Security-Aware Resource Binding to Enhance Logic Obfuscation

Michael Zuzak, *Member, IEEE*, Yuntao Liu, *Member, IEEE*, and Ankur Srivastava, *Fellow, IEEE*

***Abstract*—Logic obfuscation mitigates the unauthorized use of design IP by untrusted partners during IC fabrication. To do so, these techniques produce gate-level errors that derail typical applications run on the IC. Recent research has derived a link between the error rate and the Boolean satisfiability (SAT) attack resilience of logic obfuscation. As a result, it has been shown to be difficult for obfuscation to inject sufficient gate-level error to derail application-level function while maintaining resilience to SAT-style attacks. In this work, we explore use of architectural knowledge during the resource binding phase of high-level synthesis to automate the design of locked architectures capable of high corruption *and* SAT resilience simultaneously. To do so, we bifurcate logic obfuscation schemes into two families based on their error profile: distributed error locking and critical minterm locking. We then develop security-focused binding/locking algorithms for each locking family and use them to bind/lock 11 MediaBench benchmarks. For distributed error locking, our proposed security-aware binding algorithms designed locked circuits capable of corrupting a typical application for 52% more wrong keys than a circuit bound with conventional algorithms. For critical minterm locking, our proposed security-aware binding algorithms designed locked circuits capable of corrupting a typical application for 100% of wrong keys while also exhibiting 26x more application errors than a circuit bound with conventional algorithms. Regardless of locking family, our security-aware algorithms improved corruption without degrading SAT resilience or incurring sizable design overheads to do so. Obfuscation applied post-binding could not achieve high corruption and SAT resilience simultaneously in these benchmarks.**

***Index Terms*—Logic Locking, Logic Obfuscation, Untrusted Foundry, High-Level Synthesis, Resource Binding**

I. Introduction

The cost and complexity of cutting-edge integrated circuit (IC) fabrication has skyrocketed with shrinking technology nodes. This has driven IC design companies to adopt a fabless business model, whereby untrusted third-parties are used for IC fabrication. During fabrication, fabless designers must provide these untrusted facilities with a full layout for the design. This layout contains critical design details, driving security and privacy concerns including piracy and overproduction [1].

Logic locking (also called logic obfuscation) was proposed to address security concerns during untrusted fabrication by rendering the functionality of a design dependent on a secret *locking key* [2]–[4]. Whenever an incorrect locking key is applied to a design, there exists a deterministic set of input patterns which produce incorrect output, thereby injecting error in the design. These inputs are called *locked inputs*. By withholding the correct locking key from any untrusted fabrication partners, an IC designer can prevent unauthorized use of the design. Fundamentally, the goal of logic locking is to inject sufficient error for any wrong key to derail any unauthorized use of the design IP.

In response to logic locking, a Boolean satisfiability (SAT) attack was developed [5], [6]. This attack has proved be be quite potent. In fact, recent work has shown that logic obfuscation is often unable to induce enough error to critically impact an IC while maintaining resilience to SAT-style attacks [7], [8]. This challenge stems from a trade-off underlying combinational logic locking, regardless of construction, between the number of locked inputs per wrong key and SAT attack resilience [9]–[11]. This trade-off requires that locking protect only a small number of locked inputs per wrong key to be SAT resilient. However, because the input space of most modules is only partially utilized, the probability that an arbitrary locked input will be applied to a locked module during normal operation has been shown to be quite low in practice [8]. If no locked inputs are ever applied, logic locking provides no protection against the unauthorized use of design IP, mitigating any security guarantees. This creates a dilemma. High SAT resilience requires few locked inputs, however, we must guarantee application corruption for wrong keys using this small set of locked inputs. To overcome this dilemma, we must consider the architecture of an IC.

This leads to the key theme of the work: using the architectural plasticity and context available during the resource binding phase of high-level synthesis (HLS) to enhance logic locking and secure an IC, as a whole, against an untrusted foundry. As we show, security-aware binding can enhance the security of pre-specified logic locking configurations, enabling both corruption *and* SAT resilience to be achieved.

A. Related Work

Many prior works have explored obfuscation in the context of HLS or higher-level design processes [12]–[20]. These works can be broadly divided into two research thrusts. The first thrust we characterize by its reliance on *restricted scan-chain access*, such as TAO [17], ASSURE [15], and others [12]. These works propose methodologies to obfuscate a design during HLS. However, they assume a restrictive attacker model where the adversary cannot access a working chip with scan-chain access. This more restrictive attacker model prevents the use of SAT-style attacks [5], [6], which would otherwise unlock the high-error locking used by these schemes [21]. As a result, these techniques are more limited

This work was supported by the National Science Foundation under grants 1953285 and 2245573.

M. Zuzak is with the Department of Computer Engineering at Rochester Institute of Technology, Rochester, NY, USA. Email: mjzeec@rit.edu

Y. Liu and A. Srivastava are with the Department of Electrical and Computer Engineering at University of Maryland, College Park, MD, USA.

in utility than alternatives that promise security against an oracle-equipped attacker [13], [14], [18], [19]. The work in [16] considers how a scan-chain could be locked down during normal chip operation to restrict oracle access, but this requires the chip to be tested using incorrect *dummy keys*. Hence, the chip is never validated with its intended function.

The second research thrust we characterize by its use of *as-is* high-level synthesis algorithms, such as HLock [14], SFLL-HLS [18], DECOY [19], and others [13]. While these techniques utilize HLS in their design process, they do not tailor HLS algorithms towards security-centered design goals. For example, the work in [13], [14], [18] proposes algorithms to budget and configure logic locking prior to or during HLS. While these approaches occur alongside HLS, they do not directly integrate with HLS algorithms to inform either the design's RTL or the configuration of logic locking to improve security. Rather, this research supports the criticality of architectural context to the security of logic obfuscation. DECOY explores tighter integration with HLS, however, it still does not integrate into any phase/algorithm of HLS [19]. Instead, DECOY adds a design optimization during HLS to identify critical and non-critical IP. Critical IP is redacted and implemented in an eFPGA for protection. Non-critical IP remains in the ASIC. Thus, DECOY relies on the strong security protections of the eFPGA to protect the critical IP. While this yields security, eFPGA redaction introduces sizable complexity and overhead. Such an approach is often untenable.

This leads us to the two primary distinctions between this work and prior art. First, while the work in [12]–[19] recognizes the importance of high-level context during HLS and other high-level design processes, it fails to capitalize on the architectural decisions made during these processes to enhance security. Instead, conventional HLS algorithms optimizing for goals such as switching activity [22] or register re-use [23] are used as is. This is a missed opportunity. HLS algorithms can be designed to make RT-level design decisions that optimize/inform supply-chain security instead, as shown in this work. Second, these prior works propose end-to-end locking solutions that implement locking techniques in a design during HLS [12]–[19]. The security-aware binding approaches explored in this work do not configure or specify a locking mechanism in a design. Instead, our proposed algorithms provide the designer of the system the flexibility to select the locking technique they wish to use based on their application, security, and overhead goals and then configure these schemes based on what modules contain high-value IP. As a result, this work considers a more general problem (i.e., where the designer can choose the locking scheme that best fits their goals) with a different objective (i.e., optimizing the architecture around a fixed locking configuration).

B. Contributions

We propose security-aware resource binding to enhance logic locking. To do so, we bifurcate logic locking into 2 families with distinct security goals during HLS, called *Distributed Error Locking* and *Critical Minterm Locking*. We then propose 2 problem formulations, one for each locking family, and formalize security-aware resource binding algorithms to provably maximize the efficacy of locking. Note that all combinational logic locking techniques can be placed in one of these locking families, hence, by formalizing security-aware resource binding for each family, we formalize security-aware resource binding for combinational logic locking as a whole. Our contributions for each locking family are:

Distributed Error Locking:

1) A cost function to guide resource binding that maximizes the number of wrong keys that produce application errors for a specified locking configuration.
2) A binding algorithm based on graph theory that maps operations to locked modules to maximize the number of wrong keys resulting in application corruption. An optimal algorithm as well as a P-time heuristic is developed.

