Low Overhead System-Level Obfuscation through Hardware Resource Sharing

Daniel Xing*, Michael Zuzak†, Ankur Srivastava*
*University of Maryland
{dxing97, ankurs}@umd.edu
†Rochester Institute of Technology
mjzeec@rit.edu

***Abstract*—Logic locking techniques have been proposed to protect chip designs from malicious reverse engineering and overproduction. Stripped functionality logic locking (SFLL) has gained substantial traction as a current state of the art method, exhibiting strong resilience against a wide variety of attacks. However, secure instances of SFLL-based locking tend to have high power and area overheads, particularly in its restore units. This work presents a novel architectural approach to restore unit configuration for SFLL-like logic locking methods that treats restore units as an overhead-constrained shareable resource. We describe how resource contention caused by sharing of restore units imposes constraints on the underlying locking scheme from a graph theoretic perspective and propose both a 0-1 ILP and a heuristic clustering algorithm for finding resource-constrained shared locking configurations that satisfy these constraints. We evaluate our sharing method on SFLL-flex and find that our ILP and heuristic methods were each able to achieve a 55% and 31% reduction in power used by locked datapaths synthesized from MediaBench benchmarks while maintaining the same security and functionality compared to datapaths locked with conventional gate-level techniques.**

***Index Terms*—logic locking, hardware obfuscation, SFLL, resource sharing**

I. Introduction

The high capital cost of silicon chip design, especially at the latest process technologies, has forced most design houses to contract chip fabrication and packaging out to third-party foundries. With so many steps in the chain controlled by third parties, potentially across the world, chips with novel designs and secure applications may be at risk of counterfeiting, overproduction, and reverse engineering [1].

To thwart these supply chain attacks, logic locking techniques were developed to protect chips from attack by obscuring select combinational modules within a chip from third parties via a key mechanism [2]–[10]. Manufactured chips that implement logic locking are not fully functional unless the correct key is loaded onto the locked chip by the IP designer. Unauthorized copies without a correct key will function incorrectly, preventing their misuse. A survey of logic locking research can be found in [11], [12].

Among the most prominent class of logic locking techniques are those based on *stripped functionality logic locking*, or SFLL [2]–[4]. SFLL locks a module by replacing it with a *stripped module* that only partially implements the functionality of the original. To restore stripped behavior, additional reconfigurable circuitry (called the restore unit) is added that reimplements correct functionality when provided a correct key.

SFLL [2]–[4] is provably secure against state-of-the-art attacks such as SAT [13], [14], removal [15], [16], and structural [17] attacks. Additionally, SFLL is highly configurable to meet security and application requirements for a wide range of designs [18] and has been implemented and validated in several real-world designs [19].

Our proposal augments SFLL by taking an architectural approach to locking a design. Instead of considering each locked functional module in isolation when making locking decisions, we show that restore units can be shared across multiple locked modules, enabling a locking scope that expands *beyond gate level boundaries*. While a high level approach to logic locking is not unique [20]–[24], this work is the first to explore utilizing restore units as a *shareable architectural resource* that can be bound to multiple locked modules in a design. This high level view affords the designer significant flexibility in reducing area and power overheads as well as allowing for a mathematically rigorous and tunable optimization scheme that gives granular control over the error-overhead trade-off to the engineer.

A. Contributions

In this work, we utilize our architectural view of logic locking to explore a novel use-case. Namely, treating restore units as a shared resource for overhead reduction. We present two mathematically-rigorous methods to explore sharing. Both methods come with tunable gate-level SAT attack resilience and guarantee that all stripped functionality will be properly restored given the correct key. Our contributions are summarized as follows:

1) We define a formal mathematical model for restore unit sharing in the form of a 0-1 integer linear program (ILP). Our formulation finds solutions that meet resource contention, attack resilience, and overhead constraints while also optimally maximizing system-level locking-injected errors.
2) We present and mathematically formalize an equivalent graph-theoretic model that we build into a hierarchical clustering heuristic algorithm that satisfies the same constraints as the ILP and finds solutions in polynomial time in exchange for sub-optimal error injection profiles.

This work was supported by the NSF under Grant 1953285.

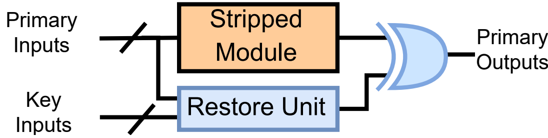


Figure 1: The general structure of SFLL locking techniques.

3) We empirically evaluate our methods on MediaBench benchmark circuits locked with SFLL-flex [2], and find that our ILP method on average yields a 55% power and 6% area reduction while maintaining the same locking efficacy as a locking architecture without sharing restore units. Similarly, our P-time heuristic achieves a 31% and 4% reduction in power and area overhead respectively.

II. Preliminaries

A. Attacker Model

The logic locking attacker model we consider is a common model used by recent works in logic locking [2]–[5], [13], [14], [20], [25]. We assume an adversary with access to

1) a locked module's netlists, obtainable via reverse engineering the locked chip's layout
2) an black-box oracle chip with scan-chain access that the attacker can query with inputs and read out correct outputs

The goal of this attacker is to find a key value that produces a fully functional chip.

