VIGILANT: Vulnerability Detection Tool against Fault-Injection Attacks for Locking Techniques

Likhitha Mankali, Satwik Patnaik, Nimisha Limaye, Johann Knechtel, *Member, IEEE,* and Ozgur Sinanoglu, *Senior Member, IEEE*

***Abstract*—Logic locking is a well-known solution that thwarts design intellectual property (IP) piracy and prevents illegal overproduction of integrated circuits (ICs) against adversaries in the globalized supply chain. The widespread prevalence of reverse-engineering tools, probing, and fault-injection equipment has given rise to physical attacks that can undermine the security of a locked design. Fault-injection attacks, in particular, can extract the secret key from an oracle, circumventing the defense offered by logic locking. When design IP is compromised through physical attacks, fixing corresponding vulnerabilities generally require a silicon re-spin, which is impractical under constrained time and resources. Thus, there is a requirement for a detection tool that can perform a pre-silicon evaluation of locked designs to notify the designer of any vulnerabilities that can be exploited using faults. In this work, we propose *VIGILANT*, a first-of-its-kind vulnerability detection tool against fault-injection attacks targeting the hardware implementation of locking techniques. More specifically, *VIGILANT* aids designers in identifying critical nets susceptible to fault-injection attacks. *VIGILANT* analyzes the underlying locked design and computes a list of candidate nets along with their fault values required for key leakage and consequently validates each candidate net as vulnerable or not, using a functional simulation model of the design (acting as an oracle). We showcase the efficacy of *VIGILANT* on different locked designs for four different locking techniques under various parameters such as technology nodes, layout-generation commands, and key-sizes. The accuracy of *VIGILANT* in identifying and validating all the candidate nets that are vulnerable to fault-injection attacks is 100%.**

***Index Terms*—Logic locking, Physical attacks, Fault-injection**

I. Introduction

With continuous advancements in lower technology nodes shepherding the integrated circuit (IC) design process, foundries are also required to constantly revamp their machinery (*e.g.,* extreme ultraviolet lithography tools) [1]. Such an effort towards re-tooling necessitates sky-rocketing investment [2] toward owning and building a state-of-the-art foundry, forcing design companies (*e.g.,* Apple, Qualcomm, and NVIDIA) to go fabless [3]. With the fabless business model, involvement of potentially untrustworthy third-party entities (*e.g.,* foundry, testing, and packaging facilities) in the supply chain leads to several threats, such as design intellectual property (IP) piracy, illegal overproduction of ICs, and insertion of hardware Trojans [4]. According to recent estimates, hardware IP piracy results in losses of billions of dollars to the US economy, reducing research and development (R&D) investment and innovation in US companies [5].

To combat design IP piracy and unauthorized overproduction of ICs, security researchers have proposed several design-for-trust solutions, such as IC camouflaging, split manufacturing, and logic locking. Amongst these, logic locking is a one-stop, holistic solution that thwarts design IP piracy against all untrustworthy entities in the globalized IC supply chain [6], [7].

A. Logic Locking

Logic locking inserts additional, key-controlled logic gates (a.k.a. key-gates) in the design IP such that the design becomes functional or non-functional on the application of the correct or incorrect secret key respectively [6]. After fabrication and testing, the IP is unlocked via loading of the secret key by a trustworthy entity (*e.g.,* a design house) into a tamper-proof memory within the IC. Note that the key feeds the key-inputs of all the key-gates.

Logic locking has received considerable attention not only from academic researchers but also from government and commercial entities. For instance, the Defense Advanced Research Projects Agency (DARPA) is investing in R&D of logic locking through various programs, namely Efficient Cross-Layered IP Protection Scheme (ECLIPSE) [8] and Automatic Implementation of Secure Silicon (AISS) [9]. On the commercial front, Synopsys has integrated logic locking in its electronic design automation tools [10] whereas Mentor Graphics has included it in its TrustChain platform [11].

Note that the ability of logic locking to provide any protection against different adversaries is contingent on the resilience of the secret key. With advancements in reverse-engineering tools and access to unlocked IPs (*i.e.,* an oracle with the secret key embedded, obtained as final IC from the open market), physical attacks pose a potent threat to the security guarantees offered by logic locking techniques.

Manuscript received August 12, 2022; revised January 16, 2023; accepted February 28, 2023. The work of Likhitha Mankali was supported by the Global Ph.D. Fellowship at New York University/New York University Abu Dhabi. This article was recommended by Associate Editor N. Mentens. (*Corresponding authors: Likhitha Mankali and Satwik Patnaik*)

Likhitha Mankali is with the Department of Electrical and Computer Engineering, Tandon School of Engineering, New York University, NY, USA (email: likhitha.mankali@nyu.edu).

Satwik Patnaik is with the Department of Electrical and Computer Engineering, Texas A&M University, College Station, USA (email: satwik.patnaik@tamu.edu).

Nimisha Limaye was with the Department of Electrical and Computer Engineering, Tandon School of Engineering, New York University, Brooklyn, NY 11201 USA. She is now with the Solutions Group, Synopsys, Inc., Mountain View, CA 94043 USA (e-mail: nimisha.limaye@nyu.edu).

Johann Knechtel, and Ozgur Sinanoglu are with the Division of Engineering, New York University Abu Dhabi, UAE (email: johann@nyu.edu; ozgursin@nyu.edu).

Digital Object Identifier 10.1109/TCAD.2023.3259300

B. Physical Attacks

Physical attacks are primarily categorized into three different types, *i.e.,* side-channel attacks, fault-injection attacks, and probing attacks. In side-channel attacks, adversaries monitor the physical interaction of ICs during execution, like electromagnetic (EM) emission [12], power consumption [13], timing behaviour [14], etc., to extract some secret data, all without directly interacting with the IC [15].

In fault-injection attacks, adversaries manipulate the IC operation directly by injecting faults via different mechanisms (*e.g.,* glitching)[1] to leak critical assets such as keys [17], analyze the underlying design [18], and/or modify the intended functionality [18]. Researchers have demonstrated fault-injection attacks on cryptosystems [17] and locked designs [19].

In probing attacks, adversaries apply tools from microelectronics failure analysis (*e.g.,* electro-optical probing, focused ion beam) for invasive or non-invasive access to the transistors to either read out [20], [21] or manipulate signals [22].

C. Motivation and Research Challenges

Once an IC is deemed vulnerable to fault-injection attacks (or any other physical attack), incorporating some countermeasures would necessitate a silicon re-spin, which is very likely impractical given constrained resources and time-to-market pressure. For example, an IBM study indicates that fixing vulnerabilities in the later stages of the supply chain incurs even higher financial costs, *i.e.,* 110% more than the original cost [23].

Researchers have developed several tools [24], [25] to detect vulnerabilities in cryptosystems (ciphers) against fault-injection attacks during pre-silicon stages. Recently, DARPA incorporated a vulnerability detection tool—Inspector-FI—against fault-injection attacks on ciphers as part of the DARPA toolbox initiative [26].

However, these tools are tailored for algorithm-specific attacks on ciphers, and none of them are suitable for assessing logic locking techniques. More specifically, existing tools exploit paradigms specific to ciphers, *i.e.,* Shannon diffusion and confusion properties. Consider the Shannon diffusion property—changing a single bit in the input (*i.e.,* plaintext for a cipher) will affect a significant amount (*i.e.,* more than half) of the outputs [27]. This property does not apply to most logic locking techniques since changing a single bit in the input pattern affects only a few outputs in the locked design. Similarly, consider the confusion property—changing a single key-bit would affect almost all outputs [27]. This property also does not apply to most logic locking techniques.

Thus, there is a need for a pre-silicon vulnerability detection tool that can detect vulnerabilities in the hardware implementation of a logic locking technique during pre-silicon and notify the designer early on of those vulnerabilities that can be exploited using fault-injection attacks (by an adversary) to retrieve the secret key.

[1]Glitches are unwanted signal variations, *i.e.,* a spike in a clock net or other nets of the design. Glitches can be induced through various physical means, *e.g.,* by injection of photocurrents via laser light, tweaking supply voltages or temperature, etc., [16].

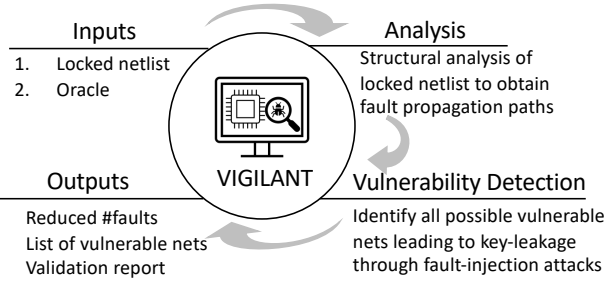


Fig. 1. High-level idea of *VIGILANT*.

Research Challenges (RC). Here, we discuss the research challenges that we seek to address through our work.

RC1 *To identify vulnerable nets against advanced fault-injection.* Note that an adversary could possess advanced fault-injection equipment, *i.e.,* multiple probes capable of injecting many faults concurrently, as considered for the t-probing attack [28]. *Can designers proactively determine vulnerable nets that, when injected with suitable faults, can lead to the retrieval of the secret key by an adversary possessing advanced fault-injection equipment?*

RC2 *To preemptively detect vulnerabilities against fault-injection attacks during the pre-silicon stage (post-layout design).* Conventionally, adversaries mount fault-injection attacks on fabricated ICs. Fixing these vulnerabilities would require a silicon re-spin, which is financially prohibitive and tarnishes the reputation of the design house. *Can a vulnerability analysis be performed at the pre-silicon phase (post-layout design) that would accurately represent the vulnerabilities in the post-silicon design?*

RC3 *To design a scalable, generic, and error-free tool.* There are various logic locking techniques and fault-injection mechanisms. *Can we develop a generic methodology scalable to larger designs and key-sizes to identify/locate vulnerabilities in the hardware implementation of any logic locking technique? Additionally, can we ensure the developed methodology is free from false positives?*

D. Our Goals and Contributions

We address the aforementioned research challenges proactively by developing *VIGILANT*, which identifies vulnerabilities in the hardware implementation of logic locking techniques against fault-injection attacks. Fig. 1 demonstrates the high-level idea of our work. *VIGILANT* (i) is applicable to different logic locking techniques, ranging from provably secure logic locking (PSLL) to SAT-hard techniques, (ii) supports standard Verilog register-transfer level (RTL) and gate-level netlist formats, (iii) works seamlessly across post-synthesis and post-layout designs, (iv) is compatible with standard industry-based IC design flows, (v) is scalable to large-scale designs and key-sizes, and (vi) does not require human intervention. The primary contributions of our work are enumerated as follows.

