Designing ML-Resilient Locking at Register-Transfer Level

Dominik Sisejkovic[1], Luca Collini[2*], Benjamin Tan[3], Christian Pilato[4],
Ramesh Karri[2], and Rainer Leupers[1]
[1] RWTH Aachen University, Germany, [2] New York University, USA, [3] University of Calgary, Canada, [4] Politecnico di Milano, Italy
{sisejkovic, leupers}@ice.rwth-aachen.de, {lc4976, rkarri}@nyu.edu, benjamin.tan1@ucalgary.ca, christian.pilato@polimi.it

ABSTRACT

Various logic-locking schemes have been proposed to protect hardware from intellectual property piracy and malicious design modifications. Since traditional locking techniques are applied on the gate-level netlist after logic synthesis, they have no semantic knowledge of the design function. Data-driven, machine-learning (ML) attacks can uncover the design flaws within gate-level locking. Recent proposals on register-transfer level (RTL) locking have access to semantic hardware information. We investigate the resilience of ASSURE, a state-of-the-art RTL locking method, against ML attacks. We used the lessons learned to derive two ML-resilient RTL locking schemes built to reinforce ASSURE locking. We developed ML-driven security metrics to evaluate the schemes against an RTL adaptation of the state-of-the-art, ML-based SnapShot attack.

CCS CONCEPTS

• **Security and privacy → Security in hardware**;

KEYWORDS

logic locking, RTL, IP protection, machine learning, deobfuscation

ACM Reference Format:
Dominik Sisejkovic[1], Luca Collini[2*], Benjamin Tan[3], Christian Pilato[4],, Ramesh Karri[2], and Rainer Leupers[1]. 2022. Designing ML-Resilient Locking at Register-Transfer Level. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC) (DAC '22), July 10–14, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3489517.3530541

1 INTRODUCTION

Integrated circuits (ICs) are a critical layer for security in modern electronic systems. However, there are security concerns due to third parties in the supply chain. As external design houses and foundries have full access to the IC intellectual property (IP) during production, attackers could reverse engineer the IP for malicious purposes, such as IP theft and hardware Trojan insertion [19]. Design-for-trust methodologies aim to counteract such threats. Logic locking has been recognized as a premier technique to safeguard ICs throughout the supply chain [10]. Logic locking builds on the concept of design obfuscation [2, 9, 15–17], where designers insert key-driven logic to functionally and structurally alter ICs, thus concealing functional intent. Only the correct activation key unlocks the intended functionality of the IC.

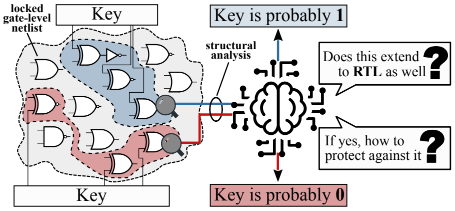

locked gate-level netlist
Key
Key
structural analysis
Key is probably **1**
Key is probably **0**
Does this extend to **RTL** as well?
If yes, how to protect against it?

Figure 1: Machine learning vs. logic locking: impact on RTL?

Recently, machine learning (ML) techniques have challenged the security of gate-level locking [8]. ML-driven attacks exploit the predictable relation between the key value and the functional or structural aspects of locking. This has led to potent attacks that can either predict the correct key value or remove the locking circuitry from the netlist [6, 11, 12, 18]. While ML-driven attacks often lack output certainty, their applicability adds another requirement for logic locking success—*prevention of key-related residue within the locking mechanics.* As long as the structural change is related to key values, it is possible to use ML to guess the keys.

Traditional gate-level locking schemes are limited to local changes and do not use the semantic information of the circuit, as logic synthesis and optimization disperse the semantics to a low granularity. Therefore, gate-level locking schemes operate "blindly" on the design without considering its functional traits. In response, RTL locking has emerged as a way to overcome this issue [1, 5]. At the RTL, locking can use the full spectrum of semantic information, including operations, constants, and control flow constructs. Hence, RTL locking is a promising basis to build ML-resilient locking. However, compared to gate-level locking, ML attacks on RTL locking remain unexplored as shown in Fig. 1.

Contributions: This study explores ML resilience of RTL locking focusing on *operation obfuscation*, where we:

- Introduce theoretical concepts to evaluate ML-resilience of RTL locking.
- Expose security faults in ASSURE RTL locking [5].
- Define ML-resilience security metrics for RTL locking.
- Introduce two ML-resilient locking algorithms: (1) ERA: **E**xact ML-**R**esilient **A**lgorithm and (2) HRA: **H**euristic ML-**R**esilient **A**lgorithm.
- Evaluate the locking algorithms against an RTL adaptation of the ML-based SnapShot attack.