Logic locking methods are subject to powerful Boolean satisfiability-based attacks (SAT attacks) [14]. The SAT attack finds a key value for the locked module that results in an input-output mapping identical to the oracle. SAT attacks convert the locked module into a Boolean expression that returns TRUE if the module's input and key value yields a value that matches the oracle's output. The attack then iteratively prunes away wrong key values by finding inputs that give different outputs from the oracle until no such inputs can be found. The resulting final key will be functionally equivalent to the correct key.

B. SFLL

SFLL [2]–[4] is one of the leading logic locking techniques resistant to SAT attacks. At its core is the idea of *functionality stripping*; altering the output of a locked module only when certain inputs are applied. The general structure of a module locked with SFLL is shown in Fig. 1. The original module is replaced with a *stripped module* which removes the functionality of selected protected input patterns (PIPs) from the original design. Stripped functionality is restored by adding a *restore unit* alongside the stripped module. The restore unit corrects incorrect outputs produced by the stripped module provided the PIPs used in the corrupting unit are supplied as key inputs to the restore unit.

By changing the number of PIPs, the rate of wrong key errors can be adjusted, allowing the designer to tune SFLL's construction to the design's target wrong-key error rate. However, inherent to this tunability is a direct inverse relationship between the number of PIPs and the expected time needed to reverse engineer the key by SAT attack [26], [27], thus forcing the designer to choose between a high level of wrong-key error and good resilience against SAT. Selecting PIPs that occur more frequently in system traces can somewhat lessen the tradeoff [20]. Fig. 2 shows the expected number of SAT queries required for a locked module with 64 bit inputs for both varying numbers of 64-bit PIPs and varying PIP widths.

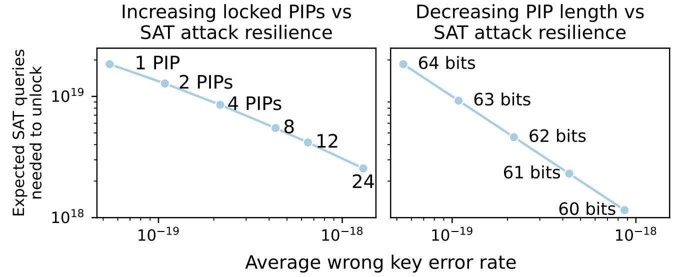


Figure 2: Relationships between number of PIPs, PIP width, average error rate, and SAT iterations derived in [26] for a locked module with 64 bit inputs. Increasing the average error rate by either of these two methods always leads to a decrease in SAT attack resilience.

clock	1	2	3	4	5	6
Module 1	$\langle 15\rangle$	$\langle 22\rangle$	$\langle 22\rangle$		$\langle 35\rangle$	$\langle 41\rangle$
Module 2	$\langle 22\rangle$	$\langle 22\rangle$	$\langle 35\rangle$	$\langle 35\rangle$	$\langle 15\rangle$	$\langle 35\rangle$

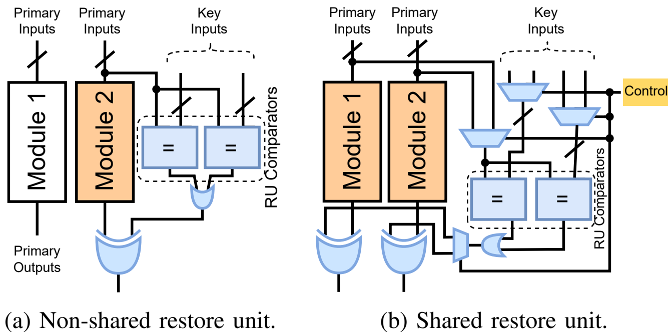
Figure 3: Two functional modules and a system input trace of the two modules for a single operation.

III. Why Share?

Conventional locking approaches decide what functionality to strip on a module-by-module basis after the overall RTL (or even gate level) architecture has been fixed. Subsequently, every locked module has its own dedicated restore unit. Since restore units are generally implemented as comparators, each locked module will contribute a significant amount of additional power and area. Sharing the comparator across multiple functional modules reduces the design overhead incurred by restore units. Additionally, restore units function independently from the specific stripping method used, so sharing them allows the designer to consider overhead and error relevant locking decisions at an architectural level, allowing greater flexibility on where to balance a design's overhead against its security. To illustrate how our method compares to the conventional module-level approach to restore units, we demonstrate how our method can be used to lock a small two module example design using a trace. For this example we assume a constrained overhead budget that only allows for the one restore unit, and only one PIP is used to strip a locked module. Additionally, the designer has access to traces fully describing all candidate PIPs that can be used for locking.

A. Conventional Module-level Locking

An example of a conventional restore unit restoring just one stripped module is shown in Fig. 4a. Note that only module 2



(a) Non-shared restore unit. (b) Shared restore unit.

Figure 4: Two 2-module locked datapaths. 4a is locked with a conventional approach, while 4b is locked with our sharing approach. Locked modules are shown in orange and each strip two PIPs.

is locked; the restore unit needed for locking module 1 would require additional overhead. The simplest locking policy would be to randomly choose one input pattern from all possible PIPs to be the PIP for each locked module. However, to differentiate between different PIP selections, we can use the trace to choose a PIP that occurs more often in order to inject more output error. Increasing an individual functional module's error rate in this way results in more errors propagated to the rest of a system, therefore decreasing the functional utility of a locked chip for a malicious user. For the example trace shown in Fig. 3, locking module 2 with $\langle 35 \rangle$ as the PIP will create the highest amount of error (three total, at clocks 3, 4, and 6) with the overhead constraint of one locked module.