Critical Minterm Locking:

1) *Critical minterm locking* is a special case of *distributed error locking* that allows broader design goals to be pursued. To utilize this expanded scope, we define a novel cost function that binds a circuit to 1) ensure all wrong keys produce application error and 2) maximize this error.
2) A graph-theoretic binding algorithm that optimally maps operations to locked resources to provably maximize security through our derived cost function in P-time.

To evaluate our security-aware binding algorithms, we applied them to 11 MediaBench benchmarks [24]. For distributed error locking, our security-aware binding algorithms produced locked circuits that corrupted a characteristic application for 52% more wrong keys than the same circuit bound with conventional binding algorithms. For critical minterm locking, our security-aware binding algorithms produced locked circuits that corrupted a characteristic application for 100% of wrong keys. Moreover, these locking configurations exhibited 26x more application errors than the same circuit bound with conventional binding algorithms. For each binding/locking solution, SAT resilience was maintained and minimal overhead was incurred compared to conventional binding schemes. Locking applied post-binding could not achieve both application corruption *and* SAT resilience. In this way, combinational locking, regardless of construction, can be enhanced via security-aware resource binding during HLS.

II. Preliminaries

A. Logic Locking

There are two primary security metrics for logic locking: 1) **corruption** and 2) **attack resilience**. Corruption is the ability to cause failures for wrong keys. This can be quantified by the number of *locked inputs* (i.e., error-causing inputs). Attack resilience is the ability to resist attempts to bypass obfuscation. This can be quantified by the complexity of specific attack strategies. For a prevalent attack against logic locking, known as the SAT attack [5], [6], a relationship between corruption and attack resilience (i.e., SAT query count) has been identified [9]–[11]. While the exact nature of this relationship is currently unknown [11], there have been upper-bounds derived [10], [11] as well as probabilistic explorations [9].

These results show that for a fixed circuit and locking configuration, there is some trade-off between the number of locked inputs (corruptibility) and the SAT query count to unlock a circuit. Because the SAT attack assumes access to an IC's scan-chain (i.e., its intermediate registers), SAT attack resilience is calculated independently on a per-module basis. Based on this relationship, research has shown that obfuscation often cannot lock enough inputs to reliably derail unauthorized use while maintaining SAT resilience [7], [8]. Thus, locking is stuck in a dilemma between security goals.

A number of combinational logic locking schemes have been proposed. While these schemes vary in construction, they all achieve security by causing output corruption for a set of inputs determined by the wrong key. We refer to these output-corruption-inducing inputs as *locked inputs*. We refer to the relationship between the set of locked inputs and the wrong key as the *error profile* of a locking scheme. In this work, we differentiate combinational locking schemes by the presence of *critical minterms*, namely inputs that produce output corruption for a large subset of wrong keys. We denote locking techniques that lack critical minterms as *distributed error locking*. This includes CAS-Lock [25], Lopher [26], InterLock [27], and others [28]–[33]. Conversely, we denote locking techniques that use critical minterms as *critical minterm locking*. This includes Stripped Functionality Logic Locking (SFLL) [34]–[36] and Strong Anti-SAT [37]. These two families form a tautology, ensuring that *all* combinational locking can be characterized in this way. We note that this categorization of logic locking differs from the commonly-used low/high-error classification categories. However, we classify techniques in this way because it enables the derivation of a provably-optimal, P-time solution to the security-aware biding problem for critical minterm locking techniques (Sec. V), a significant contribution of the work.

B. Approximate Attacks on Logic Locking

Approximate SAT attacks were developed to exploit the fact that only a small set of inputs can be locked by SAT resilient locking [38]–[40]. These attacks define early termination conditions for SAT attacks that aim to locate a key that is *good enough* to use a locked IP. Ideally, a key returned by such an attack would produce output corruption only for inputs that are never/rarely used, ensuring that locking-induced-corruption does not occur during normal operation. Doing so bypasses the security of logic locking. To resist approximate attacks, a designer must ensure that locking injects error during normal operation for the overwhelming majority of wrong keys.

C. High-Level Synthesis (HLS)

HLS transforms a behavioral description of functionality, such as a high-level language, into an RT-level design. There are generally 3 design optimizations during HLS: resource allocation, scheduling, and resource binding. Resource allocation determines the quantity and type of hardware resources available to implement the design. Upon termination, a set of allocated functional units (FUs) (e.g., multipliers, dividers, etc.) is produced. Scheduling imposes clock-cycle boundaries on the target behavioral code to resolve data dependencies. This produces a scheduled data flow graph (DFG), whereby vertices are operations and edges are dependencies between operations. Binding maps (*"binds"*) operations to allocated FUs. Common binding schemes aim to 1) minimize area (i.e., registers/muxes) [23] and 2) minimize switching power [22], [41]. During binding, the expected input space for a circuit is generally known [22], [41]. This enables switching power estimation to inform power-aware binding decisions.

D. Considered Logic Locking Techniques

This work considers how an architecture can be built around a designer-specified locking configuration. As such, our algorithms are intentionally technique-agnostic, requiring no specific locking mechanism to be used or modules to be locked. This has two advantages. 1) It allows us to rely on the strength of existing locking techniques, each with their own use-case, and instead focus on how to optimally bind the system around these techniques to improve security. 2) It allows the designer to select the locking technique from the literature based on their application, security, and overhead goals and then configure these techniques in their design based on what modules contain IP they wish to protect. Hence, **a designer can lock their system to reflect their unique security goals while using our resource binding algorithms.**

E. Attacker Model

Each proposed logic locking technique claims security under its own attacker model. In some cases, the attacker's capabilities differ substantially (e.g., with/without scan-chain access). Because the security-aware binding algorithms proposed in this work are generic to the locking mechanism used (see Sec. II-D), we inherit the attacker model of the selected locking scheme. We assume that the designer-chosen locking technique is implemented per the author's specifications in designer-specified locked modules to defend against their own attacker model. We then focus on the resource binding for each locked module, rather than locking these modules, to enhance security. Stated formally, **we target the attacker model of the locking scheme chosen by the designer.**

The way logic locking is configured within a system (e.g., what modules are locked) has a substantial impact on security. In particular, removal [42], structural reverse engineering [43], and de-synthesis [44] attacks are particularly relevant. These attacks perform structural/logical analysis that allow a netlist-equipped adversary to identify locking logic and then either remove it or infer the correct key. Given the architectural scope of the security-aware resource binding techniques we propose, it is vital that the designer carefully consider the resilience of their locking configuration to these attacks by avoiding locking easily reverse-engineered or identifiable modules in the IC.

III. Motivation for Security-Aware Binding

Based on the prior work outlined in Sec. I-A, there is a need to think *beyond the locked module* when obfuscating an IC. If we follow conventional wisdom and consider only module-level context while locking, we necessarily fall into a trade-off

between corruption and SAT resilience. Both design goals are necessary for security. Therefore, in this work, we explore how "smart" security-aware binding decisions can be harnessed to enhance logic locking and achieve both competing goals.

We begin by formalizing the security ramifications of resource binding decisions. The foundation of this relationship stems from the fact that resource binding selects the operations executed on locked functional units (FUs), thereby determining the input minterms (i.e., values) typically processed on locked FUs. Because locking corrupts output for only specific locked inputs for a given wrong key, binding decisions greatly impact security. We demonstrate this below.