1) We propose a pre-silicon vulnerability detection tool, *VIGILANT*, utilizing IC testing principles and graph theory-based algorithms to report the vulnerable nets in hardware implementation of locked designs. *VIGILANT* works in three modes considering different established and emerging methods of fault-injection attacks to leak the secret key. *VIGILANT* provides a reduced number and list of vulnerable nets that would facilitate the leakage of a secret key.
2) Without loss of generality (w.l.o.g.), we study two PSLL techniques [29], [30] and two SAT-hard techniques [31], [32] using *VIGILANT*, which includes one unbroken technique [32]. We perform extensive experiments on post-layout designs of both PSLL and SAT-hard techniques. *VIGILANT* successfully reports vulnerabilities for all the locking techniques with 100% validation accuracy.
3) We showcase the effect of fault-injection attacks and the agnostic behavior of *VIGILANT* on different benchmarks. The benchmarks include cyclic and acyclic design configurations, with varying key-sizes (ranging from 128 to 384 bits), for different technology libraries (45nm academic and 22nm commercial) and varying physical-design commands to demonstrate the efficacy of *VIGILANT* across various practically relevant scenarios. *VIGILANT* successfully identified vulnerabilities for all the scenarios with 100% validation accuracy.
4) To foster further research for pre-silicon vulnerability detection against fault-injection attacks for logic locking techniques, we open-source *VIGILANT*.[2]

Paper Organization. We provide the necessary background on the considered logic locking techniques, fault-injection mechanisms, and VLSI testing principles in Sec. II. We discuss the adversarial model in Sec. III. We present the working principles of our vulnerability assessment tool against fault-injection attacks in Sec. IV. We present detailed experimental results in Sec. V, followed by a discussion in Sec. VI. Finally, we provide concluding remarks in Sec. VII.

II. Background and Preliminaries

In this section, we summarize different logic locking techniques considered in this work, followed by a brief review of fault-injection mechanisms and principles of VLSI testing.

A. SAT-Resilient Logic Locking Techniques

Early logic locking techniques such as random logic locking [6], fault-based logic locking [33], and strong logic locking [34] were successfully thwarted by the Boolean satisfiability (SAT)-based attack [35]. The complexity of the SAT-based attack is indicated by the execution time, *i.e.,* $time_{SAT} = \sum_{i=1}^{\lambda} time_i$, where λ is the total number of SAT iterations, and $time_i$ is the time taken per SAT iteration [29].

To thwart the SAT-based attack [35], researchers proposed SAT-resilient techniques which increase (i) the number of SAT iterations to become exponential with respect to the key-size (*e.g.,* Anti-SAT [29]), and/or (ii) the time taken for each SAT attack iteration (*e.g.,* Full-Lock [31]). The techniques under (i) are commonly referred to as point function-based logic locking or provably-secure logic locking, and those under (ii) are referred to as SAT-hard locking techniques.

Provably-Secure Logic Locking (PSLL). These techniques aim to render the number of required SAT iterations to be exponential with respect to the key-size [29], [30], [38]. **Anti-SAT** [29] proposes a locking unit with two complementary functions g (AND tree) and $\overline{g}$ (NAND tree); the locking unit is controlled by the key and primary inputs (PIs) from the design The complementary functions are fed to an AND gate whose output is integrated into the original design. If the output of the Anti-SAT locking unit is 1, it corrupts the functionality of the original design, whereas if the output is 0, the design functions correctly. **CAS-Lock** [30] has a similar structure as Anti-SAT with one major difference: g consists of cascaded AND-OR gates. Compared to Anti-SAT, CAS-Lock was proposed to increase output corruption and thwart structure-based removal attacks along with SAT-based attacks.

SAT-hard Locking Techniques. These techniques thwart the SAT-based attack by considerably increasing the execution time required per SAT iteration by introducing so-called SAT-hard instances [31], [32], [39]. In **Full-Lock** [31], SAT-hard instances are designed using some key-configurable, symmetric, and logarithmic-based network. To increase the complexity further, logic gates succeeding those SAT-hard structures are converted to look-up table (LUT)-based gates. **Logic-enhanced Banyan locking (LEBL)** [32] is an improved version of Full-Lock. LEBL thwarts symmetry-breaking attacks [32] along with SAT-based attacks [35]. For Full-Lock, simple inversions are performed in the key-configuration blocks of the logarithmic network, whereas for LEBL, parts of the original design are redacted, *i.e.,* moved to the key-configurable blocks of the logarithmic network itself.

B. Prior Attacks on Locking techniques

There are different attacks on locking techniques proposed in the literature; selected attacks are listed next. **Canonical prune-and-SAT (CP&SAT)** [40] is an oracle-guided attack on the SAT-hard technique Full-Lock. CP&SAT uses the bounded variable addition (BVA) algorithm and SAT-based attack [35] to recover the key. Researchers have developed attacks that recover the secret key by exploiting the structural vulnerabilities in the locked design [36], [41]. **Circuit-recovery attack** [37] is an oracle-guided attack on Anti-SAT and CAS-Lock. It recovers the original design by exploiting the structural vulnerabilities in a locked design. We discuss the differences between *VIGILANT* and the prior attacks on Anti-SAT, CAS-Lock, Full-Lock, and LEBL in Table I.

C. Fault-Injection Mechanisms

Here, we briefly discuss different fault-injection mechanisms that can be used on ICs. In **clock glitching** [25], an adversary inserts transient faults by increasing or decreasing the frequency of the clock signal, specifically to cause hold and setup timing violations. In **voltage glitching** [42] , an adversary inserts transient faults by tampering with the power

[2]https://github.com/LL-Tools/VIGILANT

TABLE I
HIGH-LEVEL COMPARISON OF *VIGILANT* WITH STATE-OF-THE-ART ATTACKS ON THE LOCKING TECHNIQUES TARGETED IN THIS WORK.

Work	Locking Techniques	Threat model	Attack / Vulnerability detection
	Full-Lock	Oracle-guided †	Attack
SPI [36]	Anti-SAT	Oracle-guided †	Attack
Circuit-recovery attack [37]	Anti-SAT and CAS-Lock	Oracle-guided †	Attack
VIGILANT	Anti-SAT, CAS-Lock, Full-Lock and LEBL	Oracle-guided §	Vulnerability detection against fault-injection attacks

† Black box access to oracle; § Physical access to oracle

supply, *e.g.,* by increasing or decreasing the voltage for a certain period or by introducing voltage spikes.

In **EM fault-injection** [43], an adversary uses probes to inject transient faults arising from current induction by the electromagnetic fields. Note that EM fields are typically less confined (also depending on the probe size); the related fault-injection affects larger areas of the IC under attack. In **laser fault-injection** [44], an adversary uses optical probes that provide a strong and precisely focused laser beam to introduce transient faults at the targeted location in the design. With **focused ion beam** [25], an adversary can insert transient or permanent faults at targeted locations, *e.g.,* by using varying beam currents for milling or deposition of materials.

Note that an adversary may utilize any of the aforementioned mechanisms to mount an attack on a locked design to leak the secret key. *VIGILANT* detects vulnerabilities in a locked design against any fault-injection attacks independent of the employed mechanism.

D. Fundamentals of VLSI Testing

Stuck-at fault model. Such a model can consider any net in a design to be stuck-at logic value '0' (s-a-0) or logic value '1' (s-a-1).

Fault activation. It is also known as fault excitation, where a stuck-at-fault is activated by forcing the signal driving it to an opposite value from the stuck-at-fault value.

Fault propagation. It is also known as path sensitization, where the activated fault is propagated to one or multiple primary outputs (POs). To propagate a fault to some PO(s), the internal nets of the design are sensitized with required values.

Line justification. It describes a process where internal net assignments (that are required to sensitize a fault or to propagate its effect) are justified by setting the PIs of the design accordingly.

Test pattern. It describes an input pattern that justifies the internal nets to activate or propagate some faults to POs.

III. ADVERSARIAL MODEL

Although we propose and develop a vulnerability detection tool, one needs to understand the capabilities and goals of an adversary who intends to launch fault-injection attacks on locked designs.

A. Target and Resources

This work considers any IC protected with a combinational logic locking technique as a target. Such IC requires a secret key (to be stored in some tamper-proof memory); the key is the targeted-at security asset. We assume that the tamper-proof memory is resilient to fault-injection attacks, thereby requiring the adversary to insert faults into other parts of the design. We assume that an adversary can obtain multiple IC copies from the open market for two means: (i) to reverse-engineer the underlying locked design, which is required for modeling, functional simulation, etc., and (ii) as an oracle, which provides "unlocked," or correct, input-output patterns for verification of attack efforts, etc. Further, we assume that an adversary possesses the technical know-how and access to some fault-injection equipment of choice.

B. Adversarial Motivation

An adversary with physical access to working IC copies and some fault-injection equipment can leak the secret key. To understand why an adversary would take the related attack efforts, recall the importance of logic locking techniques (Sec. I-A).

In particular, note that adversaries having access to the secret key can pirate the design IP, causing losses of billions of dollars to IC design companies [5].

C. Adversarial Capabilities

We assume that an adversary can analyze the reverse-engineered locked design, query the oracle, and inject faults into the oracle using any of the fault-injection mechanisms outlined in Sec. II-C. Researchers in the logic locking community have, so far, developed attacks considering the oracle either as a black box [35] or as a target for simple fault-injection attacks that employ single faults at a time [19]. In this work, we proactively consider an advanced adversary capable of concurrently injecting multiple faults into the oracle. Such security considerations are highly relevant, as recently discussed in, *e.g.,* [28].

D. Defender Approach

VIGILANT detects vulnerabilities in the hardware implementation of a locked design, which can be exploited by an adversary capable of injecting physical faults into the oracle. *VIGILANT* aids the designer by providing pre-silicon insights on vulnerable nets and their associated fault values. Designers can use this information to safeguard their ICs at the pre-silicon stage proactively. Note that such security-aware design integration and tooling are beyond the scope of this paper.

TABLE II
COMMONLY USED ABBREVIATIONS AND NOTATIONS

Term	Definition	Term	Definition
POs	Primary outputs	G	A graph
PIs	Primary inputs	K	Secret key
PO	List of primary outputs	ATPG	Automatic test pattern generation
$\mathcal{C}_{orc}$	Oracle	$\mathcal{C}_{lock}$	Locked design
T	Set of number of faults across all key-inputs	$\|K\|$	Length of secret key
PSLL	Provably-Secure Logic Locking	L	Superset of vulnerable nets across all key-inputs
k_i	i^{th} key-input	L_i	Set of vulnerable nets for k_i
$mode$	Mode of VIGILANT	F_i	Set of fault values for all vulnerable nets of k_i
T_i	Number of faults for k_i	K_{rec}	Set of recovered key-inputs
F	Superset of fault values, for all vulnerable nets of all key-inputs	$\mathcal{P}$	Test pattern

IV. *VIGILANT*: VULNERABILITY DETECTION TOOL

VIGILANT is a vulnerability detection tool that aims to identify vulnerable nets in a locked design susceptible to key leakage via fault-injection attacks. In this section, we first provide the problem formulation, enlist the challenges faced while developing *VIGILANT*, and finally describe the detailed methodology using illustrative examples.