To the best of our knowledge, the presented concepts and locking procedure are the first to address the challenges of ML resilience on RTL. **The implementation of this study will be made available to the community once published.**

[*] This work was done while L. Collini was with Politecnico di Milano, Italy.

DAC '22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530541

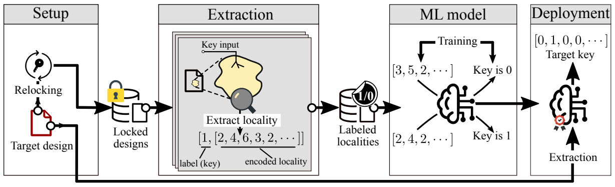


Figure 2: ML-driven SnapShot attack flow.

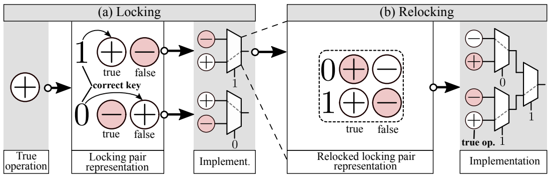


Figure 3: ASSURE operation locking and representation.

2 BACKGROUND

2.1 Threat Model

The threat model includes the following assumptions. (1) The attacker has only access to the locked design in the form of a locked gate-level netlist. As an activated chip is not available, this attack model is often referred to as *oracle-less* (OL) [6]. Starting from the provided design level, the attacker can perform reverse engineering to recover the RTL design. To simulate the *best-case scenario for the attacker*, we assume the attacker can retrieve an *exact copy* of the initial, locked RTL design. (2) The attacker is aware of the algorithmic details of the applied locking scheme. (3) The location of the key pins is known (*distinct ambiguity* [10]). In the rest of this study, we refer to the locked RTL design under attack as the *target*.

2.2 ML-Driven Structural Attacks

In the OL model, an attacker has only access to the target design without I/O patterns. An ML-based attack has to exploit *structural* key-related patterns to produce a (correct) key prediction. Thus, we selected the OL SnapShot attack [6] as a basis for the evaluation. SnapShot was initially designed to attack locked gate-level netlists, following four major steps (Fig. 2). First, the attack prepares a set of locked samples by *relocking* (self-referencing) the target benchmarks with new keys. Second, a training set is assembled by extracting a netlist sub-graph for each single-bit key input from all data samples. The extracted sub-graphs are transformed into a vector of numbers, where each entry encodes a single gate from the derived sub-graphs. These vectors are referred to as *localities*. In essence, *a locality represents a key-affected portion of the netlist*. Next, the attack trains a dedicated ML model to associate localities with their respective key values. Finally, the trained ML model is deployed to predict the key of the target design. Since SnapShot has previously only been applied on gate level, we adjust the extraction and ML model of SnapShot to support RTL locking in this work.

Besides SnapShot, the most prominent OL, ML-based attacks on gate-level locking are OMLA [12], GNNUnlock [11], and SAIL [18]. OMLA and GNNUnlock use graph neural networks, thus relying on a graph representation of the input design that is natural to gate-level netlists. SAIL exploits the deterministic and local changes of gate-level locking by learning to reverse the transformations induced by logic synthesis. Since we operate on RTL and assume a perfect reconstruction of the locked RTL, SAIL is not considered.

2.3 The Concept of RTL Locking

We focus on the locking techniques proposed in ASSURE [5]—one of the latest RTL locking policies. ASSURE offers three locking techniques: constant, branch, and operation obfuscation. Constant obfuscation extracts constants into the activation key. For example, `a = 4’b1101` is locked as `a =` K, where K is the 4-bit constant stored as the key. Branch locking works by XOR-ing the condition of the branch with a key bit, thereby inverting the condition if the value is 1. For example, the condition `a > b` is locked as `(a <= b)`$\wedge K$. Operation locking inserts a key-controlled multiplexer to choose between a real and dummy operation. For example, `a = b + c` can be locked either as `a =` K `? (b + c) : (b - c)` or `a =` K `? (b - c) : (b + c)`, depending on the value of K. In terms of security, constant obfuscation does not offer any apparent attack vectors, as the secret is fully omitted from the attacker. Branch obfuscation only affects the existing control flow based on the key, without inserting additional logic. Operation obfuscation manipulates the design by inserting additional logic *depending on the existing one*. This dependence offers the potential for an attack. Therefore, *we focus on the security of operation obfuscation.*

Operation obfuscation: The security of this locking concept lies in the assumption that the attacker cannot guess which operation of the observed pair is the correct one. The paired real and dummy operations are called *locking pairs*. Locking pairs are defined as (T, T'), where T and T' are the real and dummy operations, respectively. On RTL, a locking pair is implemented in the form of a *ternary operator*. For instance, as depicted in Fig. 3a, the real operation + can be locked in the form of $(+, -)$ for the correct key value 1, or in the form of $(-, +)$ for the correct key value 0. Hence, an addition is always locked in pair with a subtraction, and vice versa. Note that all operations have predefined locking pairs [5]. If a locked pair is relocked, both T and T' are locked separately. As shown in Fig. 3b, relocking results in a tree of multiplexers, i.e., nested ternary operators. The compact notation of locked pairs from Fig. 3 is used in the rest of this study for visualization purposes.

