Trace Logic Locking: Improving the Parametric Space of Logic Locking

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

***Abstract*—To protect against an untrusted foundry, logic locking must 1) inject sufficient error to ensure critical application failures for any wrong key (error severity) and 2) resist any attack against it (attack resilient). We begin our work by deriving a fundamental trade-off between these 2 goals which exists underlying all logic locking, regardless of construction. This relationship forces integrated circuit (IC) designers to sacrifice the error severity of logic locking to increase its attack resilience and vice versa. We proceed by exploring the consequences of this trade-off through architectural simulations of ICs incorporating locking sweeping over the derived parametric space. We find that the efficacy of logic locking is severely limited by this trade-off. In response, we propose trace logic locking (TLL), a novel enhancement of module level logic locking which enables existing art to secure arbitrary length sequences of input minterms, referred to as *traces*. Doing so injects an additional degree of freedom into the parametric space of locking, enabling locking techniques to overcome the limitations of our derived trade-off. We both theoretically and empirically prove this by using TLL to enhance cutting edge locking. In 10 large benchmarks, we show that TLL-enhanced logic locking provides exponentially stronger attack resilience than conventional locking with only modest additional overhead. Finally, we demonstrate the efficacy of TLL in a processor IC using architectural simulations. Despite prior art being unable to secure this IC, we find that TLL concurrently achieves strong error severity and attack resilience.**

***Index Terms*—Trace Logic Locking, Reverse Engineering, Logic Locking, SAT Attack, Untrusted Foundry, IP Piracy**

I. Introduction

Custom IC design has become universal due to the increasing use of embedded systems. As technology advances, the cost to create and maintain the cutting edge facilities necessary to fabricate these devices has substantially increased. As a result, many design companies have been forced to go *fabless*, relying on unaffiliated and untrusted foundries to fabricate their ICs. This has raised security concerns as an untrusted foundry can pirate, counterfeit, or overproduce an IC [1]–[3].

To mitigate these risks, a family of hardware security techniques known as *logic locking* has been proposed [4]–[22]. Logic locking protects ICs from unauthorized use by inserting auxiliary combinational logic into IC modules. This added logic is designed to link locked module functionality to additional primary inputs, known as *key inputs*. By doing so, IC functionality becomes dependent on these key inputs. Only by applying the correct value to key inputs, known as the *secret key*, can correct IC functionality be unlocked. This means that intellectual property (IP) security concerns can be mitigated by withholding the key from untrusted foundries. Doing so renders an IC functionally incorrect for any untrusted foundry. Therefore, by inducing sufficient error within locked modules to ensure critical application failures for any wrong key, logic locking prevents unauthorized use. We consider this to be the first goal of logic locking, referred to as **error severity**.

In response to logic locking, an oracle-based attack, known as a SAT attack, was developed [23] and expanded [24]–[28]. SAT attacks use a Boolean satisfiability solver to iteratively eliminate all incorrect keys from the key-space, thereby locating the secret key. These attacks proved to be quite potent, quickly unlocking most existing locking techniques [4]–[8]. As a result, logic locking began to incorporate strong attack resilience [9]–[18], [20], [21], [29]. This brings us to the second goal of locking, referred to as **attack resilience**, which requires a logic locking technique to resist attacks against it.

Recently, an inverse relationship between the error severity and SAT attack resilience of logic locking has been identified [13], [21], [30], [31]. To explore this relationship, researchers have provided limited derivations of it for specific techniques [13] and loose generic bounds for locking as a whole [21], [30], [31]. Unfortunately, the specificity/weakness of these results provides little insight into the larger ramifications of this underlying relationship. Therefore, we begin our work by rigorously deriving the exact relationship, rather than a loose bound, between the average error injection rate and the number of SAT attack iterations required to unlock logic locking, regardless of technique. Fundamentally, this result provably quantifies and relates the 2 goals of locking, error severity and attack resilience, placing them into direct contention. As a result, once a locking configuration's error severity is fixed, the corresponding SAT attack resilience can be directly quantified (and vice versa). Therefore, our derivation defines provable limits for logic locking, regardless of construction.

We continue by exploring this trade-off. To do so, we use the ObfusGEM simulation framework [32] to simulate 9 benchmarks from the PARSEC [33] benchmark suite on a cycle-accurate GEM5 [34] model of each locked netlist. Our results show that the identified trade-off prevents logic locking configurations with feasible area, delay, and power overheads from achieving error severity and attack resilience. Because this trade-off exists regardless of logic locking scheme, these results not only identify limitations of existing art, but also indicate that novel logic locking techniques (that remain bounded by this same trade-off) will also experience these limitations. This motivated us to explore methods to expand the parametric space of locking.

To do so, we developed trace logic locking (TLL), a novel

This work was supported by the ARCS Foundation and the Air Force Office of Scientific Research Grant FA9550-14-1-0351.

M. Zuzak, Y. Liu, A. Srivastava are with the Department of Electrical and Computer Engineering at the University of Maryland, College Park, MD 20742, USA. Email: {mzuzak, ytliu, ankurs}@umd.edu

enhancement to module level logic locking art which alters existing techniques to secure arbitrary length sequences of input cubes, referred to as *traces*. By doing so, TLL provably injects an additional degree of freedom, namely trace length, into the parametric space of locking. This additional degree of freedom allows existing art to overcome the limits of the derived trade-off between error severity and attack resilience. Our contributions can be summarized as follows:

1) A derivation of the exact relationship between error severity and SAT attack resilience which exists underlying *all* logic locking techniques. This result quantifies the provable security of locking, regardless of construction.
2) An exploration of logic locking at the architecture level. We show that locking with feasible design overheads cannot achieve both error severity and attack resilience.
3) A logic locking enhancement, trace logic locking (TLL), which alters existing combinational locking techniques to expand their parametric space. Doing so allows prior art to overcome the error severity/attack resilience trade-off.
4) A formalized construction of TLL. With this, we provide:
   a) A SAT resilience derivation proving that TLL injects a degree of freedom into locking's parametric space.
   b) An analysis of TLL's resilience to structural attacks.
   c) An empirical analysis of TLL's expanded design space.
   d) A quantification of TLL's design overhead.
5) An evaluation of TLL through architectural simulations of a TLL-secured processor IC. TLL, unlike prior art, simultaneously achieved error severity and attack resilience.

II. Preliminaries

A. Attacker Model

For this work, we use a SAT-capable adversary common in recent logic locking research [9]–[27], [35], [36]. Specifically, we consider an adversary who takes some strategy utilizing:

1) A locked netlist of the IC, C, obtained through reverse engineering the GDSII file.
2) A correctly-keyed, black-box oracle IC, C_o, obtained through the open market or IC testing facilities. The adversary can query the black box oracle with an input and record the correct output, denoted as $y \leftarrow C_o(x)$.

The attacker's goal is to find a key, k, that produces a functional IC. For theoretical derivations, success is defined as locating a key, k, such that $\{\forall x, C(x,k) = C_o(x)\}$.

B. SAT-Based Attacks

In response to logic locking, a Boolean satisfiability attack (SAT attack) was proposed which quickly unlocked most existing logic locking art [23]. The goal of this attack was to locate a key (k) that when applied to the locked IC (C), yielded identical output (y) to an unlocked oracle IC (C_o), regardless of input (x). Hence, a key satisfying $\{\forall x, C(x,k) = C_o(x)\}$.

To perform this attack, we must convert the locked circuit to conjunctive normal form (CNF): $C_{cnf}(x,k,y)$. This form evaluates to true only if an assignment of $x,\ k$, and y can be found such that $y = C(x,k)$. By using a CNF-SAT solver on the CNF circuit, the attack proceeds as follows:

1) An input (x_{di}) and two keys (k_1, k_2) must be found such that when this input is applied to the locked circuit, each key produces a different output (y_1, y_2).
$$C_{cnf}(x_{di}, k_1, y_1) \wedge C_{cnf}(x_{di}, k_2, y_2) \wedge (y_1 \neq y_2) \quad (1)$$
This input, x_{di}, is called a *distinguishing input (DI)*.
2) The DI is applied to C_o and the output, y_{di}, is recorded.
3) During each iteration, a pair of keys k_1, k_2 must be found which produce correct output for all previously located DIs (x_j) along with an additional new DI, x_{di}.
$$C_{cnf}(x_{di}, k_1, y_1) \wedge C_{cnf}(x_{di}, k_2, y_2) \wedge (y_1 \neq y_2) \bigwedge_{j=1}^{i-1} (C_{cnf}(x_j, k_1, y_j) \wedge C_{cnf}(x_j, k_2, y_j)) \quad (2)$$
4) The SAT solver operates on (2) until it is unsatisfiable. This indicates that no further DIs remain. A final key is found which matches the oracle's output on all tested DIs. This key is functionally equivalent to the correct key.

C. Stripped Functionality Logic Locking (SFLL)

SFLL is currently a prominent gate-level locking scheme under the SAT attack model [13]–[15], [22]. At the core of SFLL is the idea of functionality stripping, defined as the incorrect and permanent alteration of the output produced when specific inputs are applied to a locked module. This stripped functionality is then corrected by some added logic, known as the *restore unit*, when a correct key is applied. By functionality stripping more or smaller minterms, the wrong key error rate of an SFLL construction can be modified. This makes SFLL a uniquely tunable locking construction. The work in [13]–[15], [22] introduces multiple locking constructions, however, we focus on SFLL-Fault [15].

SFLL-Fault [15] is a fault injection and ATPG driven strategy to implement the SFLL-Flex construction outlined in [13]. The construction of SFLL-Fault consists of a functionality stripped module and a tamper-proof look-up table (TPLUT) as the restore unit. In SFLL-Fault, the TPLUT is indexed by the secret key. When the input to the locked module matches the index of the TPLUT (secret key), a restore signal is provided which inverts the locked module's output. In the presence of a correct key, the restore signal will correct output errors induced by stripped functionality. In the presence of a wrong key, the restore signal will corrupt correct outputs, causing more error.

III. Deriving the Parametric Space of Locking

We begin our work by identifying and deriving a parametric space which exists underlying *every* logic locking technique. Specifically, we show that an increase in the wrong key error rate of a fixed logic locking construction, while improving error severity, must reduce the average number of SAT queries required to find an unlocking key. As a result, an IC secured with logic locking that simultaneously achieves the highest error severity and SAT resilience can be shown to be to have an infeasible design overhead (Section IV).

While prior work has identified this trade-off for specific techniques [13] or as loose asymptotic bounds applying to more generic logic locking constructions [21], [30], [31], it has

not yet been precisely quantified. Hence, our work constitutes the first derivation that directly quantifies, without reliance on asymptotic bounds, the wrong key error rate (error severity) and SAT attack resilience of any logic locking construction.

Let the input and key of an arbitrary locked module be of length n and $|k|$ bits respectively (2^n total inputs and $2^{|k|}$ total keys). There exists c correct keys for the locking construction, therefore, $2^{|k|}-c$ incorrect keys exist which corrupt the output corresponding to q inputs on average. Each of these inputs, on average, produces corrupt output for x wrong keys. This implies that the average wrong key error rate, ϵ, is $\epsilon = q/2^n$. With this notation, we define Lemma 1.

