Obfuscation and Security for Digital Integrated Circuits

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Joseph Sweeney

B.S. Engineering Physics, Fordham University
B.S. Electrical Engineering, Columbia University
M.S. Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

January 2021

Acknowledgements

Thanks to my advisor, Larry Pileggi, who has guided me in my research, given me countless opportunities, and from whom I've learned a lot.

To my committee members, Shawn Blanton, Yiorgos Makris, Marijn Heule, and Rob Rutenbar, thank you for mentorship and direction in developing this thesis.

To my collaborators and coworkers Ken Mai, Mohammad Zackriya, Samuel Pagliarini, Oguz Atli, Ruben Purdy, Danielle Duvalsaint, Xiang Lin and Deepali Garg, thanks for enabling so much of this work.

I am grateful for the funding support from the Defense Advanced Research Projects Agency under contract FA8750-17-1-0059 "Obfuscated Manufacturing for GPS (OMG)" and Honeywell Federal Manufacturing & Technologies, LLC under contract A023646.

To my friends Amrit Pandey, Onur Kibar, Prashanth Mohan, Mimi Sweeney, Antonis Manousis, Dimitrios Stamoulis and Meric Isgenc, you have truly made Pittsburgh home.

Thanks to Natalia, who supports me more than I deserve.

Finally, thanks to my family to whom I owe everything.

Abstract

Globalization of IC manufacturing has led to increased security concerns, notably IP theft. A promising countermeasure is logic locking that adds programmable elements to a design, obfuscating the true functionality during manufacturing. Generally, logic locking techniques aim to provide IP security while avoiding large overheads. Towards this end, this dissertation makes several contributions.

First, a security analysis of existing locking techniques is presented, exposing several vulnerabilities. One class of techniques is analyzed using sensitivity, a property of Boolean functions. The analysis reveals the modified portions of a circuit with high probability, leading to deobfuscation. Another class of locking methods is used to demonstrate two modeling techniques, relaxed models and symmetry breaking, that can dramatically reduce attack times.

These vulnerabilities inform the development of latch-based logic locking, a novel obfuscation method that resists known attacks while maintaining low overheads. This balance is achieved by locking a design's clock tree, manipulating the functionality while avoiding timing-critical logic. To validate the technique, a set of common industrial designs has been locked and brought through the full manufacturing process. To demonstrate resistance to deobfuscation, the locking scheme is evaluated against existing and newly developed attack methods.

Finally, two metrics are established to better quantify the security of a given locking technique under common attack scenarios. These metrics are efficiently estimated using approximate model counting techniques. Importantly, they provide a means of analyzing the overhead-security trade-off of locking techniques, an essential aspect of integrating locking schemes into real systems.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Logic Locking for Protection of Intellectual Property

The downscaling of integrated circuits (ICs) has enabled computing that underlies many essential technologies. However, due to prohibitively high research and development costs, only a few foundries are manufacturing ICs in most scaled technologies. Consequently, many IC companies tend to operate fabless, relying on third-party foundries to manufacture their designs. Once a circuit is sent for fabrication, the foundry gains full visibility of the design with minimal effort, allowing theft of intellectual property (IP). This threat undermines the significant cost associated with developing digital circuits and is a growing concern in the IC industry.

To combat IP theft, a variety of logic locking techniques have been developed. Generally, these techniques add programmable elements to the logic of an IC. When programmed incorrectly, the elements disrupt the circuit functionality. The key, which correctly programs the elements, is set post-manufacture, so the correct design is never revealed to the untrusted entity. These techniques seek to provide maximal security while maintaining low overhead in the circuit's area, power, delay, and testability.

Unfortunately, logic locking is far from realizing this goal. Corresponding with logic locking schemes, attacks have been developed that can reveal the circuit's correct key under various attack conditions. To resist such attacks, lock designers have sacrificed greater overheads. In doing so, many schemes push the cost of locking a circuit to untenable levels. Furthermore, this cost increase is justified with notions of security that are tightly coupled to the execution time of specific attacks. In many cases, straightforward modifications to attacks lead to deobfuscation.

1.2 Thesis Contributions

This thesis addresses several of the preceding issues in logic locking. First, a security analysis of existing logic locking techniques is presented, exposing several weaknesses. Specifically, a new analysis method reveals locked portions of the design allowing for removal of locking circuitry and lock modeling techniques allow for significant speedups in attack time. The underlying causes of these vulnerabilities are identified, and fixes are proposed for each broken lock method. Although the security is improved, these fixed locking schemes still do not provide a satisfying security-overhead tradeoff.

This analysis informs the development of a new locking scheme, latch-based logic locking. This technique aims to provide low overhead security by exploring a new dimension in which to obfuscate a circuit. Whereas previous locks typically just manipulate a design's combinational logic, latch-based logic locking additionally keys a design's clock tree. This enables large amounts of interdependent keys to be inserted without the overheads associated with other techniques. The technique's low overhead is validated by manufacturing locked versions of several industry-standard circuits. Additionally, the security is comprehensively analyzed against existing as well as newly developed attack methods.

Finally, to enable better quantification of the security of given locking techniques, two metrics are proposed. These metrics, key corruption and minimum corruption, correspond to common attack models rather than specific attacks, enabling a designer to better discern the efficacy of various lock types. A flow for approximating these metrics on generic locked circuits is developed and evaluated several on locking techniques. We conclude with a discussion how the development of these and other metrics allows for automated logic locking schemes, reducing reliance on circuit designers for lock insertion and potentially providing substantially improved security-overhead trade-offs.

The remainder of the thesis is organized as follows:

- Chapter 2 elaborates on logic locking, providing an overview of relevant background knowledge, existing locking schemes, and attack methods.

- Chapter 3 describes new attack methods that deobfuscate locks from two classes locking methods.

- Chapter 4 introduces latch-based logic locking, a novel approach to adding programmable logic that maintains low overhead while resisting existing attacks.

- Chapter 5 proposes two new metrics that more accurately capture the security of locking techniques.

- Chapter 6 concludes with a discussion of the insights produced in this work and a promising new direction for logic locking research.

Chapter 2

Background

2.1 Digital Integrated Circuits

2.1.1 Structure, Operation, and Testing

Due to their low cost, noise tolerance, and high speed, digital integrated circuits are used in the majority of computing platforms. These circuits perform computations on Boolean-valued signals, $\{0, 1\}$. Generally, the circuits consist of sequential logic made from interconnected logic gates and state elements. Individually, the logic gates implement simple functions that are composed into more complex functions. These functions are referred to as a circuit's combinational logic. The state elements, also referred to as sequential elements, function as memory, storing values while the combinational logic is evaluated. Fig. 2.1 shows an example circuit with the state elements and logic gates colored blue and white, respectively.

To conduct the desired computation, the circuit moves through a set of state values based on the logic gate structure and inputs to the system, producing the desired sequence on its outputs. The evaluation of the circuit is dictated by a periodic clock signal. Each clock period, the logic gates compute the circuit's current output values and the next values of the state elements based on the current state and input values.

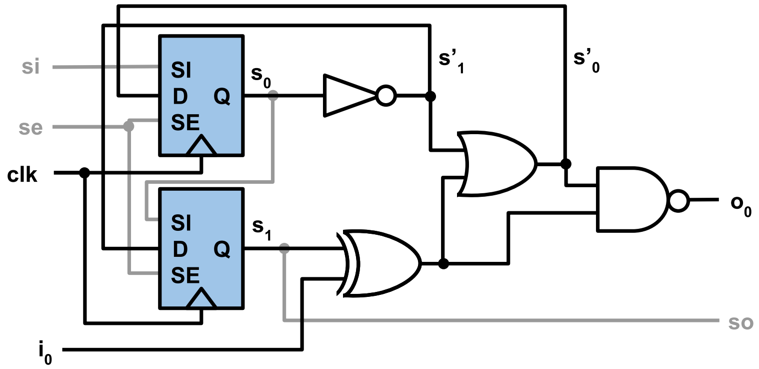
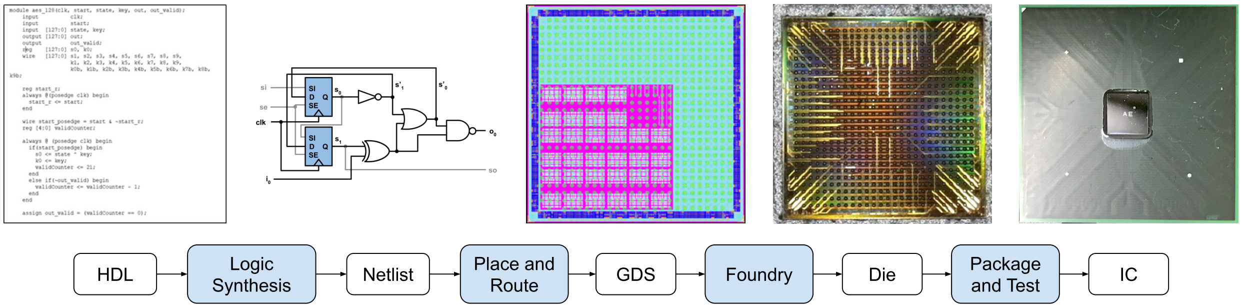


Figure 2.1: Typical digital IC structure

More formally, the functionality of the digital circuit can be specified as a finite state machine (FSM) using the Mealy model [1]. The model consists of a set of variables, functions, and initial states that describe how the circuit operates. We respectively denote the input, output, state and next-state variables as $i \in \{1,0\}^n$, $o \in \{1,0\}^m$, $s \in \{1,0\}^l$, and $s' \in \{1,0\}^l$. These variables are labeled in Fig. 2.1, where s_n is the nth bit of s. The output and next-state values are determined by an output function, λ, and state-transition function, δ. Every clock period, the variables are updated as $s := s'$, subsequently $s' := \delta(s,i)$ and $o := \lambda(s,i)$. We denote the circuit's initial states as $S^0 \subseteq \{1,0\}^l$.

To aid in testing, digital circuits typically use a scan-chain, colored gray in Fig. 2.1. The scan-chain is formed by a serial connection between the circuit's state elements, the output of each sequential element feeds into the scan-in port, *si*, of the next. During *test* mode, the scan-chain is enabled ($se = 1$) causing the sequential elements take in data from their *si* port, rather than the *d* port used in normal operation. This enables arbitrary reading and writing of a digital circuit's state, significantly reducing the cost of exercising the circuit's logic during testing. Scan-chains allow the combinational logic of the circuit to be considered separately from the state elements, significantly improving the performance of automatic test pattern generation (ATPG) tools. For convenience, we refer to the combinational logic as C, where variables $y = (s', o)$ and $x = (s, i)$ are updated as $y := C(x)$. Note that in Fig. 2.1 $x = (s_1, s_0, i_0)$ and $y = (s'_1, s'_0, o_0)$.



```
module aes_128(clk, start, state, key, out, out_valid);
    input          clk;
    input          start;
    input  [127:0] state, key;
    output [127:0] out;
    output         out_valid;
    reg    [127:0] s0, k0;
    wire   [127:0] s1, s2, s3, s4, s5, s6, s7, s8, s9,
                   k1, k2, k3, k4, k5, k6, k7, k8, k9,
                   k0b, k1b, k2b, k3b, k4b, k5b, k6b, k7b, k8b,
k9b;

    reg start_r;
    always @(posedge clk) begin
      start_r <= start;
    end

    wire start_posedge = start & ~start_r;
    reg [4:0] validCounter;

    always @ (posedge clk) begin
      if(start_posedge) begin
        s0 <= state ^ key;
        k0 <= key;
        validCounter <= 21;
      end
      else if(~out_valid) begin
        validCounter <= validCounter - 1;
      end
    end

    assign out_valid = (validCounter == 0);
```

Figure 2.2: High-level view of IC manufacturing process

2.1.2 Manufacturing Process Vulnerabilities

The significant cost of developing modern IC manufacturing processes has shifted the industry towards a fabless production model. A high-level diagram of the model is depicted in Fig. 2.2. First, a design team encodes the functionality of a design in a hardware description language (HDL). This description is passed to a logic synthesis tool that implements the functionality as a sequential circuit. The logic gates, sequential elements, and their interconnections are defined in a *netlist*. The netlist is passed to a place-and-route tool that maps the circuit to a 2-D grid, ultimately producing a GDS file containing a geometric description of the various transistor and metal layers to be printed. This GDS file is *taped-out*, i.e., sent to the foundry to manufacture the IC. The foundry produces bare IC dies that are then packaged and tested.

A critical security vulnerability results from third-party access to the design data. As a GDS file contains all necessary information for production, it is trivial for the foundry to *overproduce the design*. Furthermore, it is straightforward to reverse-engineer a design's functionality, extracting the netlist from the GDS. Other supply chain entities, such as the packaging company, can obtain the netlist through other means such as delayering the ICs [2]. The adversarial entity may then *leak design information* to other parties. Additionally, the foundry can *modify the design*, inserting hardware Trojans that disrupt functionality. These threats are a concern for both commercial IC design companies that invest large amounts of money in IC designs as well as government entities that produce

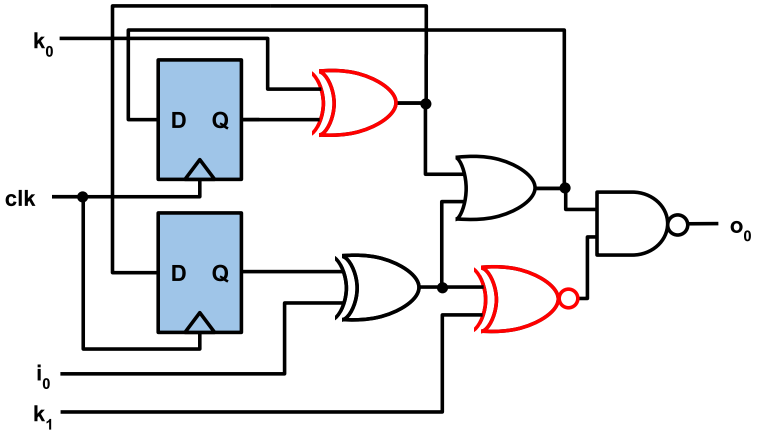


Figure 2.3: XOR logic locking example

ICs for security critical applications [3, 4, 5].

2.2 Logic Locking

Logic locking is a strategy for addressing these manufacturing security vulnerabilities. The designers add a set of key inputs to the circuit's netlist. The inputs are incorporated into the design such that under different values of the key inputs, the circuit's functionality changes. The foundry does not know what the proper key value is, thus any overproduced ICs or leaked design information will be incomplete. An adversary must select a functionally correct key out of the exponentially large space. After manufacturing, the correct key is applied to the circuit. A typical assumption is that the key is stored in a tamper-proof non-volatile memory.

A simple example is XOR logic locking in which parity gates (XOR/XNOR) invert signals depending on a key input [6]. The parity gates, termed key gates, are spliced into locations either selected randomly or with various heuristics maximizing corruption or interdependence [7, 8]. After insertion, the netlist is resynthesized, mixing the parity gates with the existing logic. A diagram is shown in Fig. 2.3 wherein two parity gates, highlighted in red, have been inserted into the circuit from 2.1. Note that the k_0 parity gate has replaced an inverter from the original netlist. These manipulations create a large amount of corruption within the circuit as the parity gate's entire upstream function is inverted.

Formally, a logic locking technique, $\mathcal{L} : C(x) \rightarrow C_{\text{lock}}(x, k)$, is a transform that takes a circuit, $C(x)$, and produces a locked version, $C_{\text{lock}}(x, k)$. Under the correct key, k_{c}, the locked circuit produces the same behavior as the original circuit, however the exact notion of equivalence may depend on the locking technique. If the transform only modifies the combinational logic, it is sufficient to show equivalence proving that the next state logic and output logic cones are equivalent, $\forall x \, C_{\text{lock}}(x, k_{\text{c}}) = C(x)$. If the state elements are not preserved, sequential equivalence definitions must be used. While several different notions of sequential equivalence exist, in this work we use three-valued safe equivalence [9]. Here, each circuit node takes on a value from $\{0, 1, X\}$, where X represents unknown values. A locked circuit is equivalent to the original if after the application of a reset input sequence $\mathcal{R}$, no input sequence exists that would produce a different output sequence between the original and locked circuits. Specifically, for all reachable states after reset, the original and locked output functions are equivalent, $(\lambda, \lambda_l) \in \{(0, 0), (1, 1), (X, X)\}$. Generally, proving sequential equivalence is significantly harder than combinational equivalence due to the sheer size of the state spaces that might be encoded differently [10].

2.2.1 Attack Models

In the characterization of the security of a logic locking technique, an attack model is used to specify assumptions regarding the adversary. Within the logic locking research community, a variety of attack models have been considered differing in regard to the artifacts, abilities, and success criteria of the adversary. As a result, understanding the targeted attack model is of utmost importance for any party implementing these locking techniques.

There are generally two artifacts to consider in defining the attack model: the locked circuit's design data and a functioning version of the unlocked circuit. Adversarial access to the design data is a basic assumption of logic locking. However, depending on where the adversary is in the supply chain, the design data may vary. A foundry will have access to a GDS file, and thus can extract a netlist. Along, with the netlist, a foundry will have intimate knowledge of transistor and interconnect models, allowing detailed physical simulation and analysis. This would allow an adversarial foundry to obtain design features such as clock frequency. Other supply chain entities will likely have some subset of this access.

A more powerful attack model assumes access to a functioning version of the unlocked design. The unlocked circuit, referred to as an oracle, has the correct key set in its tamper-proof memory, affording the attacker black-box access. Obtaining an unlocked circuit may be trivial if the IC is available on the open market or may be the result of comprised physical security. Variations of this attack model have been considered in which the adversary does or does not have full access to the unlocked circuit's scan-chains. While scan-chains are essential in testing a design, they are also a powerful attack vector for an adversary.

Other considerations include the amount of processing power available to the adversary as well as the potential for side-channel analysis such as probing signal lines. Access to high-performance machines is cheap and foundries commonly use probing techniques to aid development of manufacturing processes and circuit failure analysis. Key values have been probed in nodes as small as 28nm [11]. However, as feature sizes continue to scale, this probing becomes more difficult, limited by spatial resolution [12]. Most work in logic locking has not directly considered this threat, relying on the validity of the tamper-proof memory assumption.

The definition of a successful attack can also vary. The most rigorous definition is finding a functionally equivalent key. In the combinational context, wherein the attacker has scan-chain access, the problem solved by the attacker is one of finding an exact key, k_e, for which the locked circuit produces the same function as the original, k_e :

$\forall x\, C_{\text{lock}}(x, k_{\text{e}}) = C_{\text{lock}}(x, k_{\text{c}})$. A relaxed version of the exact recovery success criteria is approximate recovery. In this case, the attacker finds an approximate key, k_{a}, under which functionality of the locked circuit differs from the correct functionality with at most some probability ϵ, $k_{\text{a}} : P_{x \in X}[C_{\text{lock}}(x, k_{\text{a}}) \neq C_{\text{lock}}(x, k_{\text{c}})] < \epsilon$. Both definitions can be extended to sequential settings using the previous sections's definition of sequential equivalence.

2.2.2 Brief Taxonomy

As in the example from Fig. 2.3, the earliest logic locking techniques inserted keyed parity gates into the original circuit's structure. Similar locks utilize multiplexors (MUXs) and Lookup Tables (LUTs) in lieu of parity gates [8, 13]. We refer to this class of techniques that generally maintain the original structure as *insertion locks*. Insertion locks can exhibit low overhead if the added gates avoid the design's critical timing paths. However, some insertion heuristics that increase security via greater key inter-dependency tend to create paths with many key gates, significantly impacting the delay. Unfortunately, under oracle attack models, these insertion locking schemes have been largely broken using a variety of methods, the most successful of which are miter-based attacks that use either satisfiability (SAT) solvers or ATPG tools to iteratively form constraints that the correct key must respect. [14]. More detail on these attacks is provided in section 2.3.

Researchers have attempted to increase the difficulty of mounting a miter-based attack by inserting attack-resilient logic blocks into the locked circuit [15, 16]. These techniques, collectively referred to as *iteration locks*, reduce the number of keys ruled out per attack iteration, significantly increasing the overall execution time. However, the logic blocks are susceptible to removal attacks since the circuitry is typically traceable through properties such as signal probability [17]. Fixes to these vulnerabilities have been proposed with the *strip-functionality* class of locking techniques that provide a better mixing scheme, but as to be detailed in section 3.1, have limited effectiveness.

Other schemes rely on circuit properties that the miter-based attack does not model. For instance, a cyclic obfuscation scheme assumes SAT solvers can only handle acyclic circuits [18]. It creates loops in the circuit's combinational logic to corrupt the miter-based attack. Another technique, delay-locking, adds tunable delay key gates to the design. Incorrect keys lead to setup and hold timing violations that the attack does not model by default. This comes at the cost of large delay overheads as the security scales, with reported average delay overheads of 60% [19]. While the security of these defenses initially seems promising, when their properties are correctly formulated within SAT, they can be easily broken [20, 21]. Moreover, commercial ATPG tools have built-in timing and loop breaking algorithms to automatically handle many of these situations [22].

Yet another class of locking mechanisms replaces portions of the design with highly configurable logic [23, 24, 25, 26]. The densely interdependent keys of these locks overwhelm SAT and ATPG solvers. A prototypical example is replacing gates with LUTs and adding configurable routing. The resulting locks resemble a field-programmable gate array (FPGA) embedded into the circuit. The existing incarnations of these locks vary in insertion methods, density, and mixing with original logic. These methods require careful integration into the system to avoid large overheads, in some cases delay overheads north of 200% have been reported. However, generally the class has been shown to be highly resistant to the typical miter-based attacks. In section 3.2, we will investigate methods of reducing these attack times.

Finally, while the preceding techniques just modify the combinational part of the design, sequential logic locking also manipulates the state elements. One such technique modifies the circuit's FSM to require a specific input sequence to transition from the reset state to the functionally correct set of states [27]. If an incorrect sequence is given, the circuit remains in a portion of the state space with incorrect behavior. This particular scheme is susceptible to a targeted removal attack [22, 28]. In general, sequential logic locking relies on limited scan-chain access to a design. However, recent work has extended the miter-based attack, originally developed for combinational circuits, into a model checker-based attack that assumes no scan-chain access [29, 30].

As clear from the preceding examples, there have been many directions explored for locking circuits. Unfortunately, none of these solutions balances security with low overhead, motivating further research in the area.

2.2.3 Related Techniques

Other proposed strategies for IP security in manufacturing include fully programmable logic, split-fabrication schemes, camouflaged logic, and metering schemes.

Fully programmable logic solutions, such as FPGAs, are conceptually similar to logic locking wherein the device's configuration bits are analogous to the key. These commercially available devices enable the design functionality to be programmed post-manufacture, bypassing the IP security issues faced in circuit manufacturing. An adversary in this case has virtually no information, just the generic structure. Unfortunately, application requirements may prohibit the use of fully-programmable devices as the power, area, and delay overheads can be an order of magnitude higher compared to application specific circuits [31]. However, some applications may see significantly less overhead if they can utilize the hard-coded versions of common sub-circuits incorporated into most FPGAs.

Split-fabrication relies on two foundries producing portions of the design. Thus, barring collusion, neither foundry has full access to the design. *Split-Manufacturing* divides a design into front and back-end-of line partitions [32]. One partition contains the low-level metal and transistor layers, the other contains the remaining portion of the metal stack. Each partition is manufactured in a separate foundry and combined by a trusted entity. This is a potent solution but requires navigating complicated logistics between foundries. *Split-Chip* design utilizes two ICs to implement a system, a untrusted modern-node IC and a trusted legacy-node IC, providing combined security and performance [33, 34]. The trusted IC contains the security-critical control logic, and the untrusted IC contains the performance-critical components. This is a powerful solution to IP security, but requires a suitable system topology and the cost of a second IC.

Camouflage gates are logic elements that rely on subtle changes in structure to confuse delayering and imaging netlist extraction techniques. Examples of this include modifications to threshold voltages [35], dummy fill logic [36], and dummy via connections between metal layers [37]. While the foundry, which has access to the GDS, will detect these modifications, adversaries relying on netlist extraction techniques may reverse engineer an incorrect functionality. If the locations of the camouflaged gates can be identified, the possible functionalities can be modeled in a similar manner as logic locking, enabling oracle-based and potentially some netlist-based attacks.

Hardware metering consists of methods that enable tracking and controlling of ICs post-manufacturing. It is divided into passive and active categories [38]. Passive metering uniquely identifies each manufactured IC. This allows the designers to identify unlicensed copies of the design. There are several different schemes of encoding this unique identification including physically unclonable functions (PUFs), one-time programmable memories, and focused ion beam (FIB) modifications. Active metering provides a mechanism to enable or disable the IC. This can be done remotely via public-key cryptography systems [39] or locally on the IC in which case it is functionally similar to logic locking.

2.3 Attacks on Logic Locking

2.3.1 Netlist-Based Attacks

The most critical attack model to consider is clearly the netlist-based attack. If a locked circuit can be deobfuscated in this context, there is no point to locking the circuit. Examples of these attacks have already been mentioned, specifically the removal attacks for iteration-based and sequential locks. In both cases, the lock adds some isolated structure to the circuit that is easily discernible from the original circuitry. Foiling the locking scheme is just a matter of tracing and removing the outgoing connections from the lock portion of the circuit. Section 3.1 will develop another such example, using sensitivity, a property of Boolean circuits.

Other netlist-based attacks utilize machine-learning algorithms to predict key values as seen in an attack on XOR locking [40]. The authors demonstrate that, even after synthesis, the changes to a netlist locked with parity gates are generally restricted to the immediate neighbors of a key input. Using this local structure as data, they train a classification algorithm and demonstrate prediction accuracy as high as 95%.

2.3.2 Brute Force and Sensitization Attacks

Given the more powerful oracle attack model, a brute force attack establishes a baseline for the necessary key width of the circuit. This attack entails searching through the space of keys and inputs, ruling out a key if a mismatching input-output (IO) pair is found. Each key-input combination is applied to the circuit's netlist, simulating the output result. The output value is compared to the unlocked circuit's output under the same input, ruling out keys when a difference is observed. The rate at which keys can be ruled out depends on what portion of the input space is incorrect under each key and the expected number of keys to check depends on the portion of the input space that is equivalent to the functionally correct key. Assuming a typical scan-chain frequency of 100MHz and a state size of 1000 bits, a query to the oracle can occur at a rate of 100kHz. With access to 1 unlocked circuit and 1 month of attack time, an attacker can apply 2^{38} key-input vectors.

One step beyond brute force is the use of sensitization methods to propagate a key value to an output node [41, 22]. Sensitization is essential to ATPG tools and consequently is highly optimized. Sensitizing a node consists of finding an input pattern that when applied to the circuit will propagate the value (or inverse) of the node to an output. Under such an input pattern, toggling the node would toggle some output. Using sensitization as an attack is done by iterating over the key bits, applying unknown (X) values for all other key inputs. Then attempting to sensitize the current key to an output. If successful, the key value will be known and can be set to the appropriate value in subsequent iterations, simplifying the problem. This process is repeated until fixed-point where no additional key inputs can be sensitized.

2.3.3 Miter-Based Attacks

Beyond brute-force and sensitization attacks, the oracle attack model enables the mounting of a more targeted, miter-based attack. This attack uses the netlist and unlocked circuit to iteratively produce input-output (IO) relationships [14]. These relationships are used to rule out all keys that do not produce the same behavior, narrowing the space of possible circuit functionalities. As previously mentioned, these attacks can be implemented using SAT solvers or ATPG tools as the underlying kernel. We focus on SAT-based version, which is used throughout this work.

Propositional Satisfiability

A common approach to deal with hard combinatorial problems, such as finding the key of locked circuits, is to encode them into propositional logic and solve the resulting propositional formulas with a satisfiability (SAT) solver. The performance of SAT solvers improved significantly in the last two decades and they are used for many applications in hardware and software verification [42, 43]. In recent years, SAT solvers have also been successfully applied to various attacks, such as hash collisions [44] and mathematical challenges [45].

The most successful class of SAT solvers are based on the conflict-driven clause learn-

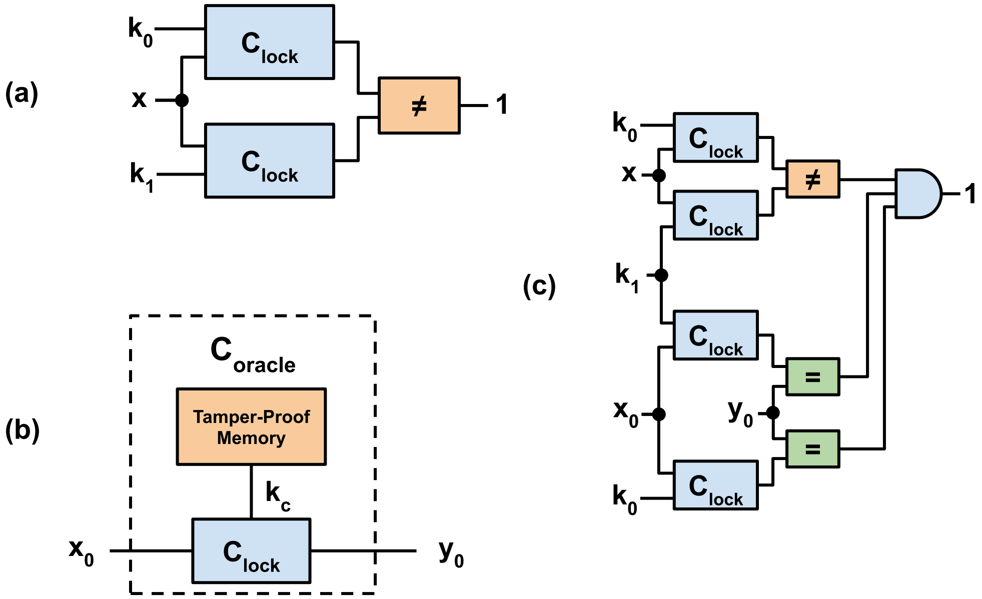


Figure 2.4: Miter-based attack steps: (a) Miter circuit construction, (b) Unlocked (oracle) circuit produces correct IO functionality (c) Addition of learned IO constraint to miter circuit

ing (CDCL) algorithm [46, 47]. Briefly, CDCL solvers work by repeatedly selecting a variable through heuristics and assigning a value. Implications from the assignments are determined using a highly optimized process called *unit propagation*. If a conflict is found, a clause is added to the formula that rules out assignments causing the conflict. Then the solver non-chronologically backtracks based on the conflict and continues, repeating this process until a solution is found or the problem is found to be unsatisfiable.

The typical encoding of the SAT problem is in the conjunctive normal form (CNF). This form consists of a set of clauses that must all be satisfied. Each clause is a disjunction of literals. A circuit can be encoded into propositional logic via the Tseitin transformation [48]. This transformation can take a circuit netlist and produce a set of clauses that, when collectively satisfied, will correspond to the original circuit's behavior.