3 LEARNING RESILIENCE FOR RTL LOCKING

Learning attacks make predictions about the key by studying the locked design. A scheme that is secure against learning attacks is considered *learning-resilient* [7]. As discussed in [7], netlists that exhibit regular, repetitive structures *maximize* the exposure of a locking scheme’s mechanics, making it is easy to identify potential leakage points. This is because key-related structural changes are more likely to be identified within repetitive constructs. To evaluate RTL locking for structural leakage, let us consider its workings on a structurally regular design that only contains connected + operations. The following challenge arises: *how do we lock it without suggesting anything about the correctness of the key?* Let us consider the two methods of operation selection in ASSURE: *serial* and *random*. Furthermore, we need to take two data sets into account: test and training. The test set consists of locking samples for which the key is *unknown* and represents the design under attack. The training set comprises locking samples that are added in additional relocking rounds of the target with *known* keys. This process is also known as self-referencing [6, 18]. The attacker uses the training set

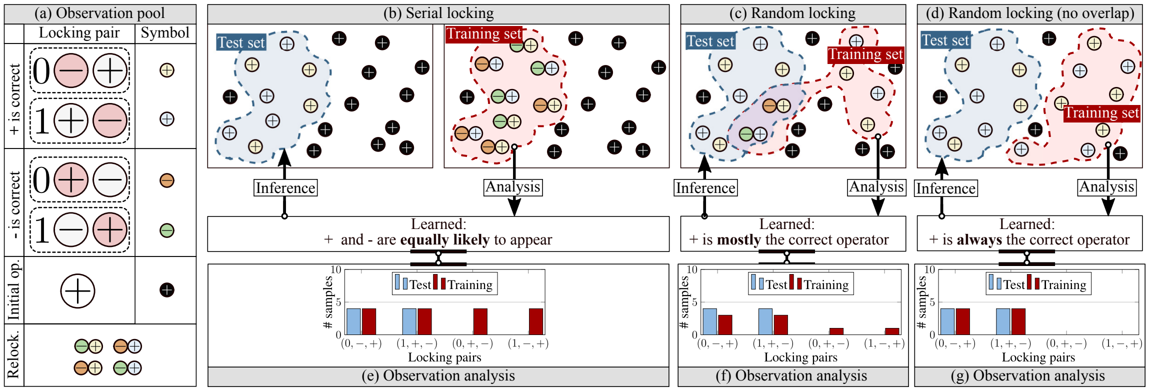


Figure 4: Impact of operation selection on learning resilience in RTL locking.

to collect observations about *the relation between the locking pairs and the key*. The ensuing discussion follows visualizations in Fig. 4, where we consider different locking scenarios using locking pairs and symbols from Fig. 4a, and the + operation network.

Serial selection is the standard selection in ASSURE. As shown in Fig. 4b, the initial locking (test set) selects + operations for locking to create locking pairs in the form of $(+, -)$ and $(-, +)$ (encoded with a single symbol for simplicity). This "serial" selection always selects the operations in a serial manner w.r.t. the design topology. Due to the serial selection, the subsequent locking rounds (training set) select the *same* operations as the test set for relocking; already-locked + operations are extended with additional locking pairs. In the example, the left operation is selected for key value 1 and the right for key value 0, according to the rule of ternary expressions. As portrayed in Fig. 4e, both the + and − operations *are equally related to the key value 0 and 1*, resulting in confusing observations. This suggests that ASSURE is, in principle, learning-resilient. However, this case arises only due to the deterministic order of selecting operations—it can easily be broken by either using a longer training key to ensure locking untouched operations during training or by randomizing the order of selection. Hence, the standard ASSURE procedure *is not secure* w.r.t. data-driven attacks.

Random selection is depicted in Fig. 4c. Here, the samples from the test and training set are likely to overlap only to some extent. Hence, the observations of the training set are contaminated by some *contradictory* observations. By analyzing the training set (Fig. 4f), one can learn that the + operation *is more likely to be the correct one.* The random selection might result in a favorable outcome for the attacker when training and test samples do not overlap (Fig. 4d). In this scenario, all observations from the training set (Fig. 4g) suggest that the + operation *is always the correct one.* This knowledge can be used to infer a correct key.

3.1 Observations

(1) Learning resilience on RTL can be achieved if the likelihood of any operation in a locking pair is equally related to key value 0 and 1. (2) Operation selection impacts learning resilience. (3) The initial distribution of operation types determines if learning-resilience is achievable. Evidently, the effectiveness of learning-resilient locking *should not depend on circuit features.* Even if real-world designs are not represented by the + network, focusing on this biased case ensures that the scheme offers security even in general cases.