Problem Formulation. Given an oracle C_{orc} with an embedded secret key K of key-size $|K|$, with $\{k_0, ..., k_{|K|-1}\}$ as key-inputs and a reverse-engineered netlist of the locked design C_{lock}. The goal of *VIGILANT* is to find the superset of vulnerable nets *i.e.,* $\{L_0, ..., L_{|K|-1}\}$ where L_i is a set of vulnerable nets corresponding to key-input k_i. Furthermore, *VIGILANT* also reports a set $\{T_0, ..., T_{|K|-1}\}$ where T_i is the number of faults required to leak a given key-input k_i.
The vulnerable nets reported by *VIGILANT* must be valid, *i.e.,* aid a designer in analyzing the security of the locked design C_{lock} by enabling retrieval of the correct secret key K.

A. Definitions of Common Functions

Next, we describe the functions implemented in *VIGILANT* and used throughout this section.

1) `ATPG(`f`,`C_{lock}`,`n`)`: Returns a test pattern to detect stuck-at-f fault at net n in design C_{lock}. ATPG denotes automatic test pattern generation. In this work, w.l.o.g., we leverage the commercial ATPG tool *Synopsys Tetramax* to compute test patterns that facilitate the propagation of a given key-input to the design's POs.
2) `create_graph(`C_{lock}`)`: Returns the graph representation of the given locked design C_{lock}. Logic gates, PIs, key-inputs, and POs are represented using vertices. Nets/wires are represented using edges.
3) `find_paths(`G`,`v_i`,`v_j`)`: Returns all paths between two vertices v_i and v_j in a graph G.
4) `find_fault_values(`L_i`,`$\mathcal{P}$`)`: Returns the set of fault values F_i to be inserted at vulnerable nets in L_i to validate the leakage of considered key-input. It is calculated by the test pattern $\mathcal{P}$ returned by ATPG.
5) `find_shortest_path(`G`,`v_i`,`v_j`)`: Returns the shortest path between two vertices v_i, v_j in a graph G. This function implements Dijkstra's algorithm.

Algorithm 1: *VIGILANT*

Input: Oracle (C_{orc}); Locked design (C_{lock}); Key-inputs (K); Primary outputs (PO); Mode ($mode$);
Output: Vulnerable nets (L), No. of faults (T), Fault values (F)

Function `Find_vulnerable_nets(`G, $path$, k_i`)`:
  $prev\_vertex \leftarrow k_i$
  for $v \in path$ **do**
    $inn \leftarrow$ `in_edges(`G, v`)`
    for $u \in inn$ **do**
      if $u \neq prev\_vertex$ **then**
        L_i`.append(`u`)`
    $prev\_vertex \leftarrow v$
  return L_i

Function $VIGILANT_{basic}$`(` k_i, K`)`:
  $L_i \leftarrow K$`.remove(`k_i`)`
  return L_i

Function $VIGILANT_{sweep}$`(`G,k_i,PO`)`:
  for $po \in PO$ **do**
    $all\_paths_{po} \leftarrow$`find_paths(`$G, k_i, po$`)`
    for $path \in all\_paths_{po}$ **do**
      $L_{po} \leftarrow$`Find_vulnerable_nets(`$G, path, k_i$`)`
      L_i`.append(`L_{po}`)`
  return L_i

Function $VIGILANT_{int}$`(`G,k_i,PO`)`:
  $len\_shortpath \leftarrow inf$
  for $po \in PO$ **do**
    $path_{po} \leftarrow$`find_shortest_path(`G, k_i, po`)`
    if $len(path_{po}) < len\_shortpath$ **then**
      $path_{int} \leftarrow path_{po}$
  $L_i \leftarrow$`Find_vulnerable_nets(`$G, path_{int}, k_i$`)`
  $T_i \leftarrow len(L_i)$
  return L_i,T_i

Function $VIGILANT_{opt}$`(`G,k_i,PO,K_{rec},K`)`:
  $L_i, T_i \leftarrow VIGILANT_{int}(G,k_i,PO)$
  for $l \in L_i$ **do**
    $flag = 0$
    for $k_j \in K$ **do**
      if $k_j \notin K_{rec}$ & $path\_exists(G, l, k_j)$ **then**
        $flag = 1$
        break
    if $flag == 0$ **then**
      L_i`.remove(`l`)`
      $T_i \leftarrow T_i - 1$
  K_{rec}`.append(`k_i`)`
  return L_i,T_i, K_{rec}

$G \leftarrow$`create_graph(`C_{lock}`)`
$K_{rec}, L, T, F \leftarrow NULL$
$f \leftarrow 0$
for $i \leftarrow 0$ ***to*** $|\mathrm{K}| - 1$ **do**
  $k_i \leftarrow K[i]$
  $\mathcal{P} \leftarrow$`ATPG(`f,C_{lock},k_i`)`
  if $mode == basic$ **then**
    $L_i \leftarrow VIGILANT_{basic}(k_i, K)$
    $F_i \leftarrow$`find_fault_values(`$L_i, \mathcal{P}$`)`
  if $mode == sweep$ **then**
    $L_i \leftarrow VIGILANT_{sweep}(G, k_i, PO)$
    $F_i \leftarrow$`find_fault_values(`$L_i, \mathcal{P}$`)`
  if $mode == int$ **then**
    $L_i, T_i \leftarrow VIGILANT_{int}(G, k_i, PO)$
    $F_i \leftarrow$`find_fault_values(`$L_i, \mathcal{P}$`)`
  if $mode == opt$ **then**
    $L_i, T_i, K_{rec} \leftarrow VIGILANT_{opt}(G, k_i, PO, K_{rec}, K)$
    $F_i \leftarrow$`find_fault_values(`$L_i, \mathcal{P}$`)`
  L`.append(`L_i`)`
  T`.append(`T_i`)`
  F`.append(`F_i`)`
return L, T, F

6) `path_exists(`G,v_i,v_j`)`: Returns 1 if there exists a path between v_i and v_j. Else, it returns 0.
7) `len(`L_i`)`: Returns length of set L_i.

8) L.append(e): Appends a given set/element e to the superset/set L.
9) T.remove(e): Removes a given element e from the set T.
10) in_degree(G, v_i): Returns the number of incident edges to the vertex v_i in graph G.
11) in_edges(G, v_i): Returns the set of incident edges to the vertex v_i in graph G.

B. Challenges in Developing VIGILANT

Before we describe our methodology, we outline the challenges faced while developing *VIGILANT*.

C1 *Identification of security-critical nets*: Any locked design consists of a large number of nets. But, only a few nets are vulnerable to fault-injection attacks launched to leak security-critical information such as the secret key. How can a designer identify vulnerable nets in the design which can propagate the secret key to observable points (*i.e.,* POs in a locked design) in the design?

C2 *Assignment of fault values*: A selected key-input propagates to an observable point once the detected vulnerable nets are injected with specific fault values. A designer has to find these fault values to validate the leakage of the key-input. How can a designer obtain the logic values required for analyzing fault-injection on the identified vulnerable nets?

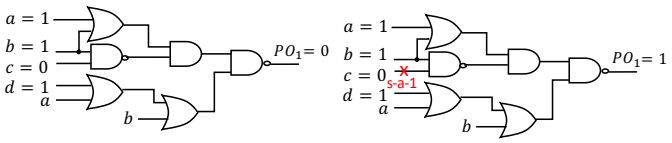
C3 *Identification of internal nets vulnerable to fault injection:* Consider a designer proactively protecting all the key-inputs against fault-injection attacks. Still, this scenario does not restrict an advanced adversary that uses multiple probes to inject faults on internal nets from attacking the locked designs. How can *VIGILANT* serve to identify vulnerable internal nets?

C4 *Identification of vulnerable nets in cyclic designs:* Consider a designer introducing cycles or combinational loops to a design to thwart simple fault-injection attacks. Still, an adversary may leak the key by inserting faults on internal nets. How can *VIGILANT* serve to identify vulnerable nets also in cyclic designs?

C5 *Finding the reduced number of faults required to leak each key-input:* An adversary with fewer resources might follow a more strategic approach to decrease the number of probes required for leaking the key. Thus, a reduced number of faults is required for a designer to understand the strength of locked design against such fault-injection attacks with fewer resources. How can *VIGILANT* serve to reduce the number of faults, *i.e.,* vulnerable nets?

C. Concept

To address C1, we analyze the structure of the locked design C_{lock} using graph-based algorithms to identify vulnerable nets susceptible to fault-injection attacks for leaking the secret key. To address C2, we rely on testing principles to obtain the fault values at identified vulnerable nets. We use the stuck-at fault model to obtain a test pattern that can detect a given stuck-at fault at a key-input. Next, we illustrate an example to detect a stuck-at fault at a considered input (Fig. 2).



(a) Fault-free design (b) Design with stuck-at fault

Fig. 2. Example for fault activation and propagation of a s-a-1 fault at input c. (a) represents the design without any fault inserted. (b) is a design with stuck-at fault at input c. Here, the input pattern $\{a = 1, b = 1, c = 0, d = 1\}$ helps to detect the fault by fault activation (setting input c to the opposite value of the fault, *i.e.,* $c = 0$), fault propagation (sensitizing other inputs such that the fault is not masked before reaching output PO_1), and comparing the outputs of the fault-free and the faulty design.

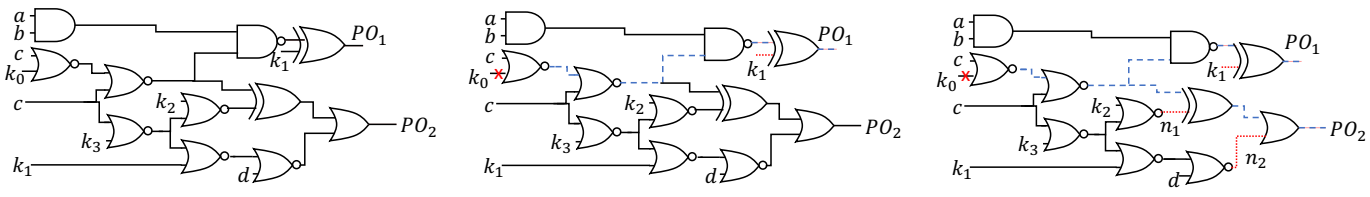
Example 1. Consider an input c in Fig. 2(a) to be checked if it is stuck at a constant logic value, say '1'. Then, a pattern $\mathcal{P}_c$ is generated to bring this input net to a logic value '0' and also to propagate some fault occurring at this input net (if any) to the output PO_1 as shown in Fig. 2(b). In *VIGILANT*, we are utilizing this stuck-at-fault concept as follows. We obtain test patterns that can detect the logic values at which key-inputs are stuck-at, given that they are statically fed ("stuck-at") through a tamper-proof memory. If the test patterns cover key-inputs, adversaries must insert physical fault at these inputs since they are not directly accessible (Sec. III). For other regular input ports, adversaries can apply patterns directly without inserting physical faults. The above-outlined approach forms the most naïve and basic one; we refer to this as the $VIGILANT_{basic}$ mode. We explain $VIGILANT_{basic}$ in more detail below (Sec. IV-D). To address C3, we propose more advanced modes of *VIGILANT*, which consider fault-injection attacks on internal nets, as follows.