Combinational Attack

If the adversary has access to the design's scan chain, the miter-based attack can be executed considering just the circuit's combinational logic. In this case, the IO relationships are efficiently learned through a three-step procedure: **I.** First, a miter circuit, M, is used to determine an input that is guaranteed to rule out at least a single key. A miter circuit consists of two copies of the original circuit with the inputs tied together, the key inputs kept separate, and the outputs connected to comparators. A diagram of the connections is shown in Fig. 2.4(a). Additional key constraints, such as timing and loop breaking, can be conjuncted with the miter output. A SAT solver is used to find a setting of the shared input (x) and key inputs (k_0, k_1) such that the output of the miter circuit is logic 1. By construction, the solution to the SAT problem will have two different keys that, at that input value, disagree on the output value. The shared input value found by the solver is termed a differentiating input. **II.** Next, as depicted in Fig. 2.4(b), the differentiating input, x_i, is applied to the oracle circuit to determine the differentiating output, y_i, forming an input-output pair that the correct key must respect; any key that does not conform to this IO pair is incorrect. **III.** Finally, as shown in Fig. 2.4(c), the IO pair is added as a constraint to the miter circuit for the next iteration. Now, any keys that satisfy the miter circuit will also satisfy the learned IO relationship. While each relationship is guaranteed to rule out at least one key, in practice, a larger portion of the key space is ruled out due to overlapping key functionalities at a given input. These steps repeat, adding more constraints until the miter circuit is unsatisfiable. At this point, any key that respects all learned IO relationships will be a functionally correct key. We outline the pseudo-code for this attack in algorithm 1.

Sequential Attack

The sequential version of the miter-based attack uses the same oracle and miter circuit but extends the concept of a differentiating input to a differentiating input *sequence*. To handle the state elements in the design, this attack substitutes the SAT solver for a model checker. The model checker is able to automatically unroll the circuit in time, connecting the state input of one cycle to the state output of the next. Every input sequence is assumed to start from a potentially unknown, but *fixed* reset state, s^0. In the

Algorithm 1: Miter-Based Attack

```
Input: $C_{\text{lock}}$, $C_{\text{oracle}}$
Output: $k_{\text{attack}}$
$M := C_{\text{lock}}(x, k_0) \neq C_{\text{lock}}(x, k_1)$;
$D := \emptyset$;
$i := 0$;
while SAT$[M \wedge D]$ do
    $x_i :=$ SAT_ASSIGNMENT$_x[M \wedge D]$;
    $y_i := C_{\text{oracle}}(x_i)$;
    $D := D \wedge C_{\text{lock}}(x_i, k_0) = y_i \wedge C_{\text{lock}}(x_i, k_1) = y_i$;
    $i := i + 1$;
end
return SAT_ASSIGNMENT$_{k_0}[D]$
```

same fashion as the SAT-based attack, a miter circuit is used to find input sequences that produce different output sequences for different keys. The miter circuit is unrolled as in Fig. 2.5(a). These input sequences are applied to the oracle and the produced outputs form IO relationships that are again encoded into the miter circuit as unrolled constraints as depicted in Fig. 2.5(b). Aside from the reduced controllability and observability from the state elements, the most significant difference between the combinational and sequential attacks is in the termination conditions.

While a SAT solver can be used to exhaustively prove that no input can satisfy a combinational miter circuit, when state is considered, the task is a much harder. This is mainly due to the state space explosion associated with allowing the model checker to unroll the circuit indefinitely [49]. Thus, in previous work, the model checker attack relies on secondary termination conditions: a check that only one key remains and a logical equivalence check for the combinational logic fan-in into each flip-flop. Without these equivalence conditions the termination relies on unbounded model checking algorithms [50, 51]. For a more complete description of the model checker-based attack, we direct the reader to [29, 30].

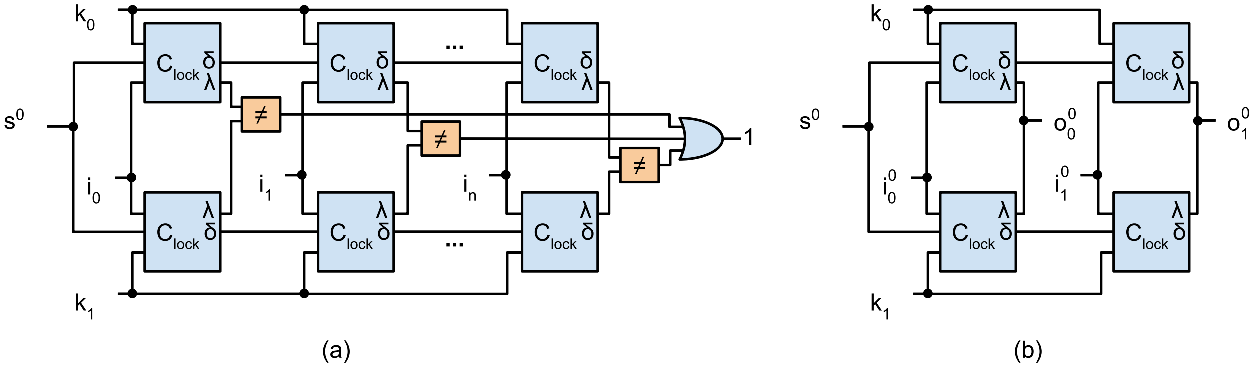


Figure 2.5: Miter-based model checker attack steps: (a) Miter circuit unrolling, (b) Differentiating sequence constraint.

Constraint Aided Attacks

The miter-based attack has become the standard attack paradigm as it is able to quickly deobfuscate many existing techniques. As mentioned in section 2.2.2, logic mechanisms have been developed that utilize non-standard design practices to prevent a miter-based attack from converging. Introducing combinational loops into circuits can cause a SAT solver to non-deterministically resolve the circuit state, corrupting the attack by breaking the correspondence between the real oracle circuit and logical miter circuit. This correspondence can also be broken by introducing key elements that modulate timing in the circuit. Without knowledge of the circuit's timing, a key produced from the miter-based attack will likely have incorrect functionality via timing violations. Despite their initial promise of miter-based attack resistance, when properly modeled, these techniques are easily broken. For each technique, critical information is withheld from the SAT solver causing it to incorrectly terminate. Reintroducing this information through additional key constraints avoids corruption. In these examples adding key constraints that rule out keys with oscillating loops and modifying the netlist to encode the change functionality with changed timing characteristics allows the attack to terminate correctly.

Chapter 3

Security of Existing Locking Schemes

In this chapter, we develop deobfuscating attacks for two families of logic locking techniques. The first attack takes advantage of traces left by the locking procedure, enabling the modified portion of the circuit's input space to be isolated and corrected. The second utilizes improved attack modeling to dramatically reduce attack times. For each broken lock scheme, we develop improvements that resist our attacks. We then use these attacks and corresponding fixes to inform the development of our own locking technique in chapter 4.

3.1 Sensitivity Analysis of Strip-Functionality Locks

Our first attack applies to the *strip-functionality* class of locking techniques, mentioned in section 2.2.2. This class was developed in response to the miter-based attack and represents a sizeable thread of logic locking research. In this section, we explore the use of Boolean sensitivity in analyzing this class of locks. Sensitivity is shown to be a powerful signal that can reveal the key of the locked circuits. We propose an improved insertion technique that mitigates this attack, but only for certain circuits.

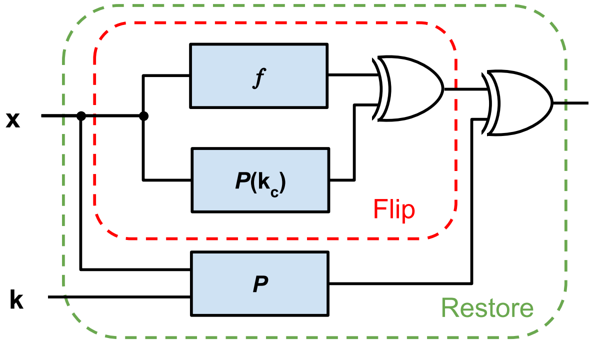


Figure 3.1: Underlying structure of strip-functionality locking

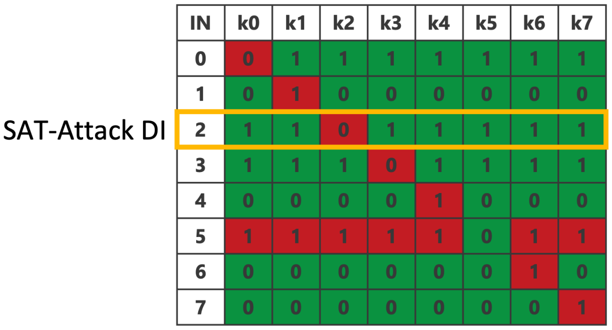
3.1.1 Strip-Functionality Locking

Strip-functionality locking refers to a class of techniques that share a similar mechanism for directly defending against the miter-based attack. The locks resist the attack by reducing the number of keys ruled out per iteration, significantly increasing the overall execution time. This class of techniques currently includes TTLock [52], TTLock* [53], SFLL-HD [54], and SFLL-Flex [54].

Under this class of locking schemes, functionality is *stripped* from the circuit by inverting the output response under certain inputs. This set of inputs is referred to as the protected inputs. The inverted functionality is re-established using restoration circuitry paired with the correct key. The restoration part of the locking circuitry can be isolated and removed by tracing key inputs, however, the resulting circuit still exhibits incorrect behavior at the protected inputs.

The generic structure of these techniques is shown in Fig. 3.1. For simplicity, we just show a single output cone of the locked circuit, which we denote as f. The locked circuit consists of two layers, flip and restore. Both layers make use of a function, $P(x,k)$, that checks if an input, x, is part of the protected set determined by the key, k. Different values of k will produce different protected sets.

The flip layer contains the original function, f, and a instance of P with the key input hard coded at the correct value, k_c. The outputs of both functions are XOR'd together, inverting the original function for the protected inputs. We refer to the flip layer function


SAT-Attack DI

IN	k0	k1	k2	k3	k4	k5	k6	k7
0	0	1	1	1	1	1	1	1
1	0	1	0	0	0	0	0	0
2	1	1	0	1	1	1	1	1
3	1	1	1	0	1	1	1	1
4	0	0	0	0	1	0	0	0
5	1	1	1	1	1	0	1	1
6	0	0	0	0	0	0	1	0
7	0	0	0	0	0	0	0	1

Figure 3.2: Truth table of circuit locked with TTLock

as f_{flipped}; it is equivalent to f except at the protected inputs. The restore layer contains another instance of P, XOR'd with the output of the flip layer. When the key input, k, matches k_c, the correct functionality of the original function is restored, each protected input value's output being flipped twice. We refer to the whole function with the restore layer as f_{locked}. The whole circuit is synthesized together, mixing and reducing the logic from both layers along with k_c. It is likely that the restore layer's P function remains intact as synthesis is unable to reduce the logic. However, the flip layer's P function and k_c are usually combined with the logic of f such that they are not recognizable via inspection.

These techniques resist the miter-based attack because the overlap between the flipped input patterns for different keys is kept low. Thus, when an IO constraint is formed in the miter-based attack, only a small number of keys are ruled out at once. An example of this is shown in the truth table of Fig. 3.2 where the highlighted differentiating input only rules out a single key. Also note that input 5 is the protected input. If this input is found, the technique is broken as only a single key agrees with the correct output. While the exact scaling of the attack resistance depends on the specific technique, the class as a whole shows greatest miter-based attack resistance when the number of protected inputs is minimal. Thus, there is an inherent trade-off between attack resistance and corruption of the circuit.

The specific techniques within the class are largely similar but each can be briefly described as following. TTLock is the original technique in this class; the method flips a single input pattern that is equal to the key. Thus, in this incarnation, P is a equality function. TTLock* is a version of TTLock that tries to mitigate any netlist-based attacks by converting the locked circuit to a reduced order Boolean decision diagram and resynthesizing. Effectively, this technique does a better job of mixing the f with the flip layer's hard coded k_c and P function. SFLL-HD is a generalization of TTLock in which every input pattern a fixed Hamming distance from the key is flipped. In SFLL-HD, P computes the Hamming distance between the key and input, then compares this to a fixed value to determine if the input is protected. Finally, SFLL-Flex stores a set of user-specified protected input patterns in a lookup table (LUT). In this case, P is a function that is logic 0 for all inputs except the selected input values.

3.1.2 Boolean Sensitivity Attack

Sensitivity is a simple complexity measure of a Boolean function [55]. Defined at a particular value in the input space, it is the number of inputs Hamming distance 1 from a particular input, for which the function produces a different value. Defined over the function, it is the maximum sensitivity of all inputs. We can specify each case more formally given a Boolean function, $f : \{0,1\}^n \rightarrow \{0,1\}$ and an input value, x. If $x_{\text{flip}(i)}$ represents the input value with the ith bit inverted, the sensitivity of f at x, $s(f,x)$, is the following:

$$s(f,x) = \frac{\sum_{i=1}^{n} f(x) \oplus f(x_{\text{flip}(i)})}{n} \tag{3.1}$$

Thus, the sensitivity of f, $s(f)$, is:

$$s(f) = \max_{x} s(f,x) \tag{3.2}$$

Finally, we define the average sensitivity of f, $\bar{s}(f)$ as:

$$\bar{s}(f) = \frac{\sum_{x \in \{0,1\}^n} s(f,x)}{2^n} \tag{3.3}$$

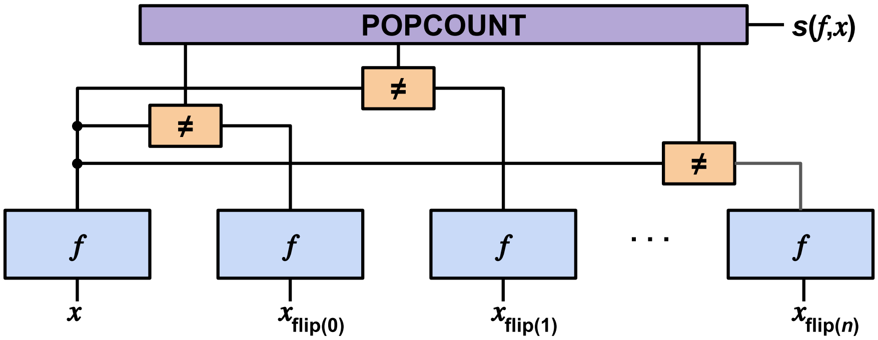


Figure 3.3: Circuit that determines sensitivity where $s(f, x)$ is the sensitivity of f at a given input x and $x_{\text{flip}(i)}$ represents x with the ith bit flipped

Sensitivity is interesting in analyzing strip-functionality circuits because it is an easily computed metric *and its value is inverted for protected inputs*. Consider an input value, $x_{\text{protected}}$, chosen at random to become a protected input. In the original circuit, if the input has a low sensitivity, most inputs Hamming distance 1 away will agree. As we know, the locking procedure will invert the output value for this input value using the flip layer's P function. This means that most of neighboring inputs now disagree with the protected input. Specifically, where f is the original circuit and f_{flipped} is the locked circuit without the restoration circuitry (ie. just the flip layer), the new sensitivity is:

$$s(f_{\text{flipped}}, x_{\text{protected}}) = 1 - s(f, x_{\text{protected}}) \tag{3.4}$$

Thus, the protected input is moved to the opposite end of the sensitivity distribution. This sensitivity inversion is potentially a usable signal to find the protected inputs.

This raises the question, what does the sensitivity distribution look like for typical circuits? To understand the behavior of sensitivity, we consider a set of benchmark circuits [56], which are commonly used in the logic locking and circuit testing communities. Each circuit contains several Boolean functions. For each function, f, we estimate the average sensitivity across all inputs, $\bar{s}(f)$, and find the sensitivity of the function, $s(f)$.

The average sensitivities are determined by sampling a set of randomly selected values in the input space of each function. The sensitivity, $s(f, x)$, is evaluated at each input value by calculating the output value of the input value and all neighbors Hamming dis-

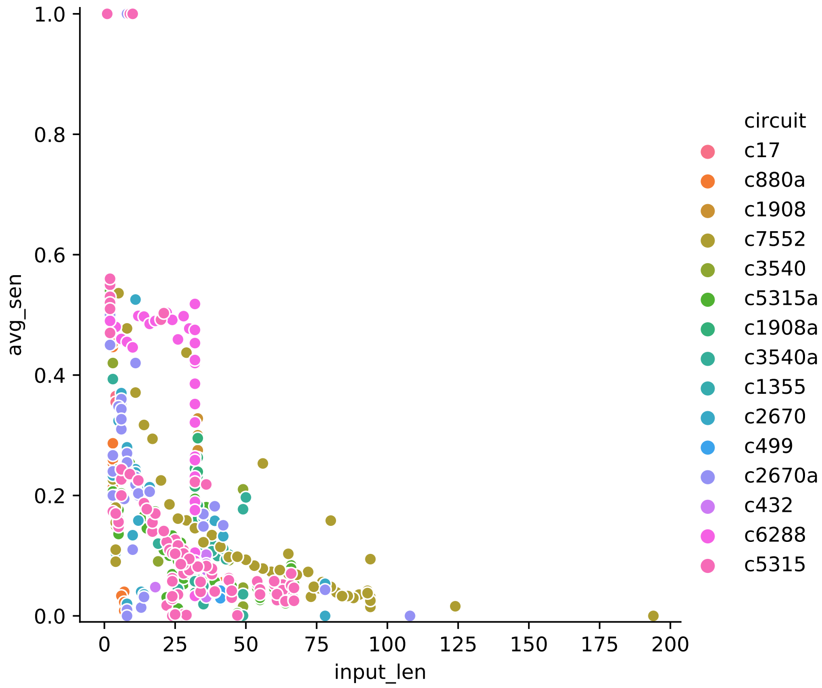


Figure 3.4: Average sensitivity, $\bar{s}(f)$, versus input width of benchmark circuits from 50 samples

tance 1 away, summing the number of disagreements. This random input selection is the same method used to pick keys in the strip-functionality techniques, giving us an idea of the likely sensitivity of the protected inputs.

To find the sensitivity of each function, $s(f)$, we build a circuit that quantifies the sensitivity at a given input. This circuit, shown in Fig. 3.3, is made up of $n+1$ copies of the function, where n is the width of the function's input. The inputs of the first copy of the function are tied to the n additional copies such that a different bit is flipped for each. The outputs are fed to comparators and subsequently a population count that determines the sensitivity at the input. The circuit is loaded into a SAT solver and the output sensitivity value is constrained to value, s. Starting from n, s is decremented until a satisfying assignment can be found for the circuit. The first input found will have a sensitivity value, s/n, the highest of any input. This defines the sensitivity value of the overall function.

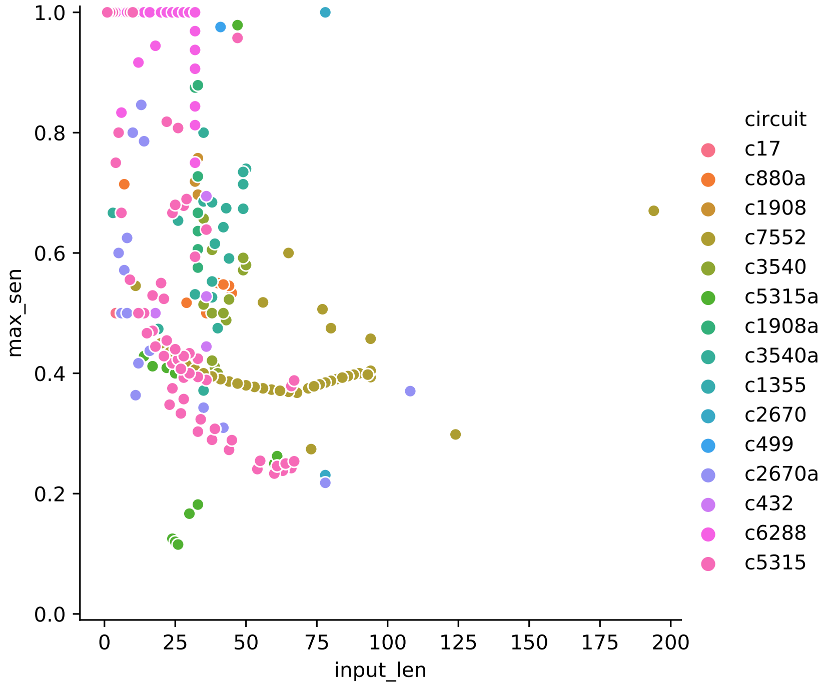


Figure 3.5: Sensitivity, $s(f)$, versus input width of benchmark circuits

The results of this analysis are shown in Fig. 3.4 and 3.5, where $\bar{s}(f)$ and $s(f)$ versus input width are respectively plotted. We see that most circuit output functions exhibit low average sensitivity, furthermore a substantial portion have maximum input sensitivities of less than 0.5. Another clear trend is that the average and maximum sensitivity is inversely proportional to the input width. From a designer's perspective, this makes intuitive sense since specifying a complex function of many inputs is difficult. A notable outlier is c6288, a multiplier circuit, that maintains a high average local sensitivity for larger inputs. This trend implies that at a sufficiently large input width, if an input is randomly selected as a protected pattern, the new sensitivity of this input will go from an *average low* to an *outlying high* value. Selecting inputs with large input width is motivated by increased brute force and miter-based attack resistance. As the selection for the strip-functionality techniques does not consider the sensitivity of the protected pattern, it is likely that the protected input pattern will have a final input sensitivity that is an outlier in the high end of the distribution.

Algorithm 2: Sensitivity-Based Attack

```
Input: $n$, $f$, $f_{\text{flipped}}$
Output: $x_{\text{protected}}$
$sen := n$;
$block := \emptyset$;
while $sen > 0$ do
    $CNF := (s(f_{\text{flipped}}, x) = sen) \wedge block$;
    if SAT$[CNF]$ then
        $x_{\text{protected}} :=$ SAT_ASSIGNMENT$_x[CNF]$;
        if $f(x_{\text{protected}}) \neq f_{\text{flipped}}(x_{\text{protected}})$ then
            return $x_{\text{protected}}$
        end
        $block := block \wedge (x_{\text{protected}} \neq x)$;
    else
        $sen := sen - 1$;
    end
end
```

Using the same sensitivity quantifying circuit from Fig. 3.3, we can build an attack algorithm that will detect inputs with high sensitivity. While the resulting key can only be definitively verified with access to an unlocked IC, finding the likely keys only requires access to the netlist. The first step in building such an attack is preprocessing the locked circuit netlist. We find a function, f_{locked}, in the circuit that has the restoration unit in its fan-in. This can be done by tracing the key inputs through the circuitry. The restoration unit is removed from the netlist, creating a circuit functionally equivalent to f_{flipped}. In TTLock, TTLock*, and SFLL-Flex this entails adding constraints such that the input is not equal to the key. In SFLL-HD, the Hamming distance between the input and key must not be at the fixed value.

After obtaining f_{flipped}, we build the sensitivity quantifying circuit. We add a constraint setting the unnormalized sensitivity to the maximum value, n (the input width of the function). This instance is put into a SAT solver searching for an input with this sensitivity level. The sensitivity is decremented until a satisfying input is found. The satisfying input is then applied to the oracle, f. If the output is the same as the simulated result from f_{flipped}, a constraint ruling out this input is added to a set of blocking clauses, *block*, and the process continues, searching for the next highest sensitivity input. If the output is different, it is a protected pattern. For TTLock and TTLock* this pattern is the key. For SFLL-Flex, this process must be repeated until all protected inputs in the LUT are found. Finally, for SFLL-HD, a total of three patterns with a mutual Hamming distance of twice the fixed value are found. The value of each key bit is then determined by the taking majority of the discovered protected inputs. The pseudo-code of the attack is listed in Algorithm 2.

3.1.3 Resistant Locking Scheme

For certain circuits, TTLock can be adapted such that the sensitivity-based attack is no longer effective. We demonstrate this process to show the limits of our attack method, however, we do not see this fixed TTLock as a viable locking method as the amount of output corruption is too small to be meaningful.

Resisting the sensitivity-based attack can be achieved by selecting an input that, after flipping its output value, is not a sensitivity outlier. This means locking an input that will subsequently be moved to a dense part of the sensitivity distribution. Here, we implement an algorithm that targets the average sensitivity as the final value. Resistance to the sensitivity-based attack must be balanced with brute force and miter-based attack resistance in which the function's input width determines the expected number of iterations. This entails locking an output function that has at least a given number of inputs. Thus, from all output functions in the circuit with input width greater than the required value, we want to find the function, f, and input value, x, such that:

$$\arg\min_{x,f} |\bar{s}(f) - (1 - s(f,x))| \tag{3.5}$$

To find the optimal input, we first rule out all functions that do not meet the desired brute force and miter-based attack resistance. For each function in the list of remaining functions, $f \in F$, we compute a mapping of input widths, $N : f \to n$, and of estimates

Algorithm 3: Sensitivity Attack Resistant TTLock

```
Input: set of functions $F$, set of input widths $N$, set of average sensitivities $A$
Output: optimal protected input $x_{\text{protected}}$, function $f$
$b := 0$;
while $b < max(N)$ do
    for $f \in F$ do
        $n := N[f]$;
        $a := A[f]$;
        $CNF := |a - (1 - s(f, x))| \leq b/n$;
        if SAT$[CNF]$ then
            $x_{\text{protected}}$ := SAT_ASSIGNMENT$_x[CNF]$;
            return $x_{\text{protected}}, f$
        end
    end
    $b := b + 1$;
end
```

of the average sensitivities, $A : f \to \bar{s}(f)$, using the same method from Section 3.1.2. We then search for the function and input pair that has a flipped sensitivity closest to the average value for the function. This is done by iteratively relaxing a bound, b, until a function is found that has an input with flipped sensitivity less than b/n from the function's average sensitivity. After finding the optimal input, it is flipped following the original TTLock method. The pseudo-code of the algorithm to find the optimal protected input is shown in Algorithm 3.

Like TTLock, a sensitivity attack resistant version of SFLL-Flex can be easily created by repeating this process multiple times. However, SFLL-HD is harder to make resistant to the sensitivity-based attack. To avoid easy detection, all flipped inputs must have low sensitivity. Finding a set of inputs with this property and are all a fixed Hamming distance from a common value is unlikely.

3.1.4 Attack Results

To assess the strength of our sensitivity-based attack against strip-functionality locking as well as our modified version of TTLock, we ran three experiments. First using a com-

Table 3.1: Sensitivity attack results for author-provided circuits using Cadence JasperGold

Technique	Circuit	N_{bits}	Time(s)	N_{iter}
TTLock	c5315	32	3	1
TTLock	c7552	32	3	1
SFLL-HD	DFX	256 (HD=32)	584	3

mercially available tool, Cadence JasperGold, we demonstrate the attack's applicability on a set of locked benchmarks provided by the authors of the respective techniques from the strip-functionality class. To further validate these results, we then extend this analysis to an additional set of generated locked circuits. Finally, we implement the attack using open-source tools and repeat the analysis on the generated circuits.

All attacks are run using a 64GB, 24-core, 2.2GHz machine. JasperGold, the commercial tool, is a formal verification suite that uses a parallel execution strategy attempting to find a solution employing several different solvers at once. In this case, our attack algorithm is implemented in TCL, a widely adopted scripting language used in digital IC design tools. Our open-source flow, uses the CaDiCaL SAT solver [57] and Python to implement the attack algorithm. For all experiments, we limit each run to a timeout of 4 hours. The implementation of this flow can be found in our repository[1].

In Table 3.1, we present the results of our attack on the circuits provided by the authors. For each circuit we show the number of bits used to lock it, the overall time to execute the attack, and the number of iterations (the number of inputs checked in the oracle). As seen, all circuits are broken, most in seconds, the largest in minutes. This is significantly faster than the expected miter-based attack time, which scales exponentially in the width of the key. As sensitivity is not considered in these locking schemes, in all cases the protected inputs are the highest sensitivity inputs keeping the required number of iterations small.

For the next experiments, we implement the strip-functionality locking techniques

[1] https://github.com/jpsety/sensitivity_attack

Table 3.2: Sensitivity attack results for generated circuits using Cadence JasperGold. For SFLL-HD, HD $= N_{\text{bits}}/8$ and for SFLL-Flex, $N_{\text{patterns}} = N_{\text{bits}}/8$.

		TTLock		TTLock*		SFLL-HD		SFLL-Flex		TTLock-Sen	
Circuit	N_{bits}	Time(s)	N_{iter}	Time(s)	N_{iter}	Time(s)	N_{iter}	Time(s)	N_{iter}	Time(s)	N_{iter}
c499	32	1	2	70	6	2	3	8	25	timeout	3106
c880	32	1	1	4	4	2	4	3	8	timeout	3480
c1355	32	1	3	3849	181	2	3	5	14	timeout	3492
c1908	32	1	1	18	2	3	4	3	6	timeout	3339
c2670	32	1	1	6	1	2	3	2	4	68	62
c2670	64	1	1	5	1	4	3	7	8	70	62
c3540	32	2	2	24	1	2	3	2	6	timeout	2096
c5315	32	2	3	1	1	2	3	2	5	timeout	3028
c5315	64	4	9	3	1	4	3	10	14	24	26
c7552	32	1	1	4	1	3	3	69	16	timeout	2700
c7552	64	2	1	19	1	3	3	4	4	timeout	2568
c7552	128	4	1	16	1	14	3	9	8	timeout	1670

Table 3.3: Sensitivity attack results for generated circuits using the SAT solver CaDiCaL. For SFLL-HD, HD $= N_{\text{bits}}/8$ and for SFLL-Flex, $N_{\text{patterns}} = N_{\text{bits}}/8$.

		TTLock		TTLock*		SFLL-HD		SFLL-Flex		TTLock-Sen	
Circuit	N_{bits}	Time(s)	N_{iter}	Time(s)	N_{iter}	Time(s)	N_{iter}	Time(s)	N_{iter}	Time(s)	N_{iter}
c432	32	0.20	1	0.15	1	0.67	3	0.63	4	timeout	11018
c499	32	0.06	2	26.64	1	0.36	3	57.14	918	timeout	12243
c880	32	0.07	1	0.14	1	0.67	3	0.37	4	timeout	13159
c1355	32	65.21	989	1271.27	377	0.59	3	425.36	2982	timeout	11938
c1908	32	0.13	1	0.67	1	1.08	3	0.46	4	timeout	17602
c2670	32	0.05	1	0.05	1	0.43	3	0.19	4	1.03	1
c2670	64	0.06	1	0.10	1	1.27	3	0.49	8	4.49	1
c3540	32	0.34	2	9.55	1	0.69	3	0.27	4	timeout	3442
c5315	32	0.19	3	0.04	1	1.58	3	0.31	4	timeout	13670
c5315	64	0.12	1	0.07	1	2.82	3	0.99	8	2.81	1
c7552	32	0.05	1	0.10	1	0.38	3	0.16	4	timeout	10079
c7552	64	0.07	1	0.72	1	2.11	3	0.53	8	timeout	12286
c7552	128	0.08	1	0.08	1	5.78	3	2.06	16	timeout	7667

and lock a commonly used set of benchmark circuits [56]. We generate locked circuits starting at 32 key bits, doubling the count until the circuit no longer has a function with at least that input width. We set both the Hamming distance used in SFLL-HD and the number of patterns used in SFLL-Flex to $N_{\text{bits}}/8$. Since the protected inputs are chosen randomly and thus will likely have high sensitivity after being flipped, the impact of these parameter choices will likely have negligible effect on the attack result.