Based on the above observations, we can conclude that *learning resilience on RTL is achievable if the occurrence frequency of every operation within a locking pair is equal for all operations in the pairings.* This is the case if the design has the same number of + and − operations after locking. In that case, any selection procedure for training results in an equal number of contradictory observations. In the next section, we introduce two locking algorithms that use this rule for learning resilience on RTL.

3.2 ASSURE Leakage Points

We analyzed the serial selection of ASSURE [5], and the *current pairing of operations is leaky* as operations are incorrectly paired. For example, ASSURE assumes these pairs: $(*, +)$, $(+, -)$, and $(-, +)$. Here, * is paired with a +, but + is also paired with -. Hence, if the locked pair $(*, +)$ is encountered, the attacker can infer * as the correct operation, as $(+, *)$ does not exist. Similarly, leakage exists for modulo, xor, power, and division. Thus, *currently ASSURE can be broken by analyzing operation pairs.* Hence, every operation must exist as a real and dummy operation with the same pair, e.g., $(*, /)$ and $(/, *)$. This fix applies to all evaluations in this study.

4 ML-RESILIENT RTL LOCKING

Based on the discussion, we introduce the following definition:

Definition 1. *An RTL design is* ***learning-resilient*** *w.r.t. operation locking if the number of operations of type T is equal to the number of operations of type T' for each locking pair for which at least one operation of type T or T' is locked.*

If neither T nor T' are involved in locking, the locked design is learning-resilient even if the number of T and T' operations is not balanced. The reason is that, during training, locking "untouched" T and T' operations does not provide feedback for the target samples. Henceforth, *secure* refers to security in the context of Def. 1. Next, we introduce two ML-resilient locking algorithms built on top of ASSURE: ERA: **E**xact ML-**R**esilient **A**lgorithm and **H**euristic ML-**R**esilient **A**lgorithm. ERA guarantees security w.r.t. Def. 1, but requires a large key budget. HRA trades-off key length with security, yielding less secure solutions if the key budget is limited.

Operation distribution: The first step in ERA and HRA is to analyze operation distribution in the input RTL. We store this information in an *operation distribution table* (*ODT*). For each T, the table

```
Algorithm 1: Lock
    Input: Locking type $T$, operation distribution table $ODT$, RTL design D, and
           pair mode P
    Output: Number of used bits
 1  $n \leftarrow 0$                                          // Initialize used bits var
 2  $T' \leftarrow$ GetPairType($T$)
 3  $o_i \leftarrow$ RndSelect($D.ops[T]$)                    // Select a T-type op.
 4  $o_j \leftarrow$ RndSelect($D.ops[T']$)
 5  if $ODT[T] > 0$ and $!P$ then
 6      AddPair($D, o_i, T'$)                                 // Add T' node to $o_i$
 7      $ODT[T] \leftarrow ODT[T] - 1$
 8      $ODT[T'] \leftarrow ODT[T'] + 1$
 9      $n \leftarrow n + 1$
10  else if $ODT[T] < 0$ and $!P$ then
11      AddPair($D, o_j, T$)                                  // Add T node to $o_j$
12      $ODT[T] \leftarrow ODT[T] + 1$
13      $ODT[T'] \leftarrow ODT[T'] - 1$
14      $n \leftarrow n + 1$
15  else
16      AddPair($D, o_i, T'$)                                 // Add T' node to $o_i$
17      AddPair($D, o_j, T$)                                  // Add T node to $o_j$
18      $n \leftarrow n + 2$
19  end
20  return $n$
```

```
Algorithm 2: $d_e$: Modified Euclidean Distance
    Input: Current vector $v_j$ and optimal vector $v_o$
    Output: Distance
    $s \leftarrow 0$                                          // Initialize sum var
    for $i \leftarrow 0; i < |v_o|; i++$ do
        /* Check if value should be considered                */
        if $v_o \neq$ 'x' then
            $s \leftarrow s + (v_o[i] - v_j[i])^2$
    end
    return $\sqrt{s}$
```

```
Algorithm 3: ERA: Exact ML-Resilient Algorithm
    Input: Key budget $k_b$ and RTL design $D$
    Output: Locked RTL design
 1  LoadODT($D$)                                              // Populate ODT
 2  $n \leftarrow 0$                                          // Initialize used bits var
 3  $\Theta \leftarrow \{(T_1, T'_1), \dots, (T_n, T'_n)\}$   // Valid locking pairs
 4  while $n < k_b$ do
 5      $\vartheta \leftarrow$ RndSelect($\Theta$)            // Select a pair
 6      $T \leftarrow$ RndSelect($\vartheta$)                 // Select a type
 7      while $|ODT[T]| > 0$ do
 8          $s \leftarrow$ Lock($T, ODT, D, False$)           // Apply lock (Algorithm 1)
 9          $n \leftarrow n + s$
10      end
11  end
12  return $D$
```

stores a number representing the difference between the distribution of T-type and the locking-pair T'-type operations. Assuming the pair $(+, -)$, a design with 7 "+" and 5 "−" has the following ODT entries: $ODT[+] = +2$ and $ODT[-] = -2$. A positive (negative) ODT value indicates that the operation type has more (less) operations than its locking-pair type. The ODT entries can inform a *secure* design by balancing the number of T and T' operations.