A. Motivational Example: Overview

In order to explore the security impact of binding decisions on logic locking, let us consider the scheduled DFG in Fig. 1A. This DFG is created from the behavioral description provided as input to HLS, usually as a high-level programming language. Vertices in the graph are operations. Edges are data dependencies between operations. The DFG in Fig. 1A is scheduled over two cycles. During the first cycle, OP_A and OP_B occur. During the second cycle, OP_C and OP_D occur. For simplicity, assume each operation in the DFG is an add. There are 2 adder FUs allocated to bind the DFG.

The resource binding phase of HLS maps the 4 add operations onto the 2 allocated adder FUs. Fig. 1B depicts 2 candidate bindings for our sample DFG. The green-shaded region encloses operations bound to FU1. The red-shaded region encloses operations bound to FU2. Let us assume that we have knowledge of the input distribution for each operation, a common assumption for HLS [22], [41]. This allows us to estimate how often various input values (minterms) occur for each operation during typical workloads. We have aggregated these estimates for 4 arbitrary input minterms, denoted $\{w, x, y, z\}$, during a typical workload below Fig. 1A.

By using the expected occurrence of input minterms, we can estimate how often each input is applied to each adder. We have compiled these estimates for each FU/binding below Fig. 1B. Let us consider how security-informed decisions can be made during resource binding to impact security for the two combinational logic locking families, defined in Sec. II-A. We emphasize that these families form a tautology. By exploring both families our analysis considers *all* combinational locking.

B. Security-Aware Binding for Distributed Error Locking

Distributed error locking schemes are characterized by a lack of critical minterms in their error profile (see Sec. II-A for definitions). This means that error is distributed throughout the input space without any one input minterm producing output error (i.e., being locked) for a large subset of wrong keys. The advantage of schemes in this locking family is their ability to produce strong and non-probabilistic SAT resilience guarantees [9]–[11]. A key limitation of these locking schemes is their susceptibility to approximate attacks (Sec. II-B). This limitation arises from the trade-off between the number of locked inputs per wrong key and SAT attack resilience underlying logic locking [9]–[11]. To be SAT resilient, distributed

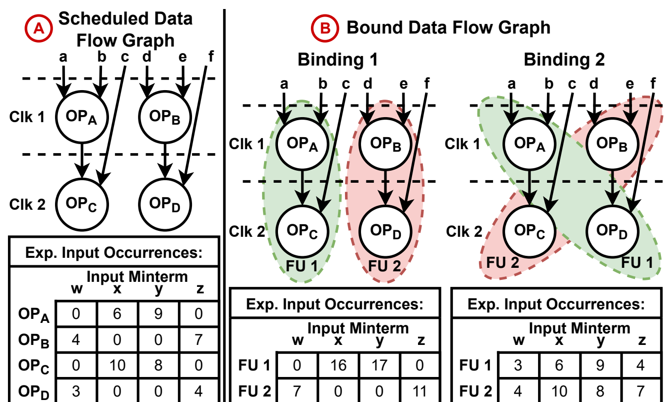


Fig. 1: Sample scheduled DFG and binding solutions.

error locking must lock only a few inputs per wrong key. These locked inputs are distributed throughout the entire input space of the locked module, with no one input producing output error for the majority of wrong keys (i.e., no critical minterms). However, because many commonly-locked modules only utilize a small fraction of their input space during normal operation [8], there is a high probability that any given wrong key will *only* produce corruption for inputs that are not used during normal operation [7], [8]. If no locked inputs are used for a given wrong key, no locking-induced corruption will occur, mitigating the security of locking. Thus, there exists a large subset of wrong keys that produce functionality *good enough* to enable unauthorized use and IP piracy. These *good enough* keys can be identified by an approximate-style attacker and be used to bypass the security of locking [38]–[40]. Hence, to secure against an untrusted foundry with distributed error locking, we must maximize the number of wrong keys that produce error during normal operation.

With this in mind, we return to our example in Fig. 1. Assume that a designer used a conventional, security-agnostic binding algorithm, which has generated binding 1 for the circuit. The designer then decides to secure design IP by locking FU 1 with a distributed error locking technique following binding. To maximize IP security (i.e., corruption), the designer would lock the majority of input minterms. This ensures high application corruption regardless of wrong key. However, SAT resilience is inversely related to the number of locked inputs, hence, such an approach would result in minimal SAT resilience. To meet SAT resilience constraints, let us assume only a single input minterm can be locked for each wrong key. Consider the following locking scenario. Assume that FU 1, which has a 3-bit input (i.e., 8 total input values), has been locked using SARLock [30], a prominent distributed error logic locking technique. The resulting error profile for locked FU 1 is in Tab. I. We note that a similar error profile could be produced by many other distributed error locking techniques, such as Anti-SAT [32]. This error profile shows which input minterms (rows of the table) produce corrupt output for a given wrong key (columns of the table). To do so, Tab. I contains an ✗ at the intersection of a wrong key/input when that input produces output corruption for a wrong key (e.g., wk_0 locks input w).

Input	Key Input							
	wk_0	wk_1	wk_2	wk_3	ck_4	wk_5	wk_6	wk_7
$s:0$	✓	✓	✓	✓	✓	✗	✓	✓
$t:1$	✓	✓	✓	✓	✓	✓	✗	✓
$u:2$	✓	✓	✓	✓	✓	✓	✓	✗
$v:3$	✓	✓	✓	✓	✓	✓	✓	✓
$w:4$	✗	✓	✓	✓	✓	✓	✓	✓
$x:5$	✓	✗	✓	✓	✓	✓	✓	✓
$y:6$	✓	✓	✗	✓	✓	✓	✓	✓
$z:7$	✓	✓	✓	✗	✓	✓	✓	✓

TABLE I: Error profile for locked FU1 that shows which inputs produce an incorrect output for each key. An ✗ at an input/key intersection indicates that the locked FU produces incorrect output when that input/key combination is applied. A ✓ indicates that the circuit produces correct output when that input/key combination is applied. Note that "wk" identifies wrong key values and "ck" identifies a correct key (i.e., a key where all inputs produce correct output.

Because we know the expected occurrence of various inputs for each operation and the error profile of the locking configuration, we can estimate how many wrong keys for this locked FU will corrupt a typical application. Notice that FU 1 in binding 1 executes OP_A, which operates on inputs x, y, and OP_C, which operates on inputs x, y. Based on the error profile in Tab. I, this means that the bound/locked circuit produces output corruption for 2 wrong keys (wk_1, wk_2) during typical workloads. Let us consider how security-aware binding can increase the number of wrong keys producing corruption.

1) Security-Aware Binding: Consider the case where a locking configuration has been specified prior to resource binding. Following our prior example, this means FU 1 will be locked by distributed error locking with the error profile in Tab. I. However, instead of binding in a security-oblivious fashion, let us bind the DFG in Fig. 1A to maximize the wrong keys that lock an input applied to the locked FU (FU 1). In this case, binding 2 would be selected, resulting in 4 wrong keys ($wk_0 - wk_3$) injecting error. The advantage of this approach is that the number of wrong keys that produce corruption is doubled (4 vs. 2) compared to our prior security-oblivious binding approach that selected binding 1. Because the error profile of the locking technique is static, this results in an increase in the number of wrong keys that derail unauthorized IP use *without* compromising SAT resilience. Remember, wrong keys must actually inject error within the circuit to derail unauthorized use, hence, by ensuring that more wrong keys inject error during normal operation, we maximize the security of locking. Thus, by binding to maximize error-producing wrong keys, a designer can create a locked circuit substantially more likely to derail unauthorized use.

C. Security-Aware Binding for Critical Minterm Locking

Critical minterm locking schemes are characterized by the presence of critical minterms in their error profile. This means that a small set of locked inputs produce output corruption for a large subset of (if not all) wrong keys. Logic locking achieves security through the injection of locking-induced corruption within the locked IC. Hence, a designer can enhance the security of critical minterm locking by maximizing the occurrence of critical minterms in locked modules during typical workloads. By doing so, a designer both 1) guarantees that a large subset (if not all) wrong keys inject output corruption, and 2) maximizes the number of errors injected. In this way, critical minterm locking serves as a special case of distributed error locking, whereby the security goal of maximizing the wrong keys that produce corruption can be achieved by ensuring that critical minterms occur during typical workloads. Therefore, to secure against an untrusted foundry with critical minterm locking, we aim to maximize the occurrence of critical minterms during normal operation.