B. Locking with a Shared Restore Unit

By sharing a restore unit, we can amortize the area and power cost of the comparators in each restore unit over multiple stripped modules. An example of a shared restore unit restoring two locked modules is shown in Fig. 4b. The shared restore unit has the same function as before: compare an input with a set of key inputs, and send a restoration signal if any match. A significant difference is that only one of the two stripped modules can be restored in any clock cycle, which will impose restrictions for the PIPs selected to lock each module.

Which module to examine and which set of key values to use at each clock is controlled by a set of MUXes. On clocks in the trace where PIPs may potentially occur, the controller will direct the restore unit to the locked module where it can occur. On clocks where no module recieves a PIP, the controller can direct the restore unit to the corresponding locked module.

To illustrate the restrictions shared restore units impose, let us examine the system trace in Fig. 3 and choose the most common input for each module to be its respective PIP: $\langle 22 \rangle$ for module 1 and $\langle 35 \rangle$ for module 2. In the first clock, neither module receives a PIP as an input, so the controller can have the comparator examine either module without affecting system functionality for correct keys. In the second clock, module 1 receives its PIP $\langle 22 \rangle$ as an input and therefore the output will need to be restored, while module 2 can operate as-is since $\langle 22 \rangle$ is *not* a PIP for module 2, only module 1. Therefore, the controller should direct the restore unit to examine module 1's input and restore if needed.

Table I: Variables and constants used in the ILP formulation

Notation	Definition
s	Total number of allocated shared restore units
r	Total number of unique input patterns
u	Number of functional modules
T	Number of clock cycles in the trace
c_j	Maximum number of PIPs allowed for functional module j
$d_{i,j,k}$	Encodes trace information. $d_{i,j,k}$ is 1 if the ith candidate minterm is an input to functional unit j at clock cycle k in the trace, and is 0 otherwise
$x_{i,j}$	Variable that encodes locked minterm assignments to functional units. $x_{i,j}$ is 1 if the ith candidate minterm is a locked minterm for functional unit j, and is 0 otherwise
$a_{j,l}$	Variable that encodes functional unit assignments to restore units. $a_{j,l}$ is 1 if functional unit j is restored by restore unit l, and is 0 otherwise
$h_{i,j,k,l}$	Variable associated with locked input minterms in the trace. $h_{i,j,k,l}$ is 1 if the following hold: 1) minterm i is used to lock module j 2) module j is restored by restore unit l and 3) minterm i is an input minterm for module j at clock cycle k in the trace, 0 otherwise. Highly sparse.

In the third clock, both modules receive PIPs as inputs, so both module outputs will need to be restored. However, the multiplexer can only choose one module per clock, leading to resource contention. Both locked modules require exclusive use of a resource—the restore unit—at the same clock, but the controller only has one restore unit available to allocate, so the corrupted output of one of the modules will go unrestored, *independent of whether or not the key inputs are correct*.

To avoid resource contention in an overhead constrained design, we can instead pick PIPs that never cause resource contention given a set of comprehensive traces. For this example, if we instead lock module 1 with $\langle 15 \rangle$ and module 2 with $\langle 35 \rangle$, then only one module needs to be restored at any clock in the trace, and therefore no resource contention occurs. This contention-free PIP selection causes error in clock cycles 1, 3, 4, and 6, for a total of four injected errors, an improvement over the three errors injected with an unshared restore unit.

This approach—choosing PIPs that never cause resource contention in a trace—is the approach we take for sharing restore units. Our methods find locking configurations that meet contention avoidance, resource budget and attack resilience constraints while optimizing for high levels of injected error.

IV. Error-Optimal Sharing via ILP

The problem of finding error optimal restore unit bindings and PIPs can be described as a 0-1 ILP. To make our formulation easy to follow, Table I lists all constants and variables used, along with their definitions.

We begin with the constraints. First, each stripped module should lock just enough minterms to be sufficiently robust against expected SAT attacks [26], [27]. This can be expressed as the following constraint:

$$\sum_{i=1}^{r} x_{i,j} \leq c_j, \quad \forall j \in [1, u] \tag{1}$$

To keep MUX overhead low, stripped modules can only by restored by one shared restore unit. Therefore, a stripped functional unit with locked minterms can only be restored by exactly one restore unit. We use the following inequalities to describe these two requirements:

$$x_{i,j} \leq \sum_{l=1}^{s} a_{j,l} \leq 1, \quad \forall i \in [1,r], j \in [1,u], \tag{2}$$

To ensure PIP assignments decided by $x_{i,j}$ do not cause contention over restore units configured by $a_{j,l}$, i.e. the sets of locked minterms belonging to different functional modules unlocked by the same restore unit should never occur in the same clock, we use inequality (3) to calculate which input patterns i in the trace are PIPs for a given restore unit l and locked module u at clock cycle k in the trace. Inequality (4) constrains the number of restored modules per restore unit per clock to be at most one, otherwise contention occurs.