Lemma 1. *The average number of wrong keys corrupting the output of each input minterm, x, is defined as $x = (2^{|k|}-c)\epsilon$.*

Proof. Given the arbitrary logic locked module which we have defined, let us refer to (x_j, k_i) as a *minterm-wrong key pair (MWP)* if the input minterm x_j produces corrupt output for the wrong key k_i. Based on this, we can write the equation: $|MWP| = (2^{|k|}-c)q = 2^n x$, so, $x = (2^{|k|}-c)\epsilon$ □

Theorem 1. *The expected number of SAT attack queries required to unlock an arbitrary logic locked module, λ, is:*

$$\lambda = \left\lceil \frac{log\left(\frac{2^{|k|}-c-\epsilon(2^{|k|}-c)}{\epsilon(2^{|k|}-c)(2^{|k|}-c-1)}\right)}{log\left(\frac{2^{|k|}-c-\epsilon(2^{|k|}-c)}{2^{|k|}-c-1}\right)} \right\rceil \tag{3}$$

Proof. A SAT-based adversary attacking an arbitrary technique will locate the unlocking key when all wrong keys within the key-space are eliminated. To accomplish this, we assume the SAT-based attacker randomly samples the input space for DIs[1]. The merits of this assumption are discussed in Section VII-A. Therefore, in each SAT query, the attacker selects a DI and removes all wrong keys that corrupt the output for this DI that have not previously been eliminated. Let a_i be the expected total number of wrong keys eliminated up to iteration i. Hence, the expected number of wrong keys eliminated by SAT iteration i is $a_i - a_{i-1}$.

Lemma 1 indicates that each DI enables x wrong keys to be eliminated on average. However, some portion of these x keys could have been eliminated during prior SAT iterations and cannot be eliminated again. This must be addressed. By definition, a DI selection must eliminate *at least* 1 undiscovered wrong key to be valid. Therefore, a given SAT iteration eliminates 1 wrong key and a fraction of the $x-1$ remaining wrong keys that have not been eliminated by prior DIs. Because DIs are randomly selected from the input-space, excluding the 1 wrong key we are guaranteed to eliminate, the likelihood for any wrong key not having been eliminated equals the ratio of # wrong keys that have not been eliminated, $(2^{|k|}-c-a_{a-1}-1)$, to # total wrong keys, $(2^{|k|}-c-1)$. Therefore, $a_i - a_{i-1}$ is defined by:

$$a_i - a_{i-1} = 1 + (x-1)\cdot\frac{2^{|k|}-c-a_{i-1}-1}{2^{|k|}-c-1} \tag{4}$$

[1]An input is not a DI if it does not produce corrupt output for any wrong key. So, if locking never corrupts the output for some input, that input should be excluded from the input space for this derivation as it is not a valid DI

We continue by simplifying the above form:

$$a_i = \beta\cdot a_{i-1} + x \quad \text{where} \quad \beta = 1 - \frac{x-1}{2^{|k|}-c-1} \tag{5}$$

Let us create an intermediate variable, δ, defined as $x = \delta - \beta\delta$, which can be substituted into the above equation:

$$a_i - \delta = \beta\cdot(a_{i-1}-\delta) \quad \text{where} \quad \delta = \frac{x}{1-\beta} \tag{6}$$

Let us define the expected number of SAT queries necessary for a successful attack as λ. Using this, we can form the following set of equations that define the expected number of eliminated keys after each SAT query.

$$\begin{aligned} a_1 - \delta &= \beta(a_0-\delta)\\ a_2 - \delta &= \beta(a_1-\delta)\\ &\vdots\\ a_\lambda - \delta &= \beta(a_{\lambda-1}-\delta) \end{aligned} \tag{7}$$

Prior to launching a SAT attack, no wrong keys are eliminated. This provides an initial condition, $a_0 = 0$, enabling induction to be applied. By induction, we arrive at:

$$a_\lambda = \delta + \beta^\lambda\cdot(a_0-\delta) = (1-\beta^\lambda)\delta \tag{8}$$

For a successful SAT attack, all wrong keys must be eliminated, therefore, $a_\lambda = 2^{|k|}-c$.

$$2^{|k|}-c = \delta + \beta^\lambda\cdot(a_0-\delta) = (1-\beta^\lambda)\delta \tag{9}$$

We substitute for the intermediate terms, β and δ, and solve for λ. λ is a positive integer so we require a ceiling function.

$$\lambda = \left\lceil \frac{log\left(\frac{2^{|k|}-c-x}{x(2^{|k|}-c-1)}\right)}{log\left(\frac{2^{|k|}-c-x}{2^{|k|}-c-1}\right)} \right\rceil \tag{10}$$

Finally, we apply Lemma 1 to arrive at the final form:

$$\lambda = \left\lceil \frac{log\left(\frac{2^{|k|}-c-\epsilon(2^{|k|}-c)}{\epsilon(2^{|k|}-c)(2^{|k|}-c-1)}\right)}{log\left(\frac{2^{|k|}-c-\epsilon(2^{|k|}-c)}{2^{|k|}-c-1}\right)} \right\rceil \tag{11}$$

□

Let us briefly explore an approximate form of this result. Assume that the total number of keys ($2^{|k|}$) is much greater than the number of correct keys (c). If this were not the case, there is a sizable probability that a random key guess would produce a functional IC, making these configurations largely useless. This allows us to assume $2^{|k|}-c \approx 2^{|k|}$. Similarly, let us assume $2^{|k|} \gg 1$, so $2^{|k|}-1 \approx 2^{|k|}$. This yields:

$$\lambda \approx 1 - \frac{log(\epsilon\cdot 2^{|k|})}{log(1-\epsilon)} \tag{12}$$

However, as $\epsilon(2^{|k|}-c) \to 1$, the removed c and -1 terms become increasingly relevant, degrading Equation 12. Equation 12 should be avoided in this case.

A. Understanding the Derived Parametric Space

Prior to analyzing this result, we emphasize its fundamental nature. Because the key length and the number of correct keys are generally fixed locking constraints, Theorem 1 quantifies a direct relationship between 2 primary goals of locking: wrong key error rate (error severity) and SAT resilience. Therefore, in addition to proving that these 2 primary goals are in direct contention, this also enables an IC designer to directly quantify the provable security of their locking technique regardless of construction. By doing so, one can consciously trade between the error severity and SAT attack resilience of locking to ensure the design of a provably secure locking configuration.

To analyze the derived result, we visualize the parametric space created by Theorem 1 as a line in Figure 1. In the figure, we have fixed the key length ($|k|$=16) and number of correct keys (c=1). This was done because key length is generally determined by the allowable design overhead and number of correct keys is determined by the locking construction utilized. We note, however, that the shape of the plot is nearly identical regardless of the value selected for these parameters. Finally, note that the number of SAT attack iterations cannot be larger than the size of a locked module's input space. As described in Section II-B, a SAT-based attacker selects specific input combinations as DIs which are used to eliminate all incorrect keys. Once all possible inputs have been selected, the SAT attack has provably eliminated all incorrect keys. Therefore, regardless of the key length, the number of SAT iterations will never exceed the size of the locked module's input space. These restrictions produce the parametric space in Figure 1.

Despite the opacity of the form of Theorem 1, the resulting parametric space is quite intuitive. The trade-off between error severity and SAT attack resilience can be characterized by a monomial inverse relationship. We note that point-function-based locking techniques (such as [10], [19]) reside at the far right of Figure 1, achieving the maximum possible SAT resilience and minimum possible wrong key error rate. This is unsurprising as these techniques were introduced to maximize SAT attack resilience. On the other hand, high error rate logic locking techniques (such as [4]–[8]) reside at the far left side of Figure 1, achieving a high error rate and an extremely low SAT attack resilience. Once again, this is unsurprising as these techniques were designed prior to the SAT attack [23] and therefore did not consider SAT attack resilience.

Finally, we have evaluated several locking configurations to experimentally support Theorem 1. To this end, we locked an 8-bit adder (n=16) using Anti-SAT [9] and SARLock [10]. Each netlist was then attacked with the open-source SAT attack from [23]. The resulting number of SAT queries to unlock each netlist was compared to the expected number of SAT queries from Theorem 1. We have plotted this comparison in Figure 1. The empirical SAT queries closely match the expected SAT queries. Hence, Theorem 1 appears to correctly quantify the relationship between average error rate and SAT resilience.

B. Understanding SAT Attack Iteration Runtime

Theorem 1 quantifies the number of SAT attack iterations necessary to unlock logic locking. Prior research, such as

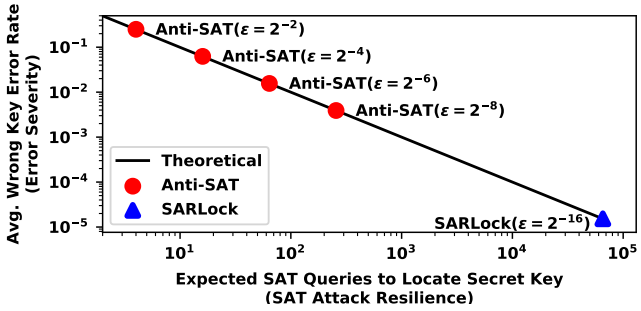

Fig. 1: Inverse relationship between error severity and SAT resilience for logic locking ($|k|$ =16, c=1) from Theorem 1.

[9], [10], [12]–[15], [18], [21], has relied upon this metric to demonstrate SAT resilience. However, recent works, such as Full-Lock [20], have taken an alternative approach to SAT resilience, attempting to make the runtime of successive SAT iterations scale exponentially. To consider this, we expand our view to total SAT attack runtime, modeled by $T_{SAT} = \sum_{i=1}^{\lambda} T(i)$, where $T(i)$ is the runtime of SAT iteration i.

Each SAT attack iteration must solve an NP-complete problem. Hence, no efficient algorithm to solve each SAT query exists, only a variety of heuristics. Thus, the time to solve each SAT query ($T(i)$) is variable and specific to 1) the design topology, 2) the Boolean SAT solver algorithm, and 3) the specifications of the machine running the attack. Hence, the runtime of each SAT query is unpredictable and empirically dependent. Regardless, the derivation in Theorem 1 holds true, even for Full-Lock-style schemes.

An analysis of SAT iteration runtime is necessary for a full view of prior art. However, due to the empirical nature of this metric, we must rely on experimental analysis. So, we have implemented Full-Lock in the $\sim$ 10k gate b14 benchmark from ITC'99 [37]. The resulting SAT attack runtime for each configuration is in Figure 2. Notice that the SAT runtime did increase exponentially in the size of Full-Lock. However, we still unlocked each configuration within 10 minutes, despite the presence of large keys, up to 384 bits. So, despite exponentially increasing SAT runtime, Full-Lock did not exhibit significant SAT resilience for reasonably sized configurations.

We also calculated the overhead of each Full-Lock con-

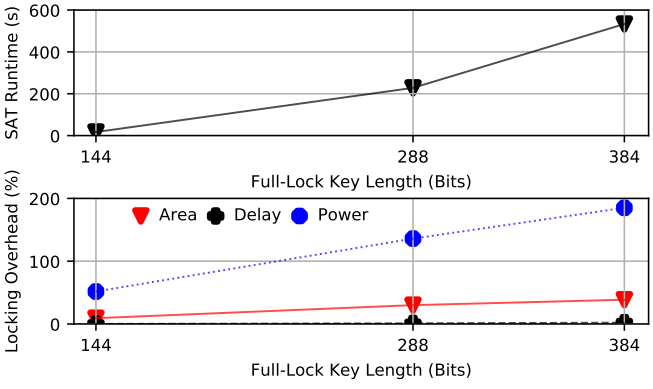

Fig. 2: SAT attack runtime with corresponding area, delay, and power overhead for Full-Lock [20] in b14 netlist.

figuration using the Cadence Encounter RTL Compiler and the Synopsys 90nm SAED library. Full-Lock showed large overheads, with nearly a 200% increase in power and 60% increase in area for the largest tested scheme. This supports our assertion in Section IV that design overhead restricts locking from simultaneously being error severe and SAT resilient. However, both iteration count and time scaling are likely needed for optimal SAT resilience. Our proposed TLL scheme can be paired with either style of locking, so we consider a Full-Lock-style approach to be particularly relevant to TLL.