Let us return to our example in Fig. 1. Assume that a designer used a conventional, security-agnostic binding algorithm, which has generated binding 2 for the circuit. The designer then decides to secure design IP by locking FU 1 with critical minterm locking that locks a single critical input, randomly selected to be x. Whenever x is applied as input to FU 1, regardless of the wrong key, output corruption will be produced. Based on our estimates for the occurrence of input x during a typical workload, we can estimate the occurrence of critical inputs for our locked adder (FU 1). This corresponds to the number of locking-induced application errors. Notice that FU 1 in binding 2 executes OP_A, which we estimate will process input x 6 times, and OP_D, which we estimate will process input x 0 times. Thus, we expect $6 + 0 = 6$ error injections to be caused by locking during a typical workload. Let us consider how security-aware binding can improve this.

1) Security-Aware Binding: Consider the case where the locking configuration is specified prior to resource binding. Thus, while binding the circuit, the designer has determined that FU 1 protects critical input x. Because the locking configuration is known, let us consider binding with locking in mind. Specifically, consider binding the DFG in Fig. 1A to maximize how often the locked input (x) will be applied to the locked FU (FU 1). This leads to choosing binding 1, which produces $6 + 10 = 16$ errors during a typical workload. This approach to binding has 3 merits. 1) Because the input x is typically applied to our locked FU, we ensure that every wrong key will inject error. 2) The errors injected by obfuscation are more than doubled (16 vs. 6) compared to a conventional binding approach. Because the locked input count is fixed, such an approach increases the corruptibility of locking *without* degrading SAT resilience to do so. 3) Errors are injected during both cycles of the schedule (clk 1 and 2) instead of only one (clk 1). Consecutive errors increase the probability of critically impacting the application. By binding to maximize critical minterm occurrence, we have designed a circuit with both more *and* higher quality application errors.

D. Security-Aware Binding to Enhance Logic Locking

For both locking families, we were able to use binding decisions to leverage architectural context to enhance security. For distributed error locking, we doubled the wrong keys that produced corruption during typical workloads. For critical minterm locking, we ensured that all wrong keys produced corruption while also causing $\sim 2x$ more error injections. In either case, the locked input count could be reduced while

simultaneously increasing the corruption of the locking construction through making security-centered decisions during binding. Essentially, *security-aware* binding decisions occur outside of the trade-off between corruption and SAT resilience, allowing them to enhance the security of logic locking.

IV. Security-Aware Binding for Distributed Error Locking

Let us formalize an algorithm to solve the *security-aware binding* problem for distributed error locking, outlined in Sec. III-B. The security-aware binding problem requires a scheduled DFG and a set of allocated resources, some of which are locked, as input. For each locked FU, we assume a SAT-resilient locking configuration (i.e., a sufficiently small number of inputs are locked per wrong key) has been specified. For locking techniques in the distributed error family, let us assume that this locking specification includes 1) the number of FUs locked and 2) the error profile for the locking scheme used. Now, given a list of FUs, a scheduled DFG, and locking details, we must bind operations to FUs to maximize the wrong keys producing application error. Doing so maximizes the corruption of the locking configuration, namely its ability to restrict unauthorized use, while maintaining SAT resilience (because the locking was configured to be SAT resilient a priori). We begin by defining a cost function that quantifies the number of error-producing wrong keys for a given binding. We use this as a security metric to inform binding decisions.

A. Objective Cost Function for Distributed Error Locking

Suppose that we have scheduled and bound a DFG onto FUs, some of which have been locked using distributed error locking. We aim to quantify the impact of these locked FUs on the number of wrong keys producing error in the DFG. We capture this by counting the number of wrong keys that lock inputs evaluated by a locked FU during the DFG's execution. The objective is to maximize these wrong keys which inject error through appropriate binding decisions. Let us define the set X_n to represent the set of inputs applied to operation n. Thus, operation n operates on input x during normal operation if $x \in X_n$. X can be calculated for a DFG by simulating characteristic input traces or applications, which are typically available during HLS [22], [41]. Given an input trace for the DFG, we can perform time simulation to generate a set of input minterms applied to each operation.

We now formalize a cost function to inform binding decisions that quantifies the number of wrong keys that produce application error for a given locking configuration in a bound DFG. To do so, assume that L is the set of obfuscated FUs. Each locked FU, $l \in L$, has a locking configuration with an error profile $E_l(x)$. For an input minterm $x \in X_n$, $E_l(x)$ returns the set of wrong keys that corrupt x for the locking configuration. We define the set of wrong keys corrupted by binding operation n to locked FU l to be:

$$K_{l,n} = \bigcup_{x \in X_n} E_l(x) \tag{1}$$

If each $l \in L$ binds a set of operations N_l, then the total number of wrong keys that produce error for the binding is:

$$K_{err.} = \sum_{l \in L} \left| \bigcup_{n \in N_l} K_{l,n} \right| \tag{2}$$

B. Optimal Security-Aware Binding Algorithm

Let us develop a binding algorithm to map operations to resources such that the wrong keys producing error (i.e., keys that lock inputs applied to locked FUs), as quantified by Eqn. 2, is maximized. Consider the scheduled DFG, S, with a depth of s cycles. The set R contains the resources allocated to bind the DFG. In order to solve this problem, we do not make any assumption regarding the type (e.g., addition) of resources and operations being bound. However, we do assume that all resources and operations are of the same type. This allows any resource in R to bind any operation in the DFG. We can make this assumption without any loss of generality by handling all resource/operation types independently. Of the R allocated resources, a subset, L, are obfuscated ($L \subseteq R$). Each $l \in L$ has a locking configuration with an error profile $E_l(x)$. This error profile is determined by the locking configuration used. As such, it is specified by the user prior to binding.

During each cycle t ($t \leq s$), a set of concurrent operations $N_t \subset S$ are scheduled. Binding maps each operation in N_t to one of the allocated FUs ($|R| \geq |N_t|$). We can trivially determine the best security-aware binding solution according to Eqn. 2 by enumerating and evaluating *every* possible binding solution. Unfortunately, this problem is not *separable* in any way. This is because any choice to bind an operation to a locked FU alters the cost calculated by Eqn. 2 when considering binding another operation to that same FU. Thus, there is no way to separate the binding problem into smaller, independent sub-problems enabling more efficient execution.

C. Heuristic Security-Aware Binding Algorithm

While a brute-force approach necessarily produces an optimal binding solution, it is not efficient. In this exhaustive approach, bindings unlikely to produce an optimal solution are still evaluated, inflating runtime. Consider the case where 2 operations, OP_A and OP_B, with input distributions, X_A and X_B. If X_A shares many members with X_B then it is unlikely that OP_A and OP_B would ever be bound to the same FU because doing so would be redundant and provide little increase in the cost calculated by Eqn. 2. A good heuristic only considers bindings capable of substantially increasing the number of wrong keys producing error. To do this, we propose a greedy agglomerative clustering heuristic that binds the operation producing the greatest increase in error-producing wrong keys at a given time. Let us define this heuristic.