Immediately clear from the locking procedure is that the smaller circuits don't contain a Boolean function with input width large enough to provide adequate security against a brute force attack; only three of the tested circuits can scale to 64 bits. However, since the functions with lower input width are likely to have higher average sensitivities and thus a lower chance of a protected input being an outlier, these circuits should be the most resistant to our attack.

The commercial tool attack results for the generated locked circuits shown in Table 3.2. We are able to deobfuscate all circuits locked with previous strip-functionality methods. With the exception of a few outliers, all protected inputs are found in the several attack iterations and in seconds of run time. The results for the sensitivity attack resistant version are mixed depending on the circuit and the number of bits. Two notable examples are c2670 and c5315. The first circuit, c2670, has very few high-sensitivity inputs; thus, when it is locked with our flow, the protected input is still in the highest portion of the sensitivity distribution. The second circuit, c5315, has a suitable protected input for the 32-bit locking, but not for the 64-bit. Therefore, in this circuit there is a distinct tradeoff between sensitivity attack and miter-based attack resistance. In general, it is clear that the resistance of this locking technique is highly circuit dependent.

Finally, in Table 3.3, we show the open-source flow results for the generated circuits. The results are similar to the commercial flow; however the execution times are lower. All previous strip-functionality circuits are again deobfuscated and the sensitivity attack resistant TTLock circuits have matching results. The open-source flow is able to explore a greater portion of the input space in the allotted time.

3.2 Modeling Techniques for Locked Circuits

The previous section developed an attack that identified the manipulated portions of a design, enabling their removal. Another approach we explore is to improve the modeling techniques used in the miter-based attack. In this section, we explore the use of two modeling techniques, relaxed encoding and symmetry breaking, that can dramatically reduce attack run time. We demonstrate their impact attacking a state-of-the-art scheme, Full-Lock. We then develop Logic-enhanced Banyan locking, an improved version of Full-Lock, not susceptible to these new modeling techniques.

3.2.1 Full-Lock

Full-Lock is specifically developed to be resistant to the miter-based attack [24] via increasing the execution time of each iteration. This is done by integrating SAT-hard logic into the circuit using a combination of routing obfuscation and look-up tables (LUT). The added logic is highly symmetric with many keys mapping to the same functionality. Symmetry is known to be difficult for SAT solvers, trapping the algorithm by spending time exploring solutions that are isomorphic [58]. Furthermore, unit propagation of the solver is hindered as each configuration depends on many keys: in order to determine any output of the Full-Lock circuitry, most keys must be assigned. Finally, the obfuscation is parameterized such that locking scheme's clauses to variables ratio is close to 4.26, the phase-transition density for uniform random 3-SAT (SAT instances with exactly 3 variables per clause) [59]. Intuitively, instances with a higher ratio are over-constrained making contradictions easier to find and those with a lower ratio are under-constrained with potentially many satisfying solutions. While the instances produced by Full-Lock are not uniform random 3-SAT, and therefore likely have a different optimal ratio, the locking still produces hard SAT instances.

Full-Lock utilizes configurable routing and LUTs to obfuscate a set of gates and their corresponding input connections. The configurable routing is implemented with Banyan networks, a class of logarithmic networks, that permutes connections based on a key [24]. The network is made up of a series of 2-input switch boxes that connect the inputs to the outputs, either directly passing through or switched. Additionally, Full-Lock adds the ability to invert the polarity of the signals in each switch box. Diagrams of the

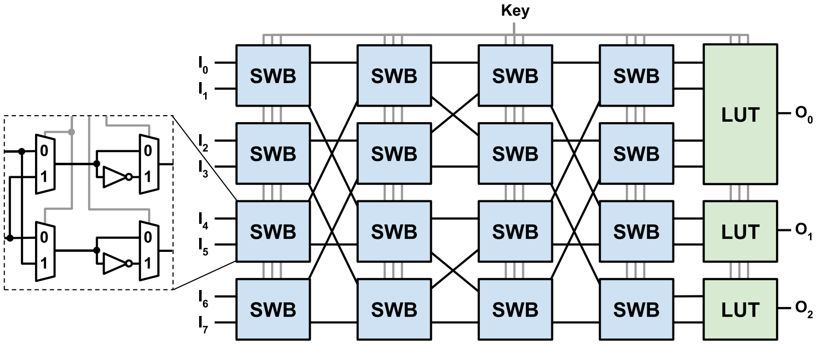


Figure 3.6: Full-Lock diagram. Each LUT replaces a gate from the original circuit; the switch boxes permute and invert their input signals.

switch boxes and overall network are shown in Fig. 3.6. The specific Banyan network configuration used has $2 * log_2(N) - 2$ stages where N is the network's input width (equal to the number of permuted lines). The Banyan network is almost non-blocking, meaning that almost all input to output connection permutations are possible.

The locking procedure is as follows. A set of gates with the desired number of total inputs is randomly selected from the circuit. These gates are then replaced with LUTs of the appropriate size. The LUT inputs are fed through a Banyan network that can permute and invert each connection. The key to the circuit is thus the concatenation of the LUT and network configuration bits. Under the correct key, each LUT will receive its original inputs with proper polarity.

The random selection of gates opens the possibility for combinational loops to be formed in the circuit. This has no impact on the circuit when the correct key is applied as all feedback paths will be broken. However, if not ruled out, these loops will corrupt the miter-based attack. Several methods of building loop-breaking key constraints have been developed to re-enable the attack [21, 60]. These loop-breaking algorithms rule out key combinations in which nodes depend on themselves determined via *taint-propagation*. We detail the specific algorithm used in section A.1.

As is, this locking method appears resistant to the miter-based attack. The authors of the original work ran the attack for 15 days without termination on instances with 32 circuit lines permuted. Additionally, the authors considered a removal attack. Even after synthesis, the added circuitry is easily identifiable due to the key lines and regular structure. Despite this, Full-Lock is also resistant to a simple removal attack as the selected gates have been replaced with LUTs and the correct interconnections and polarities of their inputs are unknown.

3.2.2 Relaxed Models

Each iteration of the miter-based attack satisfies the miter circuit while respecting the *system model*. Typically, the model is just formed from the locked circuit's netlist. The system model captures the potential behaviors of the locked circuit under different keys and is encoded into propositional logic allowing the SAT solver to generate meaningful inputs. However, the exact system model can be difficult to specify (e.g., delay-locking) or too complex for SAT solvers to efficiently handle (e.g., Full-Lock). Often, a close analog to the original behavior can be captured with a much simpler encoding. Substituting the system model can allow significant decreases in attack time, sacrificing exact functional fidelity for reduced complexity.

Several factors must be considered when building a relaxed model for a locked circuit. *First*, the model's variables do not all need to directly map to system's logic. In fact, the only requirement on the variable mapping is that the inputs and outputs remain directly mapped between the encoding and original system model so that the produced differentiating inputs can be run on the oracle and the resulting IO pair can be added to the miter. *Next*, the relaxed model must be able to produce a super-set of the IO relationships under all key values. Perhaps counter intuitively, specifying a super-set of behaviors can be easier than the exact set. *Finally*, while the key variables do not need to be directly encoded, there must be a mapping from the relaxed model back to a valid key configuration of the original system.

An example of relaxed modeling is seen in TimingSAT [61], an attack methodology for TimingCamouflage [62]. TimingCamouflage substitutes flip-flops with combinational logic delays. This disrupts a naive attack strategy because a reverse engineered netlist will be missing flip-flops that correspond to the correct functionality. It is assumed that to obtain the system functionality, an attacker must meticulously time the circuit and check all possible paths for potential combination logic delays replacing a flip-flop. However, TimingSAT simply substitutes a relaxed model, overestimating the possible locations where a combinational delay may be used as a flip-flop. In each potential flip-flop location, a MUX is inserted selecting between a flip-flop or wire. The functionality is then determined using the standard miter-based attack, solving for the proper MUX settings.

A relaxed encoding can also be used to remove key interdependence. Often the functionality of a locked circuit will depend on a large portion of the keys. To determine the output for a given input, the SAT solver must branch on many of the key variables. However, in some cases the functionality can be separated from the key variables. This allows the functionality to be selected without assigning all keys. An analogous example is encoding integers. The typical circuit for handling integers is representing them with binary numbers, however, to select an integer value all variables representing the binary number's bits must be assigned. For SAT solvers, an often more efficient strategy is one-hot encoding. Here, a value can be directly assigned by setting a single variable true (and unit propagating the others to false). In a similar sense a circuit functionality can be decoupled from the key bits, directly selecting the functionality rather than assigning all key bits.

Using this relaxed encoding strategy, we consider our example technique, Full-Lock. As previously established, the Banyan network is a SAT-hard circuit due to its large amounts of symmetry, key interdependence, and poor unit propagation behavior. Despite its complexity, the functionality is very simple: the outputs of the network are a permutation of the inputs. Due to the structure of the network, some permutations are prohibited, and others can be selected by multiple key settings. If we relax the encoding

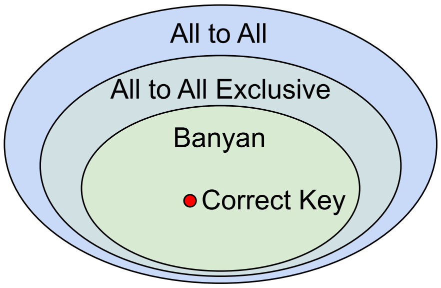


Figure 3.7: Relaxed models for Banyan network

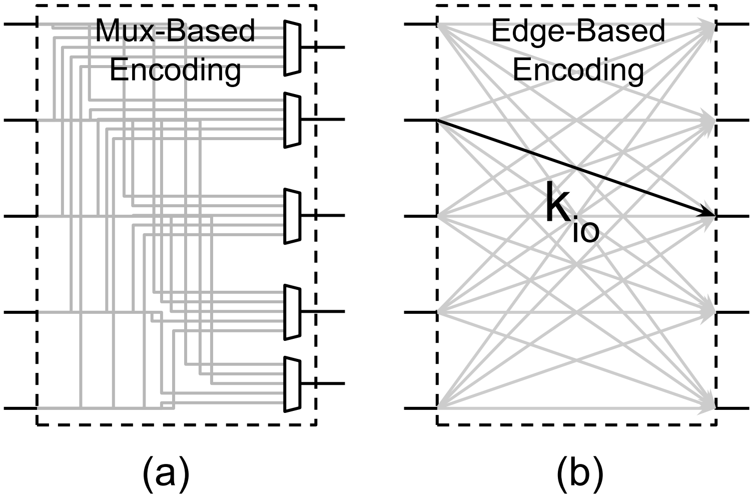


Figure 3.8: (a) MUX-based and (b) edge-based encoding schemes for the all-to-all model

of the network, allowing the prohibited permutations in our model, we can significantly reduce the complexity.

We consider two relaxed models in place of the Banyan network: *all-to-all*, wherein every input can be routed to every output, and *all-to-all exclusive*, which additionally restricts an input to be routed to only a single output. A diagram of these functionalities is shown in Fig. 3.7. The correct key is in the set of functionalities that the Banyan network allows, which is a subset of the all-to-all exclusive model functionalities, and in turn, the all-to-all model functionalities.

From a circuit designer's perspective, the natural way to encode all-to-all functionality uses an N-to-1 MUX for each output, similar to the structure depicted in Fig. 3.8(a). This can be easily specified in a high-level language such as verilog, then synthesized to a gate-level representation. The Banyan network in Full-Lock can then be substituted for these gates. Just as in the typical miter-based attack, the circuit can then be encoded into SAT via the Tseitin transformation. The all-to-all exclusive encoding can be formed in the same fashion, adding circuitry to ensure that the select bits of each MUX are different.

We also consider an edge-based strategy in which a key variable, k_{io}, is created for each possible input to output connection. A diagram of this encoding is depicted in 3.8(b). The CNF of the encoding is shown below where x_j is a variable representing a net j, I is the set of nets fanning into the Banyan network, and O is the set of nets in its fanout.

$$\bigwedge_{i \in I, o \in O} k_{io} \rightarrow (x_i \leftrightarrow x_o) \tag{3.6}$$

To ensure proper functional behavior we must also enforce that each network output is only connected to one input. This can be done using a cardinality encoding over the same variables as below:

$$\bigwedge_{o \in O} ExactlyOne(\{k_{io} : i \in I\}) \tag{3.7}$$

The edge-based all-to-all exclusive encoding is created with the additional clauses under which each input can only connect to a single output:

$$\bigwedge_{i \in I} ExactlyOne(\{k_{io} : o \in O\}) \tag{3.8}$$

Running the miter-based attack on these encodings will produce the correct mapping from the network inputs to outputs. Obtaining the corresponding key for the original system model can be done by finding a key that propagates the same paths in the Banyan network. Our models allow a greater function space, but with an encoding much more amenable to SAT solvers as we will see below.

To demonstrate the effectiveness of our relaxed models we run a series of attacks on the lock structure, comparing to the original model across several dimensions. All attacks are run using a Python implementation of the miter-based attack. The implementation uses PySAT's wrapper for the CDCL-based SAT solver CaDiCaL [63, 57]. As

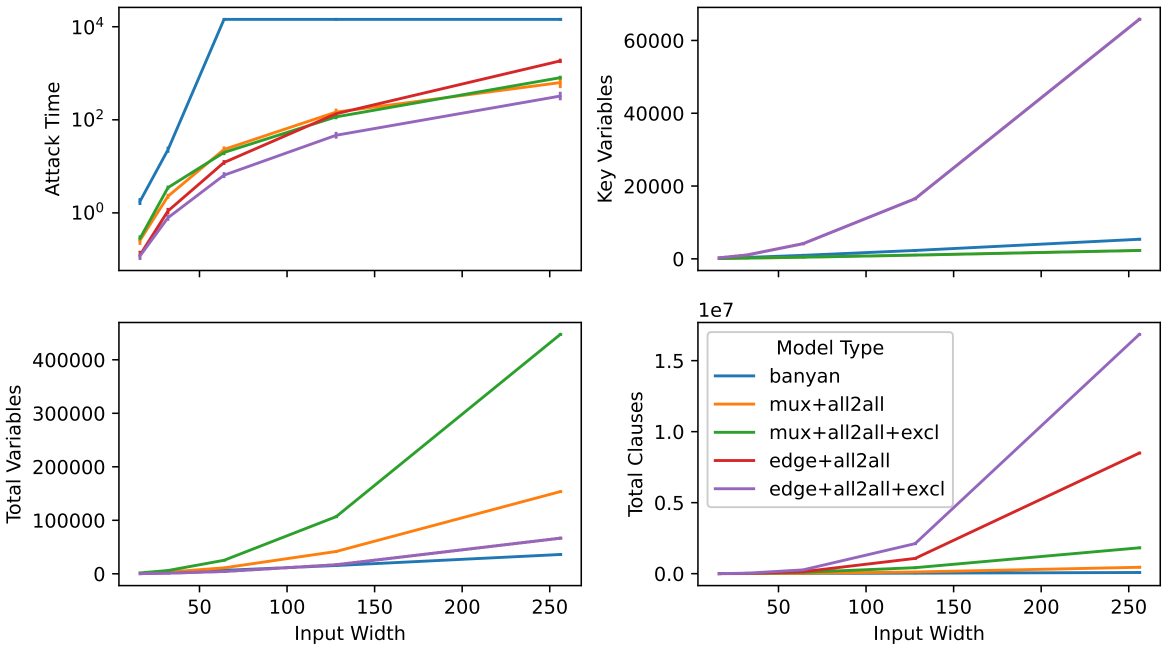


Figure 3.9: Comparison of encoding schemes for standalone Banyan network, n=10, timeout=4 hours

proposed in [64], the implementation takes advantage of the incremental interface offered by CaDiCaL adding IO constraints without restarting the solver. Each attack has a timeout 4 of hours, an iteration count of 10, and is executed on a machine with 756GB RAM and 16 2.1GHz cores. The attacks are conducted in parallel while ensuring minimal contention for resources by allotting memory greater than the maximum usage of the largest instances to each run.

We assess the impact of the model and encoding on attack run time for the standalone Banyan network; the data is shown in Fig. 3.9. We compare the five model-encoding schemes as above, namely the original Banyan network model and encoding, and all combinations of MUX-based and edge-based encodings with the all-to-all and all-to-all exclusive models. Sweeping the Banyan network size, we report several dimensions: overall attack time, number of key variables, number of total variables, and number of clauses.

Immediately obvious is the grouping of attack times. The original Banyan network times out at an input width of 64. The attack time of all proposed model-encoding pairs is significantly less, highlighting the impact of the improved models. We can clearly discern that the edge-based all-to-all encoding performs the best, quickly terminating even with an input width of 256. Considering the key variable counts, this scenario would require the original Banyan network over 5,000 keys to implement (two thirds of which are dedicated to inversions as per Fig. 3.6). While the edge-based encodings have significantly more key variables and generally the relaxed model-encoding pairs all have higher clause and variable counts, they are *significantly* easier to solve.

3.2.3 Symmetry Breaking

Another modeling technique that is not entirely exclusive from relaxed encodings, but can be applied on its own, is symmetry breaking. In the context of SAT, a symmetry is defined as a permutation of variable assignments that maps one solution onto another [65]. In the miter-based attack, symmetry results from classes of keys producing the same circuit functionality. All equivalent keys will be equisatisfiable with respect to the miter circuit inputs. If symmetry exists in the locked circuit, the attack may waste time exploring isomorphic parts of the search space.

Symmetry breaking in the miter-based attack context entails ruling out all but one key from each equivalence class. Ideally, this is done with minimal additional clauses being added to the problem, otherwise the additional problem complexity may outweigh any benefit. While not specifically labeled as symmetry breaking, this strategy has been utilized in the key-sensitization attack on Strong Logic Locking [66], wherein back-to-back key XOR gates are converted into a single XOR.

Several examples of symmetry are seen in Full-Lock. In the Banyan network, multiple keys produce the same permutations of the inputs on the outputs. Additionally, the keys that optionally invert the switch box outputs are highly symmetric: all configurations of these keys can be reduced to a single bit for each output specifying whether it is

Table 3.4: 2-Input LUT symmetries under permuted inputs.

K_0 $(I_1, I_0 = 0, 0)$	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
K_1 $(I_1, I_0 = 0, 1)$	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
K_2 $(I_1, I_0 = 1, 0)$	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
K_3 $(I_1, I_0 = 1, 1)$	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

inverted. Our relaxed models of the network already remove these two symmetries. However, there remains a significant amount of symmetry in Full-Lock's LUTs.

By themselves, LUTs have no symmetric assignments, but when coupled with external circuitry, they can become highly symmetric. This property is good for Field Programmable Gate Arrays (FPGAs) wherein flexible configurations can help meet timing, power, and area constraints, however, this flexibility hinders the miter-based attack as the same logical functionality can be specified many ways.

Full-Lock allows the LUT-inputs to be permuted, which creates *LUT configuration-input permutation* pairs that are symmetric. In Table 3.4, we show all the symmetric configurations of a 2-Input LUT with the inputs permuted. Each group with more than one equivalence is highlighted in a different shade of blue. Within the highlighted groups, permuting the inputs allows a single LUT configuration to function equivalently to the others. Thus, only one LUT configuration per group is needed. In the 2-input LUT case, only 4 out of 16 configurations are eliminated, however, as the input width increases the number of symmetric configurations grows significantly. For a 4-input LUT, there is over an order of magnitude reduction in the number of remaining configurations.

Full-Lock's input permutation symmetry can be broken by enforcing an ordering on the inputs connected to each LUT. This ensures that for every combination of inputs routed to a LUT, only a single permutation is allowed, ruling out all unnecessary configurations. To create this ordering, we add a unary-encoded number to each Banyan network output, representing the index of the connected input. This encoding uses a set of variables $\{u_{io} | i \in I, o \in O\}$. When a k_{io} is set, a series of implications rules out other keys that would break the ordering. We provide an example of these propagations in

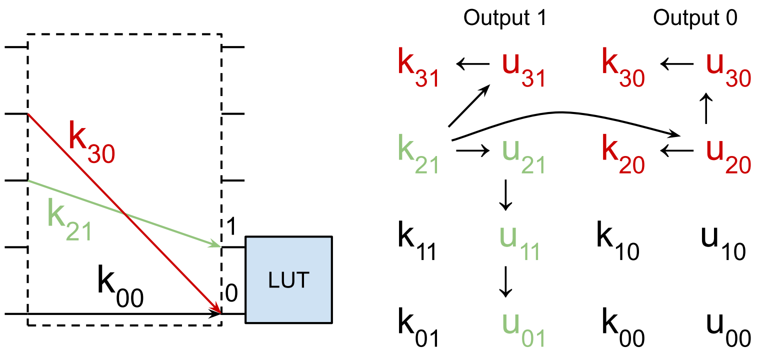


Figure 3.10: Example of propagation of ordering constraints for a 2-input LUT.

Fig. 3.10. Here, we show the unary numbers of the first two network outputs and the corresponding key variables. k_{21} is set to true, which in turn sets output 1's unary encoded number to 2. Additionally, it enforces that output 0' unary encoded number must be less than 2, ruling out k_{30}. These constraints are enforced for each LUT by adding the clauses in Eq. 3.9 to our solver. Here O_L is the set of network outputs that connect the LUT.

$$\bigwedge_{i \in I, o \in O_L} (k_{io} \rightarrow (u_{io} \wedge \neg u_{i+1o})) \wedge (k_{io} \rightarrow \neg u_{io-1}) \wedge (u_{i+1o} \rightarrow u_{io}) \tag{3.9}$$

We demonstrate the impact of the LUT symmetry breaking on attack time repeating the experiments from the previous section, using the best performing model and encoding. Since the amount of symmetry scales with LUT input size, we sweep this parameter and hold the network width fixed at 32. The resulting attack times with and without symmetry are shown in Fig. 3.11. As the LUT width increases, the advantage of symmetry breaking grows exponentially. At a LUT input width of 5, the difference in attack time about an order of magnitude.

3.2.4 Resistant Locking Scheme

Based on our attack data and the results from the original Full-Lock work, it is clear the Banyan network structure (without the modeling techniques) creates an instance that is difficult for the miter-based attack. The strengths of the network are the large number of cycles it can potentially create, the interdependence of keys, and the lack of intermediate

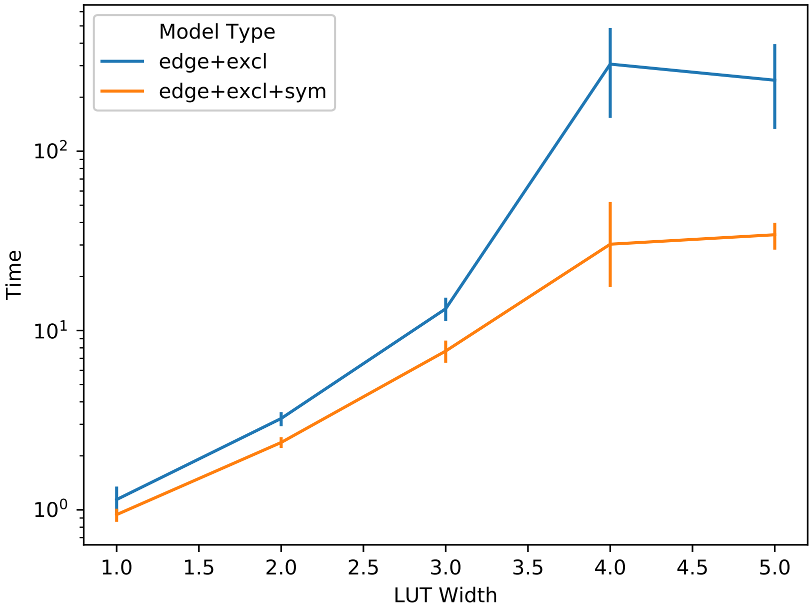


Figure 3.11: Comparison of attack time at network input width of 32 between encodings with and without LUT symmetry breaking, n=10

outputs. However, with the proposed modeling techniques, we have exposed holes in the original formulation. Here, we describe a remedy based on breaking the assumptions of the modeling techniques through the addition of logic internal to the network.

Our improved locking technique, logic-enhanced Banyan locking, uses the same Banyan structure as Full-Lock, however, the functionality is extended beyond the simple invert and permute. This is achieved by moving logic from the locked circuit into the switch boxes of the Banyan network. In the original Full-Lock switch box, two key bits are used to optionally invert the lines passing through. Now, we use these two key bits to select one of four possible functions for each switch box output. One configuration produces the correct function, the others are randomly generated decoy functions of the switch box inputs.

A diagram of the new technique is depicted in Fig. 3.12. In this small example, a 4-input Banyan network is inserted. Using the switchbox outputs as reference points, gates from the original circuit are mapped to the Banyan network. Switch box outputs

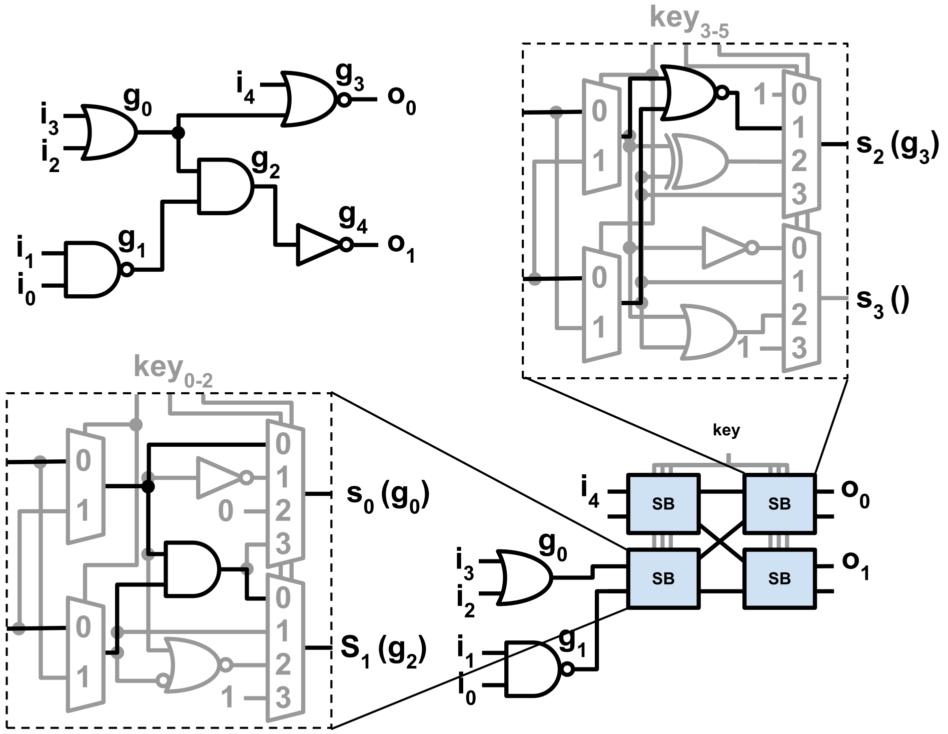


Figure 3.12: Diagram of circuit mapped to logic-enhanced Banyan network. The original circuit is shown top-left, the locked version bottom-right. The correct switch box function is highlighted in black, the decoy logic in gray.

s_0, s_1, and s_2 respectively map to gates g_0, g_2, and g_3. We show the internal logic of two of the switch boxes; the logic corresponding to the original circuit highlighted in black whereas the decoy logic is in gray. Input, i_4 feeds through the top-left switch box and gate $g4$ is mapped to the upper output of the bottom-right switch box. The network's un-mapped inputs and outputs are connected to the surrounding circuitry, ensuring no logical effect under the correct key.

As the network size is increased, it incorporates a larger portion of the design. Since there is already a significant amount of reconfiguration, we forgo the use of LUTs. The intra-network logic prohibits the use of a simplified model for the network. The correct functionality is no longer just a permutation of the inputs to the network, but rather one of a very large space of functionalities dependent on nearly all the key bits. Additionally, the large amount of symmetry has been removed; while some corner case symmetry may remain, it will be highly complex to find and probably of little value to rule out.

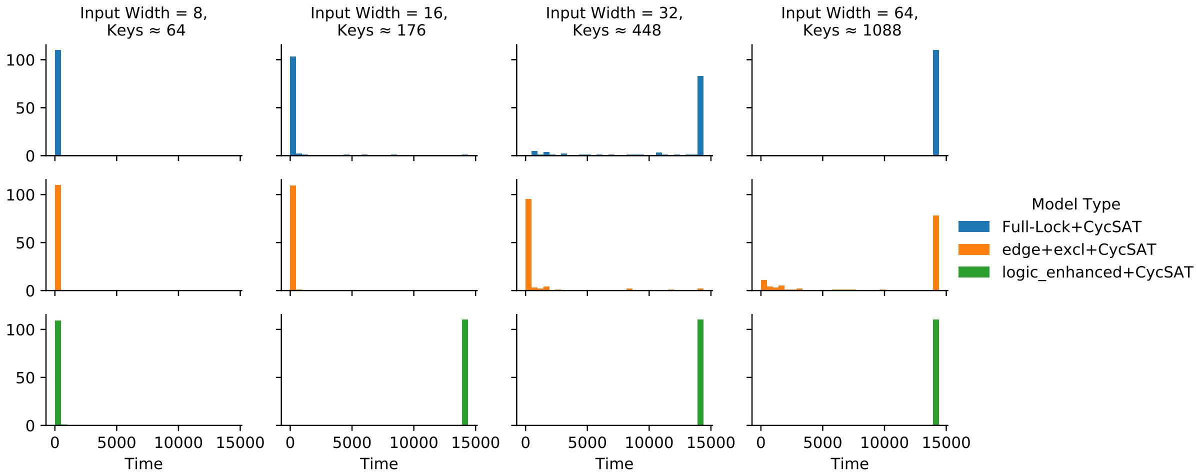


Figure 3.13: miter-based attack time for ISCAS 85 circuits locked with Full-Lock and logic-enhanced Banyan locking schemes

The insertion algorithm for logic-enhanced Banyan locking is naturally more complex than the original Full-Lock. To bring logic inside the Banyan network, we need to map the structure of the original circuit to the network. This problem can be efficiently solved using SAT, encoding the structural constraints of the Banyan network and original circuit over a set of variables that represent a mapping of a gate from the original circuit to the output of a Banyan network switchbox. We provide the full details of the algorithm in section A.2. The implementation of the insertion can be found in our repository[2].

