry that can attack the design in the following ways:

- Adversaries can perform reverse engineering on specific modules to retrieve critical information.
- If the key is known, the foundry can overproduce ICs outside the IP owner's knowledge.
- Malicious insertion can be done by either third-party IP vendors or an untrusted foundry when they can retrieve the unlocked design. We assume that an oracle will be available for SAT (and its variants) attacks while ML-based attacks do not need an oracle.

III. Background

A. Related Work

Inserting XOR/XNOR key gates at random locations to lock the netlist was proposed in [20] to overcome the overproduction problem introduced by an untrusted foundry. In [21], a fault analysis-based locking approach was proposed to ensure maximum output corruptibility in case of a wrong key insertion. In [19], key gates are inserted in a way that can thwart key sensitization attacks. This technique works on creating a dependency between locking keys so that the effects of one key cannot be propagated to the output without knowing another key value. However, with the emergence of the SAT attack [14], all these locking techniques became invalid. SAT attack uses SAT solvers to determine distinguishing input patterns (DIPs) [14]. These DIPs are used in the functional IC to extract the correct behavior. Comparing this behavior to the locked IC, incorrect keys can be discarded. Depending on the strength of the DIPs, a single iteration can prune away multiple wrong keys. Finally, when all the incorrect keys are discarded, the algorithm terminates having the correct keys.

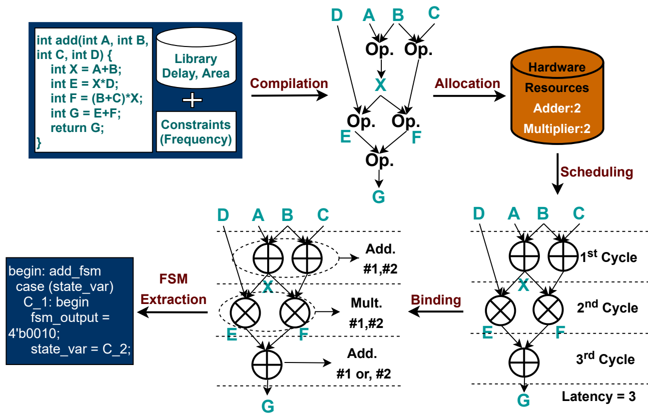
To counter the SAT attack, SARLock and Anti-SAT were proposed to limit the effectiveness of the SAT-based attack in [13], [22]. Although the techniques perform well against conventional SAT attacks, the unmodified logic cone makes them susceptible to removal attacks [23]. This type of attack identifies and removes the protection circuitry from the original circuit. For example, the signal probability skew (SPS) attack determines the skew for each logic gate and the basic construction principle for the Anti-SAT technique leads to structural traces in the netlist to identify and remove protection circuitry [23]. Some researchers proposed combining different locking techniques to strengthen the overall security of the design. However, it was shown in [24] that compound defense technique can be reduced to the one that is strongest of the two.

B. Background on HLS

High-level synthesis (HLS) enables transforming a high-level description (C/C++) of a design into hardware description language (HDL). The input to HLS is the untimed description of different functionalities. The HLS tool transforms the description into a synchronous process. While designing a complex System-on-Chip (SoC) design, using HLS greatly reduces design and verification efforts. Both government and industry are using HLS in their product development as it works best on applications containing complex algorithms and applications including machine learning, post-quantum cryptography, and hardware accelerators.

The high-level synthesis process is the combination of a series of steps comprising compilation, allocation, scheduling, binding, and finite state machine (FSM) extraction. A brief overview of these steps is as follows:

- *Compilation:* In this step of HLS, the abstract design description is translated into an intermediate representation considering data flow and control flow of the design. Data dependency throughout different operations and parallelism between blocks can be identified by CDFG of the design. [25].
- *Allocation:* In this step, different operations are allocated to available hardware resources such as logical units, computational blocks, connectivity units (e.g., buses), and



```
int add(int A, int B,
int C, int D) {
    int X = A+B;
    int E = X*D;
    int F = (B+C)*X;
    int G = E+F;
    return G;
}
```

```
begin: add_fsm
 case (state_var)
   C_1: begin
    fsm_output =
4'b0010;
    state_var = C_2;
```

Fig. 2: Steps in HLS: Compilation, Allocation, Scheduling, Binding, and FSM Extraction.

memory elements to meet specific design constraints. Hardware resource optimization is also possible based on design constraints in this step.