With $VIGILANT_{sweep}$ (Sec. IV-E), we propose a mode that sweeps or searches the entire locked design and reports all the vulnerable nets susceptible to key leakage. The exhaustive approach is important for security analysis in general. However, it has practical limitations as follows. First, attacking all the vulnerable nets is typically not required for an adversary. Second, for a designer, protecting all the vulnerable nets can result in high overheads. Thus, we propose another mode, $VIGILANT_{int}$ (Sec. IV-F), which returns the minimal set of internal nets compared to $VIGILANT_{sweep}$ required to be faulted for leaking each key-input. Finally, $VIGILANT_{opt}$ (Sec. IV-G) iteratively reduces the number of faults by stepwise discarding nets connected to already deciphered key-inputs. Algorithm 1 describes all the modes of *VIGILANT*.

D. $VIGILANT_{basic}$

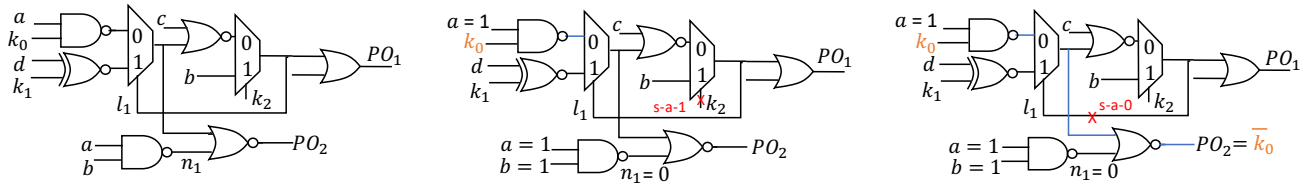
Invoking this mode returns a list of key-inputs and their corresponding fault values, along with the input test patterns, all required to leak out individual key-inputs. Next, we describe an example demonstrating the working of $VIGILANT_{basic}$ in more detail.

Example 2. Consider Consider Fig. 3(a) where a design is locked with a 4-bit key, encoded in k_0 to k_3. Further, consider that a designer wants to identify vulnerable nets in Fig. 3(a) that would facilitate leaking the key-input k_0. Toward that end, an ATPG tool is invoked to generate a test pattern $\mathcal{P}$



(a) Locked design (b) $VIGILANT_{basic}$ (c) $VIGILANT_{sweep}$

Fig. 3. Example for leaking key-input k_0. In (a), a locked design is shown. In (b) and (c), the workings of $VIGILANT_{basic}$ and $VIGILANT_{sweep}$ in detecting vulnerable nets to leak key-input k_0 are shown, respectively. Note that the red cross represents a stuck-at fault, nets highlighted in red represent vulnerable nets, and nets highlighted in blue represent the paths for key leakage (via fault propagation) for k_0.



(a) Locked design (b) $VIGILANT_{basic}$ (c) $VIGILANT_{int}$

Fig. 4. Example showcasing the challenge of leaking key-input k_0 in a cyclic design. In (a), a locked design with a combinational loop l_1 is shown. In (b) and (c), the working of $VIGILANT_{basic}$ and $VIGILANT_{int}$ modes are shown, respectively. For (b), $VIGILANT_{basic}$, we observe that k_0 cannot be leaked as we are considering here a restricted adversary capable of inserting faults only at key-inputs. For (c), $VIGILANT_{int}$, however, we can find vulnerable nets even in this cyclic design, *i.e.,* assuming an advanced adversary capable of inserting faults at internal nets.

which helps in detecting a s-a-0 fault at key-input k_0, as shown in Fig. 3(b). The ATPG tool returns an 8-bit pattern $\mathcal{P}$ that includes key-inputs and PIs. As indicated, adversaries can directly apply patterns to the PIs at the oracle but need to inject faults at key-inputs (given that key-inputs are connected to a tamper-proof memory that is statically loaded or "stuck at" with the secret key and are inaccessible to adversaries). Note that to leak a certain key-input, the faults would need to be injected at all other key-inputs in the key leakage path.

As shown in Fig. 3(b), injecting a fault at key-input k_1 is sufficient for propagating the key-input k_0 to the output port $P0_1$. Note that the path highlighted in blue is the propagation path of the fault. Comparing the related output response with the ATPG response for the generated test pattern results in key-input leakage.

A simple yet costly solution to thwart such an attack mode is applying fault-injection countermeasures at all key-inputs.

Another defense solution could be logic locking techniques introducing cycles or combinational loops to the design. For example, consider the locked design in Fig. 4(a) with some combinational loop l_1. There, key-input k_0 cannot be leaked using $VIGILANT_{basic}$ (Fig. 4(b)). More specifically, by applying an input $a = 1$, the key-input k_0 can be propagated to the input of the MUX, forming a combinational loop l_1 with its select line. Note that l_1 has to be brought to logic value '0' to propagate k_0. Since $VIGILANT_{basic}$ considers insertion of faults at key-inputs. Thus, a s-a-1 fault can be inserted at k_2, to pass the input b to l_1 such that l_1 is brought to logic value '0'. However, at the same time, input b must be at logic value '1' to propagate k_0 to PO_2. Thus, l_1 hinders fault propagation of k_0. Accordingly, when using $VIGILANT_{basic}$, a designer could conclude that such cyclic designs are resilient to fault-injection attacks. However, this does not hold true for more advanced modes, as discussed in the remainder of this section.

E. $VIGILANT_{sweep}$

Invoking this mode returns all the vulnerable nets in a locked design susceptible to fault injection-based key-leakage attacks. In this mode, *VIGILANT* first converts the locked design into a graph data structure. Next, using graph theory, $VIGILANT_{sweep}$ identifies all the paths between the key-inputs and PO ports. All the gates in these paths are tagged, and their non-propagating inputs are marked as vulnerable. Below, we describe an example comparing the $VIGILANT_{basic}$ and $VIGILANT_{sweep}$ modes.

Example 3. Consider Fig. 3(c), which shows a locked design. Nets marked in red are all the vulnerable nets responsible for key propagation via the two paths marked in blue. Recall that using $VIGILANT_{basic}$, we observed only one net k_1 to be vulnerable, as shown in Fig. 3(b). However, $VIGILANT_{sweep}$ identifies three vulnerable nets, including internal nets of all possible key-leakage paths. Next, we discuss the steps of this mode in detail.

Step 1 – *Convert design to graph.* We convert the locked design's gate-level netlist C_{lock} to a directed graph $G(V, E)$. The vertices V represent the gates, PIs, POs, and key-inputs, and the edges E represent the connections, *i.e.,* the internal nets or wires in the locked design. The function `create_graph` implements this step in Algorithm 1.

Step 2 – *Identify propagation paths.* There could be multiple paths between a key-input and all POs. We identify all the paths between each key-input k_i and all POs of the locked design using the depth-first search algorithm (DFS). These paths represent the potential propagation paths for leaking key-inputs. Next, we find all the possible vulnerable nets for all potential propagation paths where an adversary can insert faults to observe the key-input via POs. The function `find_paths` in Algorithm 1 implements DFS to find all the paths between the key-input and a given PO.

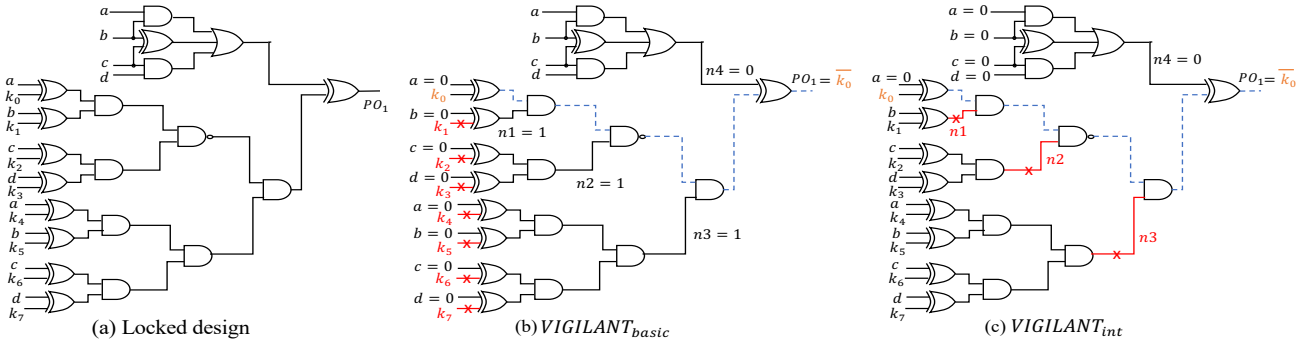


Fig. 5. Example for leaking key-input k_0 in a design locked using Anti-SAT [29], with key-size 8. The working of $VIGILANT_{basic}$ and $VIGILANT_{int}$ modes are shown in (b) and (c), respectively. As before, the red cross represents stuck-at fault, nets highlighted in red represent vulnerable nets, and nets highlighted in blue represent the key-leakage paths for k_0.

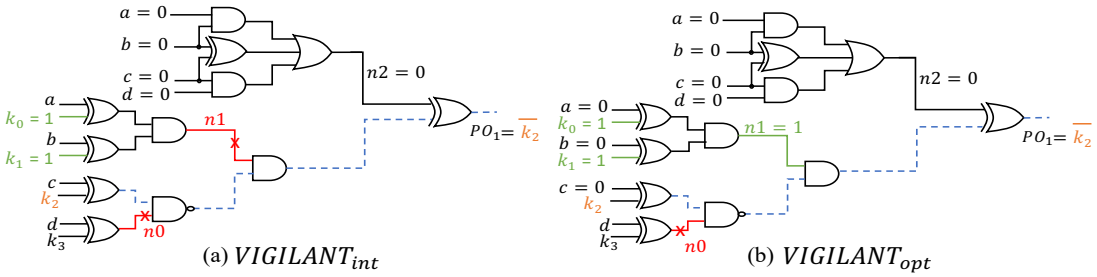


Fig. 6. Example for leaking key-input k_2 in a design locked using the Anti-SAT technique [29], with 4 bits key-length. In (a) and (b), the corresponding working of $VIGILANT_{int}$ and $VIGILANT_{opt}$ modes are shown respectively. Note that k_0 and k_1 are already recovered. Here, red cross represents a stuck-at-1 fault. As before, nets highlighted in red represent vulnerable nets, and nets highlighted in blue represent key-leakage paths for k_2. Further, nets highlighted in green correspond to recovered key-inputs.

Step 3 – *Identify vulnerable nets.* Here, we find the vulnerable nets in the propagation paths returned by Step 2. As indicated, these vulnerable nets are the locations in the locked design where an adversary would insert faults to leak the key-inputs. The function `Find_vulnerable_nets` in Algorithm 1 implements this step by processing a given path as follows.

