A Resource Binding Approach to Logic Obfuscation

Michael Zuzak, Yuntao Liu, and Ankur Srivastava
Department of Electrical and Computer Engineering, University of Maryland, College Park, MD USA
{mzuzak, ytliu, ankurs}@umd.edu

2021 58th ACM/IEEE Design Automation Conference (DAC) | 978-1-6654-3274-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/DAC18074.2021.9586179

***Abstract*—Logic locking has been proposed to counter security threats during IC fabrication. Such an approach restricts unauthorized use by injecting sufficient module level error to derail application level IC functionality. However, recent research has identified a trade-off between the error rate of logic locking and its resilience to a Boolean satisfiablity (SAT) attack. As a result, logic locking often cannot inject sufficient error to impact an IC while maintaining SAT resilience. In this work, we propose using architectural context available during resource binding to co-design architectures and locking configurations capable of high corruption *and* SAT resilience simultaneously. To do so, we propose 2 security-focused binding/locking algorithms and apply them to bind/lock 11 MediaBench benchmarks. The resulting circuits showed a 26x and 99x increase in the application errors of a fixed locking configuration while maintaining SAT resilience and incurring minimal overhead compared to other binding schemes. Locking applied post-binding could not achieve a high application error rate and SAT resilience simultaneously.**

I. Introduction

Due to the rising cost and complexity of high-end integrated circuit (IC) fabrication, IC design companies are increasingly reliant on untrusted manufacturing facilities. In order to fabricate an IC, these untrusted facilities are given GDSII files for the design. These files can be deciphered to uncover intricate design details leading to concerns such as reverse engineering, piracy, and overproduction [1].

Logic locking (also known as logic obfuscation) secures design details from an untrusted foundry by incorporating a locking key into a design that hides the functional and structural details of the circuit [2]–[9]. This locking key is then withheld from untrusted facilities. For any wrong key, locking structures produce errant output for a deterministic set of input minterms, called *locked inputs*. Whenever these locked inputs are applied to a locked circuit, error is injected into the design. Most logic locking work targets combinational circuits in a design with the goal of injecting sufficient error in these locked modules to critically impact an IC at the application level, rendering it unusable. See [1] for a survey of logic locking research.

For combinational locking, one of the most formidable attacks on locked circuits is the SAT attack [10], [11]. Recent research has suggested that logic locking often cannot induce a sufficient number of errors to critically impact a locked IC while maintaining resilience to a SAT attack [12], [13]. This is due to a fundamental trade-off between the number of locked inputs and SAT attack resilience that exists underlying all combinational logic locking schemes [2], [14]. This trade-off requires that locking corrupt only a small number of locked inputs to be SAT resilient. A small number of locked inputs causes only a small number of errors, which are often inadequate to overcome the error resilience of ICs [15]. This creates a dilemma. High SAT resilience requires low combinational module corruption, however, we also need high application corruption for wrong keys. To overcome it, an architectural view of locking is needed.

This leads to our primary goal: to use architectural knowledge available during the resource binding phase of high-level synthesis (HLS) to lock an overall IC against an untrusted foundry. As we show, by using security-aware binding algorithms, we can achieve both objectives of high application corruption *and* SAT resilience.

A. Related Work

Relatively few prior works have explored HLS in the context of logic locking [16]–[18]. The TAO technique [16] suggested transformations to obfuscate a design during HLS. However, TAO assumes a restrictive attacker model where the adversary cannot have access to a working chip. This limits the use of TAO to a small subset of logic locking use cases, where the IC is never distributed beyond the design house (e.g. government fabrication of military ICs). Doing so restricts the use of the SAT attack [10], [11], which quickly unlocks the high-error locking used by TAO [10], limiting its utility.

SFLL-HLS proposes an extra HLS step to identify IC modules with a sufficient number of inputs to support locking [17]. Then, based on simulations of the RTL design, they tune the size of their locking configuration to ensure security. While this approach occurs during HLS, it does not directly integrate with HLS algorithms to inform either the design's RTL, or the configuration of logic locking to improve security. Rather, it serves as an argument that supports architectural consideration for logic locking.

DECOY presents a tighter integration with HLS, however, it still does not integrate into any phase/algorithm of HLS [18]. Instead, DECOY adds an HLS step to partition the design into critical and non-critical IP. Critical IP is then implemented in a separate eFPGA, with non-critical IP implemented in an ASIC. As a result, the eFPGA, which exhibits strong reverse-engineering protection, obfuscates the critical IP. While this yields security, it also introduces substantial design complexity and overhead. Such an approach is often untenable.

While both SFLL-HLS and DECOY recognize the importance of HLS's context, they fail to capitalize on the RT-level design decisions made during this phase. Instead, they rely on standard HLS algorithms that optimize for parameters such as switching activity [19], or register re-use [20]. This is a missed opportunity as these algorithms can make RT-level design decisions to optimize and inform supply-chain security instead, as shown in this work.

B. Contributions

We propose security-aware resource binding. To do so, we propose 2 problem formulations, one where locked inputs are chosen prior to binding (obfuscation-informed binding) and one where the binding algorithm can inform locked input selection (binding-obfuscation co-design). Our contributions for each problem formulation are:

Obfuscation-Informed Binding:

1) A cost function to inform resource binding that maximizes the application errors caused by locking with specified locked inputs.
2) A graph theoretic binding algorithm that optimally maps operations to locked modules to maximize application error in P-time.