- *Scheduling:* Here, different operations are mapped to different clock cycles considering data dependencies and minimal design latency. After mapping the operations into functional units, they are scheduled into clock cycles. Data dependency determines the exact number of clock cycles needed. Parallel execution is possible if there are no dependencies.
- *Binding:* In the binding step, the process optimizes the total number of functional units needed to perform the functionality of the design. There are three types of bindings based on resource optimization which are: module binding, register binding, and interconnection binding. For the same resource at different clock cycles, module binding is used. Register binding works at mapping temporary values into different registers outside clock boundaries [26]. Lastly, interconnection binding optimizes connections among different resources.
- *FSM Extraction:* The final step in HLS is identifying the control signals and extracting the finite state machines (FSMs) of the design. With the control signals, FSM dictates the overall flow of the design. Once this step is done, the RTL model of the HLL is generated. Figure 2 shows the steps of HLS process.

IV. HLock

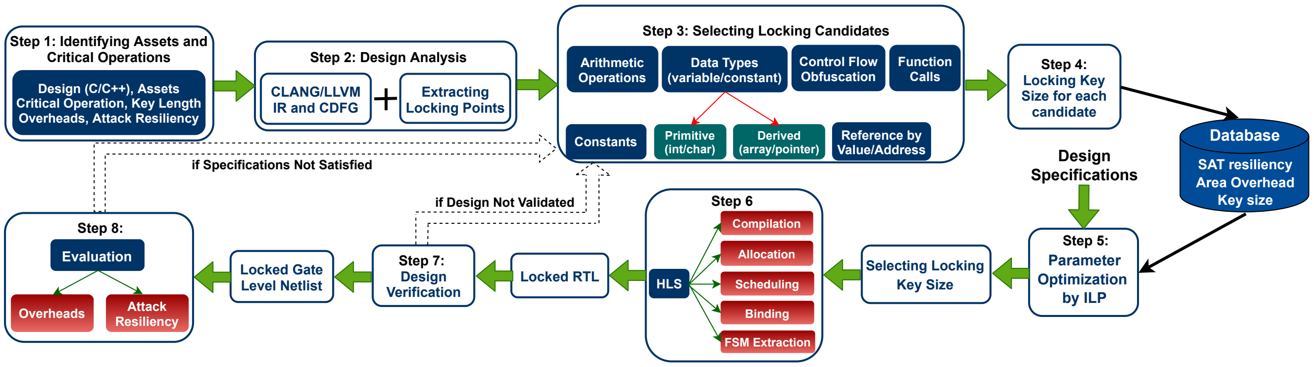
Our proposed locking framework, HLock, takes the HLL description of a design as well as the desired locking parameters and generates a locked RTL implementation of the design using HLS (see Figure 1). The locked RTL can be later synthesized to a locked netlist and layout. We consider three parameters for an optimized solution for locking design at the high level. These parameters are (i) attack resiliency, (ii) overheads (power and area), and (iii) the locking key size (K_{lock}) which are defined by the designer. To lock the design, HLock first identifies design security-critical assets (e.g., initial states, configuration bits, random numbers, encryption/decryption keys), critical operations, possible locking points, and candidates at the high level. HLock then automatically undergoes several steps to lock the design with the proper key size at the identified locking points (the locations in the C/C++ code are chosen from design analysis for locking) using different locking candidates. We describe each of the steps in HLock (see Figure 3) as follows:

1- Identifying Assets and Critical Operations: In the first step, having access to the C/C++ format of the design, its assets, functionality, and datapath information, the designer identifies which assets, security operations, or functionalities must be protected. Examples of security-critical elements are (i) message to be encrypted, (ii) encryption keys, (iii) global variables, (iv) constants, and (v) arithmetic operations.

2- Design Analysis: In this step, both intermediate representation (IR) and CDFG of the design are analyzed regarding the C/C++ abstraction to find out the information flow of the identified assets in step 1 and to understand which operations are related to the output functionality for achieving optimum corruptibility in case of the incorrect key insertion. This task becomes simple at the C/C++ level as the designers have a clear high-level overview of the assets, functionalities, and connectivities within the design. We do not strive to maximize corruptibility as it becomes easier for the SAT tool to prune more incorrect keys with fewer iterations in this case. Additionally, the candidate list for locking can be augmented by analyzing CDFG for critical design paths used for correct functionality.