IV. Exploring the Design Space of Logic Locking

Figure 1 shows that our derived trade-off creates a rigid parametric space between 2 primary goals of locking, error severity and SAT resilience, placing them in contention. Because effective locking must achieve both goals, this raises major security concerns. In this section, we empirically explore the consequences of this trade-off on logic locking art. To do so, we use the tunability of SFLL-Fault [13]–[15] to lock processor ICs with locking that sweeps over our derived parametric space. By evaluating the security of each locking configuration, we explore the resulting design space. We find that the trade-off between error severity and SAT resilience renders logic locking with a feasible design overhead unable to thwart an untrusted foundry attacker.

A. Methods for Architectural Design Space Exploration

To arrive at this assertion, we locked victim netlists with a variety of locking configurations. To select victim netlists, we aggregated the benchmarks used by several logic locking works and assessed commonality [9], [10], [13]. In these works, processor logic constituted 74% of evaluated netlists. Of the processor logic tested, data and control path netlists were roughly equally represented. Therefore, we evaluated both control and data path locking. Specifically, we locked the control and data path of a RISC (MIPS) and CISC (80186) core to provide a cross-section of processor logic.

Locking Location: Within the control path, we locked the instruction decoder because it was the largest control path module in both processors. Within the data path, we locked the adder circuit. This was due to the prevalence of ALU netlists evaluated by prior art (57% of data path benchmarks).

Locking Configuration: Each netlist was locked using SFLL-Fault [13]–[15]. Using the tunability of SFLL-Fault, we incorporated a series of constructions within each benchmark that swept over the derived parametric space. Because this same design space exists underlying all logic locking techniques, this experiment allows us to characterize the design space of logic locking as a whole.

B. Logic Locking Attack Methodology

After locking each netlist, we evaluated their security with respect to both error severity and attack resilience (specifically SAT resilience). To evaluate error severity, we must quantify the usability of each IC in the case of a wrong key. To do so, we used the ObfusGEM simulator [32] to simulate 9 benchmarks from the PARSEC [33] benchmark suite on a cycle-accurate GEM5 [34] model of each locked netlist. We outline our experimental setup in greater detail in Section IX-C. A high failure rate for these benchmarks indicates that a locked IC is thoroughly unusable when a wrong key is applied and therefore exhibits strong error severity guarantees. To measure SAT resilience, we attacked each netlist using an open-source SAT attack [23]. By measuring the iterations and execution time required to recover the secret key, we quantify the SAT susceptibility of each locking configuration.

Control Path: We launched a SAT attack on the locked netlist with the lowest achievable wrong key error rate SFLL-Fault configuration. The evaluated MIPS controller had 16 primary inputs and the evaluated x86 controller had 15 primary inputs (after removing pass-through inputs). Therefore, the evaluated SFLL-Fault constructions utilized a 16-bit and a 15-bit key that stripped a single 16-bit and 15-bit input minterm. This locking configuration corresponds to the highest possible SAT attack resilience achievable by stripping a single minterm with SFLL-Fault. Despite being the largest control circuit, the decoder is small, allowing it to be unlocked via SAT attack.

The runtime of each attack against the control path is in Table I. Each attack successfully located the key within 48 hours. We note that even a worst-case, non-logic-locking-type approach, such as removing and replacing the netlist with a LUT, could only require a maximum of 2^{16} and 2^{15} SAT attack iterations to unlock the circuit based on Theorem 1. This corresponds to the case that each input must be selected as a DI. While not ideal, it is entirely possible to brute-force this number of SAT iterations given the small size of the controller logic. Therefore, because we selected the largest control path circuitry for locking and incorporated the most SAT resilient SFLL-Fault configuration of this form, it appears that SFLL-Fault is unable to protect the control path against a SAT attack.

Data Path: The adder circuit's input size enables extremely low error rate SFLL-Fault configurations. Because SAT resilience is inversely related to wrong key error rate, this implies that extremely strong SAT resilience can be achieved by locking. However, the goal of locking is two-fold. Alongside SAT resilience, locking must also achieve error severity. Because these 2 goals are in contention, we must locate an SFLL-Fault configuration sufficient to achieve both.

To do so, we used our simulation framework to identify the minimum wrong key error rate SFLL-Fault construction capable of achieving error severity. This minimum error rate locking construction corresponds to the maximum achievable SAT resilience for a locking configuration exhibiting error severity. Therefore, if this construction can be unlocked using a SAT attack, a locking configuration capable of simultaneously achieving SAT resilience and error severity does not exist.

We aggregated the results of these simulations for the locked

Locked Circuit	SAT Runtime	
	Control Path	Data Path
MIPS (RISC)	108493 sec	893.2 sec
80186 (CISC)	95193.4 sec	993.4 sec

TABLE I: SAT attack runtime for processor logic.

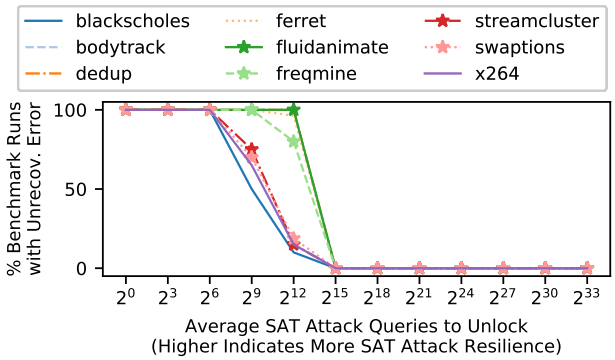

Fig. 3: Empirically derived relationship between error severity and SAT attack resilience in locked 80186 processor data path.

80186 netlist in Figure 3. To visualize the parametric space between the failure rate of common workloads (error severity) and SAT resilience, we have related the workload failure rate of each SFLL-Fault construction to the average number of SAT queries required to unlock it. This results in a map of the parametric space between error severity and SAT resilience. For both netlists, a non-zero workload error rate occurs at a wrong key error rate of 0.02%, corresponding to a 4096 (2^{12}) query SAT attack in Figure 3. This indicates that a minimum error rate exceeding 0.02% is necessary to achieve any error severity. To accomplish this, we designed an SFLL-Fault configuration to strip a 13-bit input using a 13-bit key.

We launched a SAT attack against each netlist configured with this minimal error rate SFLL-Fault construction which still achieved error severity. The runtime of each attack is in Table I. Each attack found the secret key within 1 hour, indicating that a logic locking configuration of this form that is capable of providing both SAT resilience and error severity within the data path of either IC likely does not exist. Note that security could be achieved by greatly increasing the number of stripped inputs, however, doing so requires added restore logic for each stripped input. This makes such an approach infeasible in terms of area, delay, and power overhead. Prior work also notes the infeasibility of this approach [35]. Hence, our results indicate that a logic locking configuration capable of both error severity and SAT attack resilience with a feasible design overhead likely does not exist for this core.

C. Limitations Imposed by the Parametric Space of Locking

Evaluated SFLL-Fault configurations could not achieve both error severity and SAT resilience in either the control path or the data path. This was due to the trade-off between these two objectives. While each netlist was only locked with SFLL-Fault, we have proven that this trade-off between error severity and SAT resilience exists underlying *every* logic locking technique[2] (Section III). Additionally, we used the inherent tunability of SFLL-Fault to design locking configurations which swept throughout the parametric space. Therefore, our result is not a limited example in which cutting edge logic locking was insecure, but a demonstration of the underlying limits imposed on logic locking by its rigid parametric space. As a result, simply proposing novel logic locking constructions (which remain bounded by this trade-off) will do little to overcome these limits. Instead, we must explore ways to expand this parametric space, rather than operate within it.

[2]Logic locking could achieve security by sufficiently scaling key length. For example, portions of the circuit can be replaced with re-configurable logic (e.g. an FPGA), or SFLL-Fault can strip a sizable portion of an IC's inputs. These approaches are a theoretically viable way to achieve error severity/SAT resilience. This can be confirmed by Theorem 1. However, this does not weaken our assertion. These approaches are infeasible due to their tremendous design overhead. Prior work, such as [35], arrives at a similar conclusion.

V. Trace Logic Locking (TLL)

To counter the rigidity of logic locking's parametric space, we developed trace logic locking (TLL), a novel logic locking enhancement which injects an additional degree of freedom into the parametric space of locking. TLL achieves this by locking a sequence, or *trace*, of inputs. This differs from conventional locking which locks a set of inputs. Because both a set and a trace of inputs can be locked simultaneously and independently, TLL can be integrated into any conventional logic locking technique. As we show both theoretically and experimentally in Section VII-A/IX, doing so causes the SAT attack resilience of a TLL-enhanced technique to vary exponentially in locked trace length. This is a major contribution. The derived parametric space of logic locking requires a reduction in error severity for an improvement in SAT attack resilience. However, by utilizing TLL, SAT attack resilience can be achieved by scaling locked trace length, thereby disentangling error severity and attack resilience.

A. Foundations of TLL

Prior to detailing a construction of TLL, let us formalize its notation and intended functionality. Let us assume that conventional locking (e.g. SFLL-Fault) has been applied to some arbitrary combinational module in an IC which receives an input ($x \in X$) on each clock cycle. Additionally, let us assume that some incorrect key (k_i) has been provided to the incorporated locking. In this case, logic locking will corrupt the output of some subset of the input space, $X_i \subseteq X$, such that a fixed incorrect output is produced whenever an input, $x \in X_i$, is applied. The inputs in X_i depend only on k_i.

In this work, we refer to a set of inputs occurring over l clock cycles as a *trace* of length l. TLL is designed to modify a conventional locking construction (e.g. SFLL-Fault) to lock a trace of length l. We refer to this as l-state TLL. To do so, TLL-enhanced locking must corrupt the output of a different set of inputs, $X_i \subseteq X$, on l different clock cycles. This is achieved by injecting the notion of state into conventional locking. Hence, l-state TLL incorporates an l state finite state machine (FSM) within the locking construction where each state corresponds to a unique X_i. Therefore, the FSM's state determines the currently locked inputs. We refer to the set of inputs locked in FSM state m as X_i^m. Hence, when using TLL, X_i^m depends on both k_i and the FSM's state.

B. TLL as a Logic Locking Enhancement

As noted, conventional locking secures a set of inputs (X_i) which are dependent only on the value of k_i. This functionality

is combinational, entirely lacking a sequential component. TLL, on the other hand, is entirely sequential in nature as it secures input traces. Therefore, because conventional locking lacks a sequential component, TLL can be integrated alongside any conventional locking technique. Doing so introduces a sequential component to locking without altering the underlying combinational functionality of the conventional locking technique. We refer to this as *enhancing* a locking technique with TLL. By doing so, a locking construction is created where X_i^m is dependent on both the value of k_i, determined by the conventional locking technique, and the current state of the locking construction (m), determined by TLL.