Binding-Obfuscation Co-Design:

1) A graph theoretic algorithm, occurring during HLS, to concurrently bind *and* choose locked inputs to maximize locking-induced application errors. We first develop an optimal algorithm with a non-polynomial runtime and then present a P-time heuristic.
2) A design methodology that combines exponential SAT-runtime locking schemes with our co-design algorithm to configure locking to meet arbitrary application error/SAT resilience goals.

To evaluate each algorithm, we applied them to 11 MediaBench benchmarks [21]. Obfuscation-informed binding (binding-obfuscation co-design) caused a 26x (99x) increase in the application errors caused by locking while maintaining SAT resilience and incurring an increase of only 4.7 in register count and 0.03 in switching rate compared to area and power aware binding. Conventional locking approaches that are applied to gate-level designs could not achieve both a high application error rate *and* SAT resilience simultaneously.

II. Preliminaries

A. Logic Locking

Logic locking schemes are often evaluated by 2 characteristics: 1) **error severity** and 2) **attack resilience**. Error severity is the ability of

This work was supported by the National Science Foundation Grant 1953285 and the ARCS Foundation.

logic locking to cause critical failures for wrong keys, often quantified by the number of *locked inputs* (i.e. error producing inputs for a wrong key). Attack resilience is the ability of logic locking to resist attacks. For a prevalent attack against logic locking, known as the SAT attack [10], [11], an inverse relationship between these goals has been identified [2], [14]. The work in [2] defines this relationship as:

$$\lambda = \left\lceil \log\left(\frac{2^{|k|} - c - \epsilon(2^{|k|} - c)}{\epsilon(2^{|k|} - c)(2^{|k|} - c - 1)}\right) \Big/ \log\left(\frac{2^{|k|} - c - \epsilon(2^{|k|} - c)}{2^{|k|} - c - 1}\right)\right\rceil \quad (1)$$

where λ is the expected SAT attack iterations, $|k|$ is the key length in bits, c is the # of correct keys, and ϵ is the ratio of locked inputs to total inputs. If we assume that the locking scheme, the length of the locking key, and the number of primary inputs are fixed, Eqn. 1 shows a direct, inverse relationship between the number of locked inputs and the expected SAT iterations to unlock a circuit. Note that this result is unique per module in the design because the SAT attack model assumes access to an IC's scan chain (i.e. its intermediate registers). Thus, SAT resilience is calculated separately for each locked module in the design. Based on this trade-off, recent research has suggested that logic locking often cannot lock enough inputs to induce a sufficient number of errors to critically impact an IC, while maintaining SAT resilience [12], [13]. Thus, locking is stuck in a dilemma between high corruption and high SAT resilience.

In this work, we focus on 2 prominent families of logic locking, denoted 1) critical minterm locking and 2) exponential SAT iteration runtime locking. Critical minterm locking schemes include techniques such as Stripped Functionality Logic Locking (SFLL) [3]–[5] and Strong Anti-SAT [6]. These techniques allow an IC designer to select specific *critical input minterms* in a locked module and force them to produce errant output for a large subset of wrong keys. Such techniques usually guarantee that the expected number of SAT iterations scales exponentially in key length. Exponential SAT iteration runtime locking schemes include techniques such as Full-Lock [7], LoPher [8], and Cross-Lock [9]. These techniques cause the runtime of successive SAT attack iterations to increase exponentially.

B. High-Level Synthesis (HLS)

HLS is an automated design process that converts a behavioral description of a digital system into an RT-level design. HLS consists of 3 steps: allocation, scheduling, and binding. Allocation determines the type and number of resources necessary to implement a design. After this step, a list of functional units (FUs) (e.g. adders, memories, etc.) to implement a design is created. Scheduling partitions a design into control steps, called operations, which can be completed in one clock cycle. After this step, a schedule, usually represented as a scheduled data flow graph (DFG), is created. In the DFG, nodes are operations that must be completed and edges are dependencies between operations. Binding maps (or *binds*) each operation to an FU allocated during the allocation phase. Common binding approaches target 1) minimizing required registers/multiplexers [20] and 2) minimizing switching activity [19], [22]. During this step, knowledge of the ICs input space is assumed to be known. This allows the power ramifications of binding decisions to be evaluated [19], [22].

III. Motivational Example: Security-Aware Binding

Given the findings of prior research, outlined in Sec. I, there is a strong need to think *beyond the module* when obfuscating ICs. If we follow the conventional wisdom of pursuing module level locking after the gate level configuration of modules are fixed we get stuck in the dilemma of choosing high corruption versus high SAT resilience. Achieving both objectives is critical. In this work, we show that through "smart" obfuscation-aware resource binding decisions, we can indeed satisfy both of these competing needs.

In the context of design obfuscation, binding impacts security by mapping operations onto functional units (FU). This mapping determines the types of values (input minterms) typically processed on each FU. Because locking corrupts output for specific locked inputs, this greatly impacts the lock-ability of an IC. Let us consider an example to show the utility of a security-aware binding approach.

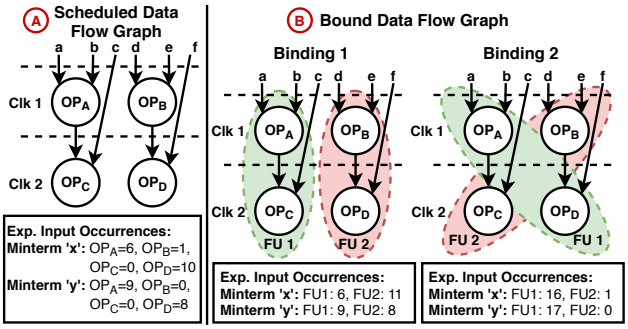

Fig. 1: Sample scheduled DFG and corresponding binding solutions.