3- Selecting Locking Candidates: Locking candidates will vary for different designs. For the HLock framework, we provide a list of locking candidates and the mechanism of using them for locking the design as described below:

- *Function calls:* Functions can be locked in multiple ways. If a function is a critical asset of the program and is called several times during execution, we can protect the function by locking its internal operations. For example, we can introduce significant changes to the output by locking the correct operation of shift row, or mix column of the AES encryption implementation, as they are called multiple times during compilation. On the other hand, if a program executes a function in a single instance, we can use either a fake function or argument for locking. For example, the key-expansion module of the AES encryption can be protected by adding a substitute function. Also, an erroneous version can be executed by calling it using an incorrect argument for the wrong key.
- *Control Flow Obfuscations:* In order to obfuscate the control flow of the design, one of the following methods can be used: i) The most common obfuscation method is to add conditional branches to protect the original branch. The designer can add a single key bit to lock each conditional branch in the program. ii) The loop execution can be tied to the key to obfuscate the actual functionality. Different key sizes may be required for each of those options. For example, a single bit key can control the overall execution of the loop. On the other hand, multiple key bits can be used to control multiple iterations. Although an increased key size provides more security to the design, the key size is also constrained by overhead. We consider this tradeoff at the parameter optimization step of HLock. iii) Array indexes can be manipulated in case of incorrect keys if the array stores secret keys. Each key bit can be responsible for one index of the array or only a single key bit can be used to corrupt a single element. iv) Fake states can be introduced when protecting recurring operations to change the original FSM of the design similar to FSM obfuscation at the RTL [27], and gate-level [17]. v) Skipping states for incorrect keys is also a type of FSM obfuscation. For adding/skipping states, the designer can use the locking key to force the program to run additional/inadequate iterations to perform the unnecessary/required operations. The key size here also can be between a single bit and the number of actual states.
- *Arithmetic operations:* Different arithmetic operations may exist in a design such as bit-wise operations, addition, subtraction, etc. A single bit key can be used to



Step 1: Identifying Assets and Critical Operations
Design (C/C++), Assets Critical Operation, Key Length Overheads, Attack Resiliency
Step 2: Design Analysis
CLANG/LLVM IR and CDFG
Extracting Locking Points
Step 3: Selecting Locking Candidates
Arithmetic Operations
Data Types (variable/constant)
Control Flow Obfuscation
Function Calls
Constants
Primitive (int/char)
Derived (array/pointer)
Reference by Value/Address
Step 4: Locking Key Size for each candidate
Database
SAT resiliency
Area Overhead
Key size
Design Specifications
Step 5: Parameter Optimization by ILP
Selecting Locking Key Size
Step 6
HLS
Compilation
Allocation
Scheduling
Binding
FSM Extraction
Locked RTL
Step 7: Design Verification
if Design Not Validated
Locked Gate Level Netlist
Step 8:
Evaluation
Overheads
Attack Resiliency
if Specifications Not Satisfied

Fig. 3: The intermediate steps of HLock for hardware locking using HLS.

introduce a fake arithmetic operation. On the other hand, multiple key bits can be used to corrupt the overall result of the operations.

- *Constant values:* Bit-wise operations such as XOR/XNOR can be added to hide constant values in the design such as initial round keys, plaintext, or states for a crypto module. Each bit of the constant can be locked using a single key bit.
- *Memory unit:* The read and write operation of the memory unit can be corrupted by changing their indices for incorrect keys. The key size will depend on the number of indices to be protected.

4- Locking Key Size for Each Candidate: In this step, the size of the locking key is selected based on locking points in the design and the required key bits of locking candidates. For example, a function $f(a, b)$ adds elements of two arrays. Now, a locking key bit can be added as an argument (i.e.,$f(a, b, k)$) so that for the incorrect key, a fake function is called performing subtraction. Also, the operation inside the function can be changed. For example, for a specific index, for the incorrect key bit, the function stores zero or a constant value. In this approach, two bits of the key can be used to obfuscate the correct behavior of the function. So, the key size will vary for each locking candidate and will depend on the structure of the candidate i.e., an assertion statement, loop iteration, branch condition, and so on. The overall locking key size will be the summation of the key bits protecting all the intended functionalities, critical operations, and assets.