To bind the operations in an arbitrary DFG, we construct a weighted bipartite graph $B = (R \cup N, E)$. Each vertex $r_i \in R$ is an FU. Each vertex n_j is an operation from the set of all operations N (i.e., $n_j \in N$). 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 between these vertices. Initially, this should be the case for all r_i-n_j pairs in the graph. If $K_{i,j}$ is the set of wrong

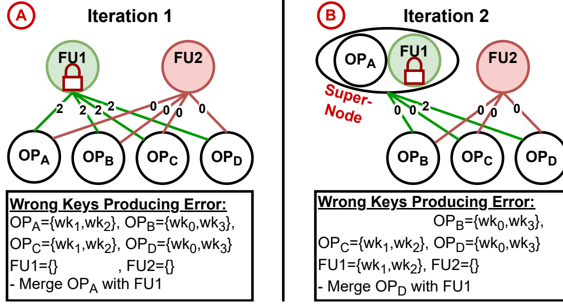


Fig. 2: Example of the security-aware binding heuristic for distributed error locking binding the DFG in Fig. 1A.

keys corrupted during normal operation when operation j is bound to for FU i, then the weight, $w_{i,j}$, is:

$$w_{i,j} = |K_{i,j}| \tag{3}$$

To bind, we select the maximum weight edge in the graph and merge the node for this operation, n_j, into the corresponding resource node, r_i. This produces a *super-node*, containing both resource r_i and operation n_j, which represents binding n_j to r_i. The weighted bipartite graph is now reconstructed with this new super-node, which we call r_i', replacing r_i.

To bind a new operation j, edge weights between non-bound operations and the super-node r_i' must be re-calculated with:

$$w_{i,j}' = \left| K_{i,j} - \bigcup_{n \in N_i} K_{i,n} \right| \tag{4}$$

where N_i is the set of operations currently bound to FU i. Thus, $w_{i,j}'$ quantifies the increase in the number of wrong keys causing error if operation j is bound to resource i that already has operations in the set N_i bound to it. Note that the edge weights connected to non-locked FUs are always 0. Similarly, any edges to operations that occur during the same cycle as an operation in super-node r_i' must not be included, as they are not valid bindings. Greedy clustering proceeds in this fashion until no operations (i.e., nodes) remain in N for the graph. The final binding solution is constructed by binding each operation to the resource present in its super-node.

To demonstrate the heuristic, consider the binding problem from Sec. III-B. There are 2 FUs, $R = \{FU1, FU2\}$, allocated to bind the DFG in Fig. 1A. $FU1$ is locked with the error profile in Tab. I. To bind this DFG, we initially produce the bipartite graph in Fig. 2A. An edge weight of 2 exists between each operation and $FU1$ because binding any of these operations to $FU1$ increases the number of error-producing wrong keys by 2. Thus, any operation can be merged with $FU1$ during iteration 1. We arbitrarily select OP_A and create a *super-node* containing $FU1$ and OP_A. Our bipartite graph is reconstructed with this super-node, depicted in Fig. 2B. At this point, only the edge between the super-node and OP_D has non-zero weight because only OP_D produces corruption for new keys (i.e., wk_0, wk_3). Thus, iteration 2 merges OP_D with the super-node containing $\{FU1, OP_A\}$, making a new super-node of $\{FU1, OP_A, OP_D\}$. The bipartite graph is again regenerated and the algorithm proceeds until no edges remain. This yields binding 2 from Fig. 1B, which indeed maximizes the error-producing wrong keys.

D. Analysis of Proposed Security-Aware Binding Algorithms

Note the following 3 properties of the proposed algorithms.

1) Runtime Complexity: The optimal security-aware binding algorithm assesses every possible binding configuration for all locked FUs. If there are $|R|$ FUs, $|L|$ of which are locked ($|R| \geq |L|$), there is at most $|R|$ operations bound during each cycle of the schedule. If we assume the schedule is of length s, then there exists at most $\binom{|R|}{|L|}^s$ binding solutions for the locked FUs. This yields a super-polynomial complexity. However, consider the greedy heuristic, which maps the operation causing the biggest increase in error-producing wrong keys to a locked FU for each iteration. In this case, edge weight must be calculated between each un-bound operation in the DFG and locked resource. There are $|N|$ total operations in the DFG, hence, edge weight must be calculated for at most $|N||L|$ operation/FU combinations. If we assume that the wrong keys corrupting an input trace for an operation is stored in a sorted list K, the edge weight (Eqn. 4) can be calculated in $O(|K|)$. This graph is constructed at most $|N|$ times to map every operation to an FU, resulting in a time complexity of $O(|N|^2|L||K|)$ for the heuristic, a P-time solution.

2) Validity and Completeness of Binding Solution:

Theorem 1. *The security-aware binding heuristic will produce a valid and complete binding solution, if it exists.*

We omit a detailed proof, however, this follows from 2 aspects of the heuristic. First, bipartite edges are only added between FUs and operations capable of being bound together. As a result, FUs that already have operations from a given cycle bound to them will not have edges to other operations in that same cycle. This guarantees a valid solution. Second, the algorithm terminates when no edges remain. This can only occur when all operations have been bound to an FU, a complete binding solution, or when there are not enough resources available to bind operations in a cycle (i.e., $|N_i| > |R|$ for cycle i), in which case there is no valid binding. Thus, a valid and complete binding solution is produced, if it exists.

3) Security Impact: The security-aware binding algorithms we have proposed bind a circuit to maximize the wrong keys causing error for a fixed locking configuration. Remember, the locking configuration dictates the SAT resilience of the design. Therefore, because the locking configuration is not altered during binding, our algorithms enhance the security of logic locking without reducing SAT resilience to do so.

V. Security-Aware Binding for Critical Minterm Locking

Now, let us consider the *security-aware binding* problem in the context of critical minterm locking. As we noted in Sec. III-C, critical minterm locking serves as a special case of distributed error locking, whereby our security goal of maximizing the number of wrong keys producing error can be trivially achieved by maximizing the occurrence of critical minterms on locked FUs. Doing so also has the added

impact of maximizing locking-induced errors. As we show, this allows the security-aware binding problem to be solved more efficiently for critical minterm locking. To begin, we define an objective cost function for critical minterm locking to quantify locking-induced application errors for a fixed binding.

A. Objective Cost Function for Critical Minterm Locking

Suppose that we have scheduled and bound a DFG onto FUs locked using critical minterm locking. We aim to quantify the impact of these locked FUs on error in 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 smart binding decisions. Let us define matrix μ 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 $\mu_{m,n}$. μ is calculated using the "typical" input traces that are available during HLS [22], [41]. 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 μ, we define an objective cost function to inform binding that quantifies the expected number of application errors for a given locking configuration in a bound DFG. To do so, assume that 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:

$$Err. = \sum_{l \in L} \sum_{m \in M_l} \sum_{n \in N_l} \mu_{m,n} \tag{5}$$

B. Security-Aware Binding Algorithm

Using the cost function in Eqn. 5, we develop a binding algorithm that maps operations to FUs such that the application errors (i.e., when locked inputs are applied to locked FUs) are maximized. Consider a scheduled DFG, S, spanning s cycles. A set of resources, R, has been allocated to bind the DFG. Of these R resources, L have 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:

$$w_{i,j} = \sum_{m \in M_i} \mu_{m,j} \tag{6}$$

where M_i is the set of locked inputs for FU i and $\mu_{m,j}$ is the expected occurrences of locked input $m \in M_i$ for operation j. Thus, $w_{i,j}$ 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. We solve the max weight bipartite matching problem for B_1, which can be solved optimally in P-time. The resulting match maps (binds) each operation during clock $t = 1$ to an available FU.

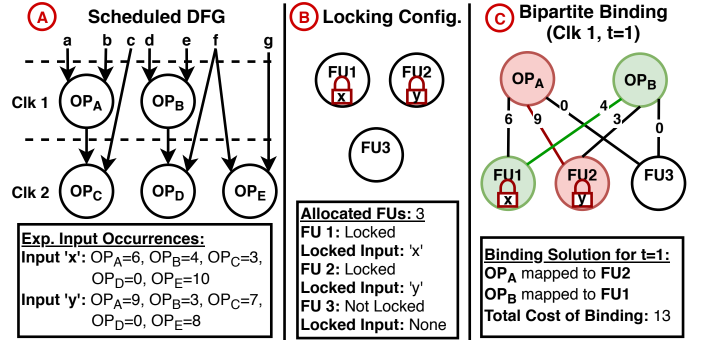


Fig. 3: Security-aware binding algorithm for clock 1 (t=1).