A. Motivational Example: Overview

Consider the scheduled data flow graph (DFG) in Fig. 1A. This DFG represents some behavioral code segment where each node is an operation that must be applied to the data and each edge shows the flow of data between operations. The DFG in the figure requires 2 cycles to execute (clk 1 and 2). During the first (second) cycle, 2 operations, OP_A and OP_B (OP_C and OP_D), must be completed. Without the loss of generality, let us assume that each operation in the figure is an add. To implement this DFG in hardware, 2 adder FUs are necessary to execute the 2 concurrent add operations.

Resource binding specifies the mapping of the 4 add operations onto the 2 allocated adder FUs. In Fig. 1B, we show the 2 possible bindings for the scheduled DFG. For each binding, the green (red) circled operations are mapped to FU 1 (FU 2). Let us assume that a security-oblivious binding algorithm has selected binding 1 for the design. After binding, a designer has decided to lock FU 1 to protect the design. In this case, the best solution would be to lock a large majority of input minterms to ensure the highest corruption at the application level. However, due to the SAT resilience constraint, let us assume we can only lock a single input minterm, randomly selected to be x. If we are aware of the input distribution for each operation, a common assumption for HLS [19], [22], we can determine the expected number of occurrences of arbitrary input minterms for each operation in the DFG during a typical workload. We have aggregated the expected number of times input minterm x and y are applied to each operation during a typical workload at the bottom of Fig. 1A.

Because we know the expected occurrences of input x for each operation, we can estimate the number of locked inputs applied to our locked adder (FU 1). This is the number of application errors caused by the locking scheme. Notice that FU 1 in binding 1 executes OP_A, which is expected to operate on input x 6 times, and OP_C, which is expected to operate on input x 0 times. This means that the bound/locked circuit is expected to inject $6 + 0 = 6$ errors. We have aggregated the expected occurrences of minterm x and y for each FU-binding combination below Fig. 1B. Let us consider how security-aware binding could increase the number of error injections.

B. Example 1: Obfuscation-Aware Binding

Consider the case where the logic locking configuration has been specified prior to resource binding. Following our prior example, this means that FU 1 will lock the input minterm x. However, instead of binding in a security-oblivious fashion, let us instead bind the DFG in Fig. 1A to maximize the number of times the locked input (x) will be applied to the locked FU (FU 1). In this case, binding 2 would be selected, resulting in $6 + 10 = 16$ application errors over a typical workload. Such an approach has 2 key advantages. 1) The number of errors injected by logic locking is more than doubled (16 vs. 6) compared to our security-oblivious binding approach. Because the number of locked inputs is static, this results in a substantial increase in the corruption caused by logic locking *without* compromising SAT resilience. 2) Errors are now injected during both clock cycles of the schedule (clk 1 and 2) instead of only one (clk 1). This opportunity for consecutive error injections increases the likelihood of critically impacting the application. By binding to maximize the

application errors, a locking configuration that simultaneously causes substantially more *and* higher quality application errors is produced.

C. Example 2: Binding-Obfuscation Co-Design

In addition to selecting which operations are bound to each FU, let us also decide which input minterms to lock. Previously, we assumed that only input x could be locked. If we simultaneously consider the locked inputs and the binding in order to maximize the number of application errors, we will lock input y in FU 1 for binding 2. Notice that input y does not have either the highest total number of expected occurrences, or the most occurrences for a single operation. However, this locking configuration causes $9+8 = 17$ application errors during a typical workload. Not only does this 1) increase the number of error injections by over 2x (17 vs. 6) compared to our security-oblivious approach and 2) inject error during both cycles of the schedule, but it also results in more errors than any configuration locking input x could achieve. Thus, a co-design approach can further improve the number and quality of application errors caused by locking.

D. Security-Aware Binding Problem Formulations

In both examples, the architectural context available during resource binding enabled us to create locked circuits causing over 2x more application errors. This allows a designer to decrease the number of locked inputs, while simultaneously increasing the application errors caused by the locking construction. Essentially, *security-aware* binding decisions enable us to achieve higher attack resilience *and* higher corruption simultaneously. Based on each example, we specify a problem formulation that is addressed in the remainder of the work.

1) **Obfuscation-Aware Binding:** For this problem, we assume that modules have already been locked to secure a known set of error-causing locked inputs. Based on this configuration, we map operations onto FUs (bind) to maximize application errors.
2) **Binding-Obfuscation Co-Design:** For this problem, we simultaneously select the binding and the locked input minterms to maximize the application errors caused by locking.

IV. Problem 1: Obfuscation-Aware Binding

The *obfuscation-aware binding* problem assumes that the allocation/scheduling phases of HLS have occurred and a SAT-resilient locking configuration (i.e. one that locks a sufficiently small number of inputs) has been specified for the allocated FUs. The locking specification must include 1) the number of FUs locked, 2) the locking scheme used, and 3) the locked inputs. We also assume that critical minterm locking schemes, such as SFLL-rem [5], have been used so that locked inputs are static between wrong keys. Now, given a list of FUs, a scheduled DFG, and locking details, we must map each operation to an FU such that the application errors caused by the locking construction are maximized. Doing so ensures that IC corruption is maximized while maintaining SAT resilience (because the locking construction was chosen to be SAT resilient a priori). To address this, we have developed an objective cost function to quantify the application errors caused by locking for a fixed binding.