Presenting TLL as a logic locking enhancement, rather than a unique locking construction, is quite advantageous. It allows TLL to leverage the strongest existing conventional techniques, while still expanding the parametric space of locking. In fact, because TLL only adds a sequential component to a conventional locking construction, it does not modify the underlying combinational functionality of conventional locking at all. This means that TLL expands the parametric space of locking, thereby exponentially improving SAT resilience, while maintaining any other security guarantees (e.g. removal resistance) of the underlying locking technique. However, because TLL is a locking enhancement, its construction depends on the technique it enhances. Moving forward, we utilize SFLL-Fault [13]–[15] to formalize a construction of TLL.

SFLL-Fault was chosen as it is currently the most prevalent logic locking technique which remains unbroken. However, there are limitations to the SFLL family. For example, structural traces unique to SFLL have been shown to be exploitable to reconstruct the secret key [38], [39]. While these traces have not yet been used to successfully unlock SFLL-Fault, they have been used to unlock SFLL-HD, indicating some limitations. While other techniques, such as [11], [16]–[18], [20], have been shown to be strong alternative locking constructions, they have not yet gained the prevalence of SFLL. Therefore, we present a locking construction for TLL based on SFLL-Fault. However, any alternative conventional locking technique could have been utilized as an equally valid backbone for TLL.

C. Comparison of TLL and FSM-Based Locking

Despite TLL's sequential nature, it differs substantially from FSM-based locking schemes, such as [40]–[44]. To distinguish TLL and sequential locking, we note 2 key differences:

1) **Target Circuitry:** FSM-based locking obfuscates an IC by altering its control FSM. To do so, a series of key authentication states are added to the control FSM to validate the key. For a wrong key, the controller enters a permanent obfuscation mode utilizing dummy states with incorrect functionality. For a correct key, the controller enters the intended FSM region, enabling correct functionality. Hence, FSM-based locking schemes obfuscate IC control flow. Rather than modifying the control path, TLL instantiates a separate FSM that directly modifies a combinational logic locking scheme. When the key is incorrect, rather than inducing errant control flow, TLL induces combinational errors in the logic locked module.
2) **Intended Use:** FSM-based locking achieves stand-alone security. While these schemes can be paired with other locking art, they operate independently. TLL must be closely integrated with a logic locking scheme because it cannot induce error on its own. TLL relies on the underlying locking scheme for error, making TLL an enhancement for logic locking, rather than a stand-alone scheme.

This leads to 2 key advantages over FSM-based schemes:

1) **Exponential Improvement in SAT Resilience:** While FSM-based locking can be integrated alongside logic locking, it operates separately, allowing it to be attacked separately. For example, automated reverse engineering attacks [44] can isolate the control FSM to infer the key of FSM-based locking. Similarly, a logic locked module can be isolated and SAT attacked separately from the locked control FSM. TLL integrates tightly into logic locking. As derived in Thm. 2, this requires both schemes be attacked together, exponentially improving SAT resilience.
2) **Reverse Engineering Attack Resilience:** FSM-based locking is susceptible to reverse engineering. In [44], the authors outline these attacks on prominent FSM-based schemes [40]–[43]. These attacks are potent due to the ease of identifying and reverse engineering FSMs, enabling the authors to infer errant control flow, thus the secret key. However, TLL 1) does not target an IC's control FSM and 2) does not rely on errant control flow for security. So, unlike FSM-based locking, TLL's FSM topology does not encode the key, thus FSM reverse engineering is irrelevant.

Therefore, we consider TLL-enhanced and FSM-based locking to be fundamentally different hardware security schemes.

VI. Enhancing SFLL-Fault With TLL

In this section, we formalize a construction of TLL to enhance SFLL-Fault ($\text{TLL}_{SFLL-Fault}$) [13]–[15]. For the remainder of this work, we rely on this construction to evaluate TLL. We begin by introducing a limited example of 2-state TLL which we later generalize to a fully tunable construction.

A. Enhancing SFLL-Fault With 2-State TLL

Assume an arbitrary combinational module receives input, $i \in I$, on each clock cycle. We refer to a sequence of l inputs applied over l clock cycles as a *trace* of length l. Therefore, 2-state TLL locks a trace of length 2. We emphasize that the intended functionality of a TLL locked module must remain combinational despite the sequential nature of TLL's locking.

For the 2-state TLL construction which we introduce in this section, let us assume that we intend to corrupt the output of a single cube within the locked module for each of the 2 TLL states. To illustrate this, let us arbitrarily refer to the locked trace inputs as i_0 followed by i_1. In this case, i_0 *or* i_1 produces incorrect output in the locked module for a given clock cycle, never both. The currently locked input switches between i_0 and i_1 whenever the input to the module matches a portion of the secret key. This yields functionality such that the order of locked inputs in the trace is critical, but the number of cycles between locked inputs is irrelevant. Figure 4A shows a block diagram of 2-state TLL.

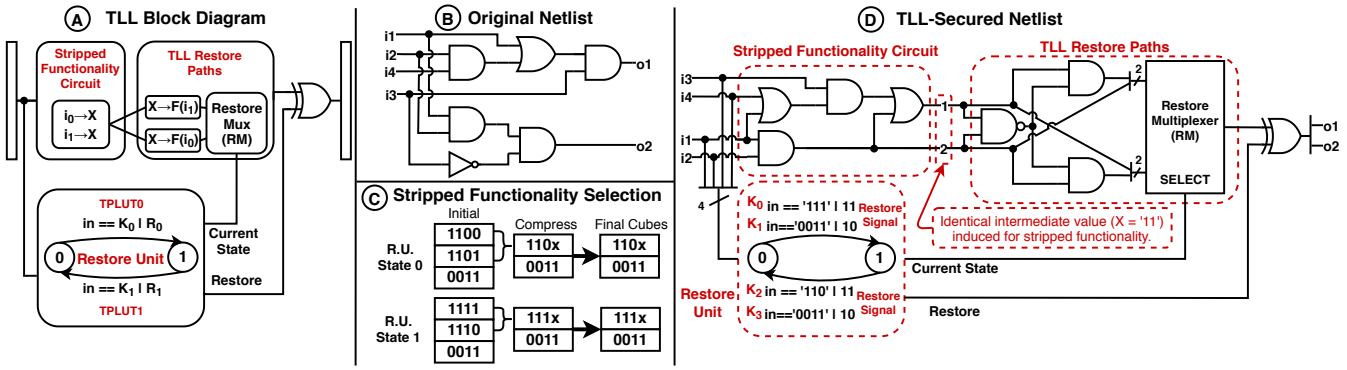

Fig. 4: 2-state TLL-secured module. A) Block diagram of 2-state TLL for input sequence "i_1, i_0". B) Original c17 netlist. C) Stripped functionality selection and compression for TLL. D) C17 netlist secured with 2-state TLL.

From the block diagram, notice that 2-state TLL consists of a stripped functionality (SF) module, a restore unit, and an XOR gate. The functionality of these components relies on a secret key, denoted as the concatenation of two independent *subkeys*, $k = (k_1, k_0)$. The correct secret key corresponds to the locked inputs of the trace, $k = (i_1, i_0)$. For the remainder of the section, we describe the functionality of TLL in depth by considering each of its 3 components.

SF module: SF is defined as the re-design of a given set of inputs within a combinational module to produce incorrect output. In 2-state TLL, the SF module contains 2 SF inputs corresponding to the locked trace, namely the inputs i_0 and i_1. However, we note that 2-state TLL only has 1 SF input enabled within the design during each clock cycle. To achieve this, both SF inputs in the module are mapped to a common intermediate value, X, rather than two separate and unrelated incorrect outputs as is done by [13]. This intermediate value is then mapped to the correct output for *either* i_0 or i_1. Finally, because both inputs are mapped to the same intermediate, X, it is impossible for both inputs to have correct output during a given clock cycle. To select between mapping X to the correct output for i_0 or i_1, additional logic called the "restore multiplexer" (RM) is added. Specifically, the select line of the RM determines whether i_0 or i_1 is mapped to correct output. This select line is controlled by the restore unit state.

Restore Unit: The restore unit is located below the SF module in Figure 4A and consists of a 2-state finite state machine (FSM). As noted, the current state of this FSM controls the select line of the RM, hence, the restore unit determines the current SF inputs within the locked module. State transitions occur within this FSM when the current input to the locked module matches one of the two secret subkeys (k_0 or k_1), with k_0 used in state 0 and k_1 used in state 1. Hence, the state sequencing of this FSM is determined by the secret subkey corresponding to TLL's restore unit state. Building upon the foundation built by SFLL-Fault [15], this secret subkey should be stored within a tamper-proof look-up table (TPLUT) for protection. This TPLUT, as defined in [13], contains a secret subkey (k_0 or k_1), acting as the index, and a restore signal, acting as the output. Additionally, when a state transition occurs (i.e. when the currently active secret subkey matches the input to the locked module), a restore signal (R_s) is applied to TLL's XOR gate, altering the module's output.

When properly keyed, the TLL restore unit will correct output corruption induced by SF inputs by applying the restore signal to the XOR gate located at the output of the SF module. In the case of a wrong key, the restore unit will inject error by applying a restore signal which corrupts otherwise correct outputs corresponding to non-SF inputs. In this case, output corruption is present in the locked IC not only for SF inputs, but also for these non-SF inputs.

XOR Gate: The final component of 2-state TLL is the XOR gate on the output of the locked module. As noted previously, when a restore unit state transition occurs, a restore signal is applied to this XOR gate which modifies module output.

Note that a TLL-secured module remains combinational. Only the restore unit is sequential. Additionally, only 1 input in the trace has SF on a given clock cycle. The SF input switches when the currently enabled secret subkey matches the input to the module, thus transitioning the restore unit FSM to the next state. This allows 2 separate locked inputs to exist without any increase in the wrong key error rate of the locked module for a given clock cycle. As we show in Section VII, by utilizing TLL to enhance SFLL-Fault, locking constructions can be created which exhibit equivalent error rates and exponentially stronger SAT resilience than SFLL-Fault alone. This allows TLL to expand the parametric space of SFLL-Fault. We summarize our construction of TLL-enhanced SFLL-Fault below.

1) TLL protects a sequence of 2 inputs, i_0 followed by i_1
2) At any given time, only a single SF input is locked
3) The RM select line dictates which SF input is locked
4) The restore unit corrects SF errors when the secret subkey matches the current SF input

B. Example $TLL_{SFLL-Fault}$ Implementation

To clarify TLL's implementation, we have locked the c17 circuit from ISCAS'89 [45] in Figure 4B-D. The un-locked c17 circuit is shown in Figure 4B. To implement 2-state TLL, the designer first selects candidate SF inputs for each TLL restore unit state. In Figure 4C, we have selected 3 candidate SF minterms for each state: (1100, 1101, 0011) and (1111, 1110, 0011) for restore unit state 0 and 1, respectively. Notice that the same inputs can be selected as SF inputs in multiple

states (e.g. 0011). Now, we can optionally compress these minterms into smaller cubes to reduce design overhead. In Figure 4C, the minterms (1100, 1101) and (1111, 1110) are compressed into single cubes during this compression step.

Given our list of SF cubes, we can implement TLL in the c17 netlist with the following process. First, each SF input must be mapped to some identical intermediate value ('X'). In this case, 'X' is '11'. With 'X' defined, we can perform standard combinational optimization to synthesize the SF circuit. The resulting SF circuit is labeled "Stripped Functionality Circuit" in Figure 4D. Now, 2 restore paths, one for each restore unit state, must be added to the output of the SF circuit to correct the SF inputs. Hence, in restore unit state 0, the intermediate value, '11', must be mapped to the intended output '01' and in restore unit state 1, '11' must be mapped to '10'. These paths are denoted "TLL Restore Paths" in Figure 4D. At their output, a 2-input multiplexer, called the "Restore Multiplexer", connects these 2 paths to an XOR gate added at the output of the module.