5- Parameter Optimization: As mentioned in Step 4, the required locking key size will vary depending on locking candidates. For a given specification, to acquire the best possible result in terms of attack resiliency, overheads, and key size, we consider the problem set as an optimization problem. So, we use ILP in this case to obtain the best solution. In this step, we create a database containing the information on the three parameters, attack resiliency, the required key size, and overhead as mentioned earlier for different locking points and locking candidates in a design. Then, we use the given specification as constraints and the results of different locking case (a specific locking point with a specific locking candidate) instances as inputs to the optimization tool. The tool outputs a combination of locking cases that satisfies the constraints of the obfuscation. Each locking case also translates into a specific locking key size. As a result, the output combination of locking cases provides the total key size required (Key_{req}) to satisfy the specifications. Let γ_{pc} and α_{pc} represent the resiliency and area overhead parameters for each locking case L_{pc}, where p and c denote locking point and candidate, respectively. Then the set of constraint equations for the ILP tool are:

$$\gamma_{1c} \times L_{1c} + \gamma_{2c} \times L_{2c} + ... + \gamma_{mc} \times L_{mc} \geq Res_{spec} \quad (1)$$

$$\alpha_{1c} \times L_{1c} + \alpha_{2c} \times L_{2c} + ... + \alpha_{mc} \times L_{mc} \leq Ov_{spec} \quad (2)$$

Here, Res_{spec} and Ov_{spec} are the designer specifications for attack resiliency time and area overhead. With these constraints, the objective function will be the minimum requirement of locking keys. The output of the ILP tool will be a combination of locking cases residing within the constraints that can be translated into the required key size $size(Key_{req})$.

As the framework provides a broad range of solutions with dynamic locking key sizes, it is possible that for some resiliency specifications, the size of the locking key required (Key_{req}) is greater than the designer defined size ($size(K_{lock})$ $=$ Key_{spec}). In this scenario, to meet the designer's target key size, HLock begins locking with $size(K_{lock})$. The additional keys are created by expanding from the existing key size. This key expansion approach requires a form of substitution box (S-box) [28] embedded in the design while performing key-index changes, substitution from S-box, and the logical combination of previous and updated keys to creating a new set of keys. The inclusion of S-box and additional operations increases the overhead of the design. However, for larger designs, and with a fewer number of additional keys, the overall overheads will be lower.

In Algorithm 1, we use the three parameters mentioned previously for different locking cases as inputs and the performance specifications by the designer as constraints to the ILP optimization tool. The objective function is the minimum requirement of locking keys. Using the constraints and objective function, the ILP tool provides a combination of the locking cases from the available list (line 2). The required $size(Key_{req})$ is the summation of keys used in all the cases. However, when this key-size exceeds the designer specification, a key-expansion algorithm will be used to create additional keys. (lines 8-12).

6- HLS: In the HLS step, the locked HLL design goes through steps of compilation, allocation, scheduling, binding, and FSM extraction as explained before. The design undergoes multiple optimizations during these steps. The output of this step is the locked RTL.

7- Design Verification: In this step, the locked RTL using the selected locking key size (New_{Key}) is verified. This step is necessary to ensure that the correct functionality cannot be achieved for wrong keys, and locking candidates do not lead to incorrect functionality in the case of correct keys. This validation can be performed using a simulation-based approach by running testbench for multiple key inputs and evaluating the outputs. Also, using the formal method by writing properties to check equivalence between designs with correct and incorrect keys.

8- Evaluation: Performing design analysis to insert locking keys effectively at critical locations of the HLL along with confusion and irregularity added by the HLS process makes

Algorithm 1: Determining Overall Key Size

```
Input: Attack Resiliency, Overheads, Locking Key
       Size for each locking case
Output: $size(New_{Key})$
1  $Constraints \leftarrow Resiliency,\ Overhead$
2  $ILP \leftarrow Inputs\ + Constraints\ +\ Obj.\ Function$
3  $L_{PC}\{L_{P1}, L_{P2}, ......, L_{Px}\},$
       $\{L_{C1}, L_{C2}, ......, L_{Cy}\} \leftarrow ILP$ // set of
     selected locking points and
     corresponding candidates
4  $Key_{PC} \leftarrow key\ size\ corresponding\ to\ each\ L_{PC}$
5  for $i \leftarrow 1\ to\ x$ do
6      $Key_{req} \leftarrow \sum_{i=1}^{x} Key_{ij}$        // j denotes
       locking candidate corresponding to
       locking points
7  end
8  if $size(Key_{req}) > Key_{spec}$ then
9      $New_{Key} \leftarrow use\ Key\ Expansion$
10 else
11     $New_{Key} \leftarrow Key_{req}$
12 end
```