A. Obfuscation-Aware Objective Cost Function

Suppose that we have scheduled and bound a DFG onto FUs, some of which have been locked using critical minterm locking. We aim to quantify the impact of these locked FUs on the error of the DFG. We capture this error by counting the number of times a locked input is evaluated by a locked FU during the DFG's execution. The objective is to maximize these error injecting events through appropriate binding decisions. Let us define matrix K to represent the occurrence of each locked input for each operation. The number of times the locked input m is applied for operation n is $K_{m,n}$. One way to calculate K for a given DFG is to simulate the execution of the DFG for "typical" input traces, or applications. These are commonly assumed to be available during HLS [19], [22]. Given an input trace for the DFG, we can perform time simulation to calculate the number of times a given locked input is applied to each operation.

Based on K, we define an objective cost function to inform binding that quantifies the expected number of application errors for a given

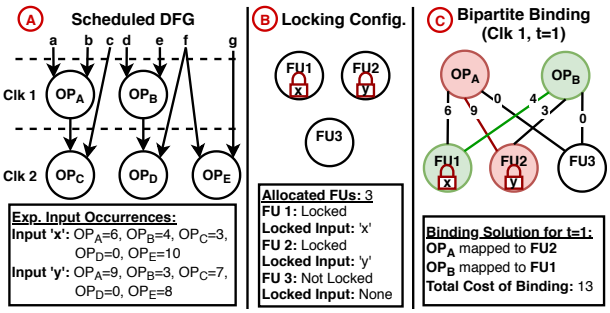

Fig. 2: Obfuscation-aware binding algorithm for clock 1 (t=1).

locking configuration in a bound DFG. To do so, assume that some set of L FUs have been locked. Each locked FU, $l \in L$, locks a set of inputs M_l and binds a set of operations N_l. The expected number of application errors caused by this locking configuration is:

$$E = \sum_{l \in L} \sum_{m \in M_l} \sum_{n \in N_l} K_{m,n} \tag{2}$$

B. Obfuscation-Aware Binding Algorithm

Using the cost function in Eqn. 2, we develop a binding algorithm that maps operations to FUs such that the number of application errors (i.e. when locked inputs are applied to locked FUs) is maximized. Consider a scheduled DFG, S, which spans s clock cycles. A set of resources, R, has been allocated to bind the DFG. While we make no assumptions as to the type (e.g. adder, multiplier, etc.) of the resources and operations, we do assume that they are all the same type. Thus, any of the resources in R can execute any operation in the DFG. By handling each operation/resource type separately, this assumption can be made without the loss of generality. Of these R resources, a subset, L, has been locked ($L \subseteq R$). Each $l \in L$ locks a set of critical inputs M_l, which are pre-determined.

During each cycle t ($t \leq s$), a set of concurrent operations $N_t \in S$ are scheduled. Binding requires us to map each operation in N_t to one of the allocated FUs (i.e. $|R| \geq |N_t|$). Consider the first cycle of the DFG, $t = 1$. To bind the operations at $t = 1$ (N_1), we build a weighted bipartite graph, $B_1 = (R \cup N_1, E_1)$. Each vertex $r_i \in R$ is an FU. Each vertex $n_j \in N_1$ is an operation. If r_i can bind n_j (i.e. the FU r_i is available and can run operation n_j), an edge of weight $w_{i,j}$ is added. This should be the case for all r_i-n_j pairs, so a complete bipartite graph is produced. The weight, $w_{i,j}$, is:

$$w_{i,j} = \sum_{m \in M_i} K_{m,j} \tag{3}$$

where M_i is the set of locked inputs for FU i and $K_{m,j}$ is the expected occurrences of locked input $m \in M_i$ for operation j. Therefore, $w_{i,j}$ is the number of times locked inputs will be applied to resource i if operation j is bound to it. Note that the edge weights connected to non-locked FUs will be 0. Now, we solve the max weight bipartite matching problem for B_1, which can be solved optimally in P-time. The resulting matching maps (binds) each operation during clock $t = 1$ to an available FU.

To demonstrate this algorithm, consider the DFG in Fig. 2A, which spans 2 clocks. There are 3 FUs, $R = \{FU1, FU2, FU3\}$, allocated to bind this DFG, shown in Fig. 2B. Of these FUs, 2 are locked, $L = \{FU1, FU2\}$, with locked inputs $M_{FU1} = \{x\}$ and $M_{FU2} = \{y\}$. For this DFG's typical input trace, the number of times each locked input (x and y) was applied to each operation is below Fig. 2A. For $t = 1$, the proposed algorithm produces the bipartite graph in Fig. 2C. A max weight matching of this graph selects the red and green colored edges, mapping OP_A to $FU2$, with edge weight 9, and OP_B to $FU1$, with edge weight 4. FU3 is unused during this clock because only 2 operations are executed. This produces a binding for clock 1 that injects $9 + 4 = 13$ errors for the typical input trace.