1) Find the vertices and edges in the path.
2) Find the edges incident to all the vertices in the path.
3) Add the incident edge to the set of vulnerable nets if it is not in the given propagation path.

F. $VIGILANT_{int}$

Invoking this mode returns a minimum set of internal nets vulnerable to leaking the secret key. Here, the set of internal nets is minimal compared to the vulnerable nets returned by $VIGILANT_{sweep}$. $VIGILANT_{int}$ follows the working of $VIGILANT_{sweep}$, except for Step 2. The total number of faults required to leak a key-input is linearly dependent on the length of the propagation path and the in-degree of each vertex in the path as per the below equation.

$$\#faults = \sum_{i=1}^{N} (in\_degree(v_i) - 1) \tag{1}$$

Assuming the same in_degree for each vertex in the path, the shortest propagation path between the key-input and the PO will result in the minimum number of faults compared to the number of faults/vulnerable nets resulting in other propagation paths. We use Dijkstra's algorithm to find the shortest path, implemented as function `find_shortest_path` in Algorithm 1. *VIGILANT* then delegates the shortest path as input to the function `Find_vulnerable_nets`, similar to other modes, to obtain the list of vulnerable nets. Here, we describe Example 4 to differentiate $VIGILANT_{basic}$ and $VIGILANT_{int}$.

Example 4. Consider the design in Fig. 5(a), locked using the Anti-SAT [29] with an 8-bit key. Fig. 5(b) shows the working of $VIGILANT_{basic}$, which reports seven key-inputs as vulnerable to leaking key-input k_0. In contrast, $VIGILANT_{int}$ (Fig. 5(c)) reports fewer nets as vulnerable, *i.e.,* only three internal nets.

Furthermore, $VIGILANT_{int}$ addresses challenge $C4$, *i.e.,* *VIGILANT* can indeed serve to identify vulnerable nets in cyclic designs. Next, we describe a related example.

Example 5. Consider some cyclic locked design in Fig. 4(a). As discussed in Sec. IV-D, this design is considered resilient when using the $VIGILANT_{basic}$ mode. However, an advanced adversary capable of fault insertion for internal nets would still be able to leak the key-input k_0, namely by inserting s-a-0 fault at l_1, as shown in Fig. 4(c). Thus, $VIGILANT_{int}$ can also find vulnerabilities in more complex designs that hold some combinational loops.

G. $VIGILANT_{opt}$

To address C5, *i.e.,* to find the reduced number/set of faults required to leak each key-input, we propose $VIGILANT_{opt}$. This mode reduces the number of faults as returned by $VIGILANT_{int}$. On a high level, $VIGILANT_{opt}$ takes the

TABLE III
OVERVIEW OF *VIGILANT*

$VIGILANT_{basic}$	$VIGILANT_{sweep}$	$VIGILANT_{int}$	$VIGILANT_{opt}$	
Input	Locked Netlist	Locked Netlist	Locked Netlist	Locked Netlist, and output of $VIGILANT_{int}$
Search Space	Key-inputs	Internal nets	Internal nets	Internal nets
Supported designs	Acyclic	Acyclic and Cyclic	Acyclic and Cyclic	Acyclic and Cyclic
Output	Vulnerable key-inputs and related input patterns*	All vulnerable nets	Minimum set of faults to independently leak each key-input	Reduced set of faults to strategically leak all key-inputs

*Related input patterns are output for all *VIGILANT* modes, not only $VIGILANT_{basic}$.

vulnerable nets returned by $VIGILANT_{int}$ as input and eliminates the nets dependent on previously retrieved key-inputs. That is if a vulnerable net depends on some previously retrieved key-input, and the fault value to insert at the retrieved key-input matches the retrieved key-input value; *VIGILANT* can eliminate the considered net from the set of vulnerable nets. $VIGILANT_{opt}$ mode decreases the set of vulnerable nets, *i.e.,* the number of faults compared to $VIGILANT_{int}$. As indicated, such insight on reducing the number of faults helps the designer understand the locked design's security level, *i.e.,* the practical number of probes an adversary would require to leak the key-input in the best-case scenario for the adversary. Below, we describe an example that contrasts the working of $VIGILANT_{int}$ and $VIGILANT_{opt}$.

Example 6. Fig. 6(a) demonstrates the working of $VIGILANT_{int}$ mode on a design locked using Anti-SAT [29] with four key-inputs. Here, $VIGILANT_{int}$ reports two vulnerable nets, $n0$ and $n1$, to leak key-input k_2 assuming that k_0 and k_1 are already retrieved. Consider Fig. 6(b) that demonstrates the working of $VIGILANT_{opt}$ on the same design. Here, only one vulnerable net, $n0$, is returned as required to leak key-input k_2. This reduction is because the retrieved key-inputs $k0$ and $k1$ result in logic value '1' at net $n1$, which matches the fault value to be inserted at $n1$. In short, the $VIGILANT_{opt}$ mode observes the logic value at the vulnerable nets connected to previously retrieved key-inputs and eliminates them from the list of vulnerable nets if the logic value of the considered vulnerable net matches the fault value to be inserted.

H. Validation Module

VIGILANT reports the vulnerable nets susceptible to fault-injection attacks and their corresponding fault values as shown in Algorithm 1. It is essential to validate these vulnerable nets, *i.e.,* to check if the vulnerable nets facilitate the leakage of the secret key of the locked design. Toward that end, the locked design C_{lock} and oracle C_{orc} are given as inputs to the validation module of *VIGILANT* and processed there as follows. First, *VIGILANT* inserts the faults at the reported vulnerable nets with their corresponding fault values. Then, *VIGILANT* inserts a s-a-0 or s-a-1 fault at the key-input to be leaked. Then, both C_{lock} and oracle C_{orc} are simulated with the test pattern $\mathcal{P}$ returned by *VIGILANT*'s vulnerability detection mode of choice. If the outputs of C_{orc} and C_{lock} are equal, we can deduce that the fault inserted at the key-input is a correct key-input value. Else, the complement of the fault value is the correct key-input. Once the complete key is retrieved, we perform an equivalence check between C_{orc} and C_{lock} loaded with the full key retrieved. Assuming both are equivalent, the retrieved key is validated as correct, and the vulnerable nets returned by the tool are valid. The accuracy of *VIGILANT* in identifying and validating all candidate nets vulnerable to fault-injection attacks is 100%.

I. Summary

Table III provides an overview and comparison between different modes of *VIGILANT*. First, $VIGILANT_{basic}$ considers an adversary capable of inserting faults only at the key-inputs. A designer may protect all the key-inputs to thwart such basic attacks. However, an adversary with access to advanced fault-injection equipment could still succeed by inserting faults at internal nets. Thus, in the $VIGILANT_{sweep}$ mode, *VIGILANT* reports all the vulnerable internal nets in a design, and in the $VIGILANT_{int}$ mode, fault propagation paths that require only minimal faults compared to $VIGILANT_{sweep}$ are considered and reported. Finally, in the $VIGILANT_{opt}$ mode, *VIGILANT* considers an adversary following a strategic approach for fault injection. This mode returns the reduced set of faults inserted by an adversary to leak key-inputs iteratively.

V. EXPERIMENTAL EVALUATION

In this section, we first discuss the details of our experimental setup for developing *VIGILANT* and conducting various experiments. Next, we discuss the evaluation metrics for *VIGILANT* and then evaluate and analyze *VIGILANT* through the experimental results.

A. Experimental Setup

***VIGILANT* tool setup.** We perform experiments on a 128-core Intel Xeon processor running at 2.4 GHz with 512 GB of RAM. We use a commercial synthesis tool *Synopsys Design Compiler* to synthesize the designs. Furthermore, we use other commercial tools *Cadence Innovus* and *Synopsys ICC II* for the layout flow to generate post-layout designs. We perform the experiments on the benchmarks generated for an academic-based technology node, Nangate 45nm, and one foundry-compatible technology node, *i.e.,* TSMC 22nm. We have developed two versions of *VIGILANT*, one using academic tools

TABLE IV
COMPARISON IN THE TOTAL NUMBER OF FAULTS (AVERAGE OVER TEN TRIALS OF LOCKED BENCHMARKS) BETWEEN $VIGILANT_{basic}$ (BASIC) AND $VIGILANT_{opt}$ (OPT) APPROACH FOR 45NM TECHNOLOGY WITH KEY-SIZE OF 128 FOR ANTI-SAT AND CAS-LOCK TECHNIQUES AND 144 FOR FULL-LOCK AND LEBL TECHNIQUES. REDUCE. (×) DENOTES THE REDUCTION IN TIMES FROM $VIGILANT_{basic}$ TO $VIGILANT_{opt}$ FOR THE TOTAL NUMBER OF FAULTS.

Benchmark	Anti-SAT			CAS-Lock			Full-Lock			LEBL		
	Basic	**Opt.**	**Reduce. (×)**	**Basic**	**Opt.**	**Reduce. (×)**	**Basic**	**Opt.**	**Reduce. (×)**	**Basic**	**Opt.**	**Reduce. (×)**
b14_C	16,256	1,130	14.4×	16,256	3,864	4.2×	15,335	2,393	6.41	16,114	2,386	6.7×
b15_C	16,256	1,257	12.9×	16,256	4,018	4×	14,584	2,046	7.1x	16,544	2,677	6.2×
b17_C	16,256	1,266	12.8×	16,256	3764	4.3×	13,840	1,973	7×	15,989	2,640	6×
b18_C	16,256	1202	13.5×	16,256	3,681	4.4×	12,527	1,818	6.9×	N/A	N/A	N/A
b19_C	16,256	1142	14.2×	16,256	3,529	4.6×	10,569	1,913	5.5×	N/A	N/A	N/A
b20_C	16,256	1,279	12.7×	16,256	3,695	4.4×	15,036	1,916	7.8×	15,420	2,292	6.7×
b21_C	16,256	1,313	12.4×	16,256	3,557	4.6×	14,389	1,999	7.2×	16,668	2,627	6.3×
b22_C	16,256	1,273	12.8×	16,256	3,539	4.6×	14,518	1,883	7.7×	16,268	2,636	6.2×
Average	**16,256**	**1232**	**13.2×**	**16,256**	**3,706**	**4.4×**	**13,849**	**1,993**	**6.9×**	**16,167**	**2,543**	**6.4×**

N/A for the benchmarks that could not be generated with the computing resources available.

and the other using commercial or industry-compatible tools. The academic-based version is developed using academic tools such as ABC, Icarus Verilog, and Python3.7, which requires the input design files in BENCH format (academic-based file format). The industry-compatible version is developed using *Synopsys Design Compiler*, *Synopsys Tetramax*, and *Synopsys VCS* integrated with Python 3.7 and TCL scripts.