it extremely difficult to reverse engineer the locked RTL. Additionally, the locked RTL is synthesized once more to generate the locked gate-level netlist. Therefore, the locked design goes through another round of optimization. This process hides the locking key gates effectively in the design. Also, it makes removal and ML-based attacks very difficult to trace the locking key gates because of the multiple transformations and optimizations of the locked design. For limiting the SAT attack, while inserting keys, we ensure output corruptibility is limited so that the SAT tool requires more iterations for extracting the locking keys. In this step, we evaluate the area overhead as well as the resiliency of the locked gate-level design against SAT-based and ML attacks. It should be noted that, for database creation and population, we estimate the effect of the selection of each candidate on locking parameters individually and feed it to the ILP tool. Therefore, the actual effect may vary due to synthesis optimizations when we consider all the locking points together. To address this issue, HLock can go back to the candidate selection step and add candidates if needed. As a result, it becomes difficult for removal attacks to identify locked functions/components.

V. Results and Analysis

To evaluate HLock, we have applied it to three C-level benchmarks: Advanced Encryption Standard (AES), Mergesort, and Needleman-Wunsch (NeedWun) algorithms. We compared the results in terms of SAT resiliency and overheads with conventional locking techniques. To keep the locking scenarios realistic based on the size of these benchmarks, we have set the area overhead less than 20% (15% for AES, 17% for Mergesort, and 18% NeedWun) for all the benchmarks. These benchmark designs are relatively smaller in size which demonstrates the efficacy of HLock as larger benchmarks are expected to offer increased locking key sizes for greater attack resiliency with lower overhead. Table I shows comparisons based on power overhead and SAT resiliency while keeping the area overhead constant across different techniques.

As is apparent from the table, for similar area overheads across different benchmarks, HLock shows superior SAT resiliency compared to the conventional techniques with very low power overhead. For MergeSort, the SAT resiliency is 10-1,000 times higher with power overhead at least 50% less. On the other hand for AES, SAT resiliency ranges

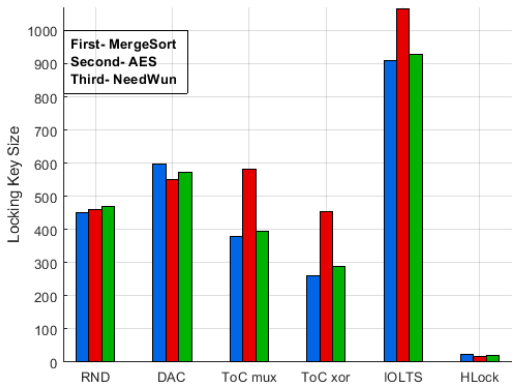


Fig. 4: Comparison based on locking key sizes for different locking techniques from Table I.

from 15 to more than 10,000 times with power overhead at least 35% less. Results in similar magnitudes can be found from NeedWun as well. It should be noted that designers can achieve even better resiliency using one of the many available key combinations using HLock. However, we have applied area-overhead constraints to HLock hence limiting SAT resiliency. HLock enables designers to find the best solution involving maximum attack resiliency if overheads are not considered. For all the other conditions, the attack resiliency will be limited by overhead constraints.

In Figure 4, different locking techniques have been compared based on the locking key sizes to achieve attack resiliency shown in Table 1. It can be seen that with HLock, even a fraction of keys compared to other techniques can make the design much more resilient against the SAT attack. The reason behind this is twofold: Firstly, performing design analysis and having access to high-level design abstraction enables embedding locking keys effectively to secure intended operations, assets, and functionality. Secondly, adding keys at the HLL enables multiple transformations and optimizations (i.e., HLS and RTL synthesis) of the locked design. It helps to blend these keys with the states of the design creating logic obfuscation as well as state obfuscation. This effect can not be replicated by inserting key gates at the gate-level representation by other techniques. From Figure 4, it is also evident that if a small number of locking keys are added in the specification, HLock can provide much higher resiliency solutions for any design. Also, if larger benchmark designs are considered, higher attack resiliency solutions will be provided by HLock with a lower overhead percentage.