$$a_{j,l} x_{i,j} d_{i,j,k} \leq h_{i,j,k,l} \quad \forall i,j,k,l \tag{3}$$

$$\sum_{i=1}^{r} \sum_{j=1}^{u} h_{i,j,k,l} \leq 1 \quad \forall k \in [1,T], l \in [1,s] \tag{4}$$

Note that inequality (3) is not linear. However, the decision variables $x_{i,j}$ and $a_{j,l}$ are constrained to be binary, and d is a constant value representing the trace. Therefore, the quadratic constraint inequality (3) can be rewritten as an equivalent set of linear inequalities by adding an additional variable. Additionally, note that if $d_{i,j,k}$ is 0 for some value of (i,j,k) (which is most values of (i,j,k)), then the corresponding constraint is redundant and can be removed, greatly reducing the number of constraints and additional h variables needed. The resulting linear expressions are

$$\left.\begin{aligned} a_{j,l} + x_{i,j} - 1 &\leq h_{i,j,k,l} \\ h_{i,j,k,l} &\leq a_{j,l} \\ h_{i,j,k,l} &\leq x_{i,j} \end{aligned}\right\} \begin{aligned} \forall (i,j,k) \in \{i,j,k | d_{i,j,k} = 1\} \\ \forall l \in [1,s] \end{aligned} \tag{5}$$

The optimization objective is to maximize the number of corrupted outputs in the trace caused by PIPs:

$$\max_{a,x,h,g} \sum_{j=1}^{u} \sum_{i=1}^{r} b_{i,j} x_{i,j} \tag{6}$$

subject to inequalities (1) to (3) and (5).

Feasible solutions to this ILP represent the set of valid locking configurations which do not cause resource contention and respect overhead and SAT attack resilience requirements. Optimal solutions are locking configurations that maximize injected errors in the trace used to configure locking.

A. ILP Usage

A designer seeking to secure a design within locking-induced overhead and security constraints while maximizing locking's impact on the functionality of a locked design using shared restore units would need to determine what functional modules should be locked, the number of PIPs each locked module should have, and the number of restore units that can be allocated from the design's power budget. Additionally, a set of traces that fully describe all conflicts between each module's prospective PIPs will be needed.

These values can then be encoded into our ILP formulation as described previously and solutions found using one of the many available ILP solvers. Optimal values of x, a, and h can then be used to set PIP assignments, configure restore unit MUXes, and program controller behavior respectively.

V. Efficient Sharing via Graphs

While in practice ILP solvers can efficiently solve some large instances of integer LPs, the worst-case runtime is still NP-complete. To provide an alternative to avoid such worst-case time complexity, we present a graph theoretic model for sharing in this section that will be used as a foundation for our heuristic algorithm in section VI.

A. Locking and Sharing Configurations with the PIP Compatibility Graph

We define entries in the trace as three dimensional vectors of the form $(i,j,k), i \in [1,r], j \in [1,u], k \in [1,T]$. Variables are defined the same as our ILP (e.g. i represents a specific input pattern, u represents the number of lockable modules, etc). We represent the entire trace and its entries as a set of these three dimensional vectors. We define a PIP assignment as a two dimensional vector of the form $(i,j), i \in [1,r], j \in [1,u]$ and represents the decision of stripping minterm i from module j.

We use the term *PIP compatibility* to refer to whether or not a pair of PIP assignments to two different locked modules will cause restore unit contention if a restore unit is shared between the two modules. It is defined as follows:

Definition V.1 (Incompatible and Compatible PIPs)**.** Two PIP assignments $v_1 = (i_1, j_1), v_2 = (i_2, j_2)$ where $i_1, i_2 \in [1,r]$ and $j_1, j_2 \in [1,u], j_1 \neq j_2$ are *incompatible* if (i_1, j_1, k) and (i_2, j_2, k) are both in the trace for some $k \in [1,T]$. A pair of PIP assignments v_1, v_2 that do not satisfy this property are *compatible*.

We can use this definition to build the multipartite PIP compatibility graph $G_P(V_P, E_p, w)$. G_P represents all pairs of compatible stripping decisions between modules in the design. The set of nodes $V_P = V_{P1} \cup V_{P2} \cup ... \cup V_{Pu}$ consist of two dimensional PIP assignment vectors $(i,j) \in V_{Pj}, i \in R, j \in U$ where each partition V_{P_j} contains all potential PIP assignments that strip module j. An edge exists between two nodes $(v_1, v_2) \in E_P$ if v_1, v_2 are compatible as defined previously. The node weight function $w : V_P \rightarrow \mathbb{N}$ weights each PIP assignment $(i,j) \in V_P$ by how often input pattern i occurs as an input to functional module j in the trace. The set of PIPs chosen to strip locked modules can then be defined as a subset $S \subseteq V_P$ of all possible PIP assignments.

Similarly to PIP assignments, we define a restore unit resource assignment to be a two dimensional vector $(j,l) \in W, j \in [1,u], l \in [1,s]$ that represents the decision of having functional module j be restored by restore unit l.