We demonstrate the resistance of our proposed technique to the miter-based attack alongside the original Full-Lock, and the best encoding-model combination from the previous results: an edge-based, all-to-all exclusive with symmetry breaking. As described in section A.1, all attacks utilize the CycSAT acyclic constraints [21]. The techniques are run on the ISCAS85 benchmark circuits, sweeping the network input width from 8 to 64. The corresponding key widths range is around 64 to 1088. The results are shown in Fig. 3.13.

The Full-Lock run times show a trend that largely agrees with the original paper's results with the exception that some circuits at 32-input width are deobfuscated. These

[2] https://github.com/jpsety/logic_enhanced_banyan_locking

improved results are likely due to the use of a different SAT solver. Our relaxed model shows run times several orders of lower than originally reported. Most circuits at 32-input width (around 448 key bits) are deobfuscated in seconds, with some outliers taking minutes, clearly demonstrating the effectiveness of these techniques. As the input width scales to 64, many circuits are still deobfuscated, but the majority take longer than our 4-hour timeout.

Even without considering the modeling techniques, the logic-enhanced Banyan scheme provides a significantly better ratio of key bits to attack time than the Full-lock predecessor. At an input width of 16, the attack times out for all circuits. While this is good, we do not suggest that such small input widths are viable locking techniques as simple enumeration attack schemes may easily deobfuscate them. However, these results overall suggest that such entangled locks may provide a direction for resisting the miter-based attacks.

3.3 Discussion

3.3.1 Attack Insights

In this chapter we have developed two attacks that take differing strategies in deobfuscation. The first uses analysis of Boolean sensitivity to isolate the modified portions of a circuit locked with strip-functionality techniques. In essence, this is simply a removal attack. The key insight is the identification of a metric under which the locked circuit's modified portions, the protected inputs, have a strong signal. Under the right lens, what may seem to be an undetectable change becomes obvious. Notably, the signal that this attack identifies depends on both the locking technique and the circuit being locked, suggesting that designers should be careful when applying techniques to new circuits.

Other metrics such as signal probability have also enabled removal attacks. In general, the effectiveness of removal attacks has been driven by locking schemes that rely on a single point of failure. Ideally, if workable metric is identified, the locking schemes would exhibit graceful degradation wherein the metric would have to deobfuscate many sub-instances correctly rather than a single point in the circuit.

Our second attack utilizes two modeling techniques, relaxed modeling and symmetry breaking, to decrease SAT-based attack run times. In the case of Full-Lock, our experiments show that these modeling techniques are highly effective, reducing run times by many orders of magnitude. Locking schemes that have easily relaxed functionality or have comprehensible symmetries are potentially vulnerable to these modeling techniques. A potential target for applying these modeling techniques are FPGA-based locking strategies. Finally, important to the success of our strategy was trying different models and encodings of the system, which produced significant differences in attack performance.

3.3.2 Shortcomings of Proposed Fixes

For both attacks, we provide resistant locking schemes. While these improvements fix the vulnerabilities, neither scheme offers a good balance between overhead and security. For strip-functionality locks, we show that if the locking procedure considers sensitivity during insertion, a designer may be able to avoid selecting inputs that will become outliers when flipped. However, the locked circuits must have inputs that move into denser areas of the sensitivity distribution when inverted, making the security of the technique highly circuit dependent. As we demonstrated, many circuits do not have output cones that have an input width large enough to resist the miter-based attack *and* the needed distribution of input value sensitivities. Thus, this class of techniques has a tradeoff between miter-based attack resistance and sensitivity attack resistance. The result is a narrowed set of situations under which strip-functionality locking is applicable. Notable examples that have the proper distributions are cryptographic circuits and algebraic circuits. While both classes of circuits will likely have many inputs that can be hidden from the sensitivity attack, it should be noted that these classes are highly regular structures that may be subject to different types of analysis.

Furthermore, if our version of TTLock or a similar construction is used, it would be prudent to more accurately characterize the sensitivity distribution before selecting inputs to lock. This could be done using approximate model counting techniques [67] on the sensitivity circuit. Additionally, it should be noted that the sensitivity attack speed can likely be substantially improved via parallelism and utilization of AllSAT solvers. A final note regarding the improved version of TTLock, as the authors of the strip-functionality techniques have noted, the small number of protected inputs decreases the amount of corruption in the circuit under an incorrect key. The trade-off is unsatisfying: the attacker has a harder time searching for the incorrect inputs, but the probability of those incorrect inputs being encountered reduces just as much.

We have also described logic-enhanced Banyan locking, an extension to the Full-Lock method, that appears to be resistant to the proposed modeling techniques. We demonstrated promising initial attack results, showing that structure of the Banyan network combined with randomly selected decoy logic is not only a mechanism of resisting convergence in the miter-based attack, but also harder to deobfuscate than the original Full-Lock for the particular CDCL-based SAT solver used. Of course, this resistance may change with some additional insight or varied attack strategies. One potential example is timing constraints, wherein every valid key must produce a circuit with a critical path less than the period. Just like the acyclic constraints that are necessary for the attack to complete, timing constraints could rule out significant portions of the key space.

While the adverse effects of such improvements are unexplored, one known issue is the overhead associated with this technique. Delay overheads as high as 70% have been observed with a 32-input width logic-enhanced Banyan locking scheme. This cost of security motivates the exploration of new locking methods that utilize similar principles of irregular, densely interconnected logic, but without exacerbation of the design's critical paths.

Chapter 4

Keyed Sequential Elements for Low-Overhead Locking

In this chapter, we propose a novel logic locking mechanism, latch-based logic locking, that adds programmable phase latches within the design. This technique enables highly interdependent key structures similar to those proposed in the preceding chapter, but without significant overhead. The latches enable the obfuscation of functionality on multiple levels: manipulating the size and location of the circuit's state, corrupting of the circuit's timing behavior, and adding decoy logic.

To validate this technique, we have developed a design flow that leverages existing commercial synthesis tools. We use this flow to provide evidence of the small design overhead on a set of industrial cores and benchmark circuits from synthesis runs as well as taped-out place-and-route runs. We assess the security of the technique by developing attacks under both the netlist and oracle attack models. Our analysis shows that latch-based logic locking is SAT/model checker-based attack resistant and requires minimal overhead, enabling the insertion of many keyed latches.

4.0.1 Latch-Based Design and Retiming

Latches are level-sensitive sequential elements that are transparent when the clock input is logic 1 and hold a sampled value when the clock is logic 0. A typical master-slave flip-flop contains two latches: a master latch with an inverted clock phase followed by a slave latch with nominal clock phase. When the clock is high, a sampled value from the last clock cycle is held by the master latch and loaded into the slave. When the clock is low, the slave latch maintains this sampled value, while the master loads a new sample. Together, the latches create an edge-triggered flip-flop, propagating the input data to the output on the rising edge of the clock.

Flip-flops can be separated into individual latches and then, through a set of transformations known as retiming, moved through the fixed logic of a circuit [68, 69]. A functionally correct retiming will maintain the cycle delay of every path through the latches. *Cycle delay* is defined as the number of required clock cycles for data to propagate along a path. Thus, all paths through the original flip-flop maintain a cycle delay of 1, passing through an inverted phase latch, then a nominal phase latch. Latch retiming is used to reduce the critical path of a design, shifting the amount path delay between the positive and negative phase of the clock, as well as reduce area by merging latches. The level sensitivity of latches enables cycle sharing between the positive and negative phases of the clock cycle allowing more flexible signal arrival times. This property is often exploited in designs with tight timing constraints [70, 71]. In this work, we use the flexible arrival time to increase the modeling difficulty of an obfuscated circuit.

4.1 Latch-Based Logic Locking

In developing a locking system, we consider maintaining circuit frequency a topmost priority. Toward this goal, we utilize latches to lock a circuit via two mechanisms: programmable path delay and programmable logic. This creates a locking system in

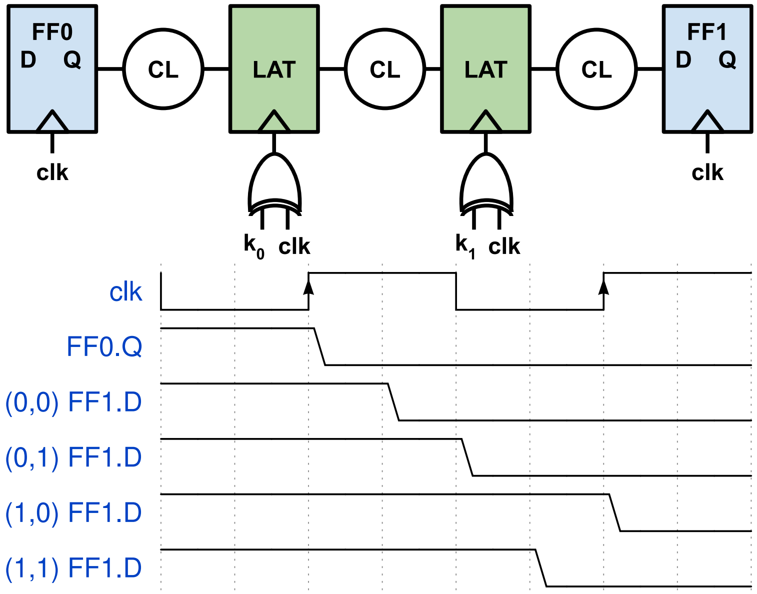


Figure 4.1: Diagram and functional waveforms of logic locked with programmable path delay via latch phase modulation.

which the critical path logic can be obfuscated while remaining largely unmodified. Both mechanisms employ decoy latch elements to cheaply create uncertainty as to the circuit's correct function.

4.1.1 Programmable Path Delay

Flip-flops from the original design can be replaced by their constituent latches and re-timed. As shown in Fig. 4.1, phase-programmable latches can be created using a key to selectively feed a latch a nominal or inverted phase clock. This not only obfuscates the design, but also potentially decreases the minimum clock period due to cycle sharing. Depending on the key setting, these phase-programmable latches can manipulate both the cycle delay *and* propagation delay along a path, thereby changing the functionality of the design via two mechanisms. *Propagation delay* is defined as the amount of time necessary for a signal to travel from its launching point to capturing point. An incorrect cycle delay will change the logical behavior of the design whereas an incorrect propagation delay will cause timing violations and undetermined behavior. The pair of latches

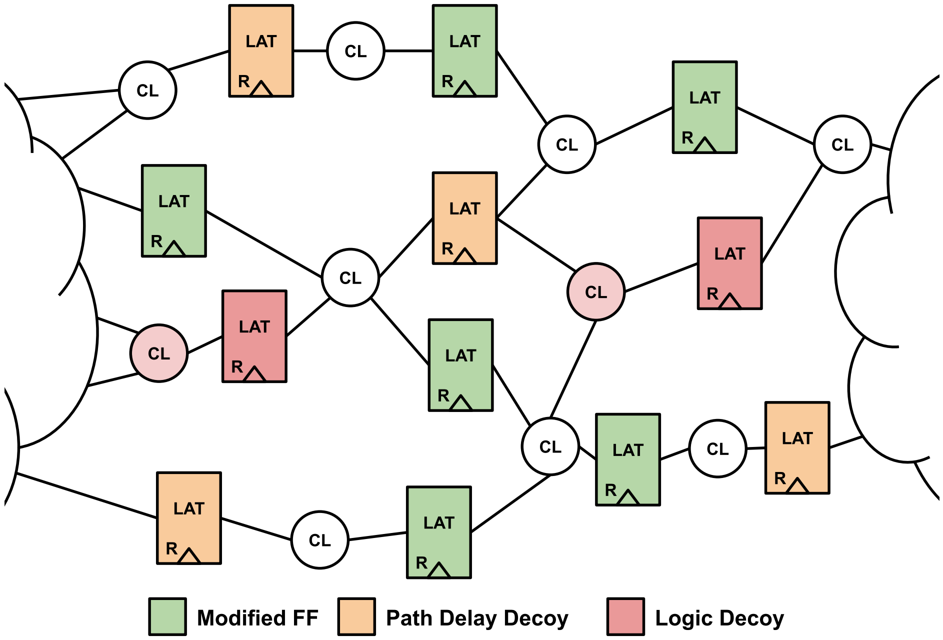


Figure 4.2: Conceptual view of latch-based logic locking. A set of interconnected flip-flops is converted to programmable latches with added decoy latches and logic. Each latch can operate on either clock phase, hold clear, or output constant logic 0.

in Fig. 4.1 can be set to four unique functionalities as both types of delay are modulated.

To add uncertainty as to a given latch's function, path delay decoy latches can be added to the circuit. These decoys allow for modulating both types of delay, forcing the adversary to solve a problem across multiple domains. The path delay decoys can be held open by setting the clock input to logic 1. In this case, the decoy latches behave as a buffer, adding only a slight propagation delay to the path. As shown in Fig. 4.2, path delay decoy latches (orange) can be inserted along paths in the original combinational logic of a circuit.

Along with their low delay impact, latches are well suited for obfuscating a circuit's path delays due to their increased complexity in modeling as compared to edge-triggered elements. The reasons that enable latches to decrease cycle times are the same that allow us to utilize them as a source of obfuscation. Both the cycle and propagation delay added by each latch are a function of the phase of the upstream latches along a given path, thus increasing the computation required to characterize a given key setting.

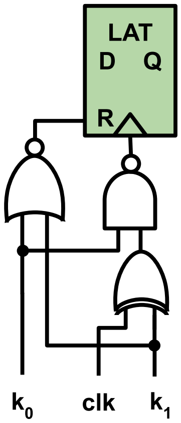
Additionally, because signals can be launched from the latch at any point within its clear phase, checking whether all signals are legally timed under a given key becomes more challenging.

4.1.2 Programmable Logic

In conjunction with the manipulation of delays in the circuit, we can add decoy logic within the design to create uncertainty in the functions being computed. Keying the reset pin of the phase-programmable latches, creates a simple method of adding decoy logic. Additional latches are inserted in the design, each along with a fan-in cone of decoy logic. The latch output is combined with existing logic such that the functionality is not changed if the latch output is logic 0. When the correct key is applied to the circuit, the reset pin is forced to logic 1 (or logic 0 depending on the reset polarity), thus maintaining the intended function. Example logic decoys and corresponding logic are shown in Fig. 4.2, highlighted in red.

The added programmable logic allows flexible insertion of latch-based logic locking. Circuits with sparse interconnection can be augmented to increase the number of paths through the locked portion. Structures known to be harder to resolve, such as reconvergent fan-out and sequential feedback paths, can be selectively added within this additional logic. Furthermore, we can emulate the highly entangled structures of the previous chapter. Importantly, these modifications can be added while leaving the critical path of the design largely unchanged.

Adding this key state results in a total of four latch configurations and thus two key bits per latch. This forces the adversary to determine for each latch whether it is a **positive phase latch**, **negative phase latch**, **delay decoy**, or **logic decoy** We show the control logic and corresponding truth table in Fig. 4.3.



k_0	k_1	LAT.R	LAT.CLK
0	0	1	1
0	1	0	1
1	0	0	!*clk*
1	1	0	*clk*

Figure 4.3: Diagram and truth table of latch clock and reset control signals. Control circuitry that determines the latch function is connected to the clock and reset pins. The truth table's row colors correspond to the associated latch type.

4.1.3 Insertion Flow

I. The first step of the insertion flow is the selection of a set of interconnected flip-flops within the original netlist. Increased interconnection forces the locked portion of the design to be considered as a whole due to the dependence of a latch's functionality on the rest of the group. In conjunction, minimizing the controllability and observability into the group makes each scenario harder to rule out. This task can be naturally modeled as community finding in graphs. We adopt a flow-based community finding algorithm to produce appropriately sized groups [72] of flip-flops from the original design. From a sample of these groups, the group with the lowest cumulative delay is selected, allowing low overhead for decoy insertion.

II. The selected group of flip-flops is subsequently converted into latches and retimed. Standard synthesis tools do not support latch retiming, however their ability to retime flip-flops can be leveraged with a previously established procedure [68, 73]. First, each latch of the original flip-flop is replaced by a flip-flop, depicted in Fig. 4.4a-b. Then, the flip-flops representing the positive phase latches are held fixed, while the flip-flops representing negative phase latches are retimed. This is then repeated, fixing the negative phase latches and retiming the positive, shown in Fig. 4.4b-c. These flip-flops are

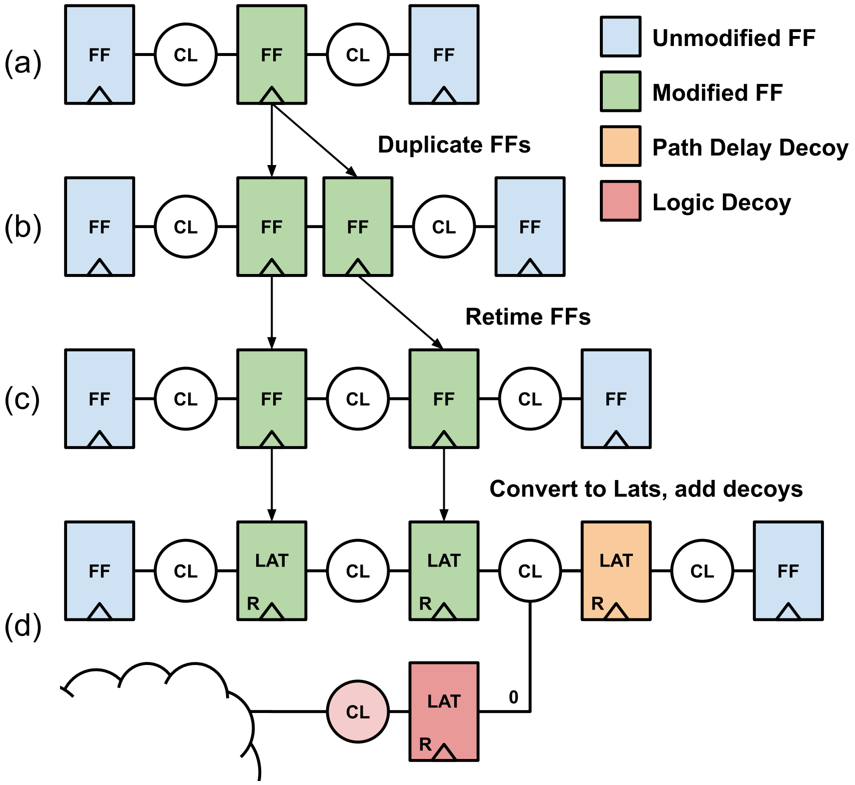


Figure 4.4: Flip-flop to latch conversion and decoy addition. A flip-flop is duplicated, retimed, and converted to latches. Two types of decoy latches can be added to the paths in the fan-in and fan-out cones of the latches. This example shows a single flip-flop, however in practice an interconnected group is converted.

then replaced by their respective latch counterparts, depicted in Fig. 4.4c-d.

III. Next, as shown in Fig. 4.4d, the two types of decoy latches are inserted, starting with the logic decoys. These decoy latches and corresponding logic are inserted to again maximize the interdependence of the existing latches in the design without creating additional control and observation points. For each decoy added, a set of fanout and fanin latches with minimum interconnection are greedily selected. This selection is elaborated in Algorithm 4. The algorithm uses the subroutines *startpoints* and *endpoints* that return the sequential elements and ports in the fanin/out cones of a set of gates. The latches with the least number of connections to the rest of the group are selected.

After the selection, a random logic cone is constructed with the fanin latches as inputs, connecting it to the input of a new decoy latch. The output of the latch is then

Algorithm 4: Logic decoy latch insertion

```
Input: $C$, $n_{\text{logic}}$, $max\_fio$
for $i \in [1..n_{\text{logic}}]$ do
    $n_{\text{fanin}} := randint(1, max\_fio)$;
    $n_{\text{fanout}} := randint(1, max\_fio)$;
    $potential\_fanin := C.latches | startpoints(C.latches)$;
    $potential\_fanout := C.latches | endpoints(C.latches)$;
    $fi\_sort\_func :=$ lambda $g : |endpoints(g)\ \&\ potential\_fanout|$;
    $selected\_fanin := sorted(potential\_fanin, fi\_sort\_func)[0 : n_{\text{fanin}}]$;
    $fo\_sort\_func :=$ lambda $g : |startpoints(g)\ \&\ potential\_fanin|$;
    $selected\_fanout := sorted(potential\_fanouts, fo\_sort\_func)[0 : n_{\text{fanout}}]$;
    $insert\_logic\_decoy\_latch(selected\_fanin, selected\_fanout)$;
end
```

Algorithm 5: Delay decoy latch insertion

```
Input: $C$, $n_{\text{delay}}$, $max\_fio$
$allowed\_startpoints := C.latches | fanin(C.latches)$;
$allowed\_endpoints := C.latches | fanout(C.latches)$;
$allowed\_gates := \{g \in C \mid (startpoints(g) - allowed\_startpoints) = \emptyset\ \&$
    $(endpoints(g) - allowed\_endpoints) = \emptyset\}$;
for $i \in [1..n_{\text{delay}}]$ do
    $g := sample(allowed\_gates)$;
    $insert\_delay\_decoy\_latch(g)$;
end
```

connected to nets in the fanin of the selected fanout latches. This output logic is created such that if the latch is controlled to constant logic 0 through the reset pin, there will be no effect on the downstream original logic.

IV. Delay decoys are subsequently inserted in locations within the collective fanout and fanin cones of the existing latches. Using Algorithm 5, the locations are selected such that they do not have control and observation points other than those of the existing latches. Additional filters can be used to select locations where the slack is greater than the delay through a clear latch, but in general we found these cells had little impact on the overall minimum period.

V. Finally, as depicted in Fig. 4.3, the logic to control the clocking and reset of all

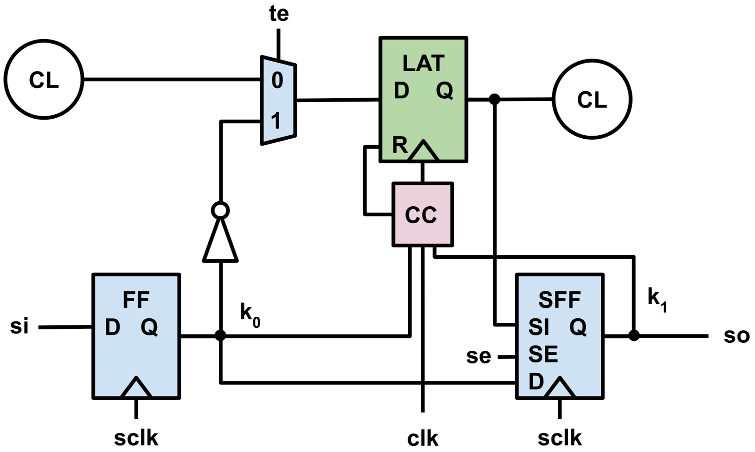


Figure 4.5: Preliminary DFT infrastructure. CC represents clock and reset control circuitry from Table 4.1.2.

the latches is added. Each latch is controlled with two key bits. The four resulting states correspond to a clear buffer, a positive phase latch, a negative phase latch, and a constant logic 0. During timing optimization and analysis, this logic is constrained to the correct key setting so the tool can target proper function. To verify the locking process did not corrupt the functionality, the correct key is fixed and a sequential equivalence check between the modified and original design is run. Additionally, timing-annotated functional simulations provide further assurance that the timing constraints have been properly specified.

4.1.4 Design for Testability

A primary concern when developing an IC is testability. As discussed in section 2.1.1, Design flows typically employ scan-based flops to serially load test vectors into the circuit; however, this design for testability (DFT) mechanism is not directly compatible with a latch-based design. An additional concern for logic locking is the storage of key bits; inherently latch-based logic locking requires two bits of storage per latch. We tackle both problems with a dual purpose scan architecture depicted in Fig. 4.5.

In order to maintain test coverage of our locked circuits, we connect the input and output of each latch to a flip-flop. During normal operation, the same flip-flops store the latch's key bits. These scannable test points are inserted such that the latch path delay is only minimally increased, adding a single MUX delay and a small capacitance. During test mode, the structure allows the output of each latch to be observed and controlled to an arbitrary value, emulating a full scan methodology. In addition to the combinational logic, faults in the added clock tree logic can be covered. We implement this initial version of the test infrastructure to demonstrate that the latch-based logic locking does not significantly impact testability; the results are shown later in Section 4.3.2. This DFT strategy as outlined maintains low delay impact but sacrifices area for ease of implementation. This strategy can likely be improved using known techniques explicitly designed for latches, namely level-sensitive scan design (LSSD).

There exists an inherent tradeoff between testability and security. The additional scan infrastructure significantly increases the controllability and observability of a design, a valuable attack vector for adversaries who have oracle access. To account for this threat, latch-based logic locking must be coupled with mechanisms to limit adversarial access to the scan chain. One viable solution is fusing external scan access with a fuse. The implementation of fuses has been previously demonstrated and widely used in practice [74, 75].

4.2 Attacking Latch-Based Logic Locking

The most critical aspect of evaluating latch-based logic locking is its attack resistance. In this section, we demonstrate the technique's resistance to several existing as well as newly developed attack methods. The methods are grouped by attack model.

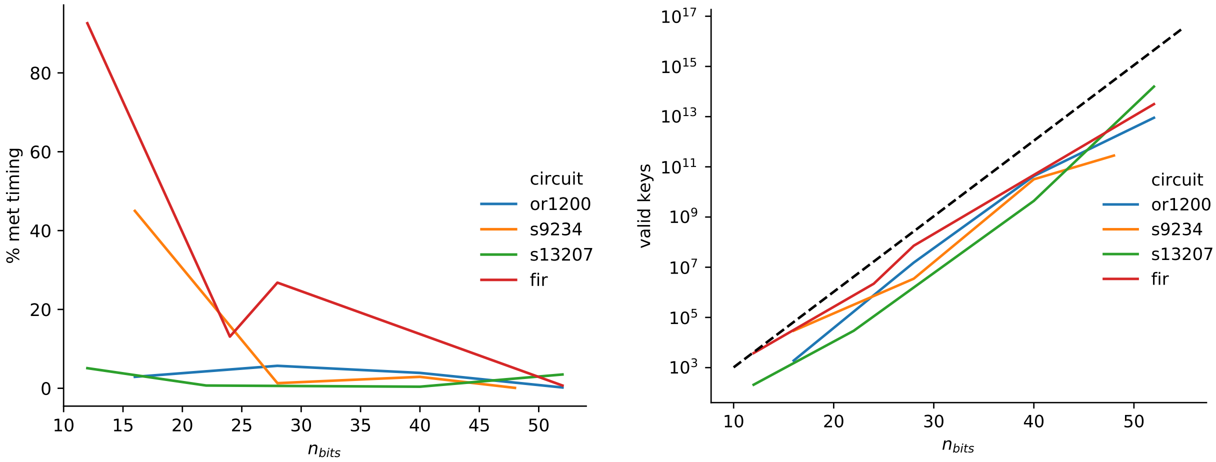


Figure 4.6: Enumeration of keys that satisfy timing constraints, $n = 1000$

4.2.1 Netlist-Based Attacks

Timing Analysis

Under the netlist attacker model, the adversary has access to the locked netlist. Furthermore, a foundry likely can infer timing information such as the clock period. The clock period can be estimated via analysis of unkeyed paths or directly obtained through analysis of the clock generation circuitry. Assuming every path in the correct circuit meets timing and that the clock period is known within some bound, this adversary can rule out key combinations that violate timing.

To understand the effect of such an attack, we run an experiment to determine what portion of the key space meets timing. Taking several circuits and sweeping the amount of locking applied, we sample key values. For each value, we apply the corresponding timing constraints to the circuit and run static timing checks at the correct design frequency. From this set of samples, we approximate the portion of the key space that meets timing. Applying this value to the key space as whole, we determine an estimate of how many keys are valid.

The results are shown in Fig. 4.6, where on the left we plot the percentage of combinations that meet timing and on the right the estimate of total valid configurations.

For some of the circuits, the portion of valid configurations decreases as the number of bits increases. However, this reduction is nullified by the exponential growth of the size of the key space. Thus, the estimated number of valid keys grows exponentially in all cases.

This exponential trend is helped in part by the timing of the latches, which makes them significantly more flexible than flip-flops. In checking timing constraints, each latch is initially allocated half the clock period to resolve the upstream combinational logic and capture the value. Because latches are clear for half a cycle, paths that have timing slack can share it with the upstream or downstream neighbors. Excluding internal latch delays, the maximum amount of cycle sharing limits the propagation delay through any contiguous set of latches on the same phase to at most the clock period, t_{period}. Additionally, the propagation delay through any contiguous set of latches with only 1 transition in phase is limited to $1.5 \times t_{period}$.

While here we have just checked the timing of sampled keys, developing constraints that map keys to a valid timing could be useful in oracle-based attacks such as the miter-based attack. This is a challenging problem as the timing of a given latch depends on more than just its immediate neighbors. Moreover, logic decoys have the ability to create physical connections that are not sensitizable under certain keys. These non-sensitizable paths are false paths that must be first detected and then removed from timing consideration. While this is straightforward for a single key, general constraints are much more difficult to specify, potentially necessitating an interface with an SMT solver [76].

Structural Analysis

While the previous timing analysis determined if a fully specified key value was invalid, we can use local structural analysis to predict a single latch function. This is similar to the local analysis discussed in 2.3.1. Here, we specify a set of metrics attempting to capture the domains in which some informative signal could be produced that would

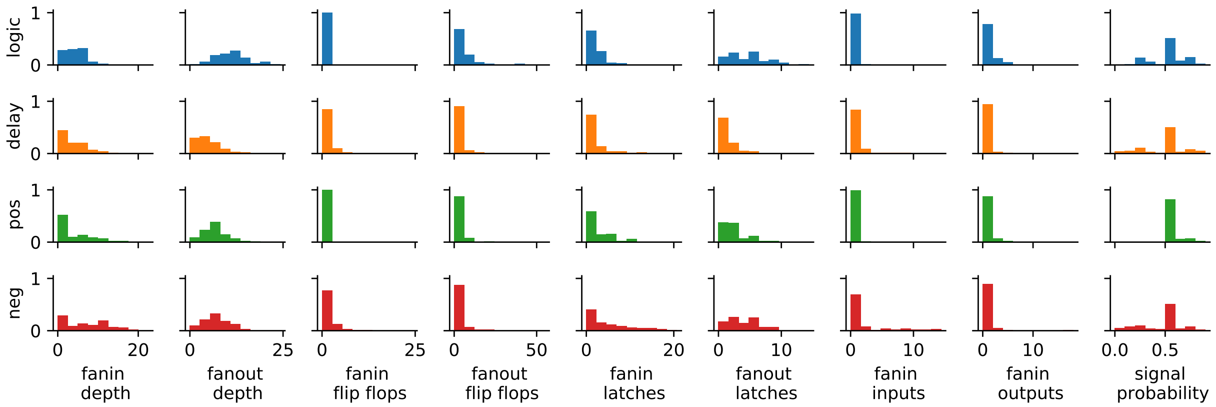


Figure 4.7: Histograms, separated by latch type, for structural metrics. Plots are generated from 60 locked circuits with varying amounts of LBLL bits.

help classify a latch into one of the four possible configurations. The metrics we select are fanin and fanout logic depth; latch, input, output, and flip-flop counts in the fanin and fanout cones; and signal probability. Collectively, these metrics give a notion of how the latch is interconnected with the rest of the circuit.