The locking step: Algorithm 1 outlines Lock, the common locking step for HRA and ERA. For a selected type T, the RTL D is locked following three cases. If $ODT[T]$ is positive (lines 6-9), pair a new T'-type operation with an existing T-type to reduce the excess of T. If $ODT[T]$ is negative (lines 11-14), pair a new T-type operation with an existing T'-type to reduce the deficiency of T. Otherwise (lines 16-18), pair new T- and T'-type operations with existing operations. This is used by *specific operation-selection algorithms* to derive HRA and ERA. Before describing the locking algorithms, we introduce a security metric for resilience w.r.t. Def. 1.

4.1 Security Metric for Learning Resilience

ODT entries can be used as a vehicle to *measure security* in the context of Def. 1. To design a metric that indicates how "far" a locked design is from the optimal distribution, let us consider the following notation. The content of ODT in iteration j of a selected locking algorithm can be represented as the vector $\boldsymbol{v_j} = [x_0, \dots, x_{l-1}]$, where l is the number of available locking pairs, and $x_i = |ODT[T]|$. Note that $|ODT[T]| \equiv |ODT[T']|$. A secure solution is reached if all entries of $ODT = 0$. Hence, the optimal distribution can be defined as $\boldsymbol{v_o} = [y_0, \dots, y_{l-1}]$, where $y_i = 0$ for $i \in [0, l-1]$. Using this notation, we can define the learning-resilience security metric as:

$$M_{sec} = 100 \cdot \left(1 - \frac{d_e(\boldsymbol{v_j}, \boldsymbol{v_o})}{d_e(\boldsymbol{v_i}, \boldsymbol{v_o})}\right), \tag{1}$$

where d_e is a modified version of the Euclidean distance, $\boldsymbol{v_i}$ the initial distribution vector of the target design, $\boldsymbol{v_o}$ the optimal distribution vector, and $\boldsymbol{v_j}$ the distribution vector after the j-th locking iteration. Note that $M_{sec} \in [0, 100]$, where the highest value indicates $\boldsymbol{v_j} \equiv \boldsymbol{v_o}$. In that case, all locking-pair operation types are equally represented within the locked design, disabling the ability of ML to learn from relocking (as discussed in Section 3). Furthermore, the formulation of the Euclidean distance was adjusted as presented in Algorithm 2. For a selected $\boldsymbol{v_o}$, the algorithm allows the exclusion of selected $|ODT|$ values from the calculation, enabling two metric variants: *restricted* and *global* learning resilience.

Global security metric (M^g_{sec}) considers all ODT entries to determine d_e, regardless of whether operations from a selected locking pair are affected by locking or not. This metric is suitable to guide heuristics when it is not clear which operation types will be locked. Thus, M^g_{sec} describes *the potential for exploitation within a design.* Since M^g_{sec} considers all ODT values, $\boldsymbol{v_o}$ does not contain any 'x' values. Hence, M^g_{sec} is monotonic.

Restricted security metric (M^r_{sec}) only considers ODT entries in which either T or T' are affected by locking. The reason is that an ML model cannot learn from operations from a selected locking pair if neither T nor T' operations are locked. In this sense, M^r_{sec} captures the security of the design when only considering locked operations, i.e., *the actual exploitability of the design.* If a selected locking pair is included during a locking procedure, certain 'x' values in $\boldsymbol{v_o}$ are set to 0. Thus, M^r_{sec} is not monotonic.

If all types in ODT are affected by locking, $M^r_{sec} \equiv M^g_{sec}$. Furthermore, $M^r_{sec} = 100$ does not imply $M^g_{sec} = 100$, since some operation types are not affected by locking. However, if $M^g_{sec} = 100$ then $M^r_{sec} = 100$. These metrics are used by the locking algorithms.