We denote W as the set containing our chosen module-to-restore unit mapping vectors (j, l) for an entire design. To keep area overhead caused by MUXes low, each locked module can only be restored by exactly one restore unit. Formally, we can define this constraint as follows: we first define W_l to be the set of locked modules restored by restore unit l

$$W_l := \{j \in U | (j, l) \in W\} \tag{7}$$

then for two different restore units l_x and l_y,

$$W_{l_x} \cap W_{l_y} = \emptyset \tag{8}$$

must hold.

We can now define a *locking configuration* (S, W) as a structure containing both a set of PIP stripping decisions S and a sharing configuration W. Now we can define a contention-free locking configuration. First, we define S_l as the set of PIP assignments restored by restore unit l:

$$S_l := \{(i, j) \in S | (j, l) \in W\} \tag{9}$$

We can formally define a *contention-free* locking configuration by using the PIP compatibility graph G_P defined earlier:

$$\begin{aligned} \text{if} \quad & (s_1, s_2) \in E_P \\ & \forall s_1, s_2 \in S_l, \;\; s_1 \in V_{Pi}, s_2 \in V_{Pj}, \;\; i \neq j, \;\; i, j \in W_l \\ & \forall l \in [1, s], \end{aligned}$$

then (S, W) will not cause resource contention.

Note that this definition of a contention-free locking configuration is equivalent to the definition of a set of partition disjoint (from equation 8) multipartite cliquess on G_P. Therefore it can be shown that a set of multipartite cliques, each containing members from disjoint partitions of G_P, represents a locking configuration that never causes contention in the trace. From here, we can define a graph problem equivalent to the ILP:

$$\underset{S}{\text{argmax}} \sum_{s \in S} w(s) \tag{10}$$

subject to

$$|\{(i, x) \in S | x = j\}| \leq c_j \tag{11}$$

and (S, W) being contention-free as defined before.

Finding a locking configuration (S, W) that meets these constraints while maximizing corrupted outputs can be seen as finding a maximum weight set of multipartite cliques on G_P with constraints on the number of cliques and cardinality of each clique. Like the ILP described previously, no polynomial time algorithm for finding such cliques is known. The rest of this section will be describe our heuristic algorithm which trades output corruption optimality for a polynomial algorithm runtime.

VI. Towards a Heuristic Algorithm

We propose a method based on hierarchical agglomerative clustering to find locking solutions that meet our other stated objectives of no contention, high corruptability, low overhead, and attack resilience in polynomial time at the expense of a suboptimal number of corrupted outputs. We start with introducing a special case of the optimization problem defined in section V-A—the two-module sharing problem—and show that optimal solutions to this special case can be found in polynomial time. We then use these two-module solutions to initialize our hierarchical clustering heuristic algorithm.

A. The Two Module Problem

The two module problem considers the specific case of two functional modules sharing a single restore unit, and is needed in the initialization step of our heuristic. We relax inequality (1) for the two-module case; guaranteeing attack resilience by limiting the number of PIPs per module will be performed later.

The PIP compatibility graph for two modules is a node-weighted bipartite graph $G_P(V_{P1}, V_{P2}, E_P, w)$. Finding a contention-free locking configuration S with maximum output corruption reduces to finding a maximum node-weighted biclique in G_P:

$$\begin{aligned} \underset{S}{\text{argmax}} \quad & \sum_{s \in S} w(s) \\ \text{subject to} \quad & S \subseteq V_{P1} \cup V_{P2} \\ & \{v_1, v_2\} \in E_P \quad\quad \forall v_1 \in V_{P1} \cap S, v_2 \in V_{P2} \cap S \end{aligned} \tag{12}$$

Fortunately, this problem has been shown to be optimally solvable in polynomial time by converting (12) into an equivalent 0-1 integer linear program and solving its LP relaxation [28].

B. Hierarchical Clustering Heuristic

Our algorithm uses a graph-based data structure $G_R(V_R, E_R, a, A)$ to store its state. G_R is a complete graph of restore units, where each node represents an allocated restore unit and each edge represents a decision to merge two restore units into one. Each node in V_R and edge in E_R has an associated set of PIP assignments stored as node and edge attributes in a and A respectively. The set of node attributes a keeps track of the current locking configuration solution state, while each edge attribute in A represents a potential decision to merge two restore units together the outcome of clustering the two restore units at each edge's endpoints.

The full algorithm is shown in Alg. 1. The algorithm first initializes G_R's nodes to represent a conventional locking configuration where every module has its own dedicated restore unit (line 1). Each node attribute contains PIP assignments that locks every possible PIP for the node's module (line 2). Every edge attribute is initialized to the two module solution introduced in the previous section (line 3).