To demonstrate this algorithm, consider the DFG in Fig. 3A, which spans two clocks. There are 3 FUs, $R = \{FU1, FU2, FU3\}$ shown in Fig. 3B, allocated to bind this DFG. Of these FUs, two 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. 3A. For $t = 1$, the proposed algorithm produces the bipartite graph in Fig. 3C. 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 two 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 locking for the characteristic workload (proved in Thm. 3).

C. Analysis of Proposed Security-Aware Binding Algorithm

Note the following 3 properties of the proposed algorithm.

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 $O(mnlog(n))$ [45]. By negating each edge weight ($w_{i,j}$) and assuming that $|N_m|$ is the maximum number of concurrent operations

in the DFG, security-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 2. *The security-aware binding algorithm will produce a valid and complete binding solution, if it exists.*

We omit a proof, however, during each clock cycle bipartite matching ensures a valid match between operations and FUs. This means that all operations in all clocks are bound to only one FU, with no more than one operation in a cycle bound to a given FU. Hence, the final solution is valid and complete.

3) Optimality of Binding Solution:

Theorem 3. *The security-aware binding algorithm yields the maximum 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. Hence, the full matching produced for each bipartite graph is independent. This means 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 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 proposed algorithm binds the circuit to optimally generate application corruption for a fixed locking scheme. Remember that this locking configuration was specified prior to binding to lock few enough inputs to resist SAT-style attacks. Therefore, the proposed security-aware binding algorithm guarantees the maximum achievable corruption is achieved without degrading SAT resilience to do so.

VI. Evaluation of Security-Aware Binding

To evaluate our security-aware binding algorithms, we applied them to bind adder and multiplier FUs in 11 benchmark DFGs. These benchmarks were created by isolating 11 C functions from 8 MediaBench benchmarks [24] and extracting their DFG with SUIF. A path-based scheduler [46] was used to schedule each extracted DFG such that no more than 3 FUs of any type were needed to bind it. The resulting benchmarks had 18.6 add and 10.6 multiply operations over 13.5 cycles on average. For each benchmark, we used the MediaBench sample workloads to generate our typical application (i.e., characteristic input trace). To do so, a trace driven simulator calculated the occurrence of input minterms for each operation in the DFG. The typical application and scheduled DFG were the inputs for each security-aware algorithm. The benchmark generation flow is shown in Fig. 4.

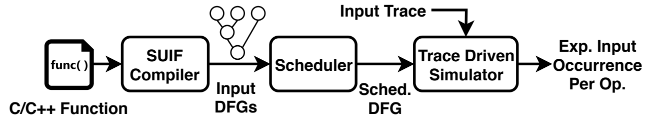


Fig. 4: Process to generate benchmark circuits.

To empirically assess the proposed security-aware binding algorithms, we compared them to alternative binding algorithms run on circuits with identical logic obfuscation configurations. For this comparison, we used an area-aware approach [23], which minimizes register count, and a power-aware approach [22], which minimizes switching frequency. For each benchmark, we enumerated combinations of {1,2,3} locked FUs with varied locking configurations. For each enumerated locked FU/locking configuration, we created a bound/locked circuit using 1) our proposed security-aware, 2) area-aware, and 3) power-aware binding algorithms. We then calculated the ratio of the effectiveness of our proposed security-aware approach compared to each area/power-aware approach with the same locking configuration. In this way, we compared each circuit created with a security-aware algorithm to the same circuit incorporating an *identical* locking configuration created with an area/power-aware algorithm. This isolates security-improvements caused by security-aware binding across numerous circuits and obfuscation configurations.

A. Experimental Analysis: Distributed Error Locking

To evaluate security-aware binding for distributed error locking, we used 4 locking configurations with different error profiles in {1,2,3} locked FUs. These configurations were selected to provide a reasonable cross-section of the error profiles used by distributed error locking schemes. The first configuration we considered was from a 256-bit implementation of CASLock (P=1) [25]. An identical error profile could be produced by many other techniques including Anti-SAT [32] and SARLock [30]. The other 3 configurations we considered randomly locked {0.01%, 0.001%, and 0.0001%}[1] of the input space for each wrong key. These configurations simulate the error profile produced by techniques such as LUT-Lock [28], InterLock [27], and others [29], [31]. For all 12 locking configurations (i.e. 4 error profiles, {1,2,3} locked FUs), we created a bound/locked circuit for each benchmark. We then calculated the ratio between the number of wrong keys corrupting the characteristic application after our security-aware binding approach compared to an area/power-aware approach with the same locking configuration. The results were averaged over every benchmark for Fig. 5.

Fig. 5 shows that optimal security-aware binding increased the wrong keys corrupting the characteristic application by an average of 52%, 30%, and 13% for 1, 2, and 3 locked

[1] Remember, locked inputs per key must be kept low to resist SAT [9], [10]

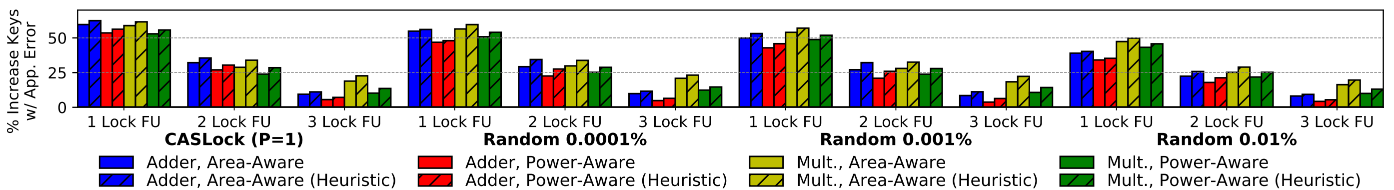


Fig. 5: Efficacy of security-aware binding for distributed error locking. Bars normalized to area [23]/power [22] aware binding.

FUs when compared to area/power-aware binding. Our P-time heuristic degraded the solution only slightly, producing a 50%, 27%, and 11% increase in application-corrupting wrong keys. This supports the utility of the security-aware binding heuristic. Because the performance degradation is small, we use this heuristic for the remaining evaluation. Based on these results, security-aware binding improved the security of distributed error locking (i.e., wrong keys producing corruption), regardless of configuration, without sacrificing SAT resilience.

We make 2 more observations from Fig. 5. First, there was a $\sim$3% degradation in the performance of our binding algorithms compared to area/power binding for every order-of-magnitude increase in the locked inputs per wrong key. This is intuitive. As more wrong keys lock each input, the difference in input distribution between operations will result in a less disjoint set of wrong keys corrupted. Moreover, as wrong keys begin to lock sizable portions of the input space, the majority of wrong keys would produce error, regardless of binding decisions, reducing the security impact of binding. However, locked inputs per wrong key must be kept very small to ensure SAT resilience, reducing the impact of this trend. Second, the security improvement provided by security-aware binding decreased as more FUs were locked in the benchmark. Again, this makes sense. As more FUs are locked in a design, more operations (and their corresponding input minterms) are bound to locked FUs. Thus, it becomes more likely that advantageous operations are bound to locked FUs given that there are simply more locked FUs in the system. This lessens the security impact of binding decisions. We note that most research tightly limits allowable logic locking overhead making it unlikely that the majority of FUs in a design would be locked due to overhead constraints [2].