4.2 Exact ML-Resilient Algorithm (ERA)

ERA (Algorithm 3) ensures that locking always yields a secure design even if the key budget is exceeded. While the key budget is not exceeded (line 4), ERA randomly selects a type T from a

Algorithm 4: HRA: Heuristic ML-Resilient Algorithm

```
    Input: Key budget $k_b$ and RTL design $D$
    Output: Locked RTL design
 1  LoadODT($D$)                                  // Populate ODT
 2  $n \leftarrow 0$                               // Initialize used bits var
 3  $v_i \leftarrow$ ExtractVector($D.ODT$)        // Initial vector
 4  $\Theta \leftarrow \{(T_1, T_1'), \ldots, (T_n, T_n')\}$   // Valid locking pairs
 5  while $n < k_b$ do
 6  |   $M_{sec}^{g} \leftarrow 0$                 // Track max metric value
 7  |   $j \leftarrow 0$                           // Track operation index
 8  |   $P \leftarrow$ RndBoolean()                // Include randomness
 9  |   if $P$ then
10  |   |   $j \leftarrow$ RndSelect($|\Theta|$)
11  |   else
12  |   |   Shuffle($\Theta$)
13  |   |   for $i \leftarrow 0; i < |\Theta|; i++$ do
14  |   |   |   Lock($\Theta[i][0], ODT, D, False$)
15  |   |   |   $v_j \leftarrow$ ExtractVector($D.ODT$)
16  |   |   |   $M_i \leftarrow$ EvalMetric($v_i, v_j$)
17  |   |   |   UndoLock($D$)                      // Undo last lock
18  |   |   |   if $M_i > M_{sec}^{g}$ then
19  |   |   |   |   $M_{sec}^{g} \leftarrow M_i$
20  |   |   |   |   $j \leftarrow i$
21  |   |   end
22  |   end
23  |   $s \leftarrow$ Lock($\Theta[j][0], ODT, D, P$)   // Apply lock (Algorithm 1)
24  |   $n \leftarrow n + s$
25  end
26  return $D$
```

randomly selected pair ϑ from valid locking pairs Θ (lines 5-6). To ensure a secure solution after each selection, the algorithm repeats the locking for the selected type until $ODT[T]$ reaches 0 (lines 7-10). This way the selected operation pairs yield a balanced solution. Thus, $M_{sec}^{r} = 100$ after each locking round even if the cost is more than allowed. Hence, ERA prioritizes security over cost. ERA always locks all selected pairs until $ODT[T]$ reaches 0—all affected pairs are guaranteed to be balanced. The security evaluation of an ERA-locked design will result in $M_{sec}^{r} = 100\%$, but not necessarily in $M_{sec}^{g} = 100\%$. The former states that all affected pairs are perfectly balanced. The latter indicates that other parts of the design are exploitable by ML *if not locked properly* (if $M_{sec}^{g} < 100\%$).

4.3 Heuristic ML-Resilient Algorithm (HRA)

HRA (Algorithm 4) performs iterative fine-grained balancing of locking-pairs in the target design to get closer to the secure solution at every step without exceeding the key budget. While key bits are available (line 5), HRA randomly chooses (line 8). Either a random operation type is chosen (line 10) or the best type is chosen (lines 12-21). The latter case evaluates all locking pairs in Θ and checks which one yields the *highest* increase in M_{sec}^{g}. In both cases, the selected pair is locked by the Lock function (line 23). As HRA performs fine-grained design adjustments, it uses the exact key budget and trades off against the guarantee to reach a secure solution. Since HRA ensures that every step increases security and decreases operation imbalances, it must be guided by the monotonic M_{sec}^{g} metric.

4.4 Metric-Guided Design

The proposed metrics can be used to design various locking algorithms targeting learning resilience. Let us consider a design with the following ODT entries: $|ODT[(+,-)]| = 25$ and $|ODT[(<<, >>)]| = 10$. As depicted in Fig. 5a, M_{sec}^{g} represents a smooth, monotonic surface. Fig. 5b presents the evolution of the metric in each step. The goal of locking is to move the target design from the initial point (bottom right) to the secure point (top left). The path between these two points represents different heuristic approaches. ERA forces selected ODT values to 0, thus jumping in two steps to the secure solution alongside the edges of the surface. HRA travels in the steepest direction, taking small steps and remaining on the highest line across the surface. A greedy approach (same as HRA where P in line 8 is always false) traverses the same points as HRA. Fig. 5b suggests that a greedy approach is more efficient than HRA since it reaches full security (i.e., metric equal to 100) with fewer key bits. However, a greedy approach has a negative consequence: reversibility. An attacker can reverse the locking procedure alongside the steepest decreasing direction. Therefore, including random locking decisions within HRA (variable P) thwarts reversibility—even if it takes longer to get to the secure solution. Similar observations hold for M_{sec}^{r} since $M_{sec}^{r} \equiv M_{sec}^{g}$ when all ODT entries are affected.