We run these metrics on a set of circuits locked with varying amounts of latches. In Fig. 4.7, we show the distributions of the individual metrics. Immediately evident is that there is no single dimension that definitively predicts a latch's function, a good first positive result. However, there are some dimensions, such as fanout latch count, that have noticeably different distributions, suggesting some predictive information exists.

While the individual metric histograms reveal very little about the value of a given latch, a classification algorithm may be able to make a prediction when all dimensions are considered. To assess our locking scheme's resistance to such an attack, we run the metric data on a set of classification algorithms. We select ten common classification algorithms with open-source implementations [77]. For each algorithm, we run a cross validation on our data with ten folds of the data. The training set consists of 1506 latch data points. In Fig. 4.8, we plot the prediction accuracy distributions across the folds. Interestingly, we see that despite the similarity among the individual distributions, the classifiers can consistently do better than chance (i.e., 25% accuracy). The best perform-

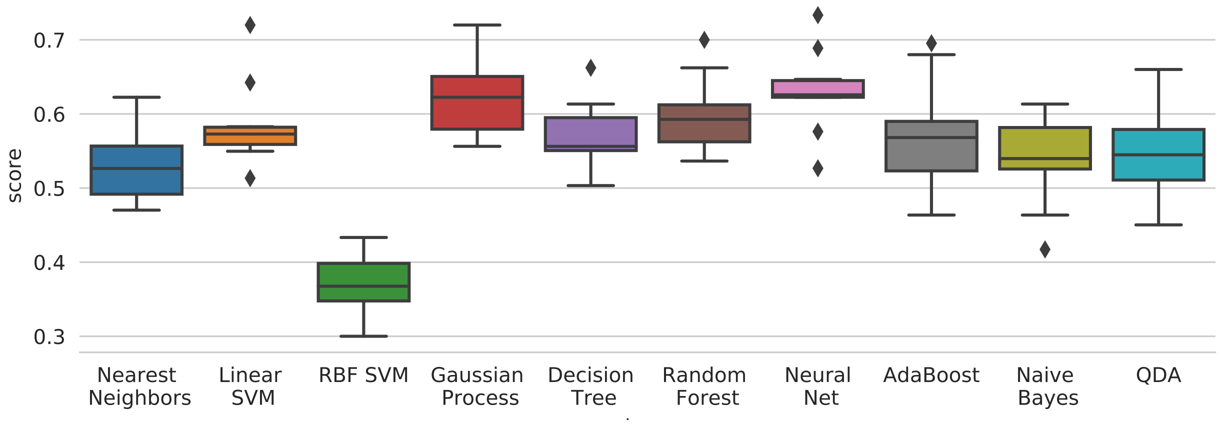


Figure 4.8: Cross-validation scores on classifiers trained on the structural metric dataset.

ing classifier is a neural network that on average can predict the functionality with 63% accuracy.

These predictions are significantly better than chance, and additional metrics may provide more information further increasing the accuracy. However, these predictions alone are unlikely to give an adversary enough information to reverse engineer a locked circuit using only the netlist. As we will discuss in chapter 5, even in scenarios where the accuracy is significantly higher, finding a circuit with minimal functional corruption is unlikely, especially when the amount of locking is scaled. A more tenable route may be the use of these predictions to increase the speed of an oracle-based attack. Nevertheless, the high prediction accuracy suggests that more should be done to reduce signal on these domains.

4.2.2 Oracle-Based Attacks

If the attacker has access to an oracle circuit, the key can be deduced via a model checker-based attack. As discussed in section 10, the model checker attack algorithms use a miter circuit to find IO sequences that produce varying behavior for different keys. The IO sequences are applied to the oracle circuit and the learned correct behavior is re-encoded into the problem as key constraints. While these algorithms are complete, meaning they will eventually terminate with a correct key, their performance is limited by the sheer size of the key, input, and state spaces. Moreover, existing attacks [29, 30] have incompatibilities that must be addressed before they can be applied to latch-based logic locking.

Incompatible Assumptions of Previous Attacks

The first issue stems from the assumption that a circuit's state is able to be reset to a potentially unknown, but *fixed* value. In many circuits this assumption is untrue. IC designers frequently include flip-flops with no reset in their designs to save both area and power. Often, only a core portion of sequential elements are set to a fixed state during reset, leaving other sequential elements in their previous, unknown states. The sequential elements that are uninitialized will eventually be set to a known state during the circuit's operation. Examples of this behavior include the register file of many processors or the datapath registers of AES.

Previous attacks use this fixed reset assumption to repeatedly find differentiating input sequences starting from the reset state. Even if the fixed reset value is unknown, these methods are still viable by treating the start state as an unknown value similar to a key input. If the fixed reset assumption is violated, assumed stable values will actually be changing over time and the learned IO relationships that depend on these variables will be corrupted. Since many circuits violate this assumption and latch-based logic locking adds state that potentially has an unfixed reset, we must generalize the attack to account for this behavior.

The second issue is that previous attacks utilize a combinational equivalence check as a termination condition to run on every iteration. These checks are incompatible with latch-based logic locking, resulting in wasted attack time. The termination condition verifies the logic faning into each output and state element is functionally equivalent under all remaining keys.

The added state elements and decoy logic of latch-based logic locking potentially

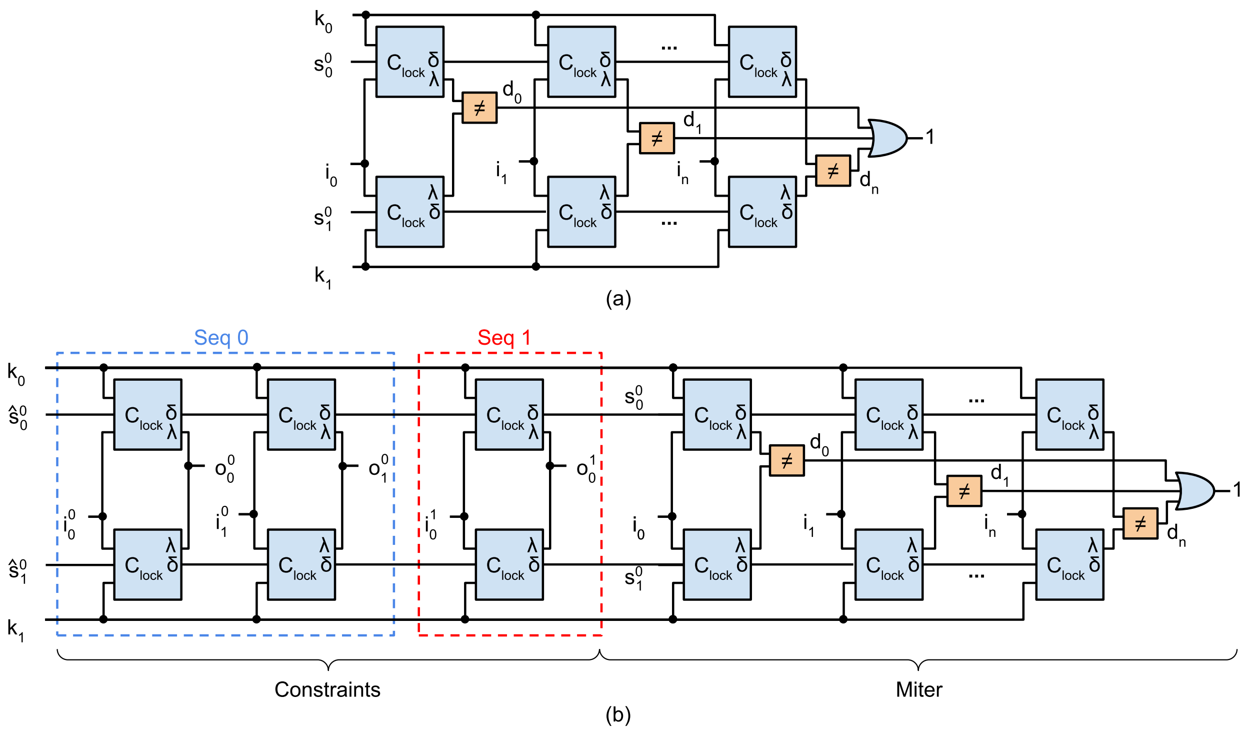


Figure 4.9: (a) Miter circuit used in our modified model checker-based attack. The free variables of the system are the two key inputs, the initial state of all sequential elements, and the inputs at every unrolled cycle. The circuit is unrolled by connecting the input of each state element from a cycle to the output of the same state element of the next cycle. (b) Addition of a learned IO constraint to the miter circuit. This limits the possible key and initial state pairs.

creating retimings of the original circuit. This means that, although all keys may be functionally equivalent, under different keys active state elements may be moved to different locations in the logic. This changes the state encoding of the system, resulting in the combinational equivalence check always failing, even when the attack has terminated.

Model Checker-Based Attack with Non-Deterministic Reset

The generalized model checker-based attack is a straightforward modification to the previous algorithms. In short, it changes the way constraints are constructed, maintaining a sequence of inputs throughout the attack as opposed to restarting from a fixed reset state each iteration. We will go through a description of the attack below.

The attack utilizes a miter circuit formed from two copies of the locked circuit. The miter circuit is specified with an FSM as introduced in section 2.1.1. The FSM's input variables are $i \in \{0,1\}^m$, the same as the locked circuit's inputs. The output is a single variable, $d \in \{0,1\}$. The output bit indicates that a difference between the circuit copies has been found. The state variables, (s_0, s_1, k_0, k_1), are the state of the locked circuit copies and the key inputs. The key inputs are modeled as state with an identity next-state function to enforce their constant value. Correspondingly, the overall miter's next-state function is $\delta_{\text{miter}} = (\delta_{\text{lock}}(i, s_0, k_0), \delta_{\text{lock}}(i, s_1, k_1), k_0, k_1)$, where δ_{lock} is the locked circuit's next state function. Notice how the keys are simply set to the same value every cycle. The miter's output function is $\lambda_{\text{miter}} = \lambda_{\text{lock}}(i, s_0, k_0) \oplus \lambda_{\text{lock}}(i, s_1, k_1)$, where λ_{lock}, is the output function of the locked circuit. On the first attack iteration all states are allowed as initial states, thus where k is the key length and l is the width of the locked circuit's state, the miter's initial state is $S^0_{\text{miter}} = \{0,1\}^{(l+l+k+k)}$.

The FSM is given to a model checker along with the target property $d = 1$. While the exact unrolling process differs depending on the model checker and algorithm used, roughly speaking, the model checker unrolls the circuit as depicted in Fig. 4.9(a), checking for the property at each cycle. The circuit's free variables are the separate key inputs, separate initial states, and shared inputs for each unrolled cycle. The first cycle input is constrained such that reset is active; in subsequent cycles, the tool is free to control reset. The model checker is given the task of finding an input sequence, $\mathcal{I}$, that satisfies the miter circuit, thereby producing a difference on the outputs. This output discrepancy can be produced by a difference in the keys or initial states applied to the circuit copies. The tool searches for such a sequence using unbounded model checking algorithms, unrolling the circuit as needed.

Once a valid input sequence is found, the oracle is used to find the corresponding outputs. The input sequence is run on the oracle, capturing the correct output and holding the clock steady on the last cycle, maintaining the state within. The produced IO relationship is then encoded as a constraint on the miter circuit's initial states, ruling

Algorithm 6: Model Checker-Based Attack with Non-Deterministic Reset

```
$\delta_{\text{miter}}, \lambda_{\text{miter}}$ := build_miter();
$S^0_{\text{constraints}} := \emptyset$;
$P_{\text{target}} := (d = 1)$;
while $M$ := unbounded_model_check($\delta_{\text{miter}}, \lambda_{\text{miter}}, S^0_{\text{constraints}}, P_{\text{target}}$) do
    $\mathcal{I}_n$ := get_assignment($M, i$);
    $\mathcal{O}_n$ := oracle_output($\mathcal{I}_n$);
    $S^0_{\text{constraints}}$ := update_constraint($S^0_{\text{constraints}}, \mathcal{I}_n, \mathcal{O}_n$);
end
return $S^0_{\text{constraints}}$
```

out keys and initial states of the locked circuit. The updated allowed miter initial states are specified by equation 5.5. In essence, the allowed initial states of the model checker, must agree with the IO sequence thus far, $(i_0, o_0), ..., (i_n, o_n)$.

$$
\begin{aligned}
\{(s_0, s_1, k_0, k_1) \in & \{0,1\}^{(l+l+k+k)} | \exists_{\hat{s_0}, \hat{s_1}} \\
& \delta_{\text{lock}}(i_n, ...\delta_{\text{lock}}(i_0, \hat{s_0}, k_0)..., k_0) = s_0 \wedge \\
& \delta_{\text{lock}}(i_n, ...\delta_{\text{lock}}(i_0, \hat{s_1}, k_1)..., k_1) = s_1 \wedge \\
& \lambda_{\text{lock}}(i_n, ...\delta_{\text{lock}}(i_0, \hat{s_0}, k_0)..., k_0) = o_n \wedge \\
& \lambda_{\text{lock}}(i_n, ...\delta_{\text{lock}}(i_0, \hat{s_1}, k_1)..., k_1) = o_n \}
\end{aligned}
\tag{4.1}
$$

A visualization of these constraints from two iterations and their connections to the miter circuit are depicted in Fig. 4.9(b). Now, satisfying the miter circuit entails finding two keys and an initial state that agree with the learned IO sequence as well as extending the input sequence to rule out additional keys. Since the state of the system is maintained between sequences, one contiguous IO relationship is formed. This avoids the issues with previous attacks by continuously modeling the unknown variables within the circuit. This process is repeated until the model checker is unable to find an input sequence that satisfies the miter and learned constraints, at which point the attack terminates. The pseudo-code for the attack algorithm is shown in algorithm 6.

Ultimately, the attack produces a set of constraints that rule out keys with incorrect functionality. At termination, there is no input sequence that will produce different

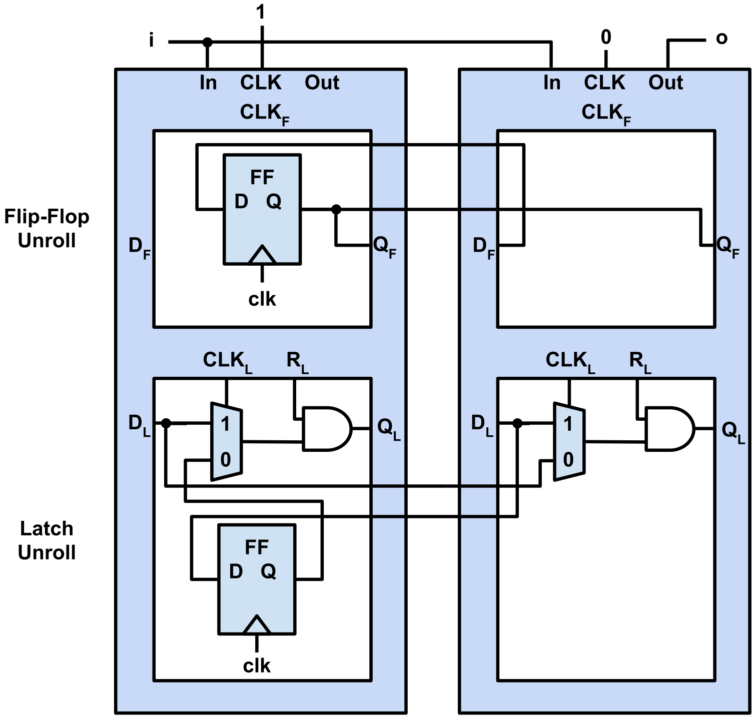


Figure 4.10: Conversion of latch-based circuit to flip-flop counterpart, enabling use in generic model checkers.

output behavior for any two remaining initial-state and key pairs. All remaining pairs are equivalent in this sense. Among these pairs is the actual pair internal to the oracle. The pair should be considered together; applying just a matching key to the oracle may still result in spurious behavior, i.e., not meeting three-valued safe equivalence as defined in 2.2. Thus, depending on the system, a homing or synchronizing sequence [78] may be needed to set the system to the corresponding initial state. It should also be noted that these keys will only produce the correct behavior under a delay-free model of the system. The keys still need to be filtered such that they meet timing constraints of the system, i.e., operating correctly at the correct clock period.

Handling Latches and Cyclic Circuits

Due to their level-sensitivity, the latches inserted by latch-based logic locking are not immediately compatible with many existing model checker implementations. To enable the use of these model checkers in attacking this technique, we adapt the circuit models to utilize only flip-flop state elements. This modeling can be achieved with two copies of the locked circuit's combinational logic as illustrated in Fig. 4.10. Here, D_F and D_L refer to the nets connected to the flip-flop and latch inputs of the locked circuit. Likewise, Q_F and Q_L refer to the nets connected to their outputs.

The copies compute the circuit's values during the two phases of the clock period. The clock inputs of each copy are hardcoded with the respective phase values. The copies share inputs and the circuit's outputs are driven by the second copy. The locked circuit's flip-flops are still modeled by a single flip-flop but connecting its output to the Q_F nets of both combinational logic copies and its input to the D_F net of the second copy. The latches are modeled by a flip-flop and two MUXs that, depending on the CLK_L, will either connect Q_L to the flip-flop value or feed-through the D_L net. Gates with no load are removed through a logic synthesis pass.

Further unrolling is used to remove combinational feedback loops from the latch-based logic locking insertion flow. Potential combinational loops are created either when converting an original flip-flop to keyed latches or when inserting programmable logic decoys. Combinational loops are avoided in real designs as the feedback may create nodes in the circuit that are independent from the current state and input values. Such nodes are uncontrollable, therefore any key that produces a combinational loop is likely incorrect. If the loops are not ruled out, their uncontrollable value will corrupt miter-based attacks. Our latch model does not directly handle these loops so additional modeling is required.

As introduced in section 2.2.2 and utilized in section 3.2.4, acyclic constraint functions can rule out keys creating combinational feedback cycles. However, the algorithms used thus far are incompatible with latch-based logic locking. The inserted decoy logic breaks assumptions about the circuit structure that the algorithms rely on. This creates scenarios in which the constraints under-approximate the set of valid keys, or conversely, rule out potentially correct keys combinations. However, a similar process can be used to produce a model of the circuit that captures the space of IO relations without uncon-

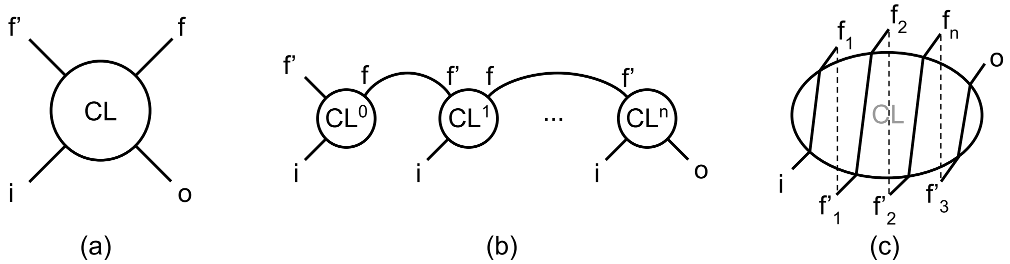


Figure 4.11: (a) Cyclic circuit with cut feedback paths. (b) Acyclic, unrolled version of the circuit. (c) Visualization of longest possible path through unrolled circuit's feedback.

trollable nodes or the potential of ruling out valid keys.

The authors in [60] describe a method of handling cyclic combinational logic, converting the cyclic logic into an acyclic model through an unrolling process adopted from [79]. The acyclic model is compatible with the miter-based oracle attacks. The conversion process finds a feedback-arc set, $f \in F$, a set of nets that when disconnected render the circuit's logic acyclic. As depicted in Fig. 4.11(a), the nets are broken, adding a new input, f', in place of the feedback. The logic is then unrolled, connecting f_n of one copy to the f'_n of the next. As in Fig. 4.11(b), the inputs of all copies are tied together and the outputs are driven by the last copy. The logic is unrolled $|F|$ times to ensure that all simple paths (i.e., a path through the logic that does not contain duplicate gates) are modeled. Borrowed from [79], Fig. 4.11(c) gives a visual intuition as to how unrolling allows the evaluation of the longest path through the logic, the path that traverses all possible feedback nets.

This acyclic model captures the combinational behavior of the logic, i.e., it produces correct output values when they are uniquely dependent on the inputs. This corresponds to the keys that do not produce sensitizable feedback. The model's behavior under sequential inputs, i.e., key *with* sensitizable feedback, will depend on the added inputs, $f' \in F'$, and will not necessarily correspond to the behavior of the real circuit. However, it is assumed that the correct key does not exhibit this sequential behavior in its logic. Namely, sequential behavior is restricted to the sequential elements, flip-flops and latches. Thus these sequential behaviors that the model captures will eventually be

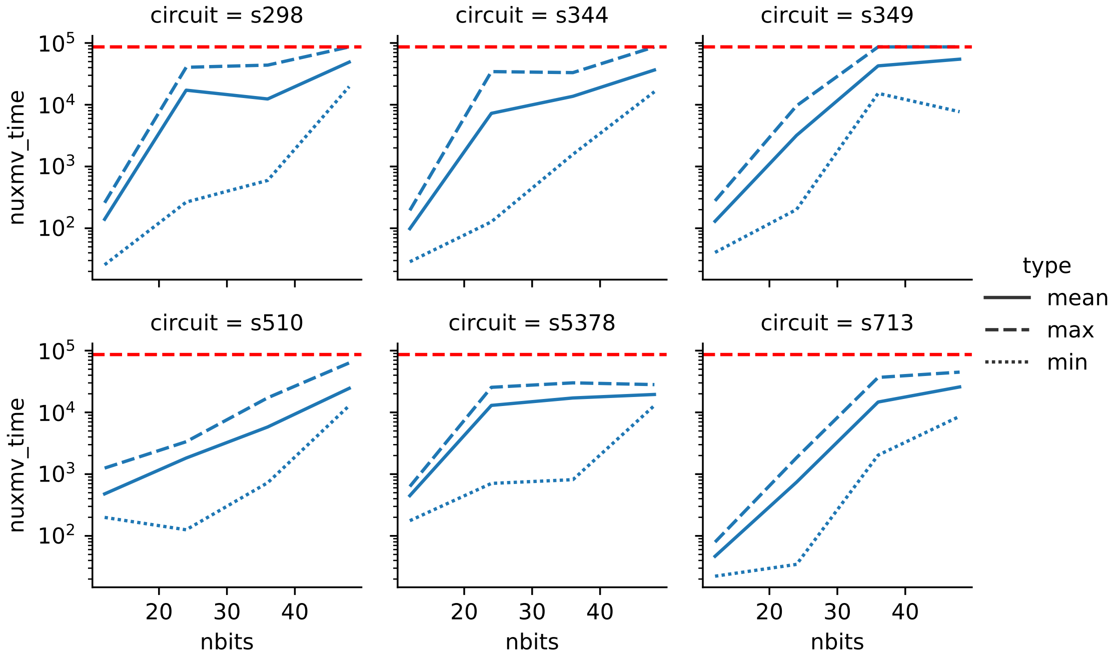


Figure 4.12: Model checker attack results; timeout of 24 hours indicated by red line.

ruled out.

Many feedback-arc sets exist, selecting smaller sets will decrease the resulting model size. We use a heuristic algorithm to identify such a set [80]. The algorithm is run on the sequential subgraph of latches as this is the only portion of the circuits with potential cyclic behavior. Once the feedback set is determined, the circuit is unrolled and then synthesized to remove the superfluous logic.

Oracle-Based Attack Results

We implement the oracle-based attack using the nuXmv [81] model checker, specifically using its IC3 [50] model checker routine. As in previous attacks, we use a Python wrapper around the tool to handle problem construction and as before the vast majority of the attack time is spent within the nuXmv solver, thus further optimization here is likely of little value. All attacks are run in parallel on a server with 750 GB RAM and 48 2.1GHz cores, limiting the number of processes such that the total memory is well below the total available.

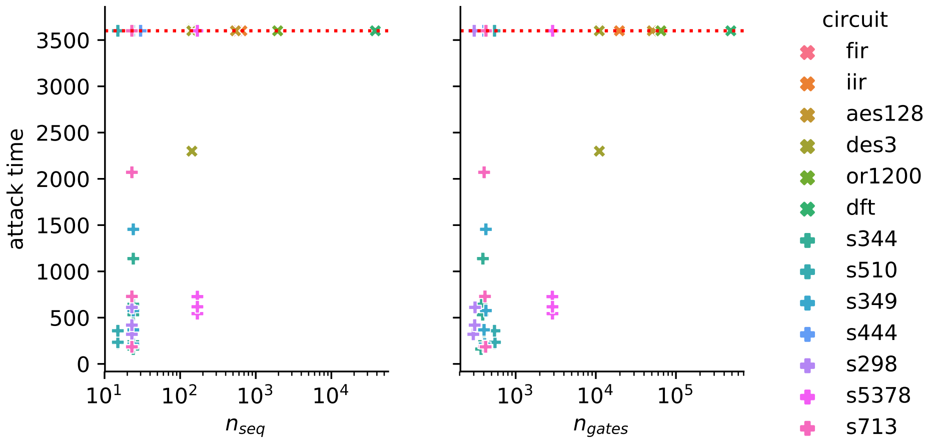


Figure 4.13: Model checker attack versus circuit size.

The first experiment we run to assess the attack time of our latch-based locking technique is a sweep of the number of key bits across a set of circuits from the ISCAS89 benchmark suite. We fix the ratio of key bits to be roughly 50% latches converted from original flip-flops, 25% path delay decoys, and 25% logic delay decoys. Starting from 8 bits, we sweep the amount to 48 bits and observe the attack times. Each run is limited to an attack time of 24 hours and each circuit, bit-count configuration is run 5 times. In Fig. 4.12, we plot the maximum, minimum, and mean attack times versus bit-count. The results show some fluctuations, but generally show an exponential increase in attack time. It should be noted that timeout trims many of the higher bit-count runs. These runs verify that as we scale the number of bits, the resulting attack termination time trends exponentially higher, an important property in ensuring the security of the locked circuits.

Our next experiment demonstrates the scaling of the attack time with circuit size. Over five iterations, we lock a set of circuits from the ISCAS89 and Common Evaluation Platform [82] benchmark sets using 20 key bits and run the attack with a timeout of an hour. In Fig. 4.13, we plot the attack times versus the number of sequential elements, gates, inputs, and the ratio of outputs to inputs. While there exist smaller circuits that timeout, generally as the circuit size grows the attack time increases. This suggests as

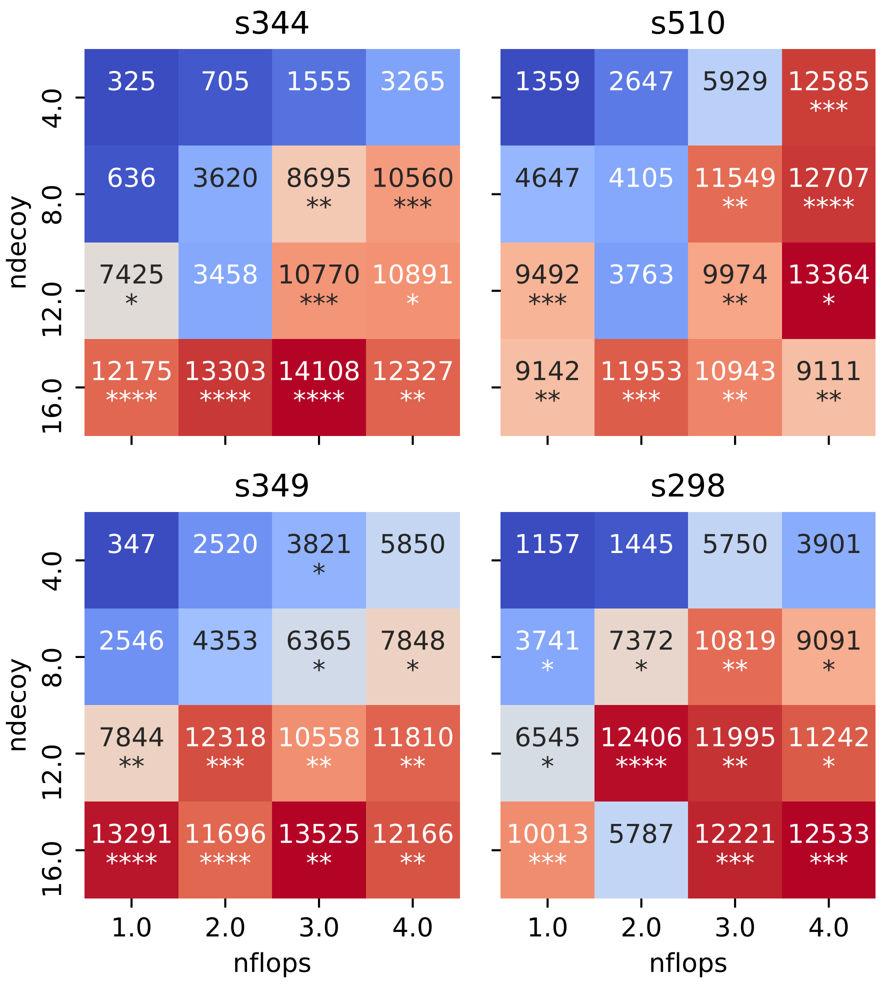
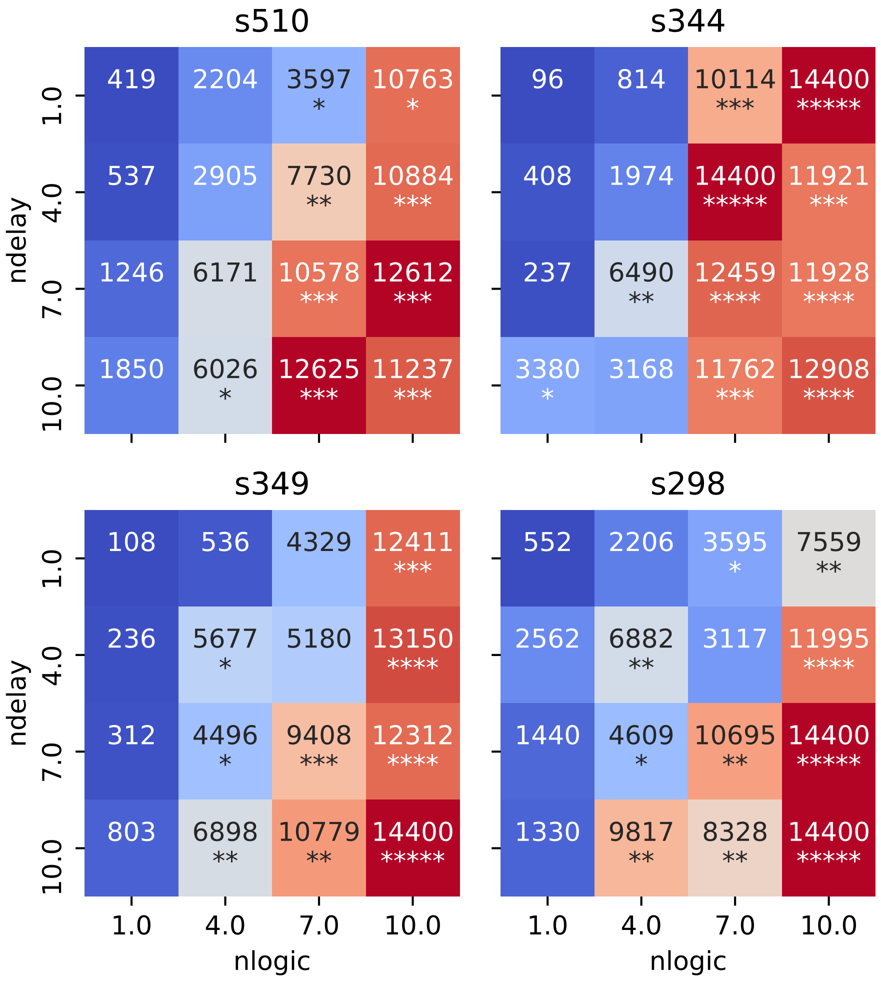


Figure 4.14: Comparison of attack time with various ratios of original flip-flops converted versus decoys add. Added decoys are evenly split between delay and logic types.

latch-based logic locking is integrated into larger systems, proving attack convergence will become more challenging. It's likely that new modeling strategies will be needed to implement a successful attack for large circuits.