Finally, we add TLL's restore unit. To do so, an FSM with a single state for each trace index must be included (i.e. 2 states). In Figure 4D, we have labeled TLL's restore unit as "Restore Unit". The state of the restore unit dictates the currently active restore path by driving the select line of the RM. Notice that the secret subkeys applied to the restore unit dictate its functionality. Whenever the active subkey matches the primary input, the restore unit FSM changes its state and applies a restore signal to the XOR gate at the module's output. For a correct key, this signal corrects errant outputs due to SF. For a wrong key, this signal corrupts otherwise correct outputs.

C. Example 2-State TLL Functionality

Assume that the 2-state TLL configuration in Figure 4A is used to lock a 2-bit input module. Therefore, the possible input space can be represented by i_3, i_2, i_1, i_0. Within this module, the trace i_0 followed by i_1 is locked. This configuration would yield correct functionality with the key $k = i_1, i_0$ and incorrect functionality otherwise. An error map exhaustively describing this 2-state TLL-secured circuit is in Table II.

The top row of this table enumerates the possible TLL secret keys as a combination of 2 subkeys. Below this row, each key combination is enumerated in the form: {subkey for restore unit state 1, subkey for restore unit state 2}. On the side of the table, each possible restore unit state and primary input combination is enumerated. As an example, let us assume we want to know TLL's output in the following situation: the restore unit is in state 1, the applied key is i_2, i_0, and the primary input is i_2. To do so, we find the intersection of the row for the current restore unit state/primary input value and the column corresponding to the current secret key. This intersection is a green-shaded cell in Table II. The ✗ in this cell indicates that TLL would produce an errant output in this scenario. Alternatively, if the primary input i_3 were applied, instead of i_2, the corresponding a ✓ symbol indicates that TLL provides correct output for this scenario.

R.U. State	**in**	**key input $\mathbf{k = k_1, k_0}$**							
		i_0, i_0	i_0, i_1	i_0, i_2	i_0, i_3	i_1, i_0	i_1, i_1	i_1, i_2	i_1, i_3
0	i_0	✓	✗	✗	✗	✓	✗	✗	✗
0	i_1	✓	✗	✓	✓	✓	✗	✓	✓
0	i_2	✓	✓	✗	✓	✓	✓	✗	✓
0	i_3	✓	✓	✓	✗	✓	✓	✓	✗
1	i_0	✗	✗	✗	✗	✓	✓	✓	✓
1	i_1	✗	✗	✗	✗	✓	✓	✓	✓
1	i_2	✓	✓	✓	✓	✓	✓	✓	✓
1	i_3	✓	✓	✓	✓	✓	✓	✓	✓

R.U. State	**in**	**key input $\mathbf{k = k_1, k_0}$**							
		i_2, i_0	i_2, i_1	i_2, i_2	i_2, i_3	i_3, i_0	i_3, i_1	i_3, i_2	i_3, i_3
0	i_0	✓	✗	✗	✗	✓	✗	✗	✗
0	i_1	✓	✗	✓	✓	✓	✗	✓	✓
0	i_2	✓	✓	✗	✓	✓	✓	✗	✓
0	i_3	✓	✓	✓	✗	✓	✓	✓	✗
1	i_0	✓	✓	✓	✓	✓	✓	✓	✓
1	i_1	✗	✗	✗	✗	✗	✗	✗	✗
1	i_2	✗	✗	✗	✗	✓	✓	✓	✓
1	i_3	✓	✓	✓	✓	✗	✗	✗	✗

TABLE II: Error map for a module with a 2-bit primary input secured with 2-state TLL. The locked trace is $in = i_1, i_0$. Correct key functionality is denoted with highlighted cells.

D. A Generalized Construction of $TLL_{SFLL-Fault}$

We initially presented a simplified construction of TLL-enhanced SFLL-Fault as a special case. A more generalized form offers the IC designer scalability in both wrong key error rate and locked trace length. Expanding to the generalized $TLL_{SFLL-Fault}$ construction relies on the same principles of functionality stripping and a sequential restore unit. We discuss the modifications necessary to tune each of these parameters separately, but present a unified $TLL_{SFLL-Fault}$ construction that enables scaling in both trace length and error rate. A block diagram of SFLL-Fault enhanced with TLL is in Figure 5.

Scaling $\mathbf{TLL_{SFLL-Fault}}$ Error Rate

The IC designer can scale the error rate of this TLL construction by incorporating additional SF inputs within the locked module for a restore unit state. The functionality stripping of additional inputs leads to a higher error rate which yields increased output corruption in an improperly keyed IC. To compensate for the added SF inputs, the restore unit must be modified to locate and restore newly locked inputs.

To modify the restore unit, we incorporate additional subkeys to match the additional SF inputs. In the worst case, a new subkey must be included for each SF input. However, combinational optimization techniques can be applied to combine subkeys or reduce the number of bits and hence reduce hardware overhead. See [13]–[15], [46] for proposed security aware synthesis algorithms which enable the combinational optimization of TLL.

Error rate can also be scaled by decreasing the length of the subkey. A shorter subkey matches fewer bits of the primary input thereby increasing the percentage of locked inputs within the module. Because we now consider the possibility of a subkey length (s) less than the length of the primary input (n), error rate becomes slightly more complex. We define the number of subkeys currently being compared to the primary input to be c throughout the remainder of this work. This means the error rate of the locked module is $c \cdot 2^{n-s}/2^n$.

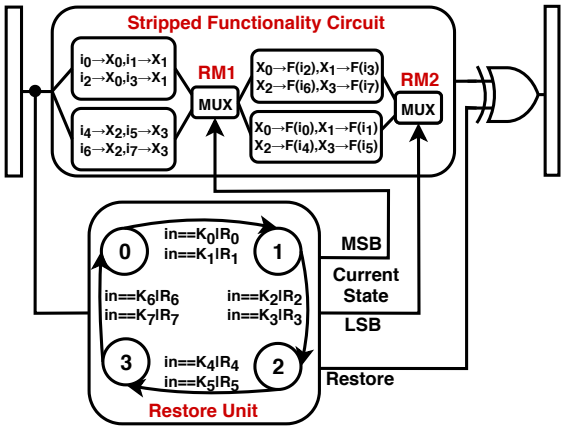

Fig. 5: 4-state TLL configuration with 2 locked primary inputs per cycle. Secured input sequence: $i_0 \vee i_1, i_2 \vee i_3, i_4 \vee i_5, i_6 \vee i_7$.

The described modifications have been applied within Figure 5. Notice that during any given restore unit state, 2 primary inputs are stripped within the locked module. This modification provides double the error rate of 2-state TLL in case of a wrong subkey for any state.

Scaling TLL$_{\mathbf{SFLL-Fault}}$ Trace Length

The length of the trace secured by TLL can be expanded from the presented 2-state case as well. Note that by expanding the trace length, exponentially stronger security guarantees against SAT-based attackers can be provided. See Section VII for proof of this claim. In order to expand locked trace length, our construction must include more restore unit states. The number of states and the length of the restricted trace must be equal. $\lceil log_2(l) \rceil$ restore multiplexers (RM1 and RM2 in Figure 5) must be included in the design as well. These RMs determine the currently exposed SF inputs in the module.

By combining these 2 modifications, we create a variable trace length TLL construction. An example of a TLL construction for a locked trace of length 4 is presented in Figure 5. Notice that the most significant bit (MSB) of the restore unit's state dictates which SF inputs are currently exposed in the module (i_0, i_1, i_2, i_3 with state 0/1, or i_4, i_5, i_6, i_7 with state 2/3). The least significant bit (LSB) of the state controls RM2. Depending on the select line of RM2, half of the stripped functionality created by RM1 is restored. Whenever any secret subkey matches the current input to the module, a state transition occurs. This causes a restore signal to be applied to the XOR gate which either corrects or corrupts the output depending on whether the secret subkey is correct.

Generalized Tunable TLL$_{\mathbf{SFLL-Fault}}$ Construction

A block diagram of a TLL$_{SFLL-Fault}$ construction scaled in both trace length and error rate is contained in Figure 5. It differs from the 2-state case contained in Figure 4 in 3 ways:

1) Increased states in restore unit (trace length scaling)
2) Additional restore multiplexer, RM1 (trace length scaling)
3) Additional SF inputs in locked module (error rate scaling)

By modifying these details, both the wrong key error rate and the locked trace length of our TLL construction can be altered yielding tunable security guarantees.

VII. Mathematical Foundations of TLL$_{SFLL-Fault}$

The goal of TLL is to expand the parametric space of locking. As shown in Section III, this parametric space is created by the trade-off between wrong key error rate (error severity) and SAT attack resilience. To this point, we have argued TLL achieves this by injecting trace length into this trade-off, thereby disentangling the direct relationship between error severity and SAT resilience. In this section, we prove this claim for our presented TLL construction. Specifically, we show that the SAT resilience of our presented construction varies in both wrong key error rate (as is the case with all conventional locking) and trace length. Proving this assertion means that wrong key error rate can be increased (improving error severity) without degrading SAT resilience by increasing locked trace length. Hence, TLL injects trace length into the parametric space of locking, thereby expanding it.

The secret key for our TLL construction, $k = k_{l-1}, k_{l-2}, ..., k_0$, is a concatenation of several independent keys corresponding to separately locked primary inputs within the locked trace of length l. We will refer to each of these concatenated keys as *subkeys* of length $c \cdot s$ bits, where c is the number of locked inputs for each position in the trace and s is the length of each locked input. Each of these subkeys correspond to a particular position within the locked trace. Additionally, each position in the locked trace corresponds to a state within the TLL restore unit. Without loss of generality, we will assume that k_0 corresponds to restore unit state 0, k_1 corresponds to restore unit state 1, and so on. For brevity, each TLL construction is presented as a triplet, TLL(s,l,c).

A. SAT Resilience of TLL$_{SFLL-Fault}$

The SAT resilience of a logic locking technique is defined as the probability of a SAT attack successfully recovering the secret key within q queries. To derive this, we assume that the attacker uniformly samples the input space for DIs. Previous research in logic locking has relied upon this assumption in attack resilience proofs such as [12], [13], [21]. We experimentally verify our resulting derivations in Section IX to ensure these assumptions are reasonable. Note that despite a focus on the SAT attack presented in [23], our result holds against a series of other SAT-based attackers such as [24]–[28].

The goal of a SAT-based attacker is to select all c SF inputs within the locked module as DIs for each of the $i = 1..l$ indices of the trace. By selecting each of the c SF inputs for a given trace index, all possible wrong subkeys for that trace index are eliminated from the keyspace. If the attacker does this for all l subkeys, corresponding to the l trace indices (restore unit states), all wrong subkeys will be eliminated. By concatenating the remaining subkeys, the adversary constructs the correct secret key. We note this approach constitutes a so-called "unrolling" attack where TLL's state machine is unrolled into a series of combinational locking configurations which are then solved by a SAT attack.

We begin characterizing SAT resilience with a derivation of the probability of selecting all c SF inputs within q selections for a given trace index by uniformly sampling the input space.

This corresponds to the necessary SAT queries to locate a secret subkey by the above methodology.