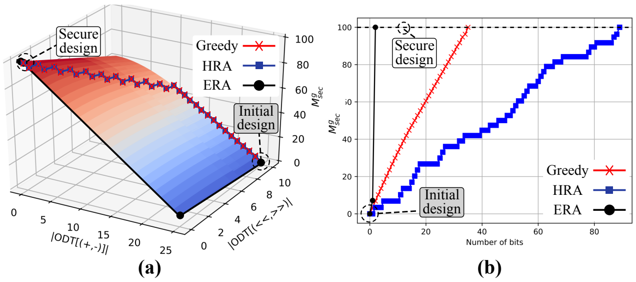


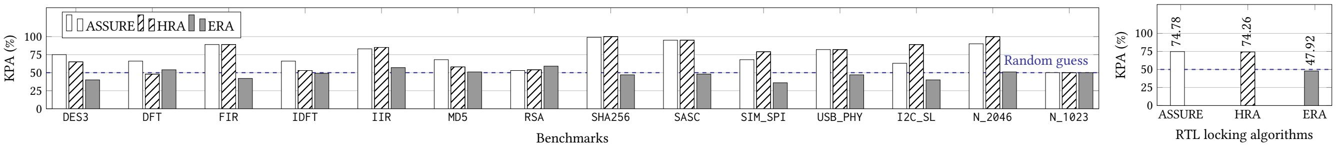
Figure 5: Security metrics: (a) search space and (b) evolution.

5 EVALUATION

We evaluate ASSURE-based locking policies against the state-of-the-art ML-based SnapShot attack on a subset of the benchmarks used in [5]. Some benchmarks were excluded due to the low number of operations. We also composed two synthetic benchmarks: `N_2046` and `N_1023`, representing a fully imbalanced (biased) design (a network of 2046 + operations) and a fully balanced design (a network with 1023 + and − operations), respectively. We consider ASSURE (serial implementation), HRA, and ERA. Note that the cost of the proposed algorithms are in line with the original ASSURE, as the cost of a locking pair per key bit has not changed [5].

SnapShot for RTL: We adapted SnapShot (Fig. 2) to learn RTL key leaks by extracting all key-controlled pairs $[K[i], C_1, C_2]$, where $K[i]$ is the key-bit value, and C_1, C_2 are encodings for an operation pair. Each type is assigned a unique integer. The extractor is based on Pyverilog [21]. Instead of one neural network type as in [6], we use auto-sklearn [13], a library for automatic ML (auto-ml) model exploration. Auto-ml searches for a suitable ML model and optimizes the hyperparameters. We selected 600 seconds per attack iteration as this was enough for the attack to converge.

Attack setup: The test set for each algorithm comprises every benchmark locked 10 times with different keys. We assembled the training set by *relocking* each test sample 1,000 times with different keys. Relocking was performed with random ASSURE locking so that all parts of the design were used for learning; thus, simulating the most effective attack. Both test and training keys are set to 75%



(a) KPA per benchmark

(b) Average KPA

Figure 6: Evaluation results for the ML-based SnapShot attack on RTL locking.

of the available operations. This was exceeded for N_2046, as its perfect imbalance requires a 100% key budget for ERA.

Accuracy metric: Key Prediction Accuracy (KPA) is used to measure attack success [6]. N% KPA indicates that N% of the key bits are correctly predicted. A random guess results in 50% KPA.

5.1 Results and Discussion

Results: Fig. 6a presents the KPA evaluation results per locking algorithm and benchmark, and Fig. 6b presents the average KPA across all benchmarks. SnapShot correctly predicts 74.78% key bits for the original ASSURE implementation, on average. The average KPA for HRA is slightly lower, 74.26%. ERA averages ~47.92% KPA with consistent KPA values around (or lower) than a random guess.

Lessons learned: SnapShot's success on HRA is at first surprising since it is supposed to have a higher level of security than non-ML-driven serial locking. However, since we use a key budget of 75% of the available operations, parts of the design remain unaffected by locking. Hence, the training step *can extract knowledge* about the design for an educated guess (~24 percentage points better than random). Once all operations are fully balanced—as guaranteed by ERA—the training fails to extract useful observations. The above leads to a significant conclusion: ***when it comes to ML-driven attacks, half measures are not effective.*** Data-driven approaches can exploit even the slightest imbalance. In contrast, half-way measures can mitigate non-ML-driven attacks, e.g., slightly increasing the key length can deteriorate a brute-force attack. While HRA appears less promising, the heuristic is useful if multiple security objectives must be reached, such as learning-resilience, output corruptibility, and Boolean Satisfiability (SAT)-resistance [3]. Since ERA makes coarse-grained modifications, it might create radical changes in the design. HRA improves learning resilience of locked designs alongside other objectives *in smaller and controlled locking steps* as it only decreases operation imbalance.

Limitations and opportunities: This study exploits individual locking pairs—but is there a "global bias" among designs? If so, this bias could help determine the correct function of locked designs. The metric in Section 4.1 can extract the initial distance for selected designs by considering the distance between the initial distribution and the optimal one. Are the locking algorithms resilient to oracle-guided attacks? Moreover, locking has recently been explored in combination with high-level synthesis [4, 14, 20]. Future efforts should evaluate the problem of learning resilience on this abstraction level and address the mentioned challenges.