In Table II, we have compared the resiliency among different locking techniques against machine-learning-based attack [30]. This attack first trains with a locked netlist with a specific locking technique. In the training phase, it takes each locking key bit and assumes both versions, 0 and 1, and acquires features such as the reduction in area and power for specific key values. Finally, it assigns weights to make a decision $0/1$ for each key bit. As can be observed from the results, for conventional locking techniques, the attack generates accuracy greater than 95%. However, for our proposed technique, the attack accuracy is much lower. It can be explained such that 50% accuracy is the worst-case scenario for these types of attacks. This is because, if the accuracy of the tool is near 0%, it means that the weighting of parameters needs to be changed to improve the accuracy. In these cases, the accuracy can be improved to near 100%. However, if the accuracy lies around 50%, then mere weight redistribution cannot improve the performance of the attack.

This lower accuracy against HLock can be explained as

TABLE I: Comparisons of SAT resiliency and power overhead among different techniques keeping area overhead constant.

Locking Type	Mergesort		AES		NeedWun	
	Power Overhead	SAT Resiliency	Power Overhead	SAT Resiliency	Power Overhead	SAT Resiliency
inserts XOR and XNOR gates at randomly chosen locations (RND) [20]	69.09%	10.75s	35.59%	3.46s	56.47%	8.74s
inserts XOR/XNOR gates carefully to avoid fault-analysis attack (DAC) [19]	103.21%	190.20s	155%	245.50s	115.70%	156.40s
Maximizes HD between correct and incorrect outputs by MUX (ToC mux) [21]	42.10%	1.34s	67.21%	2.73s	53.39%	3.27s
Maximizes HD between correct and incorrect outputs by XOR (ToC xor) [21]	82.30%	19.34s	145.30%	26.59s	103.84%	16.23s
Minimizes low controllability locations by inserting AND, OR (IOLTS) [29]	14.67%	2.90s	13.54%	0.35s	15.74%	1.60s
HLock (Proposed Framework)	**7.84%**	**1915s**	**8.08%**	**4579s**	**8.53%**	**1883s**

TABLE II: Comparisons of ML attack accuracy among different locking techniques.

Benchmark Designs	Accuracy (%) for Locking Types				
	TOCm'13 [21]	IOLTS'14 [29]	SARLock [22]	Mux2 [30]	HLock
MergeSort	96.66	100	100	92.27	68.18
AES	97.22	100	100	93.82	62.50
NeedWun	98.86	99.32	100	92.74	65.87
Avg.	*97.58*	*99.77*	*100*	*92.95*	***65.51***

most of the ML-based attacks [15], [30] focus on tracing certain key gates such as XOR/XNOR or MUX [20], [21] used for locking. So, the locking approaches using specific key gates become very susceptible to these attacks. On the contrary, in HLock, because of the intelligent insertion of locking keys into the intended locations at HLL and transformations/optimizations from C/C++ to RTL and from RTL to the gate-level netlist, key gates become immensely randomized. As a result, tracing locking key gates does not have a similar impact, and so the ML-based attacks do not provide higher accuracy.

The following points are the important takeaways about HLock:

- Performing locking at the initial stage facilitates superior results in terms of overheads and attack resiliency. Hence, the best possible solution can be ensured based on the obfuscation requirements for each design using the proposed approach.
- Instead of a specific size for the locking key, for different sets of high-level specifications, HLock provides dynamic locking key solutions to meet the specifications providing flexibility to the designers.
- HLock focuses on the overall performance parameters of the design and the trade-offs between those parameters i.e., attack resiliency vs. overhead. As a result, comprehensive locking solutions are achieved by limiting the overhead constraints.

VI. Conclusion

In this work, we have proposed a logic locking framework at the higher abstraction level based on high-level synthesis. Our approach focuses on a specific set of predefined locking parameters/goals and provides dynamic solutions covering various combinations of the requirements set by the designers. The performance shows that this approach provides greater SAT resiliency with only a fraction of the locking keys required with minimal overheads compared to the conventional techniques for different benchmark designs. Also, due to the intelligent locking at the higher abstraction combined with the transformations at different stages, this technique shows higher resiliency against ML-based attacks. Additionally, the framework presents the opportunity of locking designs at the higher abstraction providing the utmost control and flexibility to the designers. In future work, plan to provide the performance of this framework for hybrid locking techniques and also coding guidelines at the C/C++ level for the designers who seek to secure their designs using this framework.

References

[1] "A high level synthesis tool for fpga design from software binaries," *SBIR Award, Department of Defense*, 2007.

[2] J. L. Frans Sijstermans, "Working smarter, not harder: Nvidia closes design complexity gap with high-level synthesis," https://go.mentor.com/4N9cP.