Lemma 2. *The probability of selecting all c SF inputs from an input-space of size 2^s within q queries is $P = \binom{2^s-c}{q-c}/\binom{2^s}{q}$*

Proof. We have a budget of q queries. An input combination of length s constitutes a query. Any input combination is sampled with equal probability. Therefore, there are $\binom{2^s}{q}$ possible ways of doing this. A successful attack is the case in which all c SF inputs are selected in these q queries. Regardless of which order these c inputs are selected, c out of the q queries must be SF inputs. Hence, $q-c$ must be from the set of non-SF inputs. The number of possible non-SF inputs is $2^s - c$. The number of ways in which $q - c$ selections can be made from these $2^s - c$ non-SF choices is $\binom{2^s-c}{q-c}$. Therefore, the probability of locating all c SF inputs within q queries is:

$$P = \binom{2^s - c}{q - c} \Big/ \binom{2^s}{q} \quad , \quad (q \geq c) \tag{13}$$

□

Using this result, we derive the SAT resilience of TLL(s,l,c).

Theorem 2 (SAT Resilience of TLL)**.** *The probability of an adversary unlocking a TLL(s,l,c) locked module within q SAT queries per restore unit state is $P = (\binom{2^s-c}{q-c}/\binom{2^s}{q})^l$.*

Proof. As discussed, the goal of a SAT-based attacker is to select all c SF inputs as DIs for each of the $i = 1..l$ indices of the trace. This process recreates the entire secret key. This is because selecting all c SF inputs for each trace index, i, as a DI eliminates all possible wrong subkeys for all i. The remaining subkeys can be concatenated to yield the correct secret key. To accomplish this, the attacker proceeds as follows.

1) Within the locked module, initialize the restore unit state to 0. Note that the restore unit state corresponds to a specific index of the trace. Additionally, all the SF minterms, c, for each trace index, i, are independent. Hence, the SAT attack for each index can occur independently of other indices.
2) The attacker applies a SAT attack against the module in restore unit state 0. On termination, the attack returns the secret subkey for restore unit state 0. The probability of finding the correct subkey in q_0 SAT queries is given by Lemma 2: $P_0 = \binom{2^s-c}{q_0-c}/\binom{2^s}{q_0}$.
3) The remaining $l-1$ restore unit states must be attacked to find the remaining $l-1$ secret subkeys. This unlocks the circuit as a whole. For each remaining restore unit state ($i = 2 \dots l$), the adversary will initialize the restore unit to that state and repeat the attack. The probability of finding the correct subkey in q_i SAT queries for the i-th trace index is given by Lemma 2: $P_i = \binom{2^s-c}{q_i-c}/\binom{2^s}{q_i}$.

This methodology reconstructs the secret key for a TLL locked module. Therefore, the probability of reconstructing the secret key of a TLL(s,l,c) locked module within q SAT queries per restore unit state is:

$P = \prod_{i=0}^{l-1} P_i = \prod_{i=0}^{l-1} \binom{2^s-c}{q_i-c}/\binom{2^s}{q_i}$ or, equivalently:

$$P = \left(\frac{\binom{2^s-c}{q-c}}{\binom{2^s}{q}} \right)^l, \; q = q_0 = ... = q_{l-1} \tag{14}$$

□

Theorem 3. *The probability of an adversary unlocking a TLL(s,l,c) locked module in q SAT queries per restore unit state exponentially decays in the length of the trace, l.*

Proof. From Theorem 2, the probability of unlocking TLL(s,l,c) within q queries per restore unit state is:

$$P = \left(\binom{2^s - c}{q - c} \Big/ \binom{2^s}{q} \right)^l \tag{15}$$

We can expand this form:

$$= \left(\frac{(2^s - c)!(2^s - q)!q!}{(q - c)!(2^s - q)!(2^s)!} \right)^l = \frac{q^l(q-1)^l...(q-c+1)^l}{(2^s)^l(2^s-1)^l...(2^s-c+1)^l}$$

For a successful SAT attack, $c \leq q \leq 2^s$. Hence, $q/2^s \leq 1$, $(q-1)/(2^s-1) \leq 1$,...,$(q-c+1)/(2^s-c+1) \leq 1$. Therefore,

$$\frac{q(q-1)...(q-c+1)}{(2^s)(2^s-1)...(2^s-c+1)} \leq 1 \tag{16}$$

This means that Equation (15) exponentially decays in l. □

Let us approximate Theorem 2 for clarity. To do so, assume that the number of SF inputs (c) is small. Large numbers of SF inputs quickly yield infeasible design overheads. This means $q!/(q-c)! \approx q^c$ and $(2^s - c)!/2^s! \approx 2^{-c \cdot s}$, yielding the form:

$$P \approx \left(\frac{q}{2^s} \right)^{c \cdot l} \tag{17}$$

With this result, we return to our motivation: the trade-off underlying logic locking between error severity and SAT attack resilience requires locking configurations capable of protecting ICs from an untrusted foundry attacker to have an infeasible design overhead. This is because the SAT resilience of logic locking only scales efficiently in wrong key error rate (Section IV). As shown by Theorem 1, this result applies to all logic locking. However, we designed TLL to inject another parameter, trace length, into logic locking schemes that otherwise lack this parameter. When we consider the impact of TLL's trace length in Theorem 2, we find that it yields a locking scheme where SAT resilience is dependent on both wrong key error rate *and* trace length. Because the SAT susceptibility of TLL exponentially decays in the length of the trace, it can be used to efficiently achieve SAT attack resilience and error severity. This ensures that when an IC designer fixes some wrong key error rate necessary for error severity, they can always choose a value of l where SAT resilience is also guaranteed. This constitutes an expansion of the parametric space of logic locking and is the primary contribution of TLL.

To conclude, we note that our derivations are specific to the $\text{TLL}_{SFLL-Fault}$ construction. However, any conventional locking enhanced with TLL will exhibit the same functionality. Namely, a locking construction whose currently locked inputs depend on both the restore unit state and the applied wrong key. Because the underlying functionality is identical, the

ramifications of TLL will also be identical. Hence, TLL will create the same increase in SAT attack resilience for any trace length scaling, expanding the parametric space of locking.

B. Removal Resistance of $TLL_{SFLL-Fault}$

There have been several proposed removal-type attacks against logic locking techniques such as [46], [47]. When considering $TLL_{SFLL-Fault}$, these attacks can be split into 2 categories: SF removal and restore unit removal. In deriving the resistance of TLL to each of these attacks, we assume the designer has incorporated TLL into the locked module using a security aware synthesis algorithm, such as [46].

First, we consider a removal attack on the SF module. Because SF is added to a module through re-design, it cannot be removed through logic removal. A re-design attack is needed. We defer our analysis of this more complex attack to Section VII-C. Next, we consider $TLL_{SFLL-Fault}$(s,l,c)'s security against a restore unit removal attack. In this attack, the attacker has located and removed TLL's restore unit.

Theorem 4 (Restore Unit Removal Attack Resistance of TLL)**.** *In the case of a restore unit removal attack, $c \cdot 2^{n-s}$ errors remain in a TLL(s,l,c) locked module.*

Proof. Assume that an adversary has found and removed all restore unit logic within a TLL-secured module. The remaining circuit contains any errors induced through SF minterms. There exists $l \cdot c$ SF minterms in the locked module which induce $lc \cdot 2^{n-s}$ errors. $(l-1)c \cdot 2^{n-s}$ of these errors are corrected by the current RM state yielding $c \cdot 2^{n-s}$ errors present within the module after a restore unit removal attack. □

Finally, we consider the case in which $TLL_{SFLL-Fault}$'s state is fixed (e.g. by removing alternate states). We note that any active SF in the circuit can only be corrected by a restore unit state transition. Hence, an attack which fixes TLL into a single state is unable to restore any SF inputs in the circuit. This means that Theorem 4 applies in this case as well and $c \cdot 2^{n-s}$ errors are present in the module. Additionally, because TLL cannot restore errors without changing states, these errors cannot be recovered, regardless of the secret key applied.

C. Re-Design Resistance of $TLL_{SFLL-Fault}$

Let us address a re-design attack on $TLL_{SFLL-Fault}$, where the adversary alters the logic of a locked gate-level netlist to unlock it. Note that this attack is outside of the scope of a more traditional SAT-based attacker model, such as ours (or those proposed in [9], [10], [18], [21]), as it requires significant alteration of the IC's underlying logic. We discuss such an approach because it can be used to weaken $TLL_{SFLL-Fault}$, but it cannot fully unlock the circuit and is costly.

Let us consider an adversary who has located TLL's FSM using the semi-automated reverse-engineering techniques outlined in [44]. Removing this FSM results in $c \cdot 2^{n-s}$ unrecoverable errors (Theorem 4). To correct these errors, the attacker can re-design the gate-level netlist to include additional restore logic. Without loss of generality, let us assume a LUT is added with an entry for each of the c SF inputs. This yields

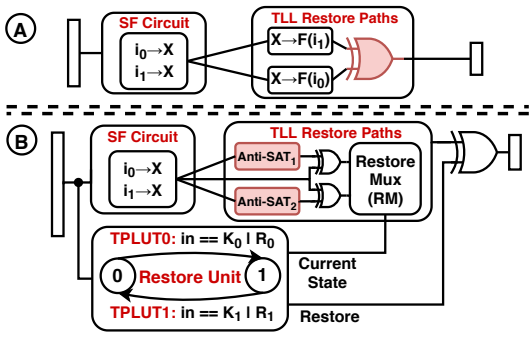

Fig. 6: A) Configuration of structural miter attack on $TLL_{SFLL-Fault}$ B) Miter-attack-resistant $TLL_{SFLL-Fault}$.

a topology consistent with SFLL-Fault [15]. Hence, if the c SF inputs are located, they can be corrected through this added LUT. This approach bypasses the sequential aspect of $TLL_{SFLL-Fault}$, but there remains 3 key limitations:

1) The circuit is not unlocked. $c \cdot 2^{n-s}$ SF-induced errors remain. To correct these errors, each SF input must be located and entered in the added LUT. A SAT attack can locate these SF inputs, but its complexity scales exponentially.
2) This attack requires netlist modification. After removing TLL's FSM, a LUT must be added. Gate-level changes force the attacker to layout, close timing, verify, etc. the IC. Performing a new tape-out is resource intensive.
3) A new mask must be created to overbuild/counterfeit the IC. This is costly. If the modified mask is used to fabricate ICs for the design house, device tampering is obvious both 1) functionally, through the presence of logically irrelevant key-bits, and 2) visually (i.e. de-layering), through major changes in key logic. This provides a watermark and allows the designer (or any future IP user) to detect tampering.

Thus, the limited and costly nature of such an attack against $TLL_{SFLL-Fault}$ makes its utility and profitability doubtful.

D. Structural Resilience of $TLL_{SFLL-Fault}$

Now, we consider a structural attack against $TLL_{SFLL-Fault}$ launched as follows: 1) locate the restore multiplexer (RM), 2) replace it with a XOR gate to create a miter, and 3) use a SAT tool to locate differences between restore paths. These differences are SF inputs, which can be used to construct secret subkeys. Notice that this attack exploits the fact that one can infer the key of SF-based locking constructions through knowledge of SF inputs. Hence, such an attack is only valid against TLL extensions of SFLL-style techniques. This attack is shown in Figure 6A.

In a base $TLL_{SFLL-Fault}$ locking configuration, such an attack is successful. To ward against it, a designer can incorporate additional combinational locking within each restore path. Let us consider replacing each restore path with a buffer locked by an Anti-SAT block [9], as shown in Figure 6B. In this case, the key applied to each Anti-SAT block determines the functionality of each restore path. Hence, the suggested miter attack now only identifies currently corrupted inputs for each Anti-SAT block, rather than the SF inputs in the circuit.