TABLE V
MINIMUM, MAXIMUM, AND AVERAGE NUMBER OF FAULTS ACROSS KEY-INPUTS FOR 10 TRIALS OF LOCKED BENCHMARKS WITH VARIED KEY SIZES FOR POST-LAYOUT BENCHMARKS OF 45NM TECHNOLOGY NODE FOR $VIGILANT_{opt}$ MODE

Bench-mark	Key-size	Anti-SAT			CAS-Lock			Full-Lock			LEBL		
		Min.	**Max.**	**Avg.**	**Min.**	**Max.**	**Avg.**	**Min.**	**Max.**	**Avg.**	**Min.**	**Max.**	**Avg.**
b14_C	128/144	9	16	12	3	56	31	3	24	14	3	32	17
	256/384	10	19	14	3	91	47	2	34	16	4	36	23
b15_C	128/144	10	16	13	3	61	31	2	27	14	4	36	19
	256/384	10	20	15	3	99	49	2	36	17	3	46	23
b17_C	128/144	10	17	14	3	56	29	2	27	14	3	35	19
	256/384	10	20	15	3	92	49	2	30	16	3	46	23
b18_C	128/144	8	10	9	3	50	26	1	24	13	N/A	N/A	N/A
	256/384	7	12	10	3	107	53	1	32	17	N/A	N/A	N/A
b19_C	128/144	8	11	10	3	52	28	3	26	15	N/A	N/A	N/A
	256/384	7	12	10	3	88	55	2	34	16	N/A	N/A	N/A
b20_C	128/144	9	18	13	2	55	29	2	36	13	4	34	18
	256/384	10	20	15	3	95	49	2	35	16	3	47	24
b21_C	128/144	8	18	13	3	54	28	2	27	14	4	34	19
	256/384	10	20	15	3	98	50	2	37	16	4	47	23
b22_C	128/144	10	18	13	3	54	25	2	29	13	4	37	19
	256/384	10	20	14	2	92	47	2	36	16	4	44	23

N/A for the benchmarks that could not be generated with the computing resources available.

Benchmarks. We demonstrate the effectiveness of *VIGILANT* across different locking techniques, *i.e.,* two PSLL techniques—Anti-SAT, CAS-Lock, and two SAT-hard techniques—Full-Lock, LEBL. We used the locking scripts or the benchmarks open-sourced by the authors of the respective publications unless they were unavailable. We obtained the locked benchmarks of Anti-SAT and CAS-Lock techniques from [45], [46]. We implemented the Full-Lock technique using *Python* 3.7. We used the locking script provided by the authors in [32] for LEBL. We evaluated the efficacy of *VIGILANT* on eight combinational ITC-99 benchmarks locked using the aforementioned locking techniques and generated different test cases with variations in (i) key-size, (ii) physical layout commands, and (iii) technology library. We generated ten trials of each locked benchmark to account for the randomness of the key-gate insertion for all locking techniques.

B. Evaluation Metrics for VIGILANT

VIGILANT analyzes the locked design structure to identify vulnerabilities against fault-injection attacks that aim to retrieve the secret key. In principle, the algorithm of *VIGILANT* depends on the underlying structure of the design. Therefore, we evaluate *VIGILANT* by performing experiments on locked benchmarks with varied structural representations. The structural representation of a locked design is governed by the technology library used to synthesize the design and the optimization commands used to generate a physical layout. The availability of logic gates in the technology library (of different technology nodes) directly transforms the structure of a design. Similarly, the settings like utilization in the layout flow also change the structure of the design. Through these experiments, we demonstrate the efficacy of *VIGILANT* with variations in key-size and structure across different benchmarks for all the locking techniques considered in this work. Next, we define metrics used to analyze the results.

Metric 1: *Minimum number of faults:* Minimum number of faults among the set of reduced number of faults reported in the $VIGILANT_{opt}$ mode for all key-inputs.

Metric 2: *Maximum number of faults:* The maximum number of faults among the set of the reduced number of faults reported by the tool's $VIGILANT_{opt}$ mode for all key-inputs.

Metric 3: *Total number of faults:* The sum of the number of faults reported by *VIGILANT* across all key-inputs.

C. $VIGILANT_{basic}$ versus $VIGILANT_{opt}$

Table IV tabulates the comparison of the total number of faults/vulnerable nets that facilitates the leakage of a complete secret key between $VIGILANT_{basic}$ and $VIGILANT_{opt}$ modes observed for ten trials of each benchmark for all the locking techniques considered in this work. There is an average of 13.2x, 4.4x, 6.9x, and 6.4x reduction in the total number of faults reported by $VIGILANT_{opt}$ mode compared

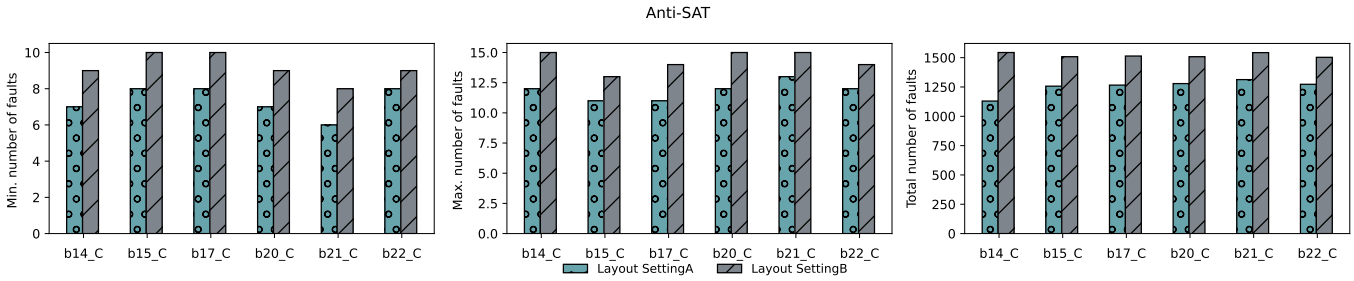


Fig. 7. Comparison of the minimum, maximum, and the total number of faults reported in $VIGILANT_{opt}$ mode (calculated over ten trials of each benchmark) between the benchmarks generated using two different layout settings (45nm) for key-size of 128 for Anti-SAT locking.

to $VIGILANT_{basic}$ across all benchmarks locked using Anti-SAT, CAS-Lock, Full-Lock, and LEBL respectively. The number of faults corresponding to $VIGILANT_{basic}$ corresponds to vulnerable key-inputs, whereas $VIGILANT_{opt}$ reports the number of faults corresponding to internal nets that may or may not include the key-inputs (depends on the structure of the benchmark). $VIGILANT_{opt}$ could detect additional vulnerable nets that $VIGILANT_{basic}$ does not report.

TABLE VI
AVERAGE (CALCULATED OVER 10 TRIALS) RUNTIME (IN MINUTES) OF $VIGILANT_{opt}$ MODE ACROSS DIFFERENT LOCKING TECHNIQUES FOR THE 45NM TECHNOLOGY NODE

Benchmark	Key-size	Anti-SAT	CAS-Lock	Full-Lock	LEBL
b14_C	128/144	6.86	7.79	4.08	6.86
	256/384	14.83	15.65	20.06	22.90
b15_C	128/144	11.57	13.94	7.81	8.81
	256/384	24.15	27.06	41.17	42.47
b17_C	128/144	33.65	35.78	16.7	17.82
	256/384	66.86	68.37	79.21	68.35
b18_C	128/144	89.75	81.61	44.02	N/A
	256/384	168.80	225.71	177.18	N/A
b19_C	128/144	195.49	233.83	103.13	N/A
	256/384	435.61	310.57	454.16	N/A
b20_C	128/144	13.53	15.25	7.41	7.86
	256/384	28.68	29.31	35.77	40.42
b21_C	128/144	13.56	15.38	7.07	8.61
	256/384	27.83	31.25	33.94	37.73
b22_C	128/144	20.08	22.67	10.58	11.34
	256/384	40.80	45.00	45.14	52.74
Average	**128/144**	**48.06**	**53.28**	**25.1**	**10.22**
	256/384	**100.94**	**94.11**	**110.83**	**44.10**

Runtime includes vulnerability detection and validation execution time. The key-sizes considered for Anti-SAT and CAS-Lock are 128 and 256. The key-sizes for Full-Lock and LEBL are 144 and 384. N/A for the benchmarks that could not be generated with the computing resources available.

D. Evaluation and Analysis of VIGILANT

Here, we outline the effect of different parameters on the performance of *VIGILANT*, specifically for the $VIGILANT_{opt}$ mode. The results of runtime, minimum, maximum, and total faults are the average values calculated for ten trials of each benchmark.

Effect of key-size. We generate locked benchmarks with different key-sizes, *i.e.,* 128 and 256 for PSLL techniques, and 144 and 384 for SAT-hard techniques, to observe the effect of key-size on the working of *VIGILANT*. The results of the minimum, maximum, and average number of faults across key-inputs reported by $VIGILANT_{opt}$ for post-layout benchmarks (for 45nm node) for all locking techniques are tabulated in Table V for varied key-sizes. The faults required to leak a secret key increase with the increase in key-size for all the benchmarks. This result indicates that the security of a locked design (for PSLL and SAT-hard locking techniques) against fault-injection attacks increases with the key-size.

Effect of layout commands. Fig. 7 represents the bar plot for comparison in the minimum, maximum, and total number of faults between two sets of post-layout Anti-SAT benchmarks generated with the difference in physical layout settings. Setting B benchmarks requires more faults than setting A. This result indicates that the structural changes in the design caused due to changes in layout flow settings affect the number of faults required to leak a secret key.

Effect of technology nodes. The number of faults required to leak a secret key depends on the technology node, as shown in Fig. 8, which demonstrates the bar plot of the minimum and the maximum number of faults for 45nm and 22nm technology-based Anti-SAT and CAS-Lock benchmarks. The structure of a design varies with technology. The minimum and the maximum number of faults vary with the technology node, *i.e.,* they have increased on an average in the 22nm TSMC technology node compared to the 45nm Nangate technology node. These results indicate that the number of faults depends on the design's structural changes that vary with the technology node due to different compositions and properties of logic gates.