The described approach produces a binding for clock $t = 1$. This algorithm must be repeated for the remaining $s-1$ clocks to produce

a complete binding solution. Thus, we must generate and match a bipartite graph, B_t, for the remaining $t = 2..s$ clocks in the schedule. Notice that the considered operations change for each cycle (t), but the FUs in R do not. Also, the bipartite graph for each cycle (B_t) has no dependence on other cycles. Thus, binding decisions made in one cycle do not conflict with another cycle, allowing each clock cycle to be bound independently and in any order (separability).

By matching each set of concurrent operations to allocated resources, we bind each operation to maximize the number of locked inputs applied to locked FUs during the typical input trace/application. This maximizes the application errors caused by the locking configuration for the characteristic workload, as proved in Thm. 2.

C. Analysis of Obfuscation-Aware Binding Algorithm

To analyze the presented algorithm, we discuss 3 key properties.

1) *Runtime Complexity:* To bind an arbitrary scheduled DFG with s cycles, the proposed algorithm must generate and match s complete weighted bipartite graphs. Each graph has $|N_t|$ operations (sources) that must be matched to one of the $|R|$ resources (destinations) with a maximum weight. A minimum weighted full match of an m-source and n-destination bipartite graph can be performed in $O(mnlog(n))$ [23]. By negating each edge weight ($w_{i,j}$) and assuming that $|N_m|$ is the maximum number of concurrent operations in the DFG, obfuscation-aware binding can be completed in $O(s|N_m||R|log(|R|))$. Thus, the algorithm runs in P-time.

2) *Validity and Completeness of Binding Solution:*

Theorem 1. *The proposed obfuscation-aware binding algorithm will always result in a valid and complete binding solution, if it exists.*

We omit a detailed proof of this claim for brevity. However, notice that during each clock cycle, bipartite matching ensures a valid matching between operations and FUs. By definition, this means that all operations in all clocks end up being bound to only one FU, with no more than one operation in a cycle being bound to a given FU. This ensures that the final solution is a valid and complete binding.

3) *Optimality of Binding Solution:*

Theorem 2. *The obfuscation-aware binding algorithm yields the maximum expected application errors for a locking configuration.*

Proof. To bind a DFG, a bipartite graph must be generated and fully matched for each cycle in the schedule ($t = 1..s$). Each graph has a source node for every operation $n \in N_t$ and a destination node for each resource in R. Every source-destination pair is connected by an edge of weight $w_{i,j}$, which is equal to the number of occurrences of each locked input for FU j during operation i. A bipartite graph defined in this way for cycle t ($1 \leq t \leq s$) is necessarily independent of the bipartite graph for all other cycles. This implies that the full matching produced for each bipartite graph is independent. Therefore, the binding for each cycle in the schedule is separable.

Now, consider that each edge weight in the bipartite graph is equal to the number of occurrences of each locked input for FU j during operation i (i.e. expected error injections). By definition, a maximum weight full matching of this bipartite graph corresponds to the operation-FU mapping (binding) that causes the most expected error injections. Hence, the full matching for each bipartite graph is optimal for a given cycle in the DFG. Because each bipartite matching produces the maximum error injections for that cycle and the bipartite graph for each cycle is separable, the total binding solution yields the maximum expected error injections for the locking scheme. □

Thus, the algorithm results in a binding with the highest possible application corruption. Remember that the locking scheme specified prior to binding ensured SAT resilience by limiting locked inputs to be sufficiently small in number. Therefore, our obfuscation-aware binding algorithm guarantees a locking scheme with the highest application corruption, while ensuring that SAT resilience is maintained.

V. Problem 2: Binding-Obfuscation Co-Design

The *binding-obfuscation co-design* problem relaxes our assumption that the identity of locked inputs are specified before binding. Instead, we assume only that the number of locked inputs are specified to ensure a SAT resilient locking solution. These locked inputs are to be chosen from some set of designer-specified candidate locked inputs to optimize application error. To formalize this, we assume that the allocation/scheduling phases of HLS have occurred and a SAT-resilient locking configuration has been specified, including 1) the number of resources locked, 2) the critical minterm locking scheme used, 3) the number of locked inputs, and 4) a list of candidate locked inputs. However, which specific inputs are locked from the candidate list is not known and needs to be chosen. We must map each operation to an FU *and* select locked inputs such that the application errors caused by locking are maximized via the cost function in Eqn. 2.

A. Binding-Informed Obfuscation Algorithm

Consider an arbitrary scheduled DFG, S, which spans s clock cycles. A set of R resources have been allocated to bind S. Once again, we assume without the loss of generality that all operations and resources are of the same type (e.g. add). This can be done by handling each set of dissimilar operations separately. Of these R resources, a subset (L) will be locked ($L \subseteq R$). Each $l \in L$ locks a set of inputs, M_l, which must be chosen from a list of candidates.

We assume that this list of candidate locked inputs, denoted C, is designer-specified. This set can be chosen by a variety of methods (e.g. randomly, most commonly occurring inputs in the DFG, etc.). We discuss strategies to choose C in Sec. V-B, however, we largely consider this to be beyond the scope of the work. For each locked FU ($l \in L$), we must select the most commonly occurring candidate locked inputs among the operations bound to it to be in M_l. In this way, when we lock each input in M_l, we produce a locked FU with the maximum application error. While the exact input distribution for each operation varies among workloads, we have applied a "typical" input trace to our DFG to estimate the number of occurrences of each input i for operation j, denoted $K_{i,j}$ (Sec. IV-A).