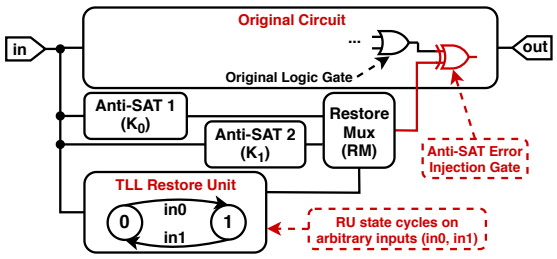

Fig. 7: 2-state TLL-enhanced Anti-SAT construction.

This modified design no longer encodes SF inputs in restore paths, hence, no longer leaks the SF minterms.

To unlock this modified configuration, the adversary still must determine all SF inputs. With this knowledge, they can 1) determine a correct key for each Anti-SAT block, which restores the necessary SF inputs for each restore path and 2) set each TLL subkey to restore the remaining SF cubes. Thus, the SAT resilience derived in Thm. 2 holds and $\text{TLL}_{SFLL-Fault}$ maintains security against this miter-based structural attack.

VIII. Enhancing Alternative Techniques With TLL

TLL is an enhancement of existing logic locking art. However, to this point, we have only provided a single concrete TLL construction, $\text{TLL}_{SFLL-Fault}$. Therefore, to show how TLL can be used to achieve equivalent functionality in alternative locking techniques, we present another example of TLL-enhanced logic locking, $\text{TLL}_{Anti-SAT}$. A block diagram of $\text{TLL}_{Anti-SAT}$ is included in Figure 7.

In the figure, 2 Anti-SAT blocks are incorporated within the circuit. These blocks serve as the core element of Anti-SAT locking, injecting error within the circuit for a specific, key-driven minterm [9]. Note that each Anti-SAT block has been provided an entirely independent subkey (K_0, K_1), which serves as the secret key of $\text{TLL}_{Anti-SAT}$. A restore multiplexer has been incorporated to select the Anti-SAT block currently injecting error within the circuit. This multiplexer is controlled by the current state of the TLL restore unit, which switches between states on 2 arbitrary (e.g. randomly selected) input minterms (in0, in1).

This construction uses 2 Anti-SAT blocks to lock a trace of length 2. Because each Anti-SAT block employs an independent key, each index in the trace corresponds to a different secret subkey. By applying the approach in Theorem 2, we can show that increases in trace length exponentially scale the SAT resilience of Anti-SAT. At the core of each presented construction is the same underlying principles and structures, which allow TLL to provably expand the parametric space of the underlying locking scheme. To enhance an arbitrary locking scheme the following criteria must be met:

1) l independent logic locking configurations with independent keys must be integrated into a single module.
2) An FSM must be integrated into this same netlist, which enables only 1 of the l independent locking configurations for each state. Each of the l locking configurations must be enabled in at least one state.

Notice that the 2 presented TLL schemes meet these criteria:

- $\textbf{TLL}_{\textbf{SFLL-Fault}}$**:** 1) l complete SFLL-Fault configurations are integrated. Each of the l configurations use an independent subkey and separate SF inputs. 2) Each TLL state enables a separate subkey and set of SF inputs.
- $\textbf{TLL}_{\textbf{Anti-SAT}}$**:** 1) l Anti-SAT blocks are added to the circuit with independent keys. 2) Each TLL state enables only one of the l Anti-SAT blocks present in the design.

By the approach in Theorem 2, a logic locking scheme that meets these criteria can be shown to exponentially increase SAT resilience as trace length is scaled. Thus, a scheme that meets these criteria has been successfully enhanced by TLL.

IX. Experimental Analysis of $\text{TLL}_{SFLL-Fault}$

We provide an experimental analysis of $\text{TLL}_{SFLL-Fault}$. This analysis is broken into 3 components. First, we demonstrate that TLL expands the parametric space of logic locking by validating its theoretical SAT resilience in a series of benchmarks. Our results show that the empirical SAT resilience of TLL-locked circuits closely match the theory derived in Section VII. Second, we use the same benchmarks to characterize TLL's overhead and the effects of trace length scaling. Based on our experiments, we found that TLL achieves exponentially stronger security than a comparable SFLL-Fault configuration while only incurring an overhead of +3.8%, -0.8%, and +3.5% for area, delay, and power. Finally, we provide an architectural example of TLL by locking the 80186 processor netlist that was un-securable using prior art. Using our locking methodology, we show that TLL can simultaneously achieve both error severity and SAT resilience with detailed architecture-level simulations of the locked IC. For these experiments, we used the 10 largest benchmarks of ISCAS'89 [45] and ITC'99 [37] to evaluate TLL. These netlists are identical to those used by SFLL [13] to enable a direct comparison.

A. Experiment 1: SAT Resilience of $\text{TLL}_{SFLL-Fault}$

We characterized the SAT resilience of TLL within our 10 benchmark circuits to experimentally verify the theory derived in Section VII-A. Specifically, we aimed to evaluate whether the probabilistic derivation in Theorem 2 is indeed practically valid, thereby empirically verifying TLL's ability to expand the parametric space of locking. To this end, we experimented with 3 TLL instances, TLL(s=11,l=$\{1,2,3\}$,c=1). For each TLL instance, we constructed 4 different locking configurations by randomly selecting alternative sets of minterms to lock. Hence,

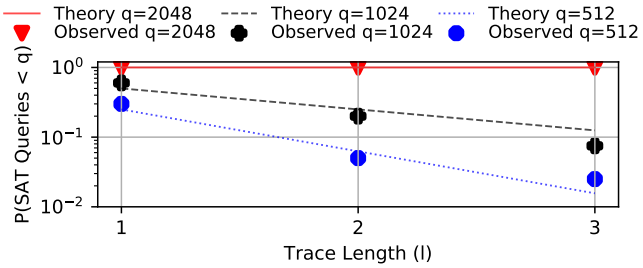

Fig. 8: Comparison of theoretical and experimental SAT resilience of TLL(s=11,l=$\{1,2,3\}$,c=1).

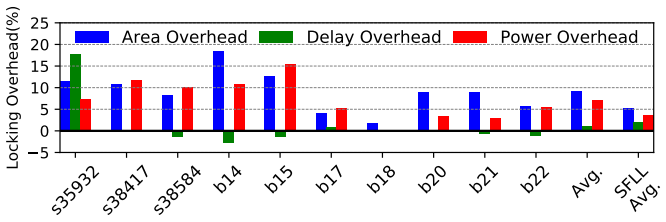

Fig. 9: ADP overhead of TLL(s=128,l=2,c=2). The average ADP overhead of an equal error rate SFLL(s=128,c=2) construction is shown for reference.

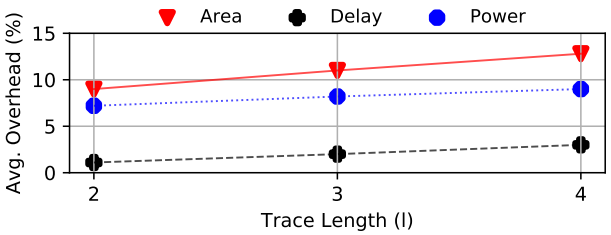

Fig. 10: Average ADP overhead of trace length scaling for TLL(s=128,l={2,3,4},c=2).

for each trace length we had 40 (4 locking configurations * 10 benchmarks) unique TLL netlists. We SAT attacked each netlist using the SAT attack from [23] and recorded the number of SAT queries required to unlock each TLL instance. With this data, we computed the probability of a SAT attack terminating within q SAT queries per restore unit state. Figure 8 displays the observed SAT resilience alongside the theoretical derivation from Theorem 2. The experimental/theoretical results closely match, supporting our derived results. Notice as trace length is increased, the experimental SAT susceptibility of TLL exponentially decays. This supports Theorem 3 which proves that the SAT resilience of TLL exponentially improves for a linear increase in trace length.

B. Experiment 2: ADP Overhead of $TLL_{SFLL-Fault}$

We characterized the area, delay, and power (ADP) overhead of our presented TLL construction. To do this, we incorporated TLL(s=128,l=2,c=2) within each benchmark circuit. The corresponding post-mapping overhead was determined using the Cadence Encounter RTL Compiler with the Synopsys 90nm SAED library. Additionally, we evaluated an SFLL-Fault implementation with the same error rate (s=128,c=2) for comparison. The ADP overhead for each benchmark is in Figure 9. On average, we found the ADP overhead of TLL to be 9%, 1.1%, and 7.2%, respectively. When compared to an equivalent error rate implementation of SFLL-Fault, TLL demonstrated a modification of +3.8%, -0.8%, and +3.5% in ADP overhead. Therefore, when SFLL-Fault is enhanced with TLL, exponentially stronger SAT resilience is achieved with only a small additional ADP overhead.

We have characterized the overhead associated with trace length scaling as well. To do so, we calculated the design overhead of TLL(s=128,l={2,3,4},c=2) within each of our 10 benchmark circuits. These results have been aggregated in Figure 10. Notice that the average design overhead of TLL increased by +2%, +0.8%, and a +1% for area, delay, and power as trace length was increased from l=2 to l=3. As we further increased trace length from l=3 to l=4, a slightly smaller ADP overhead increase of +1.8%, +0.9, and +0.7% was observed. This small reduction in added area and power overhead was due to the increased combinational optimization made possible as more functionality was stripped from the circuit. This implies that as trace length continues to scale, the overhead due to added TLL logic (i.e. LUT entries and restore unit states) will be increasingly offset by additional combinational optimization. To conclude, this experiment indicated that a linear increase in trace length yields a slightly sub-linear increase in ADP overhead. However, for this increase in overhead, TLL was shown to provide an exponential increase in SAT resilience. Hence, TLL provides an efficient trade-off between ADP overhead and security.

C. Experiment 3: Architectural TLL Security

We used TLL to lock the 80186 core found to be unsecurable by logic locking in Section IV. By evaluating TLL in this netlist, we explore its ability to secure real-world ICs.

TLL Configuration:

We began by designing a TLL construction capable of achieving security within the data path of our 80186 netlist. Note that conventional logic locking was unable to achieve security in this netlist (Section IV). Using the design space exploration in Figure 3, we were able to quantify the wrong key error rate necessary for error severity. From the figure, this corresponds to the wrong key error rate of an SFLL-Fault configuration requiring between 1024 and 4096 SAT queries to unlock on average. According to Theorem 1, this corresponds to an error rate of 0.02% and 0.08%, respectively.

We designed 2 TLL constructions, TLL(s=17,l=?,c=16) and TLL(s=17,l=?,c=64), to exceed an error rate of 0.02% and 0.08%. Next, we selected a trace length which allows TLL to achieve strong SAT resilience given each wrong key error rate. To do so, we aggregated the SAT resilience of TLL for varying trace lengths in Figure 11. Given a trace length of 8, there exists a negligible probability (2^{-128} and 2^{-512} respectively) of a SAT attack locating a correct subkey in $< 2^{16}$ queries per restore unit state. This constitutes extremely strong SAT resilience. Therefore, we used a trace length of 8 for each construction. Finally, we randomly selected input minterms for locking and incorporated TLL(s=17,l=8,c=16) and TLL(s=17,l=8,c=64) into the data path of the 80186 netlist. To evaluate each IC, we the ObfusGEM simulator [32].