Run-time. The runtime of $VIGILANT_{opt}$ for post-layout benchmarks generated using 45nm technology for Anti-SAT, CAS-Lock, Full-Lock, and LEBL is tabulated in Table VI. The benchmarks are locked with key-sizes of 128, 256 for PSLL techniques, and 144, 384 for SAT-hard techniques. The tabulated runtime includes the time taken by $VIGILANT_{opt}$ to detect and validate the vulnerable nets. The runtime of *VIGILANT* increases with the increase in key-size. We observe that $VIGILANT_{opt}$ takes an average of 48.06, 53.28, 25.10, and 10.22 minutes to detect and validate the vulnerabilities in ITC-99 benchmarks (with the number of gates varied between $7,166$ to $38,213$) locked using Anti-SAT, CAS-Lock, Full-Lock, and LEBL, respectively. Furthermore, we observe that the runtime for PSLL benchmarks is approximately $2\times$ for the key-size of 256 compared to a key-size of 128. The runtime of SAT-hard benchmarks is approximately $4\times$ for the key-size of 384 compared to a key-size of 144. Our results indicate that the runtime of *VIGILANT* increases marginally with both

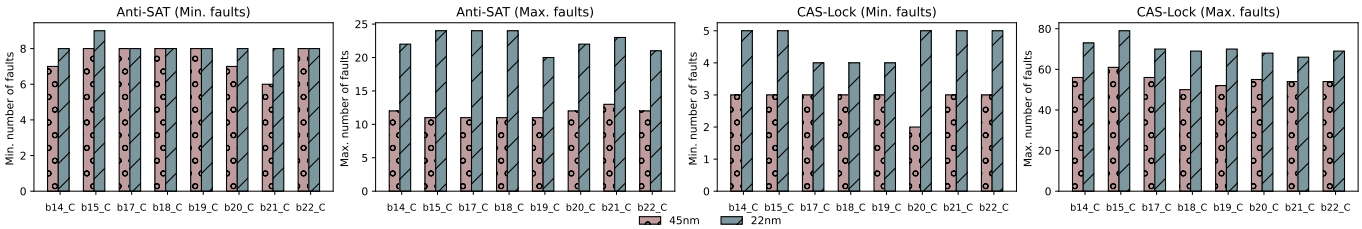


Fig. 8. Comparison of the average of the minimum and the maximum number of faults reported in $VIGILANT_{opt}$ mode (calculated over ten trials of each benchmark) between 45nm and 22nm technology nodes for Anti-SAT and CAS-Lock techniques with key-size of 128.

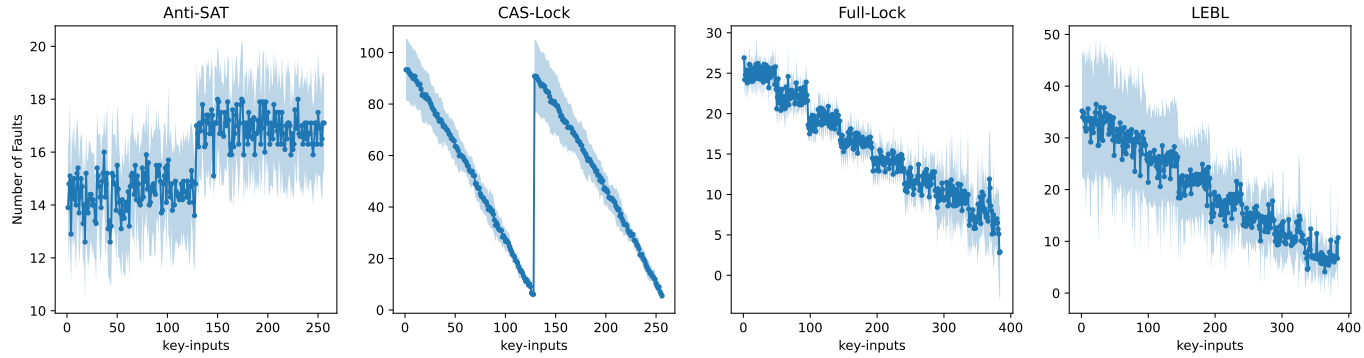


Fig. 9. The number of faults per each key-bit reported in $VIGILANT_{opt}$ mode across all locking techniques with a key-size of 256 for Anti-SAT and CAS-Lock and key-size of 384 for Full-Lock and LEBL generated using 45nm technology node. Note that the "key-inputs" reported for the x-axis refer to the key-input index.

key-size and the size of the benchmarks. Thus, *VIGILANT* is scalable to larger benchmarks and key-sizes.

E. Important Findings of VIGILANT

1) The number of vulnerable nets detected by *VIGILANT* (in $VIGILANT_{sweep}$, $VIGILANT_{int}$, and $VIGILANT_{opt}$ modes) depends on the structure of the design, which is consistent with existing notions of fault analysis like [25]. Our analysis observed that the number of vulnerable nets changes between the key-inputs of a design, as shown in Fig. 9 for both PSLL and SAT-hard locking techniques (for ten trials of the b17_C benchmark). We observe this variation since the number of vulnerable nets depends on the distance between key-input and PO. These distances depend on the index of key-inputs and the locking scheme, resulting in different structures in the locked netlist. We have observed that the trend of the line graph has not varied with the ITC-99 benchmark and key-size. Thus, the number of vulnerable nets concerning a key-input primarily depends on the structural changes introduced by the locking technique. Through this experiment, we observe that a designer can empirically strengthen the security of their locked design against fault-injection attacks by increasing the distance of the key-input or key-gate from the PO.
2) We observed that combinational loops do not hinder fault-injection attacks, especially considering an adversary capable of inserting faults on internal nets. *VIGILANT* detects vulnerabilities even in the designs with combinational loops introduced by SAT-hard locking techniques (Full-Lock and LEBL).
3) We observed that a few key-inputs are undetectable at the PO in Full-Lock and LEBL benchmarks. Fault-injection attacks cannot leak such undetectable key-inputs. In principle, the key-inputs in locked designs corrupt the POs. In contrast, an undetectable key-input would not corrupt the PO and is redundant. Thus, through *VIGILANT*, we can provide a list of redundant/undetectable key-inputs and help the designer improve the security of a locked design.

VI. Discussion and Future Work

Our experimental analysis of *VIGILANT* that the considered locked designs are vulnerable to fault-injection attacks. Therefore, designers must check for vulnerabilities in their locked designs during pre-silicon. In addition, there is a requirement for countermeasures against fault-injection attacks.

A. Why Pre-silicon Vulnerability Detection Tool?

Nowadays, sophisticated tools performing fault-injection attacks are readily accessible, and accordingly, adversaries are demonstrating successful fault-injection attacks on ICs. A designer cannot enhance the security of a design by adding a patch after the fabrication of ICs. Thus, a design has to be designed with the knowledge of fault-injection attack strategies from the start, *i.e.,* the design stage. Recall that the majority of fault-injection attacks are launched on a fabricated chip. It is financially prohibitive if designers have to reiterate the design flow if a design is found vulnerable after fabrication [23]. Thus, there is a requirement for a pre-silicon vulnerability detection tool against fault-injection attacks.

Different IPs of a design that consists of important information of a chip, such as key in the case of crypto-cores and locked chips, have to be tested using pre-silicon vulnerability detection tools. In this work, we take the first

step in developing a pre-silicon vulnerability detection tool against fault-injection by focusing on locked designs. We also consider different types of attackers with access to advanced equipment or having few resources for fault-injection.

B. Possible Countermeasures

VIGILANT detects vulnerabilities against fault-injection attacks on locked chips. Although *VIGILANT* guides in choosing the type of locking technique or the key-size, the designer would need further guidance in enhancing the security of the locked design against fault-injection. Through our experiments, we observed that the structure of the design plays a significant role in resilience against fault-injection attacks. Now arises a natural question, *i.e.,* can we use the findings of *VIGILANT* to improve the security of vulnerable locked designs? It is indeed possible to extend our work to guide the designer and use findings from *VIGILANT* to design countermeasures against fault-injection attacks. We can implement possible countermeasures against fault-injection attacks at the design and/or layout level. At the design level, we can introduce some redundant logic (*e.g.,* [47]) to the locked designs such that the number of faults required to leak the secret key increases. Another possible countermeasure could be to add detection circuitry [48], *i.e.,* sensors that can detect the fault-injection attacks and clear the key from the key-registers or tamper-proof memory. For instance, researchers have used, *e.g.,* ring oscillators in the prior art to detect fault-injection. However, such circuitry can be identified (due to its unique structure) and removed/disabled by an adversary before launching fault-injection attacks. The resulting challenge is to design and implement sensor circuitry with a non-unique structure that can be merged with the structure of the locked design post-synthesis.

VII. Conclusion

In this work, we proposed a vulnerability detection tool, *VIGILANT*, that proactively identifies and reports the vulnerabilities in the hardware implementation of locked designs in the pre-silicon (post-layout) stage. *VIGILANT* analyzes the underlying structure of the locked design using IC testing and graph theory-based principles and reports the vulnerable nets that facilitate the leakage of the secret key. Furthermore, we proposed a validation technique as a part of the overall *VIGILANT* framework to analytically validate the detected vulnerable nets. Further in-field validation using actual fault injection is the scope for future work.[3]

We verify the effectiveness of *VIGILANT* by detecting vulnerabilities in two PSLL and two SAT-hard locking techniques, including one unbroken technique. *VIGILANT* successfully identifies vulnerable nets in all the considered locking techniques with 100% validation accuracy. Through our experiments, we observed that the structure of the underlying design plays a significant role in the number of faults required to leak a secret key. Additionally, *VIGILANT* successfully reports vulnerabilities in locked designs with variations in (i) key-sizes, (ii) physical layout commands, and (iii) technology nodes. Our experimental analysis indicates that *VIGILANT* is agnostic to the aforementioned parameters.

In conclusion, our work indicates the importance of evaluating the security of logic locking techniques against fault-injection attacks during pre-silicon. In addition, our work also states the importance of developing suitable countermeasures against fault-injection attacks to build a robust, holistic logic locking technique that thwarts fault-injection attacks.

[3]In general, post-layout design files and post-silicon ICs exhibit the same graph structures. Thus, we have performed all the experimental analyses on post-layout designs and argue that our findings would also hold true in the field. Such validation is out of scope for this work, however, mainly as it requires an actual laser fault-injection attack setup, which we do not have access to at the time of writing the paper.

References

[1] "7nm vs 10nm vs 14nm: Fabrication process." [Online]. Available: https://www.techcenturion.com/7nm-10nm-14nm-fabrication

[2] D. Wu, "TSMC, Samsung Urge U.S. to Allow Them Into $52 Billion Chip Plan," March 28th, 2022, [Online; accessed 17-May-2022]. [Online]. Available: https://www.bloomberg.com/news/articles/2022-03-28/tsmc-samsung-urge-u-s-to-allow-them-into-52-billion-chip-plan

[3] J. S. Hurtarte, E. A. Wolsheimer, and L. M. Tafoya, "Understanding fabless ic technology." Burlington: Newnes, 2007, pp. 3–23. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780750679442500023

[4] M. Rostami, F. Koushanfar, and R. Karri, "A primer on hardware security: Models, methods, and metrics," *Proc. of the IEEE*, vol. 102, no. 8, pp. 1283–1295, 2014.

[5] J. Blyler, "The theft of semiconductor intellectual property (IP) is nothing new. Is it too late to act?" October 19, 2021. [Online]. Available: https://www.designnews.com/electronics/world-awakens-danger-ip-theft-again-whats-changed-time

[6] J. A. Roy, F. Koushanfar, and I. L. Markov, "Epic: Ending piracy of integrated circuits," in *Des. Autom. Test Europe*, 2008, pp. 1069–1074.