B. Experimental Analysis: Critical Minterm Locking

To evaluate critical minterm locking in each benchmark, we enumerated all combinations of {1,2,3} locked FUs locking {1,2,3} critical inputs each. Based on our trace-based simulation of the DFG for each benchmark, we determined the 10 most common input values as candidate critical minterms. For the 9 candidate locking configurations (i.e., {1,2,3} locked FUs locking {1,2,3} critical inputs), we created a bound/locked circuit securing each combination of the 10 candidate inputs for each locked FU. We then normalized the application errors produced by the security-aware circuit to those produced by an area/power-aware binding with the same locking configuration. These results, averaged over every locked FU count, critical minterm count, and locked input combination, are in Fig. 6.

Based on Fig. 6, all benchmarks had an occurrence of a critical input (i.e., locking-induced error). Thus, in all cases, 100% of wrong keys produced corruption in the characteristic application. Moreover, security-aware binding increased the application errors caused by the locking construction by 22x and 29x compared to area and power aware binding. Thus, our security-aware algorithms caused sizable increases in application errors, without sacrificing SAT resilience.

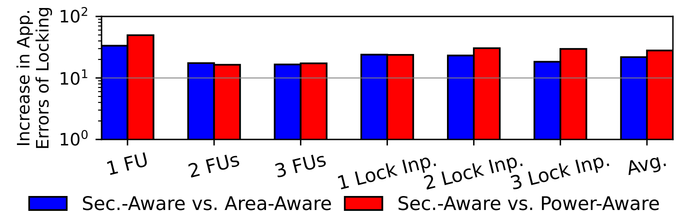


Fig. 7: Impact of critical minterm locking configuration on error from security-aware binding. Results normalized to error caused by identical locking applied after area/power binding.

We have aggregated the impact of locking configuration on the efficacy of each binding algorithm for critical minterm locking in Fig. 7. To generate Fig. 7, 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} critical inputs). In this way, we isolated the impact of each parameter on the performance of our security-aware algorithm. Based on Fig. 7, increases in error were consistent in all cases. Remember, all increases were normalized to the error caused by area/power binding for the same locking configuration (i.e., locked FU count, critical

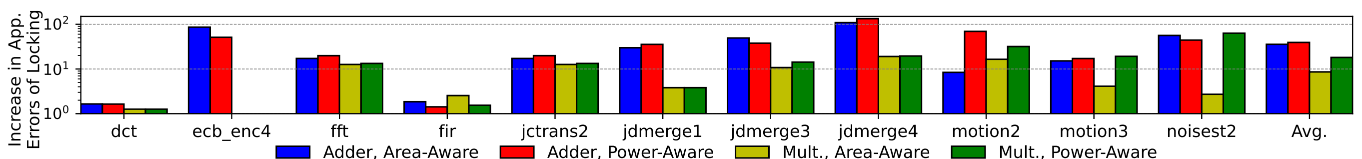


Fig. 6: Efficacy of security-aware binding for critical minterm locking. Bars normalized to area [23]/power [22] aware binding. No multipliers are in ecb_enc4, hence, these bars are not present.

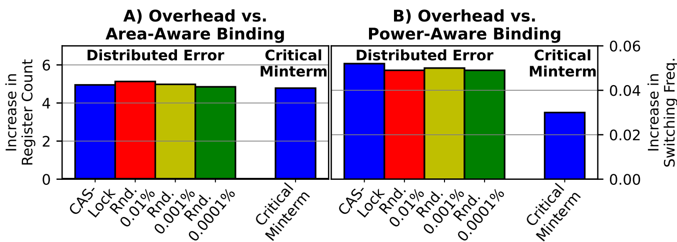


Fig. 8: Area/power overhead of security-aware binding compared to area [23]/power [22] aware binding.

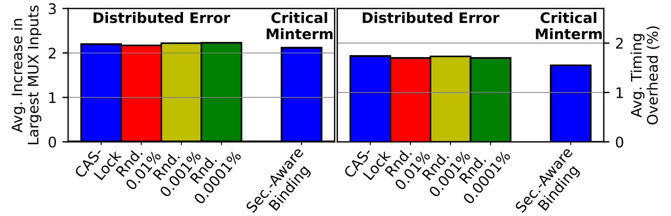


Fig. 9: Timing overhead of security-aware binding algorithms compared to area-aware [23] binding algorithm.

input count, and critical input identity). Thus, Fig. 7 suggests that security-aware binding yields a consistent increase in error, no matter the underlying locking construction.

C. Security-Aware Binding Design Overhead

Each security-aware binding algorithm makes binding decisions to enhance logic locking. These security-aware decisions are made at the expense of alternative design goals. To assess the impact of security-aware binding on these other goals, we compare circuits produced by binding the same DFG with 1) security-aware binding, 2) area-aware binding [23], which minimizes register count, and 3) power-aware binding [22], which minimizes switching frequency. The results of this comparison aggregated over all 11 benchmarks are in Fig. 8/9.

Security-aware binding used 4.8 more registers than area-aware binding, regardless of locking configuration (Fig. 8). Another side-effect of area-aware binding is the maximization of register re-use. This minimizes the size of the multiplexers on the input of each FU. Because larger multiplexers have worse timing, this approach also helps reduce timing. We have aggregated the average increase in the inputs to the largest multiplexer in the design and the average timing overhead of security-aware compared to area-aware binding using the Cadence Encounter RTL Compiler with the Synopsys 90nm SAED library in Fig. 9. On average, security-aware binding required a largest multiplexer with 2.2 more inputs causing a timing overhead of 1.7% over area-aware binding.

With respect to power-aware binding, our proposed algorithms for distributed error locking exhibited a 0.05 higher switching rate, compared to a 0.03 higher switching rate for critical minterm locking (Fig. 8). This difference between locking families makes sense. The cost function for distributed error locking (i.e., maximizing error-causing wrong keys) favors binding operations with diverse inputs together. Such binding decisions increase the probability of bit-flips between inputs, increasing switching rate. Alternatively, the cost function for critical minterm locking does not favor binding operations with disparate inputs together, instead favoring binding operations with a high occurrence of a few critical minterms to locked FUs. This results in reduced power overhead.

VII. Conclusion

We explored security-aware binding during HLS to enhance logic locking. To do so, we bifurcated logic locking schemes into 2 families: *distributed error* and *critical minterm*. We developed security-aware resource binding algorithms to enhance locking from both families without sacrificing SAT resilience. To evaluate these algorithms, we applied them to 11 MediaBench benchmarks and their typical applications. For distributed error locking, our security-aware binding algorithms designed locked circuits corrupting a typical application for 52% more wrong keys than a circuit bound with conventional schemes. For critical minterm locking, our security-aware binding algorithms designed locked circuits corrupting typical applications for 100% of wrong keys while also exhibiting 26x more errors in those applications than a circuit bound with conventional schemes. Regardless of locking family, our binding/locking solutions maintained SAT resilience while incurring minimal design overhead. Thus, security-aware binding enhances logic locking without sacrificing SAT resilience.

References

[1] M. Rostami, F. Koushanfar, and R. Karri, “A primer on hardware security: Models, methods, and metrics,” Proceedings of the IEEE, pp. 1283–1295, 2014.

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

[3] H. M. Kamali, K. Z. Azar, F. Farahmandi, and M. Tehranipoor, “Advances in logic locking: Past, present, and prospects,” Cryptology ePrint Archive, 2022.

[4] K. Shamsi et al., “Ip protection and supply chain security through logic obfuscation: A systematic overview,” ACM Transactions on Design Automation of Electronic Systems (TODAES), pp. 1–36, 2019.

[5] P. Subramanyan, S. Ray, and S. Malik, “Evaluating the security of logic encryption algorithms,” in International Symposium on Hardware Oriented Security and Trust (HOST). IEEE, 2015.

[6] K. Z. Azar, H. M. Kamali, H. Homayoun, and A. Sasan, “Smt attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the sat attacks,” Transactions on Cryptographic Hardware and Embedded Systems, 2019.