Given a list of candidate locked inputs (C), we must find the binding/locked input specification that produces the most expected application errors for the DFG. Sec. IV-B defines an obfuscation-aware binding algorithm that, given a specified set of locked inputs (M_l), returns the binding with the most expected application errors. If we enumerate all combinations of candidate locked inputs from C of size $|M_l|$ for each locked FU ($l \in L$) and apply this obfuscation-informed binding algorithm to each combination, we generate the optimal binding for each enumerated set of locked inputs. By comparing the total expected application errors for each enumerated set (i.e. the total cost of each binding solution), we can determine the optimal binding/locked input specification for the DFG.

This approach iterates over all locked input combinations, an exponential number. However, many of these combinations are unlikely to yield an optimal solution. For example, consider a set of FUs locking a set of inputs that produce minimal application error. It is unlikely that FUs locking these inputs can be combined into a high error binding solution. However, this algorithm still evaluates them. A good heuristic would focus on locked input combinations causing substantial error for each FU, regardless of how other FUs are locked, to ensure that the total error is very high. This can be achieved by evaluating each FU sequentially. If we assume that the number of candidate locked inputs (C) is upper-bounded by a predefined constant (x), we can define a P-time heuristic:

1) Choose a single FU, $l_i \in L$. Assume all other FUs are unlocked.
2) Enumerate all locked input combinations $\binom{|C|}{|M_{l_i}|}$ for l_i.
3) Apply obfuscation-informed binding on each combination. Use the max cost binding solution to fix the locked inputs for l_i, M_{l_i}.
4) Consider a new locked FU, $l_i \in L$ for which M_{l_i} has not been fixed. With all prior M_l fixed, repeat steps 2-3 to specify M_{l_i}.
5) Repeat step 4 until M_l is chosen for each $l \in L$. Run obfuscation-informed binding once more for the final binding solution.

B. Analysis of Binding-Obfuscation Co-Design

To analyze the proposed algorithms, we discuss 4 key properties.

1) *Candidate Locked Input Selection:* We consider the nuances of candidate locked input selection to be out of scope. However, we

briefly note some possible ways to choose the members of C for context. The most obvious relies on the "typical" input trace to select the most common inputs in the DFG (i.e. the top 'x' most common inputs). However, if the attacker has input distribution knowledge, such an approach could leak candidate locked inputs, making it disadvantageous. In this case, less common inputs, or even a random set can be used. Regardless of the members of C, our approach still maximizes locking-induced application errors. Thus, binding-locking co-design will still increase application error over conventional locking approaches considering the same locked inputs.

2) *Runtime Complexity:* The optimal algorithm iterates over all locked input combinations for each locked FU. Given $|L|$ locked FUs that secure at most $|M|$ inputs from a set of size $|C|$, there are $\binom{|C|}{|M|}^{|L|}$ combinations. This results in a non-polynomial runtime. However, consider the proposed heuristic, which sequentially iterates over locked inputs combinations for one FU at a time. Because $|C|$ is upper-bounded by a predefined constant 'x' for this heuristic, there are $\binom{x}{|M|}$ combinations per FU. This is upper-bounded by $x^{x/2}$. x is a constant, so this upper bound is a constant, allowing it to be discarded from the time complexity. If we use the P-time algorithm from Sec. IV-B to evaluate each locked input combination for $|L|$ FUs, our heuristic runs in $O(s|L||N_m||R|log(|R|))$, a P-time solution.

3) *Optimality of Binding Solution:* The resulting binding/locking will yield the maximum possible application errors, as quantified by the cost function in Eqn. 2. To prove this, remember Thm. 2, which proves that our binding algorithm maximizes errors when locked inputs are specified. Because we iterate over all possible locked input combinations in C, the resulting solution must cause the maximum application errors possible for the DFG given the locking parameters. On the other hand, while our P-time heuristic will still yield a locking solution with substantial error, it no longer operates on every locked input combination, so it may not be highest possible error.

4) *Impact on Security:* Consider the impact of co-design on the application errors and SAT resilience of locking. Both our optimal and heuristic algorithm configure binding/locking to optimize locking-induced application errors for a fixed number of locked inputs. Because the locked input count dictates the SAT resilience of locking, our proposed approach produces a design that maximizes corruption for an attacker without any compromise in SAT resilience.

C. Binding-Time Logic Locking Design Methodology

Consider a designer that has set a target application error rate and a minimum SAT runtime permissible for a secure locking configuration in their custom IC. With only minor modifications, the proposed binding-locking co-design approach can be used to design a locking configuration meeting both goals. Essentially, by using our co-design approach to incrementally tune the number of locked inputs in each FU, a locking configuration can be designed that achieves a sufficient application error rate with the minimum number of locked inputs, hence, the maximum SAT resilience. If the SAT resilience of this locking configuration is insufficient, exponential SAT iteration runtime locking schemes can be used alongside the binding-obfuscation co-designed locking to increase SAT runtime to a sufficient level. Exponential SAT iteration runtime schemes generally incur too much design overhead to be used on their own. For example, a 384-bit Full-Lock [7] scheme implemented in the b14 netlist of the ISCAS'85 suite incurred a 192% increase in power and 61% increase in area, while requiring < 10 minutes to unlock with a SAT attack. This overhead is infeasible. However, our co-design approach uses critical minterm locking, which incurs minimal overhead compared to these techniques. So, by using low-overhead critical minterm locking to achieve as much SAT resilience as possible, the design overhead concerns associated with exponential SAT runtime schemes can be minimized, while still meeting design goals.