The clustering step first chooses the pair of nodes $i', j' \in V_R$ whose merger yields the greatest increase in overall corrupted outputs (line 5). We perform the merger by updating the graph in-place: stripping decisions stored in the edge between i' and

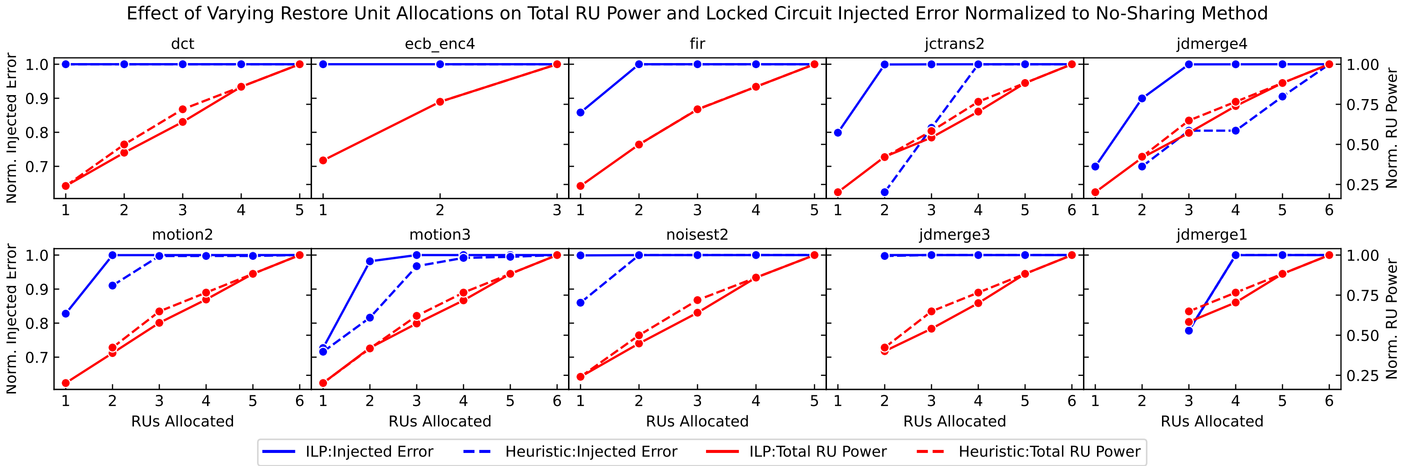


Figure 5: Breakdown of how restore unit power and total injected error is affected by changes in the number of restore units allocated for sharing. All axes are normalized to a design that locks every module with a dedicated (i.e. non-shared) restore unit. Note that low restore unit allocations resulted in no feasible solutions for some benchmarks.

Algorithm 1 Hierarchical clustering algorithm

Input: $G_P(V_P, E_P, w), s_{limit}, c$ in
Output: $G_R(V_R, E_R, a, A)$ out
G_R Initialisation :
1: $V_R \leftarrow \{1, 2, ..., u\}$, E_R is complete
2: $a_i \leftarrow V_{Pi} \quad \forall i = 1...u$
3: $A_{i,j} \leftarrow$ 2 module sol. for modules $i, j \quad \forall \{i, j\} \in E_R$
Iterative Clustering:
4: **while** $|V_R| > s_{limit}$ **do**
5: $\quad i', j' \leftarrow \underset{\{i,j\} \in E_R}{\operatorname{argmax}} \left(\sum_{s \in A_{i,j}} w(s) - \sum_{s \in a_i} w(s) - \sum_{s \in a_j} w(s) \right)$
6: $\quad a_i \leftarrow A_{i',j'}$
7: $\quad$ **for** $k \in V_R, k \neq i', j'$ **do**
8: $\quad\quad A_{i',k} \leftarrow (A_{i',j} \cap A_{i',k}) \cup (A_{i',j} \cap A_{j',k}) \cup (A_{i',k} \cap A_{j',k})$
9: $\quad$ **end for**
10: $\quad V_R \leftarrow V_R \setminus j'$ and assoc. entries in G_R
11: **end while**
12: **for** each module j restored by each RU in G_R **do**
13: $\quad$ select top-c_j most frequently occurring minterms as PIPs for module j
14: **end for**

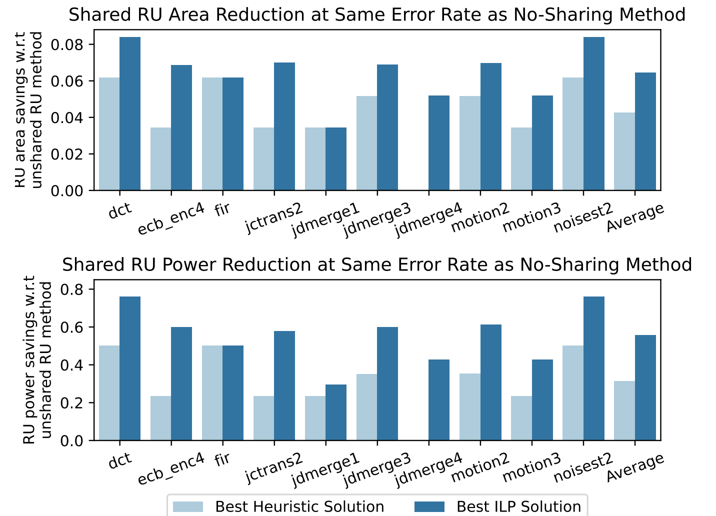


Figure 6: Locked datapath area and power for benchmarks tested, normalized to single-PIP unshared restore units. The lowest power and area achieved with the same total system error as the conventional no-sharing approach are shown here.

j' are copied to node i' (line 6). After that, edge attributes containing future possible merges between the just-merged node and all other nodes are updated. It can be shown that the set operation on line 8 preserves PIP compatibility within each restore unit. An additional check can be added here to remove edges with configurations that do not strip functionality from every module covered by restore units at each edge endpoint. Finally, as a post-processing step, PIP assignments with low weight in G_P are removed until each locked module j only strips c_j input patterns (lines 12 to 14).