[7] N. Limaye and O. Sinanoglu, "RESCUE: Resilient, Scalable, High-corruption, Compact-Key-Set Locking Framework," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[8] "A Cross-layer framework for cost-effective Intellectual Property (IP) Protection ," February, 2021. [Online]. Available: https://apps.dtic.mil/sti/pdfs/AD1122333.pdf7

[9] S. Leef, "AISS: Automatic implementation of secure silicon," 2019. [Online]. Available: https://www.darpa.mil/attachments/AISS_ProposersDay_LeefSlides.pdf

[10] A. Cron, "EDA Forms the Basis for Designing Secure Systems," https://www.synopsys.com/implementation-and-signoff/resources/blogs/looking-past-horizon/designing-secure-systems.html, 2016, [Online; accessed 15-April-2022].

[11] S. Leef, "In Pursuit of Secure Silicon," 2017. [Online]. Available: https://www.ndia.org/-/media/sites/ndia/meetings-and-events/divisions/trusted-micro-electronics--joint-working-group/787a--feb-2017/3-leef-pursuitofsecuresiliconspeechv1.ashx

[12] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *International workshop on Cryptograph. Hardw. and Embedded Syst.* Springer, 2001, pp. 251–261.

[13] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Annual International Cryptology Conf.* Springer, 1999, pp. 388–397.

[14] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Annual International Cryptology Conf.* Springer, 1996, pp. 104–113.

[15] C. Shepherd, K. Markantonakis, N. van Heijningen, D. Aboulkassimi, C. Gaine, T. Heckmann *et al.*, "Physical fault injection and side-channel attacks on mobile devices: A comprehensive analysis," *Comput. & Sec.*, vol. 111, p. 102471, 2021.

[16] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proc. IEEE*, vol. 94, no. 2, pp. 370–382, 2006.

[17] P. Dusart, G. Letourneux, and O. Vivolo, "Differential fault analysis on aes," in *International Conf. on Applied Cryptograph. and Netw. Secur.* Springer, 2003, pp. 293–306.

[18] S. Tajik, H. Lohrke, F. Ganji, J.-P. Seifert, and C. Boit, "Laser fault attack on physically unclonable functions," in *Proc. Worksh. Fault Diag. Tol. Cryptograph.*, 2015, pp. 85–96.

[19] A. Jain, M. T. Rahman, and U. Guin, "ATPG-guided fault injection attacks on logic locking," in *IEEE Physical Assurance and Inspection of Electronics*, 2020, pp. 1–6.
[20] S. Tajik, H. Lohrke, J.-P. Seifert, and C. Boit, "On the power of optical contactless probing: Attacking bitstream encryption of FPGAs," in *Proc. Comp. Comm. Sec.*, 2017, pp. 1661–1674.
[21] 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," in *IEEE Proc. Int. Symp. Hardw.-Orient. Sec. Trust*, 2020, pp. 262–272.
[22] N. Asadizanjani, M. T. Rahman, and M. Tehranipoor, *Electrical Probing Attacks*. Springer International Publishing, 2021, pp. 101–132.
[23] M. Witteman, "The price we pay for faults," 2020. [Online]. Available: https://www.gsaglobal.org/forums/the-price-we-pay-for-faults/
[24] B. Amornpaisannon, A. Diavastos, L.-S. Peh, and T. E. Carlson, "Laser attack benchmark suite," in *IEEE/ACM Int. Conf. Comp.-Aided Des.*, 2020, pp. 1–9.
[25] H. Wang, H. Li, F. Rahman, M. M. Tehranipoor, and F. Farahmandi, "Sofi: Security property-driven vulnerability assessments of ics against fault-injection attacks," *IEEE Trans. Comp.-Aided Des. Integ. Circ. Sys.*, vol. 41, no. 3, pp. 452–465, 2022.
[26] "Inspector-fi." [Online]. Available: https://www.riscure.com/security-tools/inspector-fi
[27] C. E. Shannon, "Communication theory of secrecy systems," *The Bell Syst. Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949.
[28] T. Krachenfels, F. Ganji, A. Moradi, S. Tajik, and J.-P. Seifert, "Real-world snapshots vs. theory: Questioning the t-probing security model," in *IEEE Symp. on Sec. and Priv.* IEEE, may 2021.
[29] Y. Xie and A. Srivastava, "Anti-sat: Mitigating sat attack on logic locking," *IEEE Trans. Comp.-Aided Des. Integ. Circ. Sys.*, vol. 38, no. 2, pp. 199–207, 2019.
[30] B. Shakya, X. Xu, M. Tehranipoor, and D. Forte, "Cas-lock: A security-corruptibility trade-off resilient logic locking scheme," *IACR Trans. Cryptograph. Hardw. Embedded Syst.,*, pp. 175–202, 2020.
[31] H. M. Kamali, K. Z. Azar, H. Homayoun, and A. Sasan, "Full-lock: Hard distributions of sat instances for obfuscating circuits using fully configurable logic and routing blocks," in *Proc. Des. Autom. Conf.*, 2019, pp. 1–6.
[32] J. Sweeney, M. J. H. Heule, and L. Pileggi, "Modeling techniques for logic locking," in *Proc. Int. Conf. Comp.-Aided Des.*, 2020, pp. 1–9.
[33] J. Rajendran, H. Zhang, C. Zhang, G. S. Rose, Y. Pino, O. Sinanoglu *et al.*, "Fault analysis-based logic encryption," *IEEE Trans. on Computers*, vol. 64, no. 2, pp. 410–424, 2015.
[34] M. Yasin, J. J. Rajendran, O. Sinanoglu, and R. Karri, "On improving the security of logic locking," *IEEE Trans. Comp.-Aided Des. Integ. Circ. Sys.*, vol. 35, no. 9, pp. 1411–1424, 2016.
[35] P. Subramanyan, S. Ray, and S. Malik, "Evaluating the security of logic encryption algorithms," in *Proc. Int. Symp. Hardw.-Orient. Sec. Trust*, 2015, pp. 137–143.
[36] Z. Han, M. Yasin, and J. Rajendran, "Does logic locking work with EDA tools?" in *USENIX Security*, 2021.
[37] N. Limaye, S. Patnaik, and O. Sinanoglu, "Valkyrie: Vulnerability assessment tool and attack for provably-secure logic locking techniques," *IEEE Trans. Inf. Forensics Secur.*, vol. 17, pp. 744–759, 2022.
[38] A. Sengupta, M. Nabeel, N. Limaye, M. Ashraf, and O. Sinanoglu, "Truly stripping functionality for logic locking: A fault-based perspective," *IEEE Trans. Comp.-Aided Des. Integ. Circ. Sys.*, vol. 39, no. 12, pp. 4439–4452, 2020.
[39] A. Saha, S. Saha, S. Chowdhury, D. Mukhopadhyay, and B. B. Bhattacharya, "Lopher: Sat-hardened logic embedding on block ciphers," in *IEEE Design Autom. Conf.*, 2020, pp. 1–6.
[40] H. M. Kamali, K. Z. Azar, H. Homayoun, and A. Sasan, "Interlock: An intercorrelated logic and routing locking," in *Proc. Int. Conf. Comp.-Aided Des.* IEEE, 2020, pp. 1–9.
[41] S. Patnaik, N. Limaye, and O. Sinanoglu, "Hide and seek: Seeking the (un)-hidden key in provably-secure logic locking techniques," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 3290–3305, 2022.
[42] J. Richter-Brockmann, P. Sasdrich, and T. Güneysu, "Revisiting fault adversary models - hardware faults in theory and practice," Cryptology ePrint Archive, Paper 2021/296, 2021.
[43] M. Ghodrati, B. Yuce, S. Gujar, C. Deshpande, L. Nazhandali, and P. Schaumont, "Inducing local timing fault through em injection," in *IEEE Proc. Des. Autom. Conf.*, 2018, pp. 1–6.
[44] J. Breier, X. Hou, D. Jap, L. Ma, S. Bhasin, and Y. Liu, "Practical fault attack on deep neural networks," in *Proc. Comp. Comm. Sec.*, 2018, pp. 2204–2206.
[45] K. Shamsi, "Attack tool and benchmarks," 2020. [Online]. Available: https://bitbucket.org/kavehshm/neos/src/master/
[46] "Breaking caslock." [Online]. Available: https://github.com/DfX-NYUAD/Breaking_CAS-Lock
[47] T. G. Malkin, F.-X. Standaert, and M. Yung, "A comparative cost/security analysis of fault attack countermeasures," in *Proc. Fault Diag. Tol. Cryptogr.*, ser. FDTC'06. Berlin, Heidelberg: Springer-Verlag, 2006, p. 159–172.
[48] N. Homma, Y. ichi Hayashi, N. Miura, D. Fujimoto, D. Tanaka, M. Nagata *et al.*, "Em attack is non-invasive? - design methodology and validity verification of em attack sensor," in *CHES*, 2014.

Likhitha Mankali is a Ph.D. candidate at the Department of Electrical and Computer Engineering at Tandon School of Engineering, New York University, NY, USA. She is also a Global Ph.D. fellow with New York University Abu Dhabi, UAE. Her research interests include Hardware Security and using Machine learning and fault injection techniques to enhance and quantify the security of IP protection techniques. She has won third place in CSAW-Logic Locking Conquest 2021 and HeLLO CTF competition 2022.

Satwik Patnaik received his Ph.D. degree in Electrical engineering from Tandon School of Engineering, New York University, Brooklyn, NY, USA in September 2020. He is currently a Postdoctoral researcher with the Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX, USA. His research delves into IP protection techniques, CAD frameworks for incorporating security, leveraging 3D paradigm for security, exploiting security properties of emerging devices, and applied machine learning for hardware security.

Nimisha Limaye received her Ph.D. degree in electrical engineering from the Tandon School of Engineering, New York University, New York, NY, USA, in May 2022. She is currently working as a Senior ASIC Digital Design Engineer with the Solutions Group, Synopsys, Inc. Her research interests include hardware security, specifically, logic locking, scan locking, and design and architecture of side-channel and fault-injection countermeasures for public-key and post-quantum cryptography algorithms.

Johann Knechtel received the M.Sc. degree in Information Systems Engineering (Dipl.-Ing.) and the Ph.D. degree in Computer Engineering (Dr.-Ing., summa cum laude) from TU Dresden, Germany, in 2014. He is a Research Scientist with New York University Abu Dhabi, United Arab Emirates. His research interests cover VLSI physical design automation, with particular focus on emerging technologies and hardware security. He has (co-)authored around 50 publications.

Ozgur Sinanoglu is a professor of electrical and computer engineering at New York University Abu Dhabi. He obtained his Ph.D. in Computer Science and Engineering from University of California San Diego. Prof. Sinanoglu's research interests include design-for-test, design-for-security and design-for-trust for VLSI circuits, where he has more than 200 conference and journal papers, and 20 issued and pending US Patents. His recent research in hardware security and trust is being funded by US National Science Foundation, US Department of Defense, Semiconductor Research Corporation, Intel Corp, and Mubadala Technology.