6 CONCLUSION

We introduced the first concepts on designing and evaluating RTL locking using ML-based attacks on operation obfuscation, and proposed two ML-resilient locking algorithms. The heuristic algorithm is a controlled procedure that decreases the imbalance of operations in an RTL design in small steps, adhering to the allowed key budget. The exact algorithm guarantees ML resilience but can exceed a key budget. We presented a security metric to assess resilience of RTL locking to ML attacks that can guide the design process of heuristic locking. Finally, we presented the first ML-based oracle-less attack on RTL locking by adapting the state-of-the-art SnapShot attack.

ACKNOWLEDGMENTS

R. Karri was supported in part by ONR Award # N00014-18-1-2058, NSF Grant # 1526405, NYU Center for Cybersecurity, and NYUAD Center for Cybersecurity.

REFERENCES

[1] R. S. Chakraborty and S. Bhunia. 2010. RTL Hardware IP Protection Using Key-Based Control and Data Flow Obfuscation. In *2010 VLSI Design*. 405–410.

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

[3] C. Karfa et al. 2020. Is Register Transfer Level Locking Secure?. In *2020 DATE*. 550–555. https://doi.org/10.23919/DATE48585.2020.9116261

[4] C. Pilato et al. 2018. TAO: Techniques for Algorithm-Level Obfuscation during High-Level Synthesis. In *2018 DAC*. https://doi.org/10.1109/DAC.2018.8465830

[5] C. Pilato et al. 2021. ASSURE: RTL Locking Against an Untrusted Foundry. *IEEE TVLSI* 29, 7 (2021), 1306–1318. https://doi.org/10.1109/TVLSI.2021.3074004

[6] D. Sisejkovic et al. 2021. Challenging the Security of Logic Locking Schemes in the Era of Deep Learning: A Neuroevolutionary Approach. *ACM JETC* 17, 3, Article 30 (May 2021), 26 pages. https://doi.org/10.1145/3431389

[7] D. Sisejkovic et al. 2021. Deceptive Logic Locking for Hardware Integrity Protection against Machine Learning Attacks. *IEEE TCAD* (2021). https://doi.org/10.1109/TCAD.2021.3100275

[8] D. Sisejkovic et al. 2021. Logic Locking at the Frontiers of Machine Learning: A Survey on Developments and Opportunities. In *2021 VLSI-SoC*. 1–6. https://doi.org/10.1109/VLSI-SoC53125.2021.9606979

[9] J. A Roy et al. 2008. EPIC: Ending Piracy of Integrated Circuits. In *2008 DATE*. 1069–1074. https://doi.org/10.1109/DATE.2008.4484823

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

[11] L. Alrahis et al. 2021. GNNUnlock: Graph Neural Networks-based Oracle-less Unlocking Scheme for Provably Secure Logic Locking. In *2021 DATE*. 780–785. https://doi.org/10.23919/DATE51398.2021.9474039

[12] L. Alrahis et al. 2021. OMLA: An Oracle-less Machine Learning-based Attack on Logic Locking. *IEEE TCSII* (2021), 1–1. https://doi.org/10.1109/TCSII.2021.3113035

[13] M. Feurer et al. 2015. Efficient and Robust Automated Machine Learning. In *Advances in Neural Information Processing Systems 28 (2015)*. 2962–2970.

[14] M. Muttaki et al. 2021. HLock: Locking IPs at the High-Level Language. In *2021 DAC*. 79–84. https://doi.org/10.1109/DAC18074.2021.9586159

[15] M. Yasin et al. 2016. On Improving the Security of Logic Locking. *IEEE TCAD* 35, 9 (2016), 1411–1424. https://doi.org/10.1109/TCAD.2015.2511144

[16] M. Yasin et al. 2017. Evolution of logic locking. In *2017 VLSI-SoC*. 1–6.

[17] M. Yasin et al. 2020. *Trustworthy Hardware Design: Combinational Logic Locking Techniques*. Springer. https://doi.org/10.1007/978-3-030-15334-2

[18] P. Chakraborty et al. 2021. SAIL: Analyzing Structural Artifacts of Logic Locking Using Machine Learning. *IEEE TIFS* 16 (2021), 3828–3842.

[19] S. Amir et al. 2017. Comparative Analysis of Hardware Obfuscation for IP Protection. In *2017 GLSVLSI*. 363–368. https://doi.org/10.1145/3060403.3060495

[20] S. A. Islam et al. 2020. High-Level Synthesis of Key-Obfuscated RTL IP with Design Lockout and Camouflaging. *ACM Trans. Des. Autom. Electron. Syst.* (2020).

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