[3] M. Rostami *et al.*, "A primer on hardware security: Models, methods, and metrics," *Proceedings of the IEEE*, pp. 1283–1295, 2014.

[4] U. Guin *et al.*, "Counterfeit integrated circuits: A rising threat in the global semiconductor supply chain," *Proceedings of the IEEE*, pp. 1207–1228, 2014.

[5] M. Tehranipoor *et al.*, "A survey of hardware trojan taxonomy and detection," *IEEE design & test of computers*, pp. 10–25, 2010.

[6] R. Torrance and D. James, "The state-of-the-art in IC reverse engineering," in *CHES*, C. Clavier and K. Gaj, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 363–381.

[7] U. Guin *et al.*, "A novel design-for-security (DFS) architecture to prevent unauthorized IC overproduction," in *VTS*, 2017, pp. 1–6.

[8] J. Rajendran *et al.*, "Security analysis of integrated circuit camouflaging," in *CCS*, 2013, pp. 709–720.

[9] R. W. Jarvis and M. G. Mcintyre, "Split manufacturing method for advanced semiconductor circuits," 2007, uS Patent 7,195,931.

[10] Y. Alkabani and F. Koushanfar, "Active hardware metering for intellectual property protection and security." in *USENIX security symposium*, 2007, pp. 291–306.

[11] Y. Lee and N. A. Touba, "Improving logic obfuscation via logic cone analysis," in *LATS*, 2015, pp. 1–6.

[12] S. M. Plaza and I. L. Markov, "Solving the third-shift problem in IC piracy with test-aware logic locking," *IEEE TCAD*, pp. 961–971, 2015.

[13] Y. Xie and A. Srivastava, "Anti-sat: Mitigating sat attack on logic locking," *IEEE TCAD*, pp. 199–207, 2018.

[14] P. Subramanyan *et al.*, "Evaluating the security of logic encryption algorithms," in *HOST*, 2015, pp. 137–143.

[15] P. Chakraborty *et al.*, "Sail: Machine learning guided structural analysis attack on hardware obfuscation," in *AsianHOST*, 2018, pp. 56–61.

[16] R. S. Chakraborty and S. S. Bhunia, "Rtl hardware ip protection using key-based control and data flow obfuscation," in *VLSI Design*, 2010, pp. 405–410.

[17] C. Pilato *et al.*, "Tao: techniques for algorithm-level obfuscation during high-level synthesis," in *DAC*, 2018, pp. 1–6.

[18] H. Badier *et al.*, "Transient key-based obfuscation for hls in an untrusted cloud environment," in *DATE*, 2019, pp. 1118–1123.

[19] J. Rajendran *et al.*, "Security analysis of logic obfuscation," in *DAC*, 2012, pp. 83–89.

[20] J. A. Roy *et al.*, "Epic: Ending piracy of integrated circuits," in *DATE*, 2008, pp. 1069–1074.

[21] J. Rajendran *et al.*, "Fault analysis-based logic encryption," *IEEE TC*, pp. 410–424, 2013.

[22] M. Yasin *et al.*, "Sarlock: Sat attack resistant logic locking," in *HOST*, 2016, pp. 236–241.

[23] M. M. Yasin *et al.*, "Removal attacks on logic locking and camouflaging techniques," *IEEE Transactions on Emerging Topics in Computing*, 2017.

[24] K. Shamsi *et al.*, "Appsat: Approximately deobfuscating integrated circuits," in *HOST*, 2017, pp. 95–100.

[25] P. Coussy *et al.*, "An introduction to high-level synthesis," *IEEE Design & Test of Computers*, pp. 8–17, 2009.

[26] L. Stok, "Data path synthesis," *Integration*, pp. 1–71, 1994.

[27] Y. Lao and K. K. Parhi, "Obfuscating dsp circuits via high-level transformations," *IEEE TVLSI*, pp. 819–830, 2014.

[28] Wikipedia contributors, "Rijndael s-box — Wikipedia, the free encyclopedia," 2020, [Online; accessed 22-September-2020]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Rijndael_S-box&oldid=956141554

[29] S. Dupuis *et al.*, "A novel hardware logic encryption technique for thwarting illegal overproduction and hardware Trojans," in *IOLTS*, 2014, pp. 49–54.

[30] A. Alaql *et al.*, "Sweep to the secret: A constant propagation attack on logic locking," in *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 2019, pp. 1–6.