VI. Experimental Evaluation of Proposed Algorithms

To evaluate each proposed algorithm, we applied them to bind the adders and multipliers in 11 benchmarks. Each benchmark was made by isolating a C function from 1 of 8 MediaBench benchmarks [21] and extracting the corresponding DFG with SUIF. Each DFG was scheduled to be executed on up to 3 FUs using a path-based scheduler [24]. The resulting DFGs contained an average of 18.6 add and 10.6 multiply operations spanning 13.5 cycles. To serve as the "typical" input trace/application for each benchmark, we used the MediaBench-provided sample workloads. For this input trace, each DFG was simulated and expected occurrences of input minterms for internal operations were computed. This information, along with the scheduled DFG, was used as input for each proposed algorithm. An overview of the process to generate each benchmark is in Fig. 3.

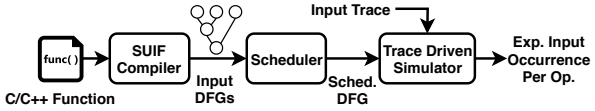

Fig. 3: Experimental flow to generate benchmarks.

To evaluate each algorithm we compared them to other common binding algorithms with identical locking configurations. Specifically, we used an area-aware approach [20], which minimizes register count, and a power-aware approach [19], which minimizes switching frequency, for comparison. For each benchmark, we enumerated all combinations of {1,2,3} locked FUs locking {1,2,3} inputs each. We then aggregated a list of the 10 most common inputs for each DFG to serve as the candidate locked inputs. For the 9 possible locking configurations (i.e. {1,2,3} locked FUs locking {1,2,3} inputs), we created a bound/locked circuit securing each combination of the 10 candidate inputs for each locked FU. We used 1) obfuscation-aware, 2) binding-obfuscation co-design (optimal and P-time heuristic), 3) area-aware, and 4) power-aware binding algorithms to generate each locked circuit. We then calculated the ratio between the number of application errors caused by each security-aware approach compared to each area/power-aware approach with the same locking configuration. The results were averaged over every locked FU count, locked input count, and locked input combination for Fig. 4.

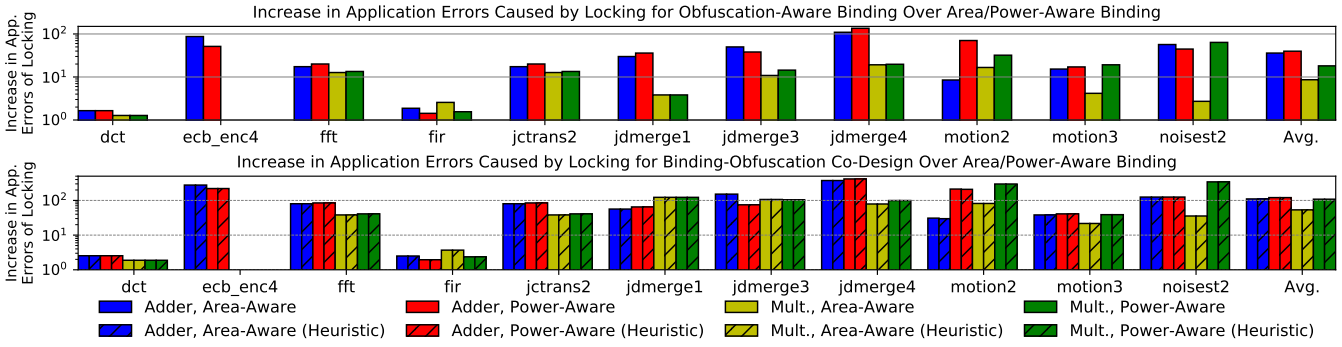

Fig. 4: Impact of security-aware binding on the application errors caused by locking during a typical workload compared to area-aware [20] and power-aware [19] binding. Adder/multiplier binding were considered separately. No multipliers were present in the ecb_enc4 benchmark.

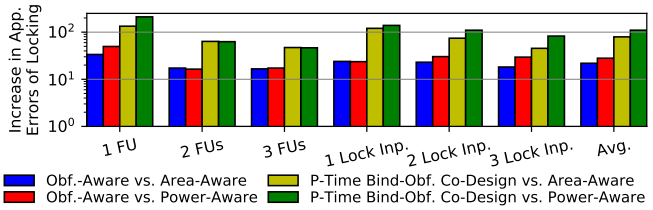

Fig. 5: Impact of locking configuration on errors caused by security-aware binding. All results are normalized to the errors caused by the same locking configuration applied after area/power-aware binding.

In this way, we compared each circuit created with a security-aware algorithm for each enumerated locking/locked input configuration to the same circuit incorporating an *identical* locking configuration created with an area/power-aware algorithm. This directly quantifies any increase in application errors due to our security-aware algorithms across a variety of circuits and locking configurations.

A. Experimental Analysis

As shown by Fig. 4, obfuscation-aware binding increased the application errors caused by the locking construction by 22x and 29x compared to area and power aware binding, respectively. The optimal binding-obfuscation co-design algorithm increased application errors by 82x and 115x. Our P-time heuristic for this algorithm resulted less than a 0.5% solution degradation, again increasing application errors by 82x and 115x. This confirms the efficacy of our heuristic. As a result, we rely on this P-time heuristic for the remainder of binding-obfuscation co-design analysis. Each algorithm caused sizable increases in application errors, without sacrificing SAT resilience.