We then analyze the effect of varying our locking parameters. We first consider, the number of added decoys n_{decoy}, and the number of flip-flops selected for latch conversion, n_{flops}. For several circuits, we sweep the number of original flops selected and the number of decoys added versus attack time. Each flop selected is limited to creating a maximum of 4 latches and the n_{decoys} parameter is incremented in steps of 4. This maintains roughly equal steps in the latch types, subject to the variations in latch re-



s510

	1.0	4.0	7.0	10.0
1.0	419	2204	3597 *	10763 *
4.0	537	2905	7730 **	10884 ***
7.0	1246	6171	10578 ***	12612 ***
10.0	1850	6026 *	12625 ***	11237 ***

s344

	1.0	4.0	7.0	10.0
1.0	96	814	10114 ***	14400 *****
4.0	408	1974	14400 *****	11921 ***
7.0	237	6490 **	12459 ****	11928 ****
10.0	3380 *	3168	11762 ***	12908 ****

s349

	1.0	4.0	7.0	10.0
1.0	108	536	4329	12411 ***
4.0	236	5677 *	5180	13150 ****
7.0	312	4496 *	9408 ***	12312 ****
10.0	803	6898 **	10779 **	14400 *****

s298

	1.0	4.0	7.0	10.0
1.0	552	2206	3595 *	7559 **
4.0	2562	6882 **	3117	11995 ****
7.0	1440	4609 *	10695 **	14400 *****
10.0	1330	9817 **	8328 **	14400 *****

Figure 4.15: Comparison of attack time with various ratios of logic versus delay decoys.

timing. The heatmap in Fig. 4.14 shows the trends in attack time. The heatmap values correspond to the average of 5 runs with a timeout of 4 hours. The asterisks below the values represent the number of runs that reached the timeout limit. Considering the skew between the attack times for steps in the different dimensions, we see that the n_{decoys} parameter has a slightly greater impact. This disparity suggests that these decoys add more resistance, but of course this may depend on the specific circuit. Interestingly, as we will see in the following section, the overheads do not vary significantly with the ratio of decoy latches to latches from flip-flops.

Finally, we analyze the impact of the ratio between the types of latches inserted, determined by the parameters, n_{delay} and n_{logic}. For the same benchmark circuits and timeout, we fix the number of original flops selected to 1, then sweep the number and type of added decoys. The heatmap in Fig. 4.15 shows the trends in attack time. A significant skew exists between the types of decoys in which the logic decoys appear to have a larger impact on attack time. This is likely due in part to the added feedback paths that must be unrolled to maintain an acyclic circuit and the generally increased interdependence between the key bits that comes from the logic decoys.

It is important to emphasize, under the returned keys, timing violations are likely to remain from incorrect propagation delays. After running the model checker-based attack, the adversary must still find a key that passes timing. If modifying the design is an option for the adversary, this can be done though recombining the latches into flip-flops and retiming. Otherwise finding a key that meets timing requires more complicated processing mapping the known cycle delays to a key that meets timing in a circuit that likely uses cycle-stealing between latches.

As a whole, the performance against this oracle-based attack bodes well for the security of this latch-based locking technique; existing attack schemes are unable to determine the locked circuit's key. Of course as discussed in chapter 3, this may change with new insights into attack modeling or the discovery of more powerful structural metrics that more reliably determine key bits.

4.3 Overhead Analysis

We demonstrate the design overheads associated with latch-based logic locking using block designs from the Common Evaluation Platform. All circuits were synthesized using a modern commercial standard cell library in a 22nm FinFET process. The characteristics of each design are displayed in Table 4.1. For modifying the netlists and synthesis, we used Cadence Genus. For place-and-route runs, we used Cadence Innovus. All synthesis and place-and-route runs target the maximum obtainable frequency. For assessing

Circuit	FFs	Gates	Frequency(GHz)
IIR	646	5100	1.31
FIR	566	5221	1.33
DES3	134	3595	2.14
AES	530	15533	2.77
OR1200	1939	14366	1.46
DFT	38163	83457	2.15

Table 4.1: Characteristics of each considered benchmark circuit

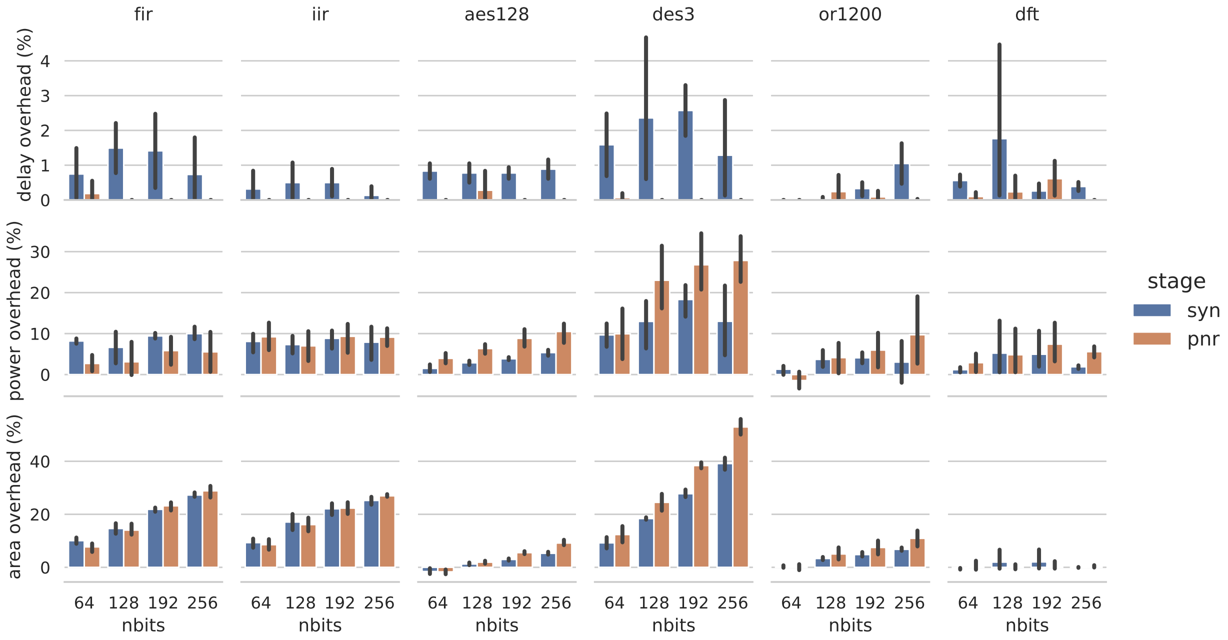


Figure 4.16: Latch-based logic locking delay, power, and area overheads vs. number of key bits normalized to the original design.

ATPG overheads, we use Mentor Fastscan.

4.3.1 Power, Performance, Area Overhead

We obtain power, performance, and area overheads as follows. First, the maximum frequency of the design is found via a synthesis run targeting an unattainably high frequency. The synthesis tool produces a critical path that determines the maximum frequency, but with excessive buffering. The synthesis process is then *rerun from scratch targeting this maximum frequency*. This result serves as the baseline implementation to which various amounts of locking are applied and to which the results are normalized.

For DFT, the baseline circuits have all flip-flops converted to their scan-able versions. The locked circuits implement our preliminary DFT structure outlined in section 4.1.4.

For our first experiment, each circuit's baseline is keyed with 64, 128, 192, and 256 bits of latch-based logic locking, targeting the design's maximum frequency. As the latch-based logic locking insertion has an element of randomness depending on the inserted decoy logic, all experiments are run with 5 samples per circuit, bit-count pair. We target a 50% decoy latch to real latch ratio; however the actual value depends on the circuit due to the variability of the retiming process. The resulting synthesis and place-and-route overheads are displayed in Fig. 4.16, respectively in blue and orange.

As expected, the observed overheads are low. Considering delay, we see that the synthesis results have some fluctuation in overhead, reaching roughly 5% overhead in the worst case. However, these delays do not impact the final place-and-route result, wherein the overhead is negligible. This difference is likely the result of logic synthesis not building the full clock tree. Place-and-route tools construct the tree and are able to balance clock edge arrival times to take advantage of a latch's level sensitivity. These results confirm we have met our primary overhead goal of maintaining performance while locking the design. The power overheads are generally low, scaling slightly with the number of latches added. Naturally, the smallest tested design, DES3, shows the largest overhead, still under 30%. The remaining blocks show overheads under 10%, even with up to 256 bits of locking inserted. In several cases, we see an increase of power overhead going from synthesis to place-and-route. This again is likely due to the addition of the full clock tree. Finally, we see that the area overheads scale with the amount of locking added and also depends on the size of the design being locked. As previously noted, this area overhead can be significantly reduced when using LSSD techniques as opposed to the current area-expensive test points.

Next, we sweep the decoy ratio, comparing the overheads from different amounts of latches converted from original flip-flops and those added as decoys. The number of key bits is fixed at 256. The results of these experiments are shown in Fig. 4.17. Delay

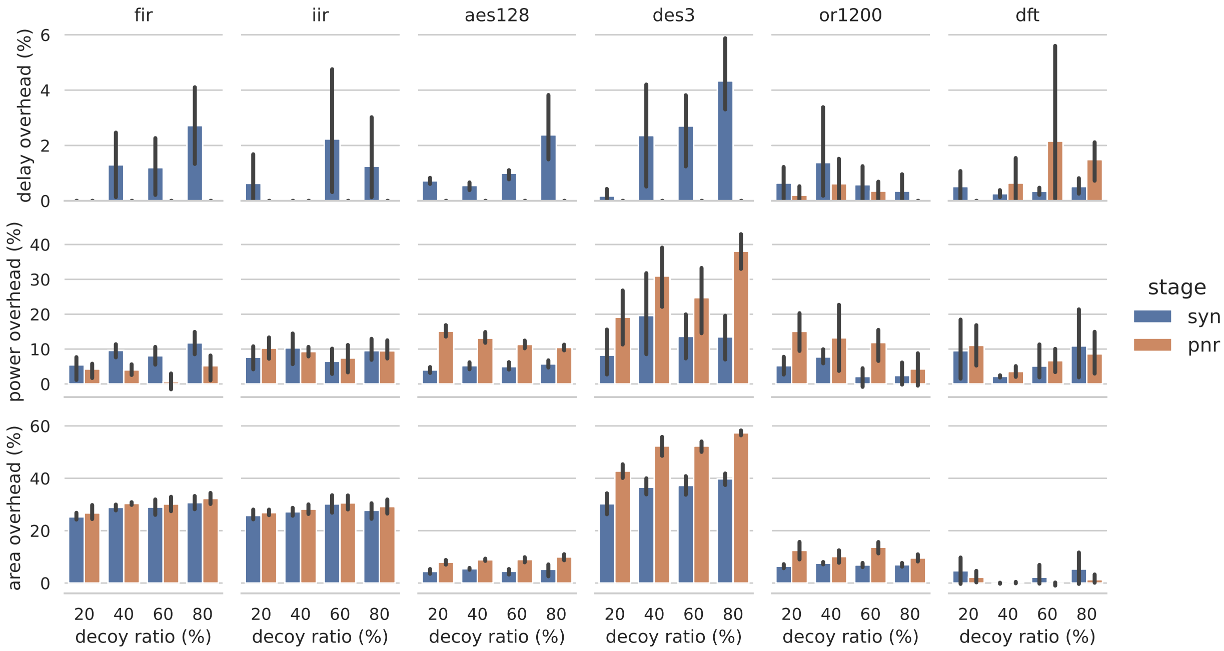


Figure 4.17: Latch-based logic locking delay, power, and area overheads vs. decoy ratio normalized to the original design, nbits=256.

overheads remain negligible, power overheads fluctuate depending on the circuit and intuitively, area slightly increases with the % of decoys. Importantly, we only see small variations when sweeping the decoy ratio. This enables an arbitrary ratio to be selected during insertion, reducing bias as to the total number of actual state elements in the locked portion of the design.

In general, there exists a good balance between the benefits of cycle sharing from replacing original flip-flops with latches, and the added delays from inserting decoys. When we increase the decoy ratio, more decoy logic is created that provides new insertion points for other decoy latches without impacting the delay. Overall, these results show that the amount of locking can be greatly increased before significant overhead costs are observed.

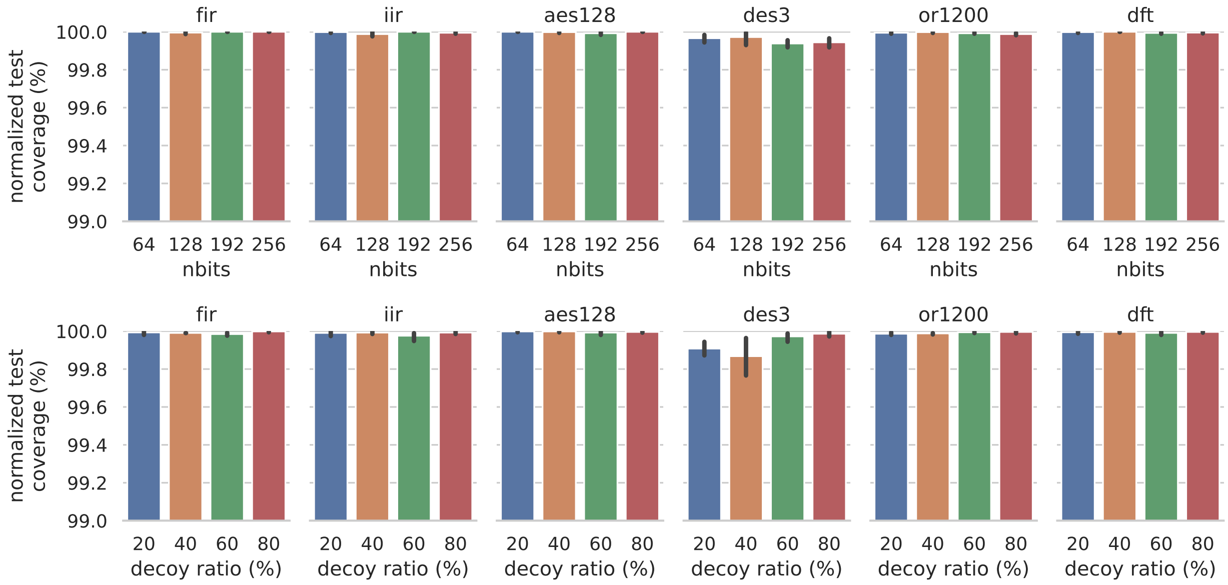


Figure 4.18: Normalized SSL test coverage for circuits locked with latch-based logic locking

4.3.2 Testing Overhead

For each of the previous runs, we generate the test coverage overheads. We use ATPG to detect all possible single stuck line (SSL) faults. SSL faults assume that a single logic gate pin is fixed at either logic 1 or 0. The ATPG tool generates a pattern that can propagate the effect of such a defect to an output. The resulting coverages, normalized to the original circuit with full scan, are displayed in Fig. 4.18. At 256 bits of locking, we see an average test coverage of 99.98% of the original value and a minimum of 99.71%. Furthermore, inspecting the outstanding faults reveals a significant portion of are located along the modified clock path to the latches. Thus, higher coverages are likely obtainable using the same DFT techniques used in clock gating. Assuming that design security is a top priority, these coverage overheads are likely acceptable.

4.3.3 Tapeout

To validate these results and demonstrate feasibility, we have taken five of these designs through the full tape-out flow. The taped-out version of the locking scheme is an older

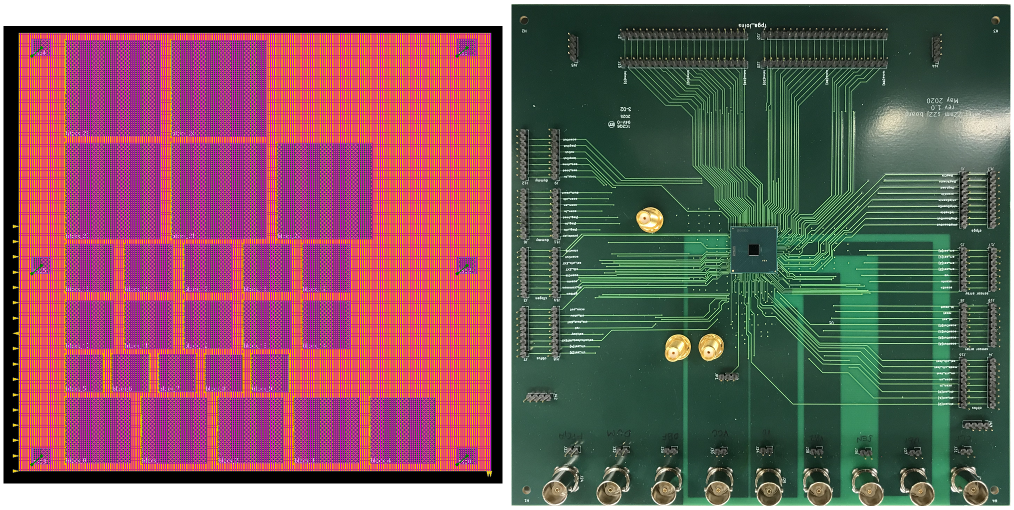


Figure 4.19: GDS of test IC and PCB with IC mounted.

iteration of the insertion flow. It lacks the DFT structures and utilizes a different decoy insertion algorithm that leads to larger overheads than the current version. Nevertheless, the essential elements of the locking scheme are present and have been verified through this manufactured design.

Five copies of each circuit are taped-out, the original baseline along with four locked versions with increasing amount of locking. Each design is wrapped in a testbench circuit that generates random inputs at-speed using linear feedback shift registers (LFSRs). The system is driven with an internally generated clock signal. The clock has gating signals that are used to selectively enable a single design. In Fig. 4.19, we show the GDS layout of the designs along with an image of the final IC mounted to our test printed circuit board (PCB).

Power values are recorded setting the voltage to the nominal value (0.85V) and setting the frequency to targeted value from place-and-route. The LFSRs are set to continuously run and the current is averaged over a 1s interval. This provides the total power value. As the circuits share power domains, there is a shared static power that is removed, producing just the dynamic power values. We report dynamic power values of the locked designs normalized to the original. These results, along with the values reported from place-and-route, are displayed in Fig. 4.20. As we can see, in all cases the power

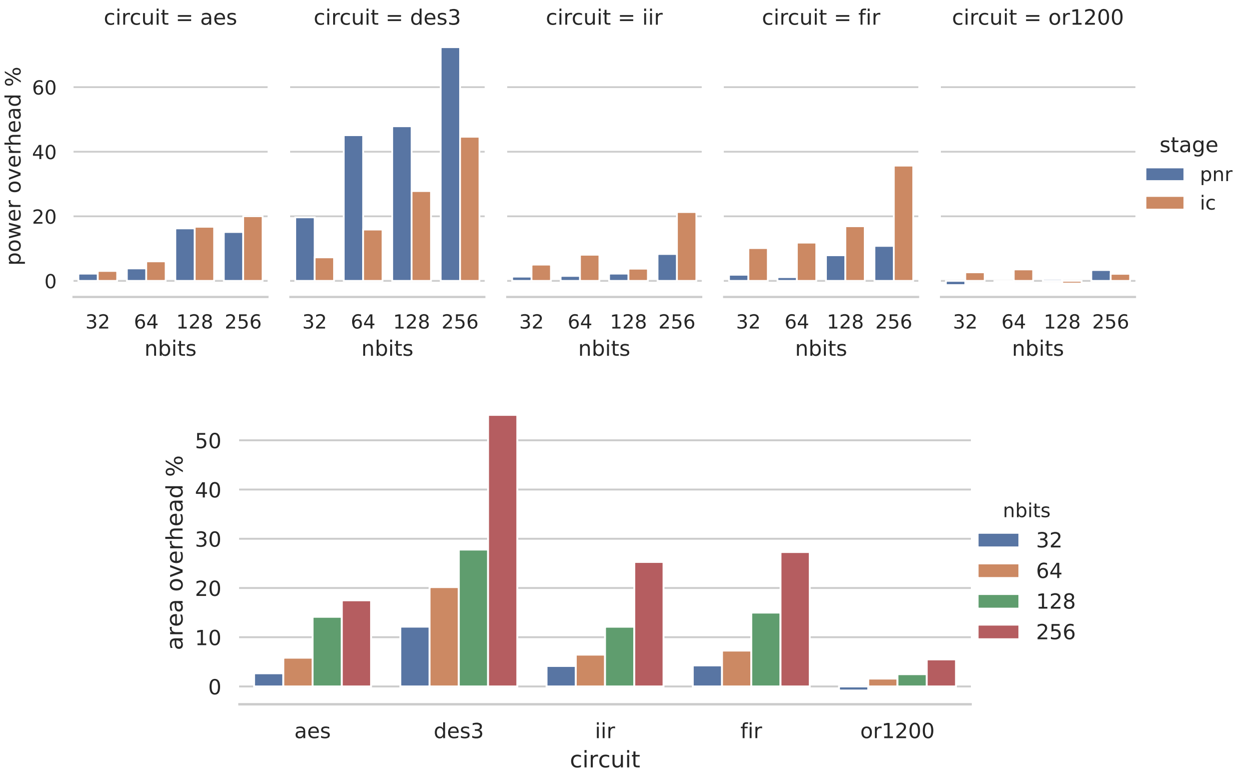


Figure 4.20: Power and area overheads for the taped-out circuits.

trends follow the predicted patterns. However, the accuracy of the predicted power varies. The predicted and measured values for AES track very closely, whereas other under or overshoot the measured value. This is most likely due to differences in the vectors being run in the actual system as produced by the LFSRs and the predicted signal activities during power analysis.

To verify functionality and determine the maximum frequencies, we run the system for a given number of cycles, then scan out the system state. Comparing these values to the simulation enables us to see if the circuits are behaving as expected. We sweep the voltage and frequency applied to each design reporting if functionality matches the simulations. The results are compiled as schmoo plots that allow us to visualize where the frontier where the design starts to fail. Schmoo plots for four of the circuits are depicted in Fig. 4.21. Comparing the original schmoo plots to the locked versions, we see that the performances are nearly identical, verifying the minimal delay overheads.

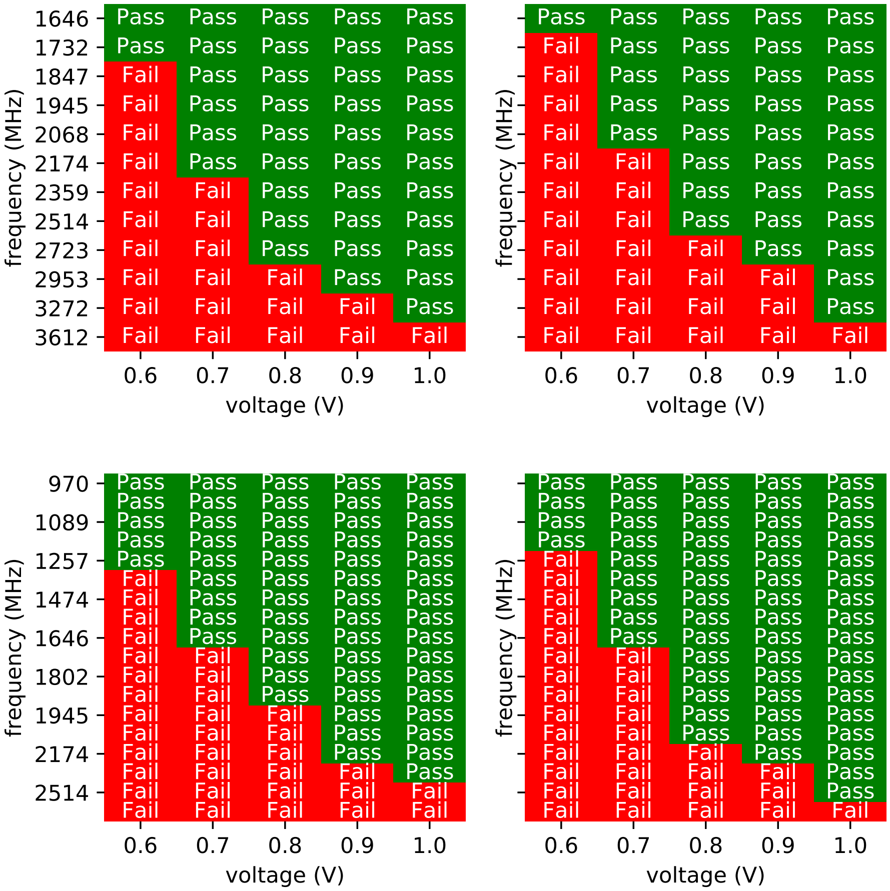


Figure 4.21: Schmoo plots for AES (top left), AES with 64 bits LBLL (top right), IIR (bottom left), IIR with 256 bits LBLL (bottom right).

4.4 Discussion

As we have demonstrated, latch-based logic locking is resistant to current attack strategies and the low overhead allows the amount of obfuscation to be increased significantly without performance loss. We believe there are several key factors that enable this strategy's attack resistance.

First, there exists large amounts of state unavailable to the attacker. Under existing attack strategies, this state must be modeled and unrolled in order to compute and reason about the functionality of the circuit. This sequential modeling is known to be expensive as our experiments here reinforce.

Second, the densely-entangled decoy logic makes propagating information out of the circuit difficult. In chapter 3, we saw a similarly entangled structure, logic-enhanced Banyan locking, also provided strong attack resistance. However, our latch-based technique adds entanglement in a low overhead manner. The added delays are spread among many timing paths as opposed to stacked on a single path as in logic-enhanced Banyan locking and other combinational techniques.

Third, the formation of combinational cycles creates a circuit that is expensive to model. As mentioned in section 2.2.2, the addition of such potential cycles has been explored using MUXs. When enough MUXs are added, the constraints to produce an acyclic model grow substantially, slowing down attacks. Our latch-based locking technique exhibits similar behavior, creating potential cycles with programmable phase latches and the added decoy logic. In order to effectively attack these techniques new less-expensive acyclic models must be developed.

Forth and finally, the technique does not have a single point of failure. In some previous locking techniques, identification and removal of the locked circuitry enabled an easy bypass in the netlist-based attack model. In our technique, the adversary must individually characterize each latch in the design to ensure proper functionality. Even if a high percentage of the latches are known, an adversary will not have a working system under the netlist-based attack model.

While these factors all contribute to the attack resistance, as seen in previous locking techniques, improved attack modeling can significantly reduce the difficulty of deobfuscation. We have modeled our technique with significant detail, despite this, better equivalence models and constraints may exist. But just as attacks may improve, there are many other directions to add to the problem complexity.

Straightforward improvements could be updating the latch insertion. The best structure and insertion technique for decoy latches has yet to be determined. Our current method, randomly forming logic cones from existing signals, leaves signal that can give an attacker some bias in guessing keys. Ideally, the patterns being detected could be ironed out, maintaining low traceability. These improvements could be coupled with logic structures with increased SAT-hardness, resisting both netlist- and oracle-based attacks.

Another simple method of increasing complexity is combining latch-based logic locking with other locking schemes, forcing the attacker to understand and model multiple interacting circuit transformations. As previously mentioned, embedded FPGAs have shown extremely high attack resistance. Augmenting this scheme with latch-based logic locking could provide a way to maintain performance in the timing critical paths of the design while maximizing security elsewhere.

Finally, this project has explored just one potential dimension for atypical logic locking strategies. Unconventional latch-timing or clocking schemes, closely related to this technique, could significantly increase the complexity of attack modeling. Typical design strategies make use of abstractions to reduce complexity. While these abstractions make design cheaper, perhaps breaking some may enable a new generation of locking schemes. One such example is in the static timing of combinational logic. Industry standard tools only consider worst-case scenarios when timing a design. However, under different input transitions, signal arrival time exhibit different delays. Capturing these differences could be utilized in a locking scheme that, like latch-based logic locking, modulates the launch and capture points of timing paths. Perhaps such a scheme would exacerbate the modeling complexity, forcing the attacker to rely on less efficient attack methods.

Ultimately, latch-based logic locking represents a new dimension in which to lock a circuit. Since we have demonstrated promising initial results, we believe this and strategies like it are a path forward to providing low overhead security during the IC manufacturing process.