[7] K. Shamsi, D. Z. Pan, and Y. Jin, “On the impossibility of approximation-resilient circuit locking,” in Intl. Symp. on Hardware Oriented Security and Trust, 2019.

[8] M. Zuzak and A. Srivastava, “Obfusgem: Enhancing processor design obfuscation through security-aware on-chip memory and data path design,” in International Symposium on Memory Systems. ACM, 2020.

[9] M. Zuzak, Y. Liu, and A. Srivastava, “Trace logic locking: Improving the parametric space of logic locking,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2020.

[10] H. Zhou, A. Rezaei, and Y. Shen, “Resolving the trilemma in logic encryption,” in International Conference on Computer-Aided Design (ICCAD), 2019.

[11] K. Shamsi et al., “On the approximation resiliency of logic locking and ic camouflaging schemes,” IEEE Transactions on Information Forensics and Security, vol. 14, no. 2, pp. 347–359, 2018.

[12] C. Pilato et al., “Optimizing the use of behavioral locking for high-level synthesis,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2022.

[13] S. A. Islam, L. K. Sah, and S. Katkoori, "High-level synthesis of key-obfuscated rtl ip with design lockout and camouflaging," ACM Transactions on Design Automation of Electronic Systems, pp. 1–35, 2020.

[14] M. R. Muttaki, R. Mohammadivojdan, M. Tehranipoor, and F. Farahmandi, "Hlock: Locking ips at the high-level language," in ACM/IEEE Design Automation Conference (DAC), 2021, pp. 79–84.

[15] C. Pilato et al., "Assure: Rtl locking against an untrusted foundry," IEEE Transactions on Very Large Scale Integration Systems, pp. 1306–1318, 2021.

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

[17] C. Pilato, F. Regazzoni, R. Karri, and S. Garg, "Tao: techniques for algorithm-level obfuscation during high-level synthesis," in Design Automation Conference, 2018.

[18] M. Yasin, C. Zhao, and J. J. Rajendran, "Sfll-hls: Stripped-functionality logic locking meets high-level synthesis," in Intl. Conf. on Computer-Aided Design, 2019.

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

[20] M. Zuzak, Y. Liu, and A. Srivastava, "A resource binding approach to logic obfuscation," in ACM/IEEE Design Automation Conference (DAC). IEEE, 2021, pp. 235–240.

[21] C. Karfa et al., "Is register transfer level locking secure?" in 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2020, pp. 550–555.

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

[23] C.-Y. Huang, Y.-S. Chen, Y.-L. Lin, and Y.-C. Hsu, "Data path allocation based on bipartite weighted matching," in Design Automation Conference, 1991.

[24] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in International Symposium on Microarchitecture. IEEE, 1997.

[25] B. Shakya, X. Xu, M. Tehranipoor, and D. Forte, "Cas-lock: A security-corruptibility trade-off resilient logic locking scheme," IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 175–202, 2020.

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

[27] H. M. Kamali, K. Z. Azar, H. Homayoun, and A. Sasan, "Interlock: An intercorrelated logic and routing locking," in 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD). IEEE, 2020, pp. 1–9.

[28] H. M. Kamali et al., "Lut-lock: A novel lut-based logic obfuscation for fpga-bitstream and asic-hardware protection," in IEEE Computer Society Annual Symposium on VLSI. IEEE, 2018, pp. 405–410.

[29] H. M. Kamali, K. Z. Azar, H. Homayoun, and A. Sasan, "Full-lock: Hard distributions of sat instances for obfuscating circuits using fully configurable logic and routing blocks," in Design Automation Conference (DAC), 2019.

[30] M. Yasin, B. Mazumdar, J. J. Rajendran, and O. Sinanoglu, "Sarlock: Sat attack resistant logic locking," in 2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). IEEE, 2016, pp. 236–241.

[31] M. Yasin, J. J. Rajendran, O. Sinanoglu, and R. Karri, "On improving the security of logic locking," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, no. 9, pp. 1411–1424, 2015.

[32] Y. Xie and A. Srivastava, "Anti-sat: Mitigating sat attack on logic locking," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 38, no. 2, pp. 199–207, 2018.

[33] K. Shamsi, M. Li, D. Z. Pan, and Y. Jin, "Cross-lock: Dense layout-level interconnect locking using cross-bar architectures," in Great Lakes Symp. on VLSI, 2018.

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

[35] A. Sengupta, M. Nabeel, M. Yasin, and O. Sinanoglu, "Atpg-based cost-effective, secure logic locking," in IEEE 36th VLSI Test Symposium (VTS). IEEE, 2018.

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

[37] Y. Liu et al., "Robust and attack resilient logic locking with a high application-level impact," ACM Journal on Emerging Technologies in Computing Systems (JETC), vol. 17, no. 3, pp. 1–22, 2021.

[38] K. Shamsi et al., "Appsat: Approximately deobfuscating integrated circuits," in 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). IEEE, 2017, pp. 95–100.

[39] Y. Shen and H. Zhou, "Double dip: Re-evaluating security of logic encryption algorithms," in Great Lakes Symposium on VLSI, 2017, pp. 179–184.

[40] Y. Shen, A. Rezaei, and H. Zhou, "A comparative investigation of approximate attacks on logic encryptions," in Asia and South Pacific Design Automation Conference. IEEE, 2018, pp. 271–276.

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

[42] M. Yasin, B. Mazumdar, O. Sinanoglu, and J. Rajendran, "Removal attacks on logic locking and camouflaging techniques," IEEE Transactions on Emerging Topics in Computing, vol. 8, no. 2, pp. 517–532, 2017.

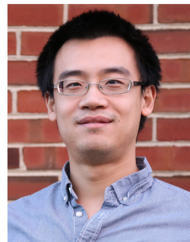
[43] D. Sirone and P. Subramanyan, "Functional analysis attacks on logic locking," IEEE Transactions on Information Forensics and Security, vol. 15, pp. 2514–2527, 2020.

[44] M. E. Massad, J. Zhang, S. Garg, and M. V. Tripunitara, "Logic locking for secure outsourced chip fabrication: A new attack and provably secure defense mechanism," arXiv preprint arXiv:1703.10187, 2017.

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

[46] S. O. Memik, E. Bozorgzadeh, R. Kastner, and M. Sarrafzadeh, "A super-scheduler for embedded reconfigurable systems," in International Conference on Computer Aided Design, 2001.

Michael Zuzak (S'19, M'22) is an Assistant Professor in the Department of Computer Engineering at the Rochester Institute of Technology, Rochester, NY, USA. He received his Ph.D. in Electrical Engineering from the University of Maryland, College Park, MD, USA in 2022. His current research interests include hardware security, computer architecture, and electronic design automation.



Yuntao Liu (S'16, M'21) is an Assistant Research Scientist at the University of Maryland, College Park. He received his Ph.D. in Electrical Engineering from the University of Maryland, College Park, MD, USA in 2021. His research focus is hardware security, including physical unclonable functions, security in emerging fabrication technologies, logic locking, and the security of machine learning hardware.

Ankur Srivastava (S'00, M'02, SM'15, F'23) Dr. Srivastava received his B.Tech in Electrical Engineering from Indian Institute of Technology Delhi in 1998 and PhD in Computer Science from UCLA in 2002. He was awarded the prestigious Outstanding Dissertation Award from the CS department of UCLA in 2002. His primary research interests lie in the field of high performance, low power and secure electronic systems and applications such as computer vision, data and storage centers and sensor networks. He has published numerous papers on these topics at prestigious venues. He has been a part of the technical program & organizing committees of several conferences such as ICCAD, DAC, ISPD, ICCD, GLSVLSI, HOST, and others. He has served as the associate editor for IEEE Transactions on VLSI, IEEE Transactions on CAD and INTEGRATION: VLSI Journal. His research and teaching contributions have also been recognized through various awards.