We have aggregated the impact of locking configuration on the efficacy of each proposed algorithm in Fig. 5. To generate Fig. 5, we fixed a single locking parameter, listed on the x-axis, and averaged our results over all other locking parameters (e.g. the "1 FU" bars average over locking with {1,2,3} locked inputs). In this way, we isolated the impact of each locking parameter on the efficacy of each security-aware algorithm. Based on Fig. 5, increases in application error remained consistently high in all cases. Remember, all increases were normalized to the application errors caused by area/power-aware binding for the same locking configuration (i.e. locked FU count, locked input count, and locked input identity). Thus, Fig. 5 suggests that security-aware binding will consistently produce a 10-150x increase in errors, no matter the underlying locking construction.

Finally, we compared the design overhead of each proposed algorithm to 1) area-aware binding [20], which minimizes register count, and 2) power-aware binding [19], which minimizes switching frequency. We show the corresponding increases incurred by our security-aware algorithms on the register count and the switching rate in Fig. 6. On average, our proposed algorithms performed similarly, requiring ~4.7 more register count than area-aware binding and incurring a 0.03 higher switching rate than power-aware binding. This confirms the low overhead nature of each security-aware algorithm.

VII. Conclusion

In this work, we explored security-aware binding for HLS with 2 problem formulations. To solve them, we developed an objective cost function that quantified the application errors injected by a locking configuration for a fixed binding. We then proposed a security-aware algorithm for both problems to maximize this cost function without degrading SAT resilience. To evaluate each algorithm, we applied them to 11 MediaBench benchmarks and their sample workloads. Our proposed obfuscation-aware binding (binding-obfuscation co-design) algorithm caused a 26x (99x) increase in locking-induced application errors over alternative binding schemes with no reduction in SAT resilience and only minimal degradation in area/power overhead. Thus, each approach can ensure that locking achieves the highest application corruption without sacrificing SAT resilience.

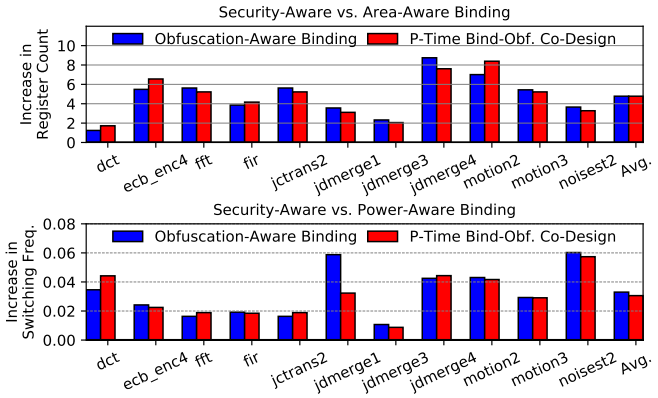

Fig. 6: Design overhead of proposed security-aware binding algorithms compared to area-aware and power-aware binding algorithms.

References

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

[2] 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.

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

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

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

[6] Y. Liu et al., "Strong anti-sat: Secure and effective logic locking." in International Symposium on Quality Electronic Design (ISQED), 2020.

[7] H. M. Kamali et al., "Full-lock: Hard distributions of sat instances for obfuscating circuits using fully configurable logic and routing blocks," in Design Automation Conference (DAC), 2019.

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

[9] K. Shamsi et al., "Cross-lock: Dense layout-level interconnect locking using cross-bar architectures," in Great Lakes Symp. on VLSI, 2018.

[10] P. Subramanyan et al., "Evaluating the security of logic encryption algorithms," in International Symposium on Hardware Oriented Security and Trust (HOST). IEEE, 2015.

[11] 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.

[12] K. Shamsi et al., "On the impossibility of approximation-resilient circuit locking," in Intl. Symp. on Hardware Oriented Security and Trust, 2019.

[13] M. Zuzak et al., "Obfusgem: Enhancing processor design obfuscation through security-aware on-chip memory and data path design," in International Symposium on Memory Systems. ACM, 2020.

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

[15] X. Li et al., "Application-level correctness and its impact on fault tolerance," in International Symposium on High Performance Computer Architecture. IEEE, 2007, pp. 181–192.

[16] C. Pilato et al., "Tao: techniques for algorithm-level obfuscation during high-level synthesis," in Design Automation Conference, 2018.

[17] M. Yasin et al., "Sfll-hls: Stripped-functionality logic locking meets high-level synthesis," in Intl. Conf. on Computer-Aided Design, 2019.

[18] J. Chen et al., "Decoy: Deflection-driven hls-based computation partitioning for obfuscating intellectual property," in Design Automation Conference (DAC). IEEE, 2020.

[19] J.-M. Chang et al., "Register allocation and binding for low power," in ACM/IEEE Design Automation Conference (DAC), 1995.

[20] C.-Y. Huang et al., "Data path allocation based on bipartite weighted matching," in Design Automation Conference, 1991.

[21] C. Lee et al., "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in International Symposium on Microarchitecture. IEEE, 1997.

[22] A. Stammermann et al., "Binding allocation and floorplanning in low power high-level synthesis," in International Conference on Computer Aided Design. IEEE, 2003.

[23] R. M. Karp, "An algorithm to solve the m× n assignment problem in expected time o (mn log n)," Networks, 1980.

[24] S. O. Memik et al., "A super-scheduler for embedded reconfigurable systems," in International Conference on Computer Aided Design, 2001.