The G_R initialization step runs an instance of the LP relaxation of eq. (12) for every pair of modules in the design, for a complexity of $O(LP(r)u^2)$, where $LP(r)$ is the runtime complexity of solving a LP with r variables. The iterative clustering step evaluates a linear expression for every edge in E_R and performs $O(s)$ set intersection and union operations. This step runs up to $s-1$ times, so the clustering portion runs in $O(s^4)$. Finally, selecting the c_j most common input patterns in each locked module runs in $O(r\log r)$. Therefore, our heuristic runs in $O(LP(r)u^2 + s^4 + r\log r)$, a P-time solution.

VII. Results

To evaluate our resource sharing approach to logic locking, we used datapaths synthesized from C functions extracted from the MediaBench suite [29]. The DFG of each function was extracted using SUIF and was scheduled with a path-based

scheduler [30]. An allocation of up to three adders and three multipliers was used, and each functional unit was bound using a security-aware binding algorithm [20]. We used test inputs supplied by Mediabench to generate input traces that represent typical use scenarios. Traces were generated by simulating the DFG with test inputs.

Stripped modules were protected with SFLL-flex, although any stripping method that relies on restore units could have been used. To maximize security against SAT attack, every locked module is allocated only one PIP (i.e. $c_j = 1 \forall j$). We compared our optimal and heuristic formulations to a error-maximum locking solution that does not share restore units.

We used Gurobi [31] as the ILP solver and NetworkX [32] to implement the graph heuristic algorithm. We used the constraint set from the ILP to verify that the heuristic algorithm returns a feasible and valid solution. All experiments were performed with a standard desktop computer equipped with an AMD Ryzen 3.4 GHz processor and 32GB of system memory. Scripts to implement and manage the solvers were written with Python and run under Linux. Power and area overhead numbers were calculated using designs built using FreePDK45 [33] and synthesized with Cadence Genus.

In order to compare our techniques against the conventional no-sharing approach, Fig. 6 shows how power and area overhead used by restore units is reduced by our heuristic algorithm and ILP formulation while maintaining the same number of injected errors. Data shown are normalized by the area and power consumed by the same datapath locked with a conventional error-maximizing approach. Our heuristic found locking solutions that reduced locking-induced power and area overhead by 31% and 4% relative to the conventional approach in most benchmarks, while our optimal ILP solution was able to save 55% and 6% power and area overhead in all benchmarks. Area savings were less significant due to the additional MUXes needed to switch between restore units. Since the restore units do not lie on a critical timing path, no design experienced any change in their timing under any locking scenario.

While both of our methods do not explicitly consider area and power, the clustering heuristic consistently shows reduced area and power savings compared to ILP-derived sharing configurations. Close examination of the sharing configuration solutions derived from each method shows that ILP optimal sharing configurations tend to favor an unbalanced distribution of restore units to functional modules, where one restore unit restores a large set of functional modules and the rest of the restore units each only restore a single module. On the other hand, the heuristic tends to favor a more balanced distribution, where the restoration of modules is evenly divided among the available restore units. This results in more MUXes over the ILP, and therefore a greater area and power overhead.

We can further break down the results by examining how power and total injected errors change with different restore unit resource allocations in Fig. 5. Resource allocations range from allocating one restore unit for each functional module to allocating just one restore unit for all functional modules. Note that allocating a restore unit to each functional module will yield a solution that is identical to the conventional approach, and is reflected in our graphs. From Fig. 5 we can observe that as allocations decrease, the number of injected errors does not decrease until allocations start to approach 1. The heuristic tends to drop in error sooner than the optimal solution for some benchmarks. Meanwhile, overhead trends roughly linearly with resource allocations. We can see that there is significant power overhead margin that can be reclaimed by efficiently sharing restore units at the architectural level while maintaining the same error severity. This tunability can enable the designer greater control on the overall system, allocating more resources towards security features and meeting area and power constraints.

VIII. Conclusion

In this work we present a resource sharing methodology for restore units used by SFLL-based locking techniques. Our method decreases design overhead when compared to a standard non-shared locking configuration at the same level of injected error while meeting SAT resilience requirements. We model resource-contention free resource sharing as a 0-1 integer linear program, and present a polynomial time heuristic algorithm based on hierarchical clustering to find good feasible solutions. We apply the technique on sample traces for synthesized functions from the MediaBench [29] suite of benchmarks and compare overhead reductions for the optimal ILP solution, the heuristic solution, and error-maximal non-shared minterm selection algorithm. Our optimal and heuristic formulation reduces total power consumed by a locked design 55% and 31% respectively when compared against an equivalent locking construction without shared restoration hardware.

References

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

[2] M. Yasin *et al.*, "Provably-secure logic locking: From theory to practice," in *Conference on Computer and Communications Security*, 2017.

[3] A. Sengupta *et al.*, "Atpg-based cost-effective, secure logic locking," in *IEEE 36th VLSI Test Symposium (VTS)*. IEEE, 2018.