Chapter 5

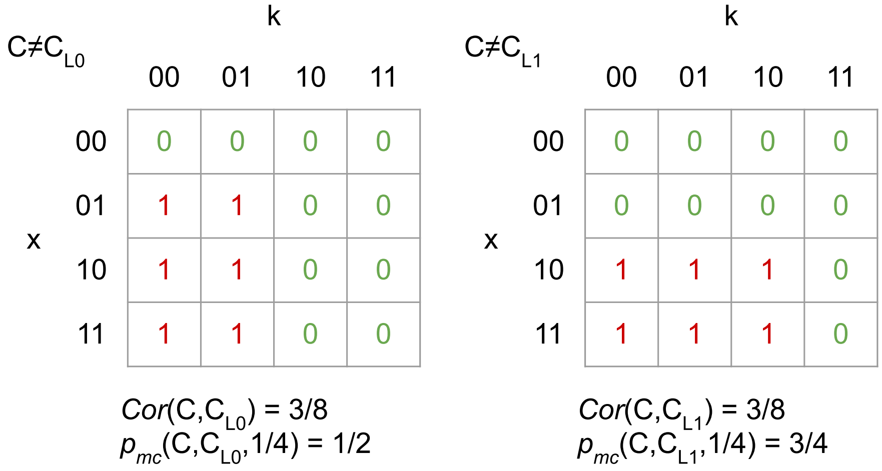
Quantifying the Efficacy of Locking Methods

5.1 Locking Metrics

A pervasive problem in the logic locking community is the lack of metrics that adequately capture the intended notions of security. Additionally, the metrics that do exist are only evaluated on trivially small circuits or rely on closed-form equations that correspond to simple structured locking schemes. In this chapter, we describe the limitations of existing metrics. We then define two metrics that capture intuitive definitions of security for the netlist and oracle attack models and subsequently relate the metric evaluation problem to the well-established field of model counting. Finally, we demonstrate three applications of these metrics, revealing critical information about existing lock schemes.

5.1.1 Existing Security Metrics

The run-time of the miter-based attack has been a ubiquitous metric since the attack's debut. The typical demonstration sweeps the number of key bits and produces a (hopefully) exponentially scaling attack time. While resistance to this attack is essential if the



$C \neq C_{L0}$	k: 00	01	10	11
x: 00	0	0	0	0
01	1	1	0	0
10	1	1	0	0
11	1	1	0	0

$Cor(C, C_{L0}) = 3/8$
$p_{mc}(C, C_{L0}, 1/4) = 1/2$

$C \neq C_{L1}$	k: 00	01	10	11
x: 00	0	0	0	0
01	0	0	0	0
10	1	1	1	0
11	1	1	1	0

$Cor(C, C_{L1}) = 3/8$
$p_{mc}(C, C_{L1}, 1/4) = 3/4$

Figure 5.1: Miter truth tables and corresponding locking metrics for two locked circuits.

oracle attack model is considered, this run-time may give an over-optimistic notion of security. Importantly, *while running the attack to completion may be infeasible, the intermediate results may produce keys that are functionally close to the original design*. In this case, the adversary need not complete the attack, but rather run until the keys produced exhibit low enough error rates.

Another previously proposed metric for assessing a lock quality in terms of error rates is corruptibility [83], defined as

$$Cor(C, C_l) \equiv P_{x \in X, k \in K}[C(x) \neq C_l(x, k)] \tag{5.1}$$

This metric captures the likelihood across all keys and inputs that a locked circuit is incorrect, essentially the total amount of inaccuracy in a locked circuit. However, it gives no notion of the distribution of incorrect values. An example of why this is important is depicted in Fig. 5.1. Here, a circuit is locked with two different schemes that produce locked circuits C_{L0} and C_{L1}. The resulting miter truth tables, $C \neq C_{L0}$ and $C \neq C_{L1}$ are shown. Both locked circuits exhibit the same corruptibility, however, the quality of the locking schemes is clearly different. For half of the possible keys, C_{L0} is completely correct whereas C_{L1} has incorrect values for every key other than the correct one. This motivates the development of a metric that can capture this disparity.

5.1.2 Key Corruption

Our first proposed metric, key corruption, is meant to capture a more precise notion of resistance to oracle-based attacks beyond run-time. Key corruption is the portion of the input space that is mapped to incorrect outputs for a given key. Specifically, it is defined as

$$KeyCor(C, C_l, k) \equiv P_{x \in X}[C(x) \neq C_l(x, k)] \tag{5.2}$$

This metric directly corresponds to the approximate key recovery outlined in section 2.2.1. This is useful to the designer in assessing the accuracy of intermediate keys produced by an attack and also as a kernel in computing more complex metrics.

To evaluate the key corruption, we build a miter circuit, $M \equiv C(x) \neq C_l(x, k_{\text{attack}})$. Counting the number of input values that satisfy this miter and normalizing by the size of the circuit's input space will determine the key corruption. Typical circuits can have input widths upwards of 64 bits, thus it is necessary to utilize approximation techniques as outlined in section 5.1.4. The error of the estimated count of satisfying solutions can be tied to the bounds of the approximation method.

Depending on the circuit's application, the targeted key corruption definition can be adapted. The above definition counts an input value as incorrect if it has at least one output bit incorrect. Some applications may require several output bits to be incorrect. In this case, key corruption could be defined as $P_{x \in X}[(\sum_{i=1}^{m} C(x)^i \oplus C_l(x, k)^i) \geq t]$, where t is the required number of incorrect bits and $C(x)^i$ is the ith bit of $C(x)$. Other definitions could weight certain bits more heavily, potentially useful in ensuring significant error in arithmetic operations.

5.1.3 Minimum Corruption

While key corruption can assess the progress of oracle-based attacks, under a netlist attack model, a more useful metric to a designer is the probability that a sampled key meets a certain corruption threshold, $\epsilon \in [0, 1]$. This threshold can be determined by the application. For example, disabling a cryptographic function may only require a small portion of the inputs to be corrupted whereas a neural network accelerator may need significantly more. We capture this notion using minimum corruption, defined as

$$MinCor(C, C_l, k, \epsilon) \equiv KeyCor(k) \geq \epsilon \tag{5.3}$$

We then can define a probability of selecting a key that meets this minimum corruption value, p_{mc}.

$$p_{mc}(C, C_l, \epsilon) \equiv P_{k \in K}[MinCor(C, C_l, k, \epsilon)] \tag{5.4}$$

For a given key, minimum corruption discounts the corruption beyond the threshold. Again considering Fig. 5.1, we see that p_{mc} captures the difference between the two locking scenarios. The designer can determine a suitable threshold, then can scale the amount of locking until the probability of meeting the corruption threshold is acceptable.

Again, the techniques discussed in section 5.1.4 can be used to efficiently approximate this metric. To obtain a p_{mc} estimate, we take a set of uniform random samples from the key space. For each of these sampled keys, we estimate the key corruption and compare it to the threshold to determine if it meets the minimum corruption. The fraction of key samples for which key corruption is greater than the threshold computes p_{mc}.

5.1.4 Estimating Metrics

In general, the metrics we propose will be used to evaluate the amount of discrepancy between the original and locked circuits under various scenarios. Calculation of these metrics directly maps to model counting (#SAT). The evaluation of metrics can be encoded as a Boolean formula wherein the number of satisfying solutions over the total space gives the value. Due to the high dimensionality of the problem, exact solutions can only be obtained for a limited key and input width. To understand how locks scale, we resort to approximation methods. Luckily, approximate model counting is a widely studied area with many efficient, open-source solvers.

We use the solver ApproxMC [84, 85] as a kernel in estimating our proposed metrics. This solver uses hash functions to split a circuit's input space into small, countable partitions of roughly equal size. By counting a single partition and multiplying by the number of partitions, the tool can give an estimation of the number of solutions to a formula. Repeating this process allows increased confidence in the estimation. Conveniently, ApproxMC has a rigorous formulation of probably approximately correct (PAC) bounds. The relation between the real count, N, and the estimated count, N_{est}, is parameterized by $\delta \in (0,1]$ and $\epsilon_a > 0$. Specifically, the relationship is $P[N/(1+\epsilon_a) \leq N_{est} \leq N(1+\epsilon_a)] \geq 1-\delta$.

5.2 Application of Metrics

The metrics proposed in the previous section can be used to evaluate the efficacy of locking techniques under the netlist and oracle attack models. We demonstrate this process using representative locks from the classes described in section 2.2.2. We implement the locks using our open-source python library *circuitgraph* [86], which allows for easy manipulation of netlists. To allow for reproducibility, we share our lock and metric implementations in our repository[1]. From the insertion-based lock types, we implement XOR, LUT, and MUX locking [6, 8, 13]. From the point-function based locks, we implement SFLL-Flex [54]. Finally, from the densely-interconnected locks, we implement Full-Lock [24] and reuse logic enhanced Banyan locking from section 3.2.4. We also run the relaxed model version of Full-Lock that uses MUXs. Our miter-based attack implementation uses the SAT solver CaDiCaL [57]. For the locking techniques that produce cyclic circuits, we use CycSAT [21] to form acyclic key conditions so that the attack terminates correctly. Each of these techniques is used to lock circuits from the ISCAS 85 combinational benchmark set [56].

[1] https://github.com/circuitgraph/logic_locking

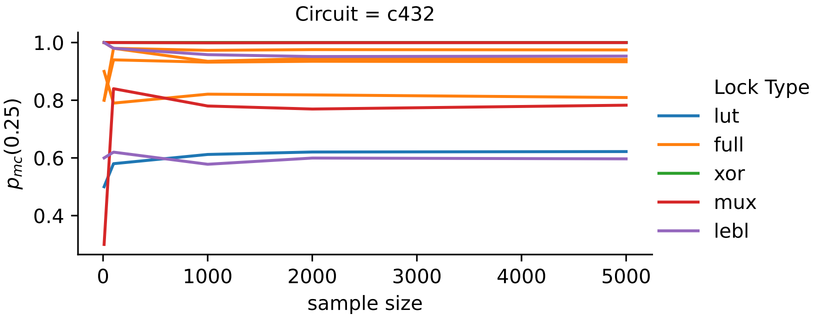


Figure 5.2: Approximate p_{mc} of benchmark versus #(keysamples).

5.2.1 Minimum Corruption under Netlist Attack Model

As discussed in section 5.1.3, the probability of meeting minimum corruption provides the designer with a good understanding of how likely it is for an adversary to select a key that functions close to the correct design. In Fig. 5.2, we show an experimental run to determine an appropriate number of key samples to use for the estimation. As seen from the trend for our set of locked circuits, the value tends to converge around 1000 samples. We use this value for the remainder of our p_{mc} estimates.

Assuming that the adversary has no a priori knowledge of the key bits, we evaluate p_{mc} for the selected locking techniques. We lock four circuits from our benchmark set with roughly 128 bits of locking. (The widths are not exact as the densely-interconnected techniques scale in unequal increments.) Using our estimation process, we uniformly sample keys and evaluate p_{mc}, showing the results in Fig. 5.3. Several interesting conclusions can be drawn from this data. First, we see that XOR-locking shows the highest p_{mc} value. At this key width, all keys sampled are corrupted for all inputs on at least one output. This shows the significant amount of corruption obtained from the inversion of random nets in the circuit. Considering the other techniques, we see that they all, except SFLL-Flex, have a high probability of meeting a 0.2 ϵ value. Generally, these probabilities decrease with ϵ at rates depending on the circuit and lock type.

We can also integrate more complex attacks into this analysis. For example, as dis-

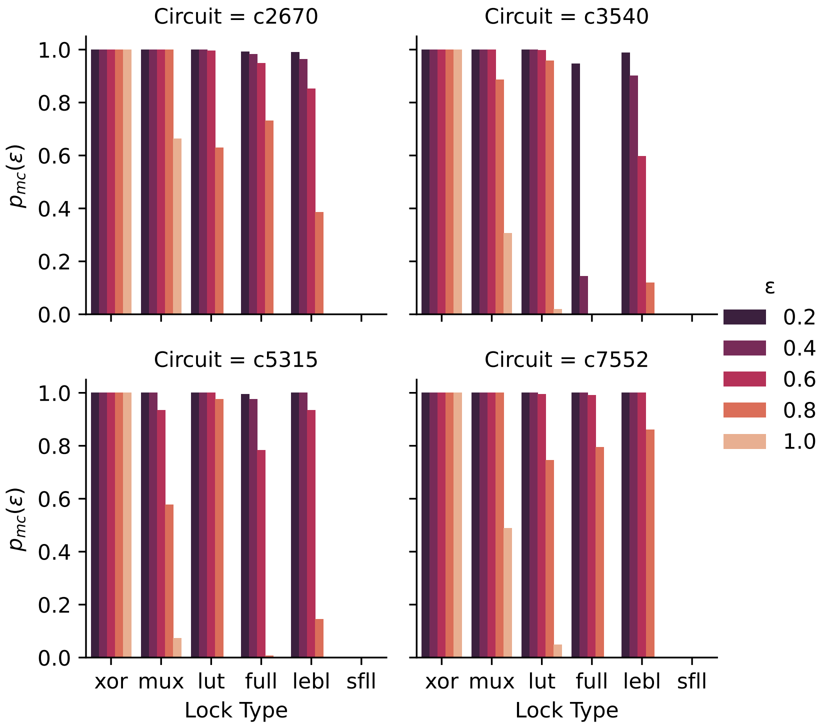


Figure 5.3: Approximate p_{mc} of benchmark circuits locked with selected techniques. $w \approx 128$ and $\#(\text{keysamples}) = 1000$

cussed in [40], an adversary can analyze the local structure of a circuit locked with XOR-locking and determine a likely key with reported accuracy up to 95%. The effect of this bias in the key space can be assessed with our minimum corruption metric. We assume that the designer requires at least 10% of the input space to be corrupted. As the analysis of each locked gate is local, we assume that each key bit is independently drawn from a Bernoulli distribution with the probability parameter, p, set to the accuracy of the model, $P[k^i = k_c^i] \sim Bernoulli(p)$. We sweep p from 0.5 (i.e., no information) to 0.95, the highest reported accuracy of the models. For each accuracy level, we determine the minimum corruption for a set of circuits with varying amounts of XOR-locking. As seen from the results in Fig. 5.4, even at $p = 0.95$ and 96 bits of locking, the value of p_{mc} is very high. This shows that, while the local-structure analysis for likely key can significantly narrow the distribution of the correct keys, it does not necessarily translate into a circuit that is functionally close to the original, largely due to the high corruption of

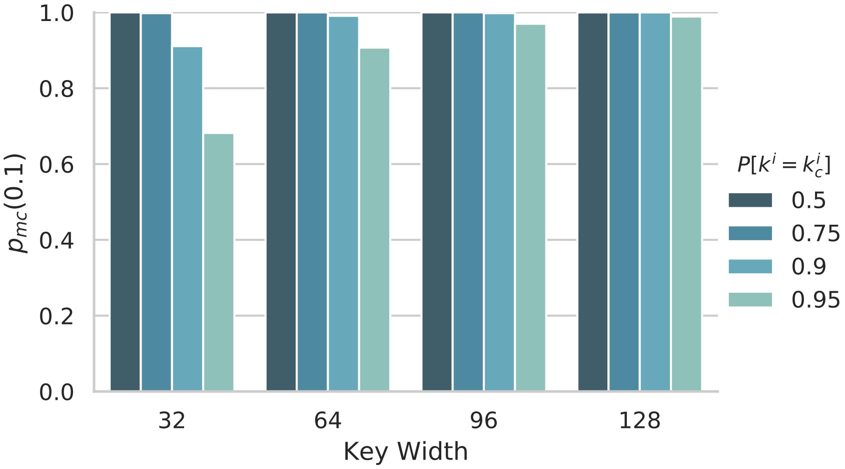


Figure 5.4: Approximate p_{mc} of c3540 locked with XOR-locking, sweeping key width w and $P[k^i = k_c^i]$ with $\#(\text{keysamples}) = 1000$.

parity gates. If the attacker's goal is to produce a functionally correct or approximately correct circuit with the netlist alone, this attack scheme alone is unlikely to succeed.

5.2.2 Incremental Key Corruption of Oracle Attack Model

The *de facto* metric evaluated for oracle-based attacks is attack termination time. In this dimension, both the point function-based and densely-interconnected techniques exhibit very strong attack resistance. However, these attack times mean little if unaccompanied by a notion of corruption for the remaining keys. Typically, when executing oracle-based attacks, a plausible key is produced in each iteration. Solving for additional keys is costly, likely motivating the attacker to simply use this incremental key. Evaluating the key corruption of the key from the attack can be used to indicate the progress of the attack.

In Fig. 5.5 we demonstrate the use of key corruption to augment the miter-based attack for our selected locking techniques. We lock the circuits with roughly 448 key bits and run the attack with a timeout of 1 hour, evaluating the corruption at each iteration. The data is shown on a log plot where key corruption values of zero are mapped to 10^{-15}. The results show several critical insights. First, we see that in most cases XOR and

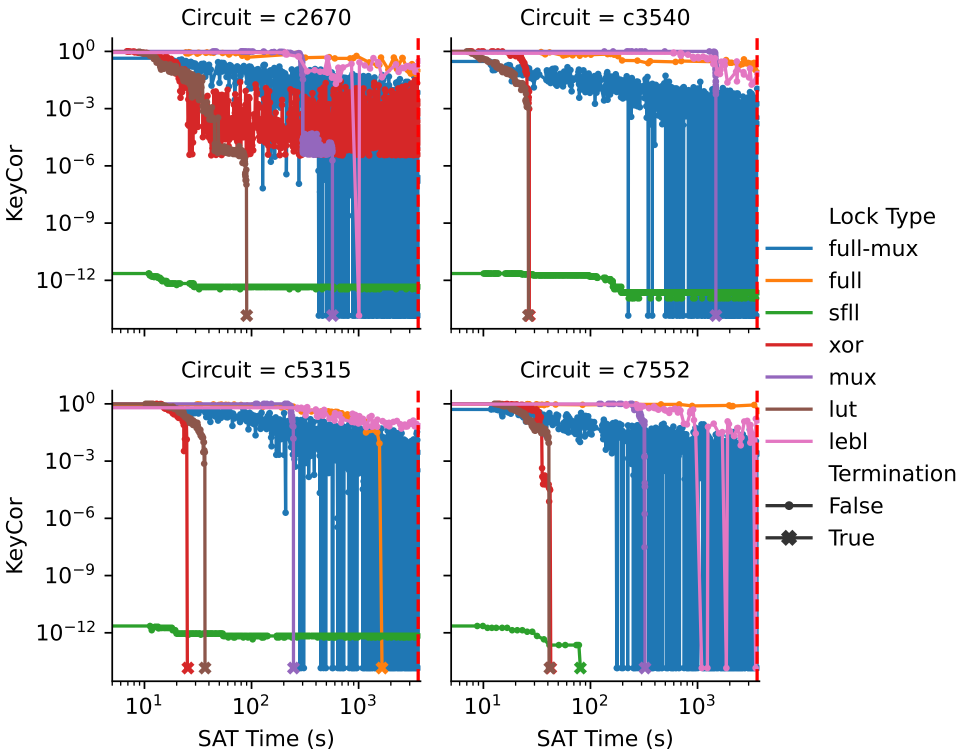


Figure 5.5: Key corruption for incremental keys returned from miter-based attack, $w \approx 448$. The attacks are run with a timeout of 1 hour, indicated by the dashed red line. The zero value of key corruption is mapped to 10^{-14}

LUT-locking terminate in under 100 seconds. At such large key widths, it is clear that these techniques do not hold up under this attack model. MUX-locking takes about an order of magnitude longer to terminate. Unexpectedly, we see termination in one of the SFLL runs (c7552), likely due to the structure of the locking mechanism biasing the SAT solver to pick the corrupted inputs, thus immediately breaking the lock. However, even when SFLL isn't broken, the key corruption remains too low to likely have a significant effect.

The densely-interconnected techniques Full-Lock and logic-enchanced Banyan locking (LEBL) generally show the best results, with the highest corruption levels at the timeout, but with some concerning caveats. We do see one run terminating under an hour for Full-lock. Furthermore, the trends in the key corruption for the relaxed model, Full-Lock MUX, and LEBL reveal an interesting pattern undetectable by just considering the attack termination time. There exist intermediate results that are functionally correct keys. Full-Lock MUX produces many such keys whereas logic-enhanced Banyan locking produces only a few functionally correct keys out of ~ 100 intermediate results. In both cases, this represents a major security vulnerability. An astute attacker could thoroughly test the intermediate key results and confirm that these keys have arbitrarily low corruption with the oracle. *Under the oracle attack model, even at these extremely high key widths, none of the tested combinational locking techniques provides a secure solution for all circuits.*

5.2.3 Overhead-Security Trade-Offs

Overhead in the typical VLSI metrics, delay, area, and power is a critical concern of logic locking. The application of the IC may enforce limitations on the acceptable overhead. Even if this is not the case, too much overhead may motivate the use of commercial solutions such as FPGAs or microprocessors, rather than design a locked ASIC. We analyze our selected locking techniques across these metrics to show their scaling with the number of key bits.

Using Cadence Genus along with a commercial standard cell library in a 28nm process, we obtain overheads as follows. The maximum frequency of the design is found via iterative logic synthesis runs. This result serves as the baseline implementation to which various amounts of locking are applied and to which the results are normalized. Each design is locked with the different techniques, varying key widths such that they produce roughly the same overhead range. We combine power, area, and delay into a single overhead value using:

$$\begin{aligned} Overhead \equiv & [(power_{locked}/power_{orig}) \times \\ & (delay_{locked}/delay_{orig}) \times \\ & (area_{locked}/area_{orig})] - 1 \end{aligned} \tag{5.5}$$

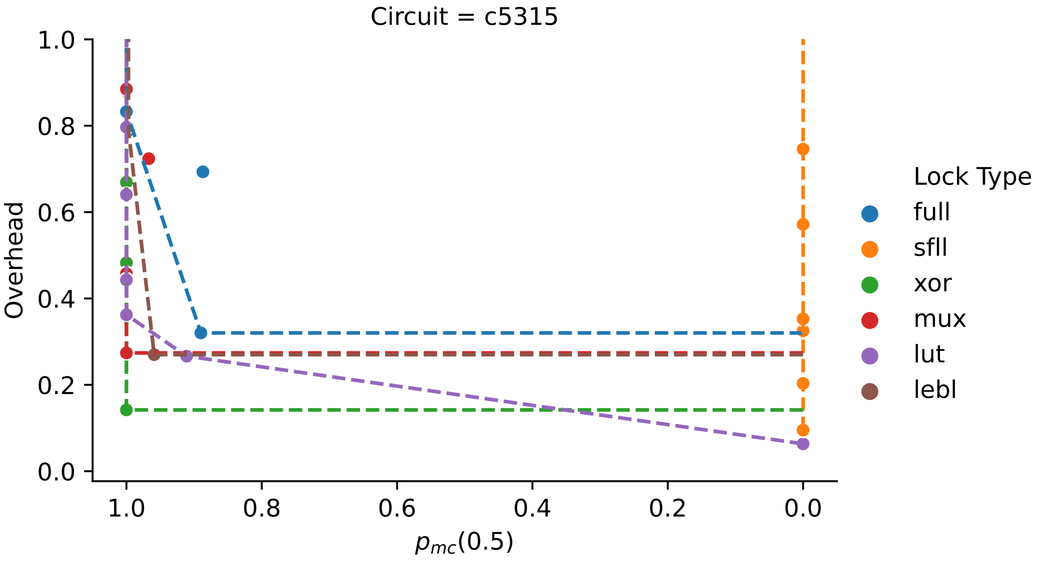


Figure 5.6: Overhead vs. p_{mc} Pareto fronts for selected locking techniques.

Coupling this data with an attack model and corresponding security metric, we can visualize the overhead-corruption trade-off.

Fig. 5.6 displays the trade-off under the netlist attack model. We use p_{mc} at $\epsilon = 0.5$ plotted against the overhead. For each locking scheme, we draw the Pareto front. By a significant margin, the best performing locking scheme under these criteria is XOR-locking. With overhead less than 20%, it shows a $p_{mc}(0.5) \approx 1$.

The oracle attack model results are plotted in Fig. 5.7. To handle the oscillations, we use the average *KeyCor* of the keys produced in the last 100 seconds of the attack. Full-Lock shows the best result, closely followed by MUX-locking and LEBL. The other techniques appear as vertical lines since they terminated for all runs. Even though in Fig. 5.5, MUX-Locking has a lower per-bit key corruption, its low overhead makes it comparable to Full-Lock and LEBL.

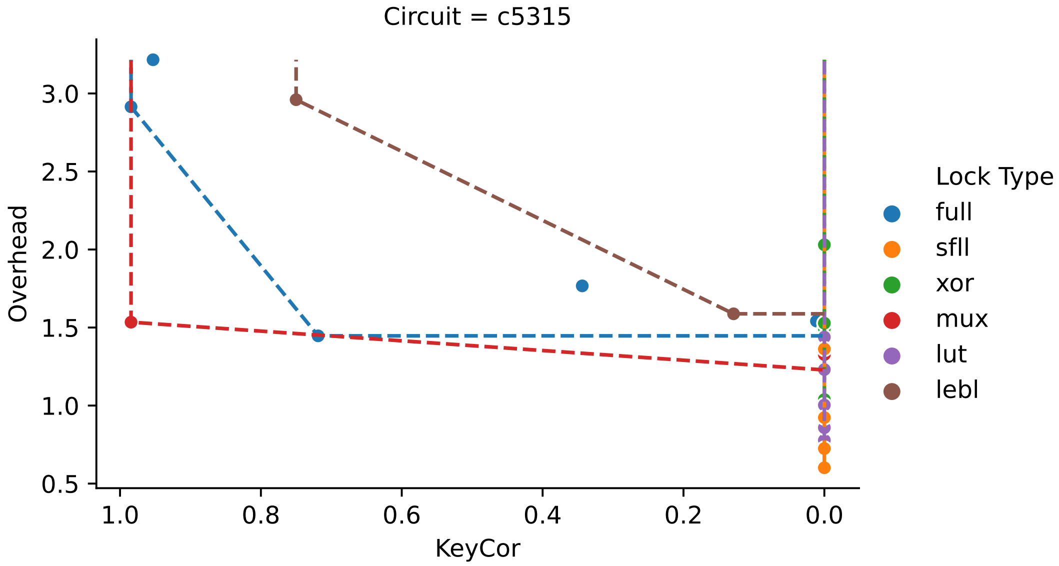


Figure 5.7: Overhead vs. key corruption Pareto fronts for selected locking techniques. The zero value of key corruption is mapped to 10^{-19}

5.3 Discussion

The proposed metrics capture a more nuanced picture of a locking scheme than previous evaluation methods. By tailoring our metrics to specific scenarios, we can better understand what the adversary will be able to achieve along with the overhead costs. Our metrics provide a framework that can allow more detailed attack comparisons, as demonstrated with our analysis of XOR-locking under a structural analysis attack.

Notably, our results capture the disparity between the netlist and oracle attack models. Under the netlist attacks, it is reasonable to expect the sampled keys will exhibit significant corruption. However, once an oracle is available, the corruption values drop by many orders of magnitude, even when substantially scaling the allowed overheads. This suggests that *finding ways to prevent oracle access should be a topmost priority for the logic locking community*.

Several interesting directions exist for future work. We have shown key corruption to be a useful metric. Its counterpart, input corruption, may also provide important information to a lock designer. For example, if most keys produce the correct answer for an input, a simple voting scheme between randomly selected keys could produce a circuit that closely emulates the correct functionality. Input corruption, along with other metrics, will likely give a more complete notion of security, especially when considering the diverse array of possible attack methods.

Utilizing model counting techniques, as we have done here, may provide a means to estimate other metrics in a scalable fashion that also provides bounds useful for assessing risk. We have also shown that our proposed metrics can be evaluated for circuits with several thousands of gates, within the range of the logic cone sizes of many industrial circuits. Understanding the limits of these approximation techniques, specifically their applicability to other metrics and scaling with circuit size, is a promising research direction.

Finally, because the vast majority of locking techniques simply modify the next state logic, we have evaluated these metrics in a combinational setting. As more sequential locking techniques are developed, such as our latch-based logic locking presented in the previous chapter, extending the metrics to capture the sequential setting will be valuable. This can be achieved by unrolling the circuits as is done for sequential ATPG methods.

Chapter 6

Conclusion and Future Work

Logic locking techniques seek to provide maximal security while maintaining low overhead in the circuit area, power, delay, and testability. This thesis has taken several important steps towards realizing this goal, but there remain many open avenues for improvement. Examples include parallelized attack algorithms that integrate different types of analysis, structural attacks that match locked circuits or portions of them to known circuits, further exploration of circuit properties that are typically abstracted away for use in security, and many others. In the following sections, we describe two such lines of research that we believe are critical to the field and promising directions for improvements. Finally, we conclude with a summary of the thesis contributions.

6.1 Improving and Impeding Oracle-Based Attacks

Naturally on both sides of logic locking, designer and adversary, there is much outstanding work. A common thread throughout this thesis has been the power of the oracle-based attack model. As we discuss in chapter 3, an unlocked circuit is a powerful tool to decipher the keys of a circuit. And while our latch-based logic locking technique shows exponential resistance to the termination of known oracle-based attacks, as we saw in chapter 5, this may not imply that the technique is secure. For latch-based logic locking but also locking techniques in general, more analysis must be conducted to verify the intermediate keys produced by the attack are sufficiently incorrect. In conjunction, further exploration of which specific logic structures are most resistant to the miter-based attacks while maintaining low overhead and high corruption is needed. Towards that end, a parameter exploration of known hard structures may produce an answer.

Furthermore, we have demonstrated key bit predictions greater than chance when applying classifiers to our latch-based technique. While on its own, this bias is not great enough to produce a working circuit, a potential use is in an oracle-based attack. Predictions about the correct portion of the key space may be useful in selecting inputs to test, potentially ruling out incorrect keys more quickly. Beyond the incorporation of these predictions, oracle-based attacks can likely be sped up using parallelization techniques and other algorithms beyond formal methods such as probabilistic methods.

A promising approach to thwart this oracle-attack model is the development of methods to prohibit access to the circuit. In certain contexts, this can be solved with physical security measures, such as ensuring the IC is monitored at all times. In other scenarios, like cloud computing, strict regulation of the IC interfaces may impede an adversary. However, in some commercial applications this may be impractical. Here, removing consumer access to test infrastructure and perhaps even online monitoring methods may be required. Design-for-security strategies, similar to the notion of design-for-testability, may provide system level strategies to restricting oracle-based attacks.

6.2 Security-Aware Logic Synthesis

In general, existing locking solutions have several issues. Described in our work here and that of other authors, many locking schemes rely on the functionality of specific logic structures. In such cases, it is often straightforward to find and remove the modifications [87, 88]. Additionally, many techniques do not significantly modify the structure of the original circuit, which can trivialize the process of identifying the circuit (e.g., arithmetic

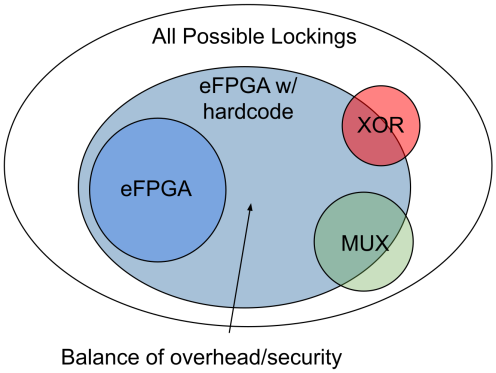


Figure 6.1: Space of possible logic lockings.

circuits). Furthermore, attacks exist that can analyze local structure and predict keys [40]. These issues extend to deployment as well. Locking schemes are not tailored to a given circuit, resulting in large variations in attack resistance and overhead. In applying the locks, existing techniques implicitly rely on a circuit designer to insert the locking. For critical applications, evaluating the security vs. overhead tradeoffs requires a security expert or an automated process that has been proven across a wide variety of circuits.