Simulation Framework:

Using the ObfusGEM simulator, we performed cycle-accurate simulations of locked ICs to quantify the rate with which a given locking construction induced critical failures in IC workloads. This failure rate serves as a measurement of the error severity of a given locking configuration. For these simulations, we chose 9 PARSEC benchmarks [33] to serve as a reasonable cross-section of common computing applications.

We performed architectural IC simulations by modeling and simulating both a locked and unlocked processor core in

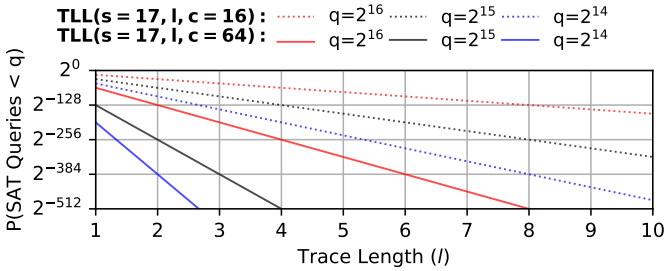

Fig. 11: SAT attack resilience of a locked 80186 netlist with varying trace length TLL constructions.

GEM5 [34]. In the locked simulation instance, locked input cubes, located via a fault analysis of a locked and keyed netlist, were mapped to a deterministic error state within the simulated IC. As these locked cubes were provided as input to the simulated IC, a multi-bit fault was injected into the simulation instance by forcibly overwriting the output of the locked module with the output of the locked and wrongly keyed netlist. Error severity was classified by comparing the processor state of a locked and unlocked GEM5 simulation over a variable number of clock cycles. A divergence of a locked core from an unlocked core which was not corrected in the variable window was classified as an unrecoverable error.

Experimental Results:

We quantified the error severity of each TLL construction with our simulation framework. Specifically, we performed 40 Monte Carlo simulations of each PARSEC benchmark with a different, randomly-selected wrong key for each trial. Hence, each Monte Carlo trial induced output corruption for a different set of input minterms in the processor. The percent of PARSEC benchmark runs with unrecoverable errors for each TLL configuration is in Figure 12 alongside the corresponding ADP overhead in Table III. Based on the simulation results, both TLL configurations achieved error severity. The first configuration, TLL(s=17,l=8,c=16), achieved an average benchmark failure rate of over 80%. This result implies that an untrusted foundry pirating this locked IC would find 80% of workloads to fail. The second TLL configuration, TLL(s=17,l=8,c=64), showed even stronger error severity, causing 97% of workloads to fail given a wrong key. We now turn our attention to SAT resilience. In Figure 11, we have aggregated the SAT resilience achieved by each TLL construction over varying trace lengths. Because the SAT resilience of TLL-enhanced locking techniques grows exponentially in trace length, relatively short trace lengths achieved extremely strong SAT resilience. Given our selected trace length of 8, there exists a negligible probability (2^{-128} and 2^{-512} respectively) of a SAT attack successfully locating a subkey in less than 2^{16} SAT queries per restore unit state. This indicates strong SAT resilience within each netlist.

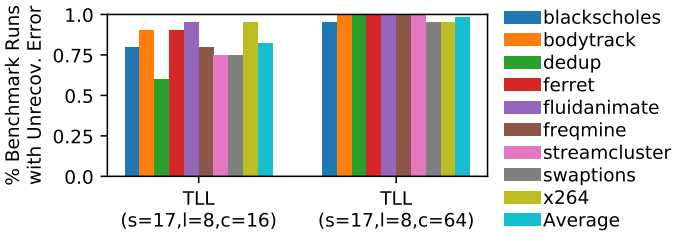

Fig. 12: Experimentally derived error severity for TLL-secured 80186 netlist running PARSEC workloads.

TLL Construction	Area	Delay	Power
TLL(s=17,l=8,c=16)	3.30%	0.00%	3.42%
TLL(s=17,l=8,c=64)	12.40%	6.43%	15.6%

TABLE III: ADP overhead for TLL-secured 80186 core.

Finally, we note that the first TLL configuration, TLL(s=17,l=8,c=16), achieved security with minimal design overhead, with only a ∼3% degradation in area/power and no delay overhead. The second TLL configuration, TLL(s=17,l=8,c=64), exhibited a higher overhead, however, provided much stronger security guarantees to compensate for this additional overhead. Therefore, both TLL constructions simultaneously achieved strong error severity and SAT resilience guarantees with only modest design overhead degradation. The conclusions of this experiment can be summarized as follows:

1) Due to the identified parametric space, logic locking techniques with a feasible overhead were unable to achieve error severity and SAT resilience in this netlist (Sec. IV).
2) By enhancing conventional locking with TLL, a locking configuration was designed and empirically shown to achieve both error severity and SAT resilience. Therefore, through trace length scaling, TLL expanded the parametric space of locking, overcoming the limits of prior art.

X. Conclusion

We began our work by deriving a fundamental theoretical trade-off which directly related 2 primary goals of logic locking, error severity and SAT resilience, regardless of construction. To quantify the ramifications of the parametric space created by this trade-off, we performed extensive architectural simulations of 2 real logic locked ICs. These experiments showed that there did not exist a logic locking configuration capable of achieving both error severity and SAT resilience concurrently due to the rigidity of the derived parametric space. This led us to investigate methods to expand upon this trade-off. As a result, we proposed trace logic locking (TLL). TLL is a provably secure and scalable enhancement to existing logic locking techniques which locks a sequence of primary inputs, known as a *trace*. By locking traces, TLL expands the parametric space of logic locking. This allows an IC designer to achieve both error severity and SAT resilience by varying trace length. We provided a theoretical and empirical demonstration of this. The low overhead nature of TLL was verified as well in 10 benchmark circuits. Finally, TLL was used to secure a real-world 80186 core. Through architectural simulations, we showed that TLL achieved both error severity and SAT resilience simultaneously.

References

[1] A. Chakraborty et al., "Keynote: A disquisition on logic locking," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 09 2019.

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

[3] 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), 2019.

[4] A. Baumgarten et al., "Preventing ic piracy using reconfigurable logic barriers," IEEE Design & Test of Computers, pp. 66–75, 2010.

[5] S. Dupuis et al., "A novel hardware logic encryption technique for thwarting illegal overproduction and hardware trojans," in 2014 IEEE 20th International On-Line Testing Symposium (IOLTS). IEEE, 2014.

[6] J. Rajendran et al., "Security analysis of logic obfuscation," in Proceedings of Design Automation Conference, 2012.

[7] ——, "Fault analysis-based logic encryption," IEEE Transactions on computers, vol. 64, no. 2, pp. 410–424, 2013.

[8] J. A. Roy et al., "Epic: Ending piracy of integrated circuits," in Conference on Design, automation and test in Europe, 2008.

[9] Y. Xie et al., "Mitigating sat attack on logic locking," in Conference on Cryptographic Hardware and Embedded Systems, 2016.

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

[11] M. Zuzak et al., "Memory locking: An automated approach to processor design obfuscation," in 2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2019, pp. 541–546.

[12] M. Yasin et al., "Ttlock: Tenacious and traceless logic locking," in Intl. Symposium on Hardware Oriented Security and Trust, 2017.

[13] ——, "Provably-secure logic locking: From theory to practice," in Conference on Computer and Communications Security, 2017.

[14] 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.

[15] ——, "Atpg-based cost-effective, secure logic locking," in IEEE 36th VLSI Test Symposium (VTS). IEEE, 2018.

[16] S. Roshanisefat et al., "Srclock: Sat-resistant cyclic logic locking for protecting the hardware," in Great Lakes Symposium on VLSI, 2018.

[17] A. Rezaei et al., "Cycsat-unresolvable cyclic logic encryption using unreachable states," in Proceedings of the 24th Asia and South Pacific Design Automation Conference. ACM, 2019, pp. 358–363.

[18] B. Shakya et al., "Cas-lock: A security-corruptibility trade-off resilient logic locking scheme," IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 175–202, 2020.

[19] M. Yasin et al., "On improving the security of logic locking," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2016.

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

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

[22] A. Sengupta et al., "Customized locking of ip blocks on a multi-million-gate soc," in International Conference on Computer-Aided Design, 2018.

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

[24] Y. Shen et al., "Double dip: Re-evaluating security of logic encryption algorithms," in Great Lakes Symposium on VLSI 2017, 2017.

[25] 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.

[26] 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.

[27] H. Zhou et al., "Cycsat: Sat-based attack on cyclic logic encryptions," in IEEE/ACM International Conference on Computer-Aided Design, 2017.

[28] M. El Massad et al., "Reverse engineering camouflaged sequential circuits without scan access," in 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 2017, pp. 33–40.

[29] D. Zhang et al., "Dynamically obfuscated scan for protecting ips against scan-based attacks throughout supply chain," in 2017 IEEE 35th VLSI Test Symposium (VTS). IEEE, 2017, pp. 1–6.

[30] H. Zhou, "A humble theory and application for logic encryption." IACR Cryptology ePrint Archive, vol. 2017, p. 696, 2017.

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

[32] 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.

[33] C. Bienia et al., "The parsec benchmark suite: Characterization and architectural implications," in Proceedings of the 17th international conference on Parallel architectures and compilation techniques, 2008.

[34] N. Binkert et al., "The gem5 simulator," ACM SIGARCH Computer Architecture News, vol. 39, no. 2, pp. 1–7, 2011.

[35] K. Shamsi et al., "On the impossibility of approximation-resilient circuit locking," in 2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). IEEE, 2019, pp. 161–170.

[36] ——, "On the approximation resiliency of logic locking and ic camouflaging schemes," Trans. on Information Forensics and Security, 2018.

[37] F. Corno et al., "Rt-level itc'99 benchmarks and first atpg results," IEEE Design & Test of computers, vol. 17, no. 3, pp. 44–53, 2000.

[38] D. Sirone et al., "Functional analysis attacks on logic locking," in Design, Automation & Test in Europe Conference & Exhibition, 2019.

[39] F. Yang et al., "Stripped functionality logic locking with hamming distance based restore unit (sfll-hd)–unlocked," IEEE Transactions on Information Forensics and Security, 2019.

[40] R. S. Chakraborty et al., "Harpoon: an obfuscation-based soc design methodology for hardware protection," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2009.

[41] Y. Alkabani et al., "Active hardware metering for intellectual property protection and security." in USENIX security symposium, 2007.

[42] J. Dofe et al., "Novel dynamic state-deflection method for gate-level design obfuscation," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 37, no. 2, pp. 273–285, 2017.

[43] A. R. Desai et al., "Interlocking obfuscation for anti-tamper hardware," in Proceedings of the eighth annual cyber security and information intelligence research workshop, 2013, pp. 1–4.

[44] M. Fyrbiak et al., "On the difficulty of fsm-based hardware obfuscation," Transactions on Cryptographic Hardware and Embedded Systems, 2018.

[45] F. Brglez et al., "Combinational profiles of sequential benchmark circuits," in IEEE international symposium on circuits and systems, 1989.

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

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

Michael Zuzak (S'19) received his M.S. and B.S. in Electrical Engineering from the University of Maryland, College Park, MD, USA in 2016 and 2014, respectively. He is currently pursuing his Ph.D. in Electrical Engineering from the University of Maryland, College Park, MD, USA. His current research interests include hardware security, computer architecture, and electronic design automation.

Yuntao Liu (S'16) is a Ph.D. candidate with the University of Maryland, College Park, advised by Prof. Ankur Srivastava. 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) 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.