[4] ——, "Truly stripping functionality for logic locking: A fault-based perspective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[5] Y. Liu *et al.*, "Strong Anti-SAT: Secure and Effective Logic Locking," in *2020 21st International Symposium on Quality Electronic Design (ISQED)*, Mar. 2020, pp. 199–205, iSSN: 1948-3287.

[6] H. M. Kamali *et al.*, "Full-lock: Hard distributions of sat instances for obfuscating circuits using fully configurable logic and routing blocks," in *Proceedings of the 56th Annual Design Automation Conference*, 2019.

[7] ——, "Interlock: An intercorrelated logic and routing locking," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020, pp. 1–9.

[8] J. A. Roy *et al.*, "EPIC: Ending Piracy of Integrated Circuits," in *2008 Design, Automation and Test in Europe*, Mar. 2008, pp. 1069–1074, iSSN: 1558-1101.

[9] M. Yasin *et al.*, "Sarlock: Sat attack resistant logic locking," in *Intl. Symposium on Hardware Oriented Security and Trust*, 2016.

[10] A. Saha *et al.*, "Lopher: Sat-hardened logic embedding on block ciphers," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020.

[11] S. Dupuis and M.-L. Flottes, "Logic locking: A survey of proposed methods and evaluation metrics," *Journal of Electronic Testing*, vol. 35, no. 3, pp. 273–291, 2019.

[12] A. Chakraborty *et al.*, "Keynote: A disquisition on logic locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[13] K. Z. Azar *et al.*, “Smt attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the sat attacks,” *Transactions on Cryptographic Hardware and Embedded Systems*, 2019.

[14] P. Subramanyan *et al.*, “Evaluating the security of logic encryption algorithms,” in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2015, pp. 137–143.

[15] M. E. Massad *et al.*, “Logic locking for secure outsourced chip fabrication: A new attack and provably secure defense mechanism,” *arXiv preprint arXiv:1703.10187*, 2017.

[16] M. Yasin *et al.*, “Removal attacks on logic locking and camouflaging techniques,” *Transactions on Emerging Topics in Computing*, 2017.

[17] “ENTANGLE: An Enhanced Logic-locking Technique for Thwarting SAT and Structural Attacks,” ser. GLSVLSI ’22, New York, NY, USA. [Online]. Available: https://doi.org/10.1145/3526241.3530371

[18] M. Yasin *et al.*, “SFLL-HLS: Stripped-Functionality Logic Locking Meets High-Level Synthesis,” in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2019, pp. 1–4, iSSN: 1558-2434.

[19] B. Tan *et al.*, “Benchmarking at the Frontier of Hardware Security: Lessons from Logic Locking,” *arXiv:2006.06806 [cs]*, Jun. 2020, arXiv: 2006.06806. [Online]. Available: http://arxiv.org/abs/2006.06806

[20] M. Zuzak *et al.*, “A resource binding approach to logic obfuscation,” in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 235–240.

[21] C. Pilato *et al.*, “On the optimization of behavioral logic locking for high-level synthesis,” *arXiv preprint arXiv:2105.09666*, 2021.

[22] M. R. Muttaki *et al.*, “Hlock: Locking ips at the high-level language,” in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 79–84.

[23] C. Pilato *et al.*, “Assure: Rtl locking against an untrusted foundry,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 7, pp. 1306–1318, 2021.

[24] N. Limaye *et al.*, “Fortifying rtl locking against oracle-less (untrusted foundry) and oracle-guided attacks,” in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 91–96.

[25] D. Sisejkovic *et al.*, “Challenging the Security of Logic Locking Schemes in the Era of Deep Learning: A Neuroevolutionary Approach,” *arXiv:2011.10389 [cs]*, Nov. 2020, arXiv: 2011.10389. [Online]. Available: http://arxiv.org/abs/2011.10389

[26] M. Zuzak *et al.*, “Trace logic locking: Improving the parametric space of logic locking,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[27] H. Zhou *et al.*, “Resolving the trilemma in logic encryption,” in *International Conference on Computer-Aided Design (ICCAD)*, 2019.

[28] M. Dawande *et al.*, “On Bipartite and Multipartite Clique Problems,” *Journal of Algorithms*, vol. 41, no. 2, pp. 388–403, Nov. 2001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S019667740191199X

[29] C. Lee *et al.*, “MediaBench: a tool for evaluating and synthesizing multimedia and communications systems,” in *Proceedings of 30th Annual International Symposium on Microarchitecture*, Dec. 1997, pp. 330–335, iSSN: 1072-4451.

[30] S. Ogrenci Memik *et al.*, “A super-scheduler for embedded reconfigurable systems,” in *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*, Nov. 2001, pp. 391–394, iSSN: 1092-3152.

[31] Gurobi Optimization, LLC, “Gurobi Optimizer Reference Manual,” 2022. [Online]. Available: https://www.gurobi.com

[32] A. A. Hagberg *et al.*, “Exploring Network Structure, Dynamics, and Function using NetworkX,” in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux *et al.*, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.

[33] “FreePDK: An Open-Source Variation-Aware Design Kit,” ser. MSE ’07, USA. [Online]. Available: https://doi.org/10.1109/MSE.2007.44