One promising locking mechanism is the use of embedded field-programmable arrays (eFPGAs) and similar structures [26, 23, 87]. These methods of redacting a portion of the design and replacing it with an eFPGA fabric or logical equivalent have shown resistance to existing attacks, but often at a significant overhead cost. The eFPGAs provide densely interconnected programmable bits that enable the implementation of many diverse functionalities within a locked circuit. This leaves the adversary with a massive space of designs to decipher. Unfortunately, this security comes at a cost, since embedding an FPGA into a circuit will tend to increase area, power and worsen critical paths. Steps have been taken to reduce the overheads by hard-coding portions of the eFPGA [26]. While these methods along with an astute designer can mitigate this overhead cost, the eFPGA insertion relies on cumbersome human intervention to do it effectively.

We seek a locking method that provides security comparable to the eFPGA based methods but without the prohibitively high overheads. Moreover, this method should

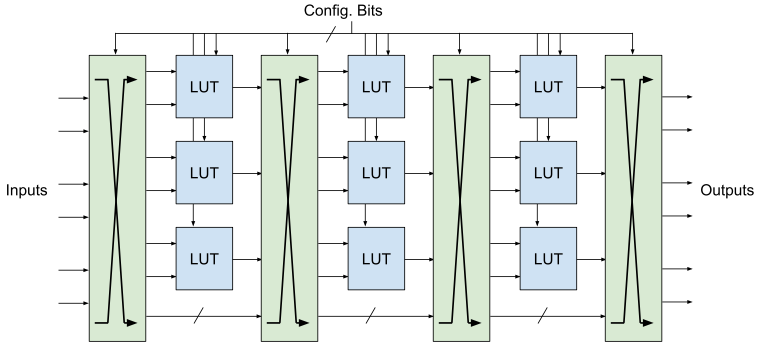


Figure 6.2: Example programmable fabric construction.

allow for straightforward deployment, circuit-independent security, and global manipulation of the circuit. Towards this end, we are proposing a novel strategy for the development of a locking technique. Consider Fig. 6.1, which depicts how the eFPGA method corresponds to a relatively small subset of the overall space of all possible lockings of a circuit. The structure of the method limits what portion of the locking space can be explored by the designer. Adding the ability to hardcode portions of the eFPGA allows a larger space of lockings to be considered, thereby incorporating solutions with potentially better security-overhead tradeoffs and even other locking types. Unfortunately, the size of this space limits the effectiveness of a human-driven exploration. Our strategy is to enable algorithmic exploration of this extended space.

We propose a locking flow that begins with a fully programmable fabric – an example is depicted in Fig. 6.2. Conceptually similar to an eFPGA, the fabric is able to implement any circuit within a certain size and resistant to deobfuscation. Unlike an eFPGA, we choose a fabric that has no feedback loops since they are problematic for logic synthesis, and our starting point does not strongly impact the quality of the final mapped circuits. The circuit to be locked is programmed on the programmable circuit using a set of configuration bits, analogous to the configuration bits in an FPGA, but also to the key bits for locking techniques in general. Using this fabric structure and the configuration bits, we optimize the circuit, hardcoding portions of the configuration bits. The hardcoded bits allow logic synthesis to be applied to reduce overheads and

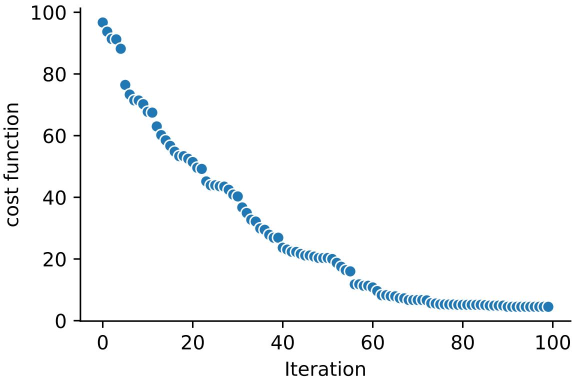


Figure 6.3: Initial optimization results.

superfluous gates.

The optimization is driven by newly developed security metrics, such as those developed in chapter 5 or available attacks that characterize a locked circuit's resistance to a given attack model. We start with the fabric and no hardcoded bits, therefore just as secure as an unprogrammed FPGA. Then, using a security metric, we perform optimization to select which logic values can be hardcoded, and apply logic synthesis to map the partially hardcoded fabric to a logic cell-based circuit that maintains sufficient security but provides far superior performance, power and area.

Incorporating these measures into the synthesis flow will allow optimization methods to build effective locking schemes. A security-based logic synthesis flow can explore the vast space of possible lockings for a specific circuit. Whereas previous solutions have been limited by human-scale manipulations, we envision an extensible synthesis framework in which cloud-based scaling could run thousands of security-constrained optimizations in parallel and utilize machine learning based search for the best designs.

We have created a small proof of concept example to demonstrate the efficacy of this approach that begins with selection of a structure for the programmable fabric (Fig. 6.2). For this demonstration example we utilize a genetic algorithm to select the hardcoded configuration bits based on a cost function that captures the overhead of the design compared to the original circuit. The optimization is constrained by our minimum corruption security metric that ensures a very low probability of randomly selecting a key that has low corruption. Specifically, we set $p_{mc}(0.1) = 0.95$. The algorithm minimizes the overhead while respecting this metric. Fig. 3 demonstrates its progress on a small logic circuit example. As the iteration count increases, portions of the design are hard-coded that lead to improved overheads, along with maintained security.

This is a promising direction forward for logic locking and there are many outstanding problems to tackle, including the following:

- Development of a cloud-based parallelizable logic synthesis
- Creation of additional security metrics that more diverse attack schemes
- Exploration of optimization techniques, pulling from the wide range of machine learning algorithms
- Analysis of relationships between number of key bits, positioning of key bits to determine security and overhead impact
- Interfacing with black-hat adversaries to analyze attack resistance versus optimization time and compute power

6.3 Conclusion

Previously proposed locking schemes do not provide adequate security while maintaining low overheads. With each new lock, new attacks have been developed that can reveal the circuit's correct key under various attack conditions. To resist such attacks, lock designers have sacrificed greater overheads. In doing so, many schemes push the cost of locking a circuit to impractical levels for some applications.

In the preceding chapters, we have described novel attacks on existing families of locking techniques as well as corresponding fixes. While these patched locking techniques solve the immediate threats, they are not satisfying locking solutions, continuing the existing trend in the field of *ad hoc* security mechanisms, developed in response to specific attacks.

To break this trend and produce a low-overhead locking solution, we have developed latch-based logic locking that introduces a new paradigm in manipulating circuits. Keying portions of the design that are not in the timing critical paths, specifically the clock tree, enables large amounts of interdependent key bits and decoy logic to be inserted without the large delays associated with other techniques. Moreover, latch-based locking has been shown to be resistant to existing attack schemes as well as newly developed approaches.

Beyond the development of this technique, we have taken steps towards better quantization of existing attack models. For the two prevalent attack models, we have developed metrics that capture notions of how effective a given technique is. This analysis has revealed interesting results under both attack models, suggesting the effectiveness of the early XOR-based techniques under the netlist attack model, and the need to analyze intermediate solutions in the SAT-based attacks.

Overall, this thesis strives to mitigate the threat of IP theft in manufacturing. We have explored several new directions in this domain, demonstrating existing weaknesses, showing promising results with a new locking scheme, and building a framework for better quantification of security. While many important challenges remain open, the work here is a step forward, that in turn will enable additional advancement towards this ultimate goal.

Bibliography

[1] George H Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955. 5

[2] D. Cheng, Y. Shi, B. Gwee, K. Toh, and T. Lin. A hierarchical multiclassifier system for automated analysis of delayered ic images. *IEEE Intelligent Systems*, 34(2):36–43, 2019. 6

[3] Ujjwal Guin, Ziqi Zhou, and Adit Singh. A novel design-for-security (DFS) architecture to prevent unauthorized IC overproduction. In *Proceedings of the IEEE VLSI Test Symposium*, 2017. 7

[4] Farinaz Koushanfar, Gang Qu, and Miodrag Potkonjak. Intellectual property metering. In *International Workshop on Information Hiding*, pages 81–95. Springer, 2001. 7

[5] M. Rostami, F. Koushanfar, and R. Karri. A primer on hardware security: Models, methods, and metrics. *Proceedings of the IEEE*, 102(8):1283–1295, 2014. 7

[6] F. Koushanfar J. A. Roy and I. L. Markov. EPIC: Ending Piracy of Integrated Circuits. *2008 Design, Automation and Test in Europe*, 2008. 7, 88

[7] Kyle Juretus and Ioannis Savidis. Increasing the sat attack resiliency of in-cone logic locking. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2019. 7

[8] Jeyavijayan Rajendran, Huan Zhang, Chi Zhang, Garrett S. Rose, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. Fault Analysis-Based Logic Encryption. *IEEE Transactions on Computers*, 64(2):410–424, 2015. 7, 10, 88

[9] M. N. Mneimneh and K. A. Sakallah. Principles of sequential-equivalence verification. *IEEE Design Test of Computers*, 22(3):248–257, 2005. 8

[10] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. 8

[11] M. T. Rahman, S. Tajik, M. S. Rahman, M. Tehranipoor, and N. Asadizanjani. The key is left under the mat: On the inappropriate security assumption of logic locking schemes. *Cryptology ePrint Archive, Report 2019/719, 2019, https://eprint.iacr.org/2019/719, to appear at HOST 2020.* 9

[12] Benjamin Tan, Ramesh Karri, Nimisha Limaye, Abhrajit Sengupta, Ozgur Sinanoglu, Md Moshiur Rahman, Swarup Bhunia, Danielle Duvalsaint, R. D., Blanton, Amin Rezaei, Yuanqi Shen, Hai Zhou, Leon Li, Alex Orailoglu, Zhaokun Han, Austin Benedetti, Luciano Brignone, Muhammad Yasin, Jeyavijayan Rajendran, Michael Zuzak, Ankur Srivastava, Ujjwal Guin, Chandan Karfa, Kanad Basu, Vivek V. Menon, Matthew French, Peilin Song, Franco Stellari, Gi-Joon Nam, Peter Gadfort, Alric Althoff, Joseph Tostenrude, Saverio Fazzari, Eric Breckenfeld, and Kenneth Plaks. Benchmarking at the frontier of hardware security: Lessons from logic locking, 2020. 9

[13] H. Mardani Kamali, K. Zamiri Azar, K. Gaj, H. Homayoun, and A. Sasan. Lut-lock: A novel lut-based logic obfuscation for fpga-bitstream and asic-hardware protection. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 405–410, 2018. 10, 88

[14] Pramod Subramanyan, Sayak Ray, and Sharad Malik. Evaluating the security of logic encryption algorithms. *Proceedings of the 2015 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2015*, pages 137–143, 2015. 10, 15

[15] Yang Xie and Ankur Srivastava. Anti-SAT: Mitigating SAT Attack on Logic Locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(2):199–207, 2 2019. 10

[16] Muhammad Yasin, Bodhisatwa Mazumdar, Jeyavijayan J V Rajendran, and Ozgur Sinanoglu. SARLock: SAT attack resistant logic locking. *Proceedings of the 2016 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2016*, pages 236–241, 2016. 10

[17] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. Security Analysis of Anti-SAT. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017. 10

[18] Kaveh Shamsi, Meng Li, Travis Meade, Zheng Zhao, David Z Pan, and Yier Jin. Cyclic Obfuscation for Creating SAT-Unresolvable Circuits. *Proceedings of the on Great Lakes Symposium on VLSI*, pages 173–178, 2017. 11

[19] Yang Xie and Ankur Srivastava. Delay Locking: Security Enhancement of Logic Locking against IC Counterfeiting and Overproduction. *Design Automation Conference (DAC)*, 2017. 11

[20] Abhishek Chakraborty, Yuntao Liu, and Ankur Srivastava. TimingSAT. In *Proceedings of the International Conference on Computer-Aided Design*, 2018. 11

[21] Hai Zhou, Ruifeng Jiang, and Shuyu Kong. CycSAT: SAT-based attack on cyclic logic encryptions. In *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, 2017. 11, 34, 45, 88, 113

[22] Danielle Duvalsaint, Zeye Liu, Ananya Ravikumar, and Ronald D. Blanton. Characterization of Locked Sequential Circuits via ATPG. In *2019 IEEE International Test Conference in Asia (ITC-Asia)*, pages 97–102. IEEE, 9 2019. 11, 14

[23] M. M. Shihab, J. Tian, G. R. Reddy, B. Hu, W. Swartz, B. Carrion Schaefer, C. Sechen, and Y. Makris. Design obfuscation through selective post-fabrication transistor-level programming. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 528–533, 2019. 11, 99

[24] Hadi Mardani Kamali, Kimia Zamiri Azar, Houman Homayoun, and Avesta Sasan. Full-Lock. pages 1–6. Association for Computing Machinery (ACM), 2019. 11, 33, 88

[25] Kaveh Shamsi, Meng Li, David Z Pan, and Yier Jin. Cross-Lock: Dense Layout-Level Interconnect Locking using Cross-bar Architectures. 2018. 11

[26] Prashanth Mohan, Oguz Atli, Joseph Sweeney, Onur Kibar, Lawrence Pileggi, and Ken Mai. Hardware redaction via designer-directed fine-grained soft efpga insertion. In *Design, Automation and Test in Europe (DATE-21)*. IEEE, 2021. 11, 99

[27] Rajat Subhra Chakraborty and Swarup Bhunia. HARPOON: An obfuscation-based SoC design methodology for hardware protection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1493–1502, 2009. 11

[28] Travis Meade, Zheng Zhao, Shaojie Zhang, David Pan, and Yier Jin. Revisit Sequential Logic Obfuscation : Attacks and Defenses. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017. 11

[29] Mohamed El Massad, Siddharth Garg, and Mahesh Tripunitara. Reverse engineering camouflaged sequential circuits without scan access. In *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, 2017. 12, 18, 63

[30] Kaveh Shamsi, Meng Li, David Z. Pan, and Yier Jin. KC2: Key-Condition Crunching for Fast Sequential Circuit Deobfuscation. In *Proceedings of the 2019 Design, Automation and Test in Europe Conference and Exhibition, DATE 2019*, pages 534–539. Institute of Electrical and Electronics Engineers Inc., 5 2019. 12, 18, 63

[31] I. Kuon and J. Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007. 12

[32] Kaushik Vaidyanathan, Bishnu P Das, Ekin Sumbul, Renzhi Liu, and Larry Pileggi. Building trusted ics using split fabrication. In *2014 IEEE international symposium on hardware-oriented security and trust (HOST)*, pages 1–6. IEEE, 2014. 12

[33] S. Pagliarini, J. Sweeney, K. Mai, S. Blanton, S. Mitra, and L. Pileggi. Split-chip design to prevent ip reverse engineering. *IEEE Design Test*, pages 1–1, 2020. 13

[34] Joseph Sweeney, Samuel Pagliarini, and Lawrence Pileggi. Securing digital systems via split-chip obfuscation. *GOMACTech Technical Program*, 2019. 13

[35] B. Erbagci, C. Erbagci, N. E. C. Akkaya, and K. Mai. A secure camouflaged threshold voltage defined logic family. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 229–235, 2016. 13

[36] R. P. Cocchi, J. P. Baukus, L. W. Chow, and B. J. Wang. Circuit camouflage integration for hardware ip protection. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–5, 2014. 13

[37] A. Vijayakumar, V. C. Patil, D. E. Holcomb, C. Paar, and S. Kundu. Physical design obfuscation of hardware: A comprehensive investigation of device and logic-level techniques. *IEEE Transactions on Information Forensics and Security*, 12(1):64–77, 2017. 13

[38] Farinaz Koushanfar. *Hardware Metering: A Survey*, pages 103–122. Springer New York, New York, NY, 2012. 13

[39] Ioannis Karageorgos, Mehmet M Isgenc, Samuel Pagliarini, and Larry Pileggi. Chip-to-chip authentication method based on sram puf and public key cryptography. *Journal of Hardware and Systems Security*, 3(4):382–396, 2019. 13

[40] P. Chakraborty, J. Cruz, and S. Bhunia. Sail: Machine learning guided structural analysis attack on hardware obfuscation. In *2018 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 56–61, 2018. 14, 90, 99

[41] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri. Security analysis of logic obfuscation. In *DAC Design Automation Conference 2012*, pages 83–89, 2012. 14

[42] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, London, UK, 1999. Springer-Verlag. 15

[43] Franjo Ivančić, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient sat-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3):256 – 274, 2008. International Symposium on Leveraging Applications of Formal Methods (ISoLA 2004). 15

[44] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 570–596, Cham, 2017. Springer International Publishing. 15

[45] Boris Konev and Alexei Lisitsa. A sat attack on the erdős discrepancy conjecture. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 219–226, Cham, 2014. Springer International Publishing. 15

[46] Roberto J Bayardo Jr and Robert Schrag. Using csp look-back techniques to solve real-world sat instances. In *Aaai/iaai*, pages 203–208. Providence, RI, 1997. 16

[47] João P Marques-Silva and Karem A Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999. 16

[48] Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983. 16

[49] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pages 1–30. Springer, 2011. 18

[50] Aaron R Bradley. Sat-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011. 18, 70

[51] Kenneth L McMillan. Interpolation and sat-based model checking. In *International Conference on Computer Aided Verification*, pages 1–13. Springer, 2003. 18

[52] Muhammad Yasin, Abhrajit Sengupta, Benjamin Carrion Schafer, Yiorgos Makris, Ozgur Sinanoglu, and Jeyavijayan Rajendran. What to Lock? Functional and Parametric Locking. 21

[53] Mohamed El Massad. *On the Foundations of Integrated Circuit Intellectual Property*. PhD thesis, New York University, 2019. 21

[54] Muhammad Yasin, Abhrajit Sengupta, Mohammed dari Nabeel, Mohammed Ashraf, Jeyavijayan Rajendran, and Ozgur Sinanoglu. Provably-Secure Logic Locking: From Theory To Practice. Technical report. 21, 88

[55] Nadia Creignou and HervÃ© Daudé. Sensitivity of Boolean formulas. *European Journal of Combinatorics*, 34(5):793–805, 7 2013. 23

[56] Franc Brglez and Hideo Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a targeted translator in fortran. *Special session on ATPG and fault simulation, Proc. IEEE International Symposium on Circuits and Systems*, pages 663–698, 06 1985. 24, 31, 88

[57] Armin Biere. Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2018. In *Proceedings of SAT Competition 2018*, pages 13–14, 2018. 30, 38, 88

[58] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. *KR*, 96:148–159, 1996. 33

[59] Jian Ding, Allan Sly, and Nike Sun. Proof of the satisfiability conjecture for large k. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 59–68, 2015. 33

[60] Kaveh Shamsi, David Z Pan, and Yier Jin. Icysat: Improved sat-based attacks on cyclic locked circuits. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7. IEEE, 2019. 34, 69, 113, 114

[61] Meng Li, Kaveh Shamsi, Yier Jin, and David Z. Pan. TimingSAT: Decamouflaging Timing-based Logic Obfuscation. In *Proceedings - International Test Conference*, volume 2018-October. Institute of Electrical and Electronics Engineers Inc., 1 2019. 35

[62] G. L. Zhang, B. Li, B. Yu, D. Z. Pan, and U. Schlichtmann. Timingcamouflage: Improving circuit security against counterfeiting by unconventional timing. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 91–96, 2018. 36

[63] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018. 38

[64] Cunxi Yu, Xiangyu Zhang, Duo Liu, Maciej Ciesielski, and Daniel Holcomb. Incremental sat-based reverse engineering of camouflaged logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(10):1647–1659, 2017. 39

[65] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. Shatter: Efficient symmetry-breaking for boolean satisfiability. In *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, page 836–839, New York, NY, USA, 2003. Association for Computing Machinery. 40

[66] M. Yasin, J. J. Rajendran, O. Sinanoglu, and R. Karri. On improving the security of logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(9):1411–1424, 2016. 40

[67] Mate Soos and Kuldeep S Meel. Bird: Engineering an efficient cnf-xor sat solver and its applications to approximate model counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1592–1599, 2019. 48

[68] K. Yoshikawa, K. Kanamaru, S. Inui, Y. Hagihara, Y. Nakamura, and T. Yoshimura. Timing optimization by replacing flip-flops to latches. pages 186–191. Institute of Electrical and Electronics Engineers (IEEE), 10 2004. 50, 54

[69] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 1991. 50

[70] Perry H. Wang, Sebastian Steibl, Hong Wang, Jamison D. Collins, Christopher T. Weaver, Blliappa Kuttanna, Shahram Salamian, Gautham N. Chinya, Ethan Schuchman, Oliver Schilling, and Thorsten Doil. Intel® atomâ„¢ processor core made FPGA-synthesizable. page 209. Association for Computing Machinery (ACM), 2 2009. 50

[71] Graham Schelle, Jamison Collins, Ethan Schuchman, Perry Wang, Xiang Zou, Gautham Chinya, Ralf Plate, Thorsten Mattner, Franz Olbrich, Per Hammarlund, Ronak

Singhal, Jim Brayton, Sebastian Steibl, and Hong Wang. Intel© Nehalem Processor Core Made FPGA Synthesizable. In *FPGA*, page 296. ACM, 2010. 50

[72] Ferran Parés, Dario Garcia-Gasulla, Armand Vilalta, Jonatan Moreno, Eduard Ayguadé, JesÃºs Labarta, Ulises Cortés, and Toyotaro Suzumura. Fluid Communities: A Competitive, Scalable and Diverse Community Detection Algorithm. 3 2017. 54

[73] David Chinnery, Kurt Keutzer, Jagesh Sanghavi, Earl Killian, and Kaushik Sheth. Automatic Replacement of Flip-Flops by Latches in ASICs. In *Closing the Gap Between ASIC & Custom*. 2004. 54

[74] Min Shi, Jin He, Lining Zhang, Chenyue Ma, Xingye Zhou, Haijun Lou, Hao Zhuang, Ruonan Wang, Yongliang Li, Yong Ma, Wen Wu, Wenping Wang, and Mansun Chan. Zero-mask contact fuse for one-time-programmable memory in standard CMOS processes. *IEEE Electron Device Letters*, 32(7):955–957, 7 2011. 58

[75] Kuei-Sheng Wu, Tseng Ching-Hsiang, Wong Chang-Chien, Chi Sinclair, Su Titan, Liu Yensong, Wei Huan-Sheng, Lien Wai Yi, and Chen Chuck. Investigation of Electrical Programmable Metal Fuse in 28nm and beyond CMOS Technology. In *2011 IEEE International Interconnect Technology Conference*. IEEE, 2011. 58

[76] Kimia Zamiri Azar, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. Smt attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the sat attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 97–122, 2019. 60

[77] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. 61

[78] Sven Sandberg. 1 homing and synchronizing sequences. In *Model-based testing of reactive systems*, pages 5–33. Springer, 2005. 67

[79] S. Malik. Analysis of cyclic combinational circuits. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 618–625, 1993. 69

[80] Peter Eades, Xuemin Lin, and William F Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, 1993. 70

[81] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In Armin Biere and Roderick Bloem, editors, *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014. 70

[82] Matthew Hicks, Paul Fiscarelli, engll, and Brendon Chetwynd. mit-ll/cep: Cep release v3.01, December 2020. 71

[83] K. Shamsi, T. Meade, M. Li, D. Z. Pan, and Y. Jin. On the approximation resiliency of logic locking and ic camouflaging schemes. *IEEE Transactions on Information Forensics and Security*, 14(2):347–359, 2019. 85

[84] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic sat calls. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 7 2016. 88

[85] Mate Soos and Kuldeep S. Meel. Bird: Engineering an efficient cnf-xor sat solver and its applications to approximate model counting. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 1 2019. 88

[86] Joseph Sweeney, Ruben Purdy, Ronald D Blanton, and Lawrence Pileggi. Circuitgraph: A python package for boolean circuits. *Journal of Open Source Software*, 5(56):2646, 2020. 88

[87] J. Sweeney, M. J. H. Heule, and L. Pileggi. Modeling techniques for logic locking. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2020. 98, 99

[88] M Yasin, B Mazumdar, and O Sinanoglu. Security analysis of anti-sat. *(Asp-Dac), 2017 . . .*, pages 342–347, 2017. 98

Appendix A

Supporting Algorithms

A.1 Acyclic Key Constraints

Logic locked circuits will often contain feedback cycles that corrupt the formal attack methods if they are not handled appropriately. Previous work has devised algorithms that will create key constraints, ruling out any key that creates feedback through the combinational logic [21, 60]. These algorithms are based on taint-propagation, ensuring that for a sufficient set of nodes in the circuit's combinational logic all paths from the nodes back to themselves are broken. The sufficient set of nodes is a feedback node set for the combinational logic. The feedback node set is a set of nodes in the combinational logic that when disconnected renders the logic acyclic, i.e., no paths exist from one node back to itself.

The algorithms find a feedback node set, F. For each $f \in F$, a function, C_{f} is built that indicates there is not feedback through f. To build this function, we assume the node is split, forming two nodes f and f'. The function is specified recursively as $C_{\mathrm{f}}(f,n)$ indicating there is no structural path from node f to node n. Initially, we define $C_{\mathrm{f}}(f,f) = 0$. $bk(i,n)$ is the condition on the key that i does not effect n. $NK(n)$ is the set of fanin nodes of n that aren't keys. For any node that does not have a key input, $bk(i,n) = 0$.

Then, we recursively build the following:

$$C_{\mathrm{f}}(f,n) = \bigwedge_{i \in NK(n)} C_{\mathrm{f}}(f,i) \vee bk(i,n) \tag{A.1}$$

Simply put, $C_{\mathrm{f}}(f,n)$ is true if the cyclic path to f is broken at node n with $bk(i,n)$ being true or the path is broken upstream. The root node, $C_{\mathrm{f}}(f,f')$ is true when the path is broken. The complete acyclic constraint is defined as:

$$C_{\mathrm{acyclic}}(k) = \bigwedge_{f \in F} C_{\mathrm{f}}(f,f') \tag{A.2}$$

This constraint can then be utilized in various attack procedures. While the specification is most easily understood recursively, building the constraint function is more efficient using dynamic programming methods, sharing logic cones. Ultimately, the same key constraints are produced but with much smaller encoding. We roughly follow a more efficient algorithm defined here [60].

A.2 Logic-Enhanced Banyan Locking Insertion

Here, we describe the insertion algorithm for our logic-enhanced Banyan locking technique. While this lock is resistant to the modeling techniques, the insertion of the Banyan network is more complex than in its predecessor, Full-Lock. Now, gates from the original circuit must be mapped onto the structure of the Banyan network, instead of just being randomly selected. This entails finding a set of gates from the original circuit with connectivity that maps onto the Banyan network's. We automate this process to enable scalable exploration of the mapping solution space. To augment the ability to map onto the Banyan structure, we split all gates from the original circuit with three or more inputs into two input gates. We start with a Banyan network of the desired input width, W and encode the problem of finding a mapping as a SAT instance through constraints that we specify below.

The encoding uses a set of variables representing a mapping between an original gate g and a banyan switch box output s. See Fig. 3.12 for a visualization of the gates and switch box outputs. The switch box outputs provide a reference point within the Banyan network that naturally correspond to gate outputs in the original circuit. For all pairs of gates in the original circuit and switch box outputs in the Banyan network, $(g, s) \in C \times B$, we create a mapping variable m_{gs}. The variable is true if gate g is mapped to switch box output s. Over these variables, we encode constraints that ensure the amount of mapping is sufficient. First, we ensure at least W gates are mapped to the network.

$$AtLeastW(\{\bigvee_{s \in B} m_{gs} : g \in C\}) \tag{A.3}$$

Then we encode that at most one gate is mapped per switch box output.

$$\bigwedge_{s \in B} AtMostOne(\{m_{gs} : g \in C\}) \tag{A.4}$$

We allow the same gate to be mapped to multiple switch box outputs enabling the mapping of gates with fanout and to allow a gate to feed through a switch box. Finally, we prohibit any path directly feeding all the way through from the network inputs to outputs, avoiding the simplest mappings. This is done by prohibiting a gate to be mapped to both the first and last layer of the Banyan network. We show the encoding below where B_i and B_o are respectively the sets of switch box outputs in the first and last layers of the network.

$$\bigwedge_{g \in C} \bigwedge_{s_i \in B_i} \bigwedge_{s_o \in B_o} AtMostOne(m_{gs_o}, m_{gs_i}) \tag{A.5}$$

To maintain the structure of the circuit, we add constraints that enforce a correspondence between the connectivity of the mapped gates and the switch box outputs. If a gate is mapped to a switch box output, the fanin of the gate in the original circuit must be mapped to the fanin of the switch box (i.e., the switch box outputs from the preceding network layer). Similarly, we also ensure that at least one of the gate's fanout is mapped to the fanout of the switch box. We allow an exception to this rule if the gate is simply fed through the switch box, which adds flexibility to the circuit structures that can be mapped. Note that here we are allowing feed through for a switch box but prohibit it through the entire network. More formally, m_{gs} implies that every fanin of g is mapped to the fanin of s or, in the case of a feedthrough, g itself is mapped to the fanin of s. This encoding is shown below.

$$\bigwedge_{s \in B} \bigwedge_{g \in C} \bigwedge_{g_f \in fanin(g)} m_{gs} \rightarrow \bigvee_{s_f \in fanin(s)} m_{g_f s_f} \vee m_{g s_f} \tag{A.6}$$

Additionally, m_{gs} implies that at least one fanout of g is mapped to the fanout of s or, in the case of a feedthrough, g is in the fanout of s. This encoding is shown below.

$$\bigwedge_{s \in B} \bigwedge_{g \in C} m_{gs} \rightarrow \bigvee_{s_f \in fanout(s)} \bigvee_{g_f \in fanout(g) \cup \{g\}} m_{g_f s_f} \tag{A.7}$$

This system of constraints is solved and the gates in the resulting mapping are inserted into their corresponding switch boxes and removed from the original circuit. The other MUX inputs are connected to randomly selected decoy functions of the switch box inputs. The network inputs and outputs are connected depending on which gates have been mapped to the first and last layer of switch boxes. It's important to emphasize that no intermediate connections are made to or from the network. Outputs with no mapping are randomly connected to remaining gates such that they have no impact on the logic of the system under the correct key.