The definitive version of this article has been accepted for publication in the IEEE Transactions on Information Forensics & Security (TIFS)

X-DFS: Explainable Artificial Intelligence Guided Design-for-Security Solution Space Exploration

Tanzim Mahfuz[1], Swarup Bhunia[2], and Prabuddha Chakraborty[1]

[1]Department of Electrical & Computer Engineering, University of Maine, Orono, ME, USA
[2]Department of Electrical & Computer Engineering, University of Florida, Gainesville, FL, USA

***Abstract*—Design and manufacturing of integrated circuits predominantly use a globally distributed semiconductor supply chain involving diverse entities. The modern semiconductor supply chain has been designed to boost production efficiency, but is filled with major security concerns such as malicious modifications (hardware Trojans), reverse engineering (RE), and cloning. While being deployed, digital systems are also subject to a plethora of threats such as power, timing, and electromagnetic (EM) side channel attacks. Many Design-for-Security (DFS) solutions have been proposed to deal with these vulnerabilities, and such solutions (DFS) relays on strategic modifications (e.g., logic locking, side channel resilient masking, and dummy logic insertion) of the digital designs for ensuring a higher level of security. However, most of these DFS strategies lack robust formalism, are often not human-understandable, and require an extensive amount of human expert effort during their development/use. All of these factors make it difficult to keep up with the ever growing number of microelectronic vulnerabilities. In this work, we propose X-DFS, an explainable Artificial Intelligence (AI) guided DFS solution-space exploration approach that can dramatically cut down the mitigation strategy development/use time while enriching our understanding of the vulnerability by providing human-understandable decision rationale. We implement X-DFS and comprehensively evaluate it for reverse engineering threats (SAIL, SWEEP, and OMLA) and formalize a generalized mechanism for applying X-DFS to defend against other threats such as hardware Trojans, fault attacks, and side channel attacks for seamless future extensions.**

***Index Terms*—Design-for-security, explainable artificial intelligence, hardware security, reverse engineering, logic locking.**

I. Introduction

A horizontal and distributed supply chain is at the heart of the booming semiconductor industry (see Fig. 1). Creation of digital hardware such as integrated circuits (IC) typically involves the development of sub-designs (intellectual properties - IP) by small entities, the integration of 3rd party sub-designs with the in-house components at the main design house, IC layout creation, fabrication at a foundry, testing at a testing facility, and assembly by the original electronic manufacturer (OEM). This model reduces the time to market for digital systems, allows for specialization, and enables small businesses. All these steps are typically carried out by different entities in different geographical locations. Such a flow can lead to a series of problems, such as: (1) design or sub-design theft, (2) hardware cloning, and (3) malicious hardware modification/tampering. Microelectronic ICs and devices in the field also face diverse threats, such as power side channel attacks and reverse engineering. A variety of Design-for-Security (DFS) strategies such as IC metering [1], [2], watermarking [3], camouflaging [4], [5], split manufacturing [6], [7], logic locking [8], [9], gate parameter optimization [10], variable delay module insertion [11], and dummy logic insertion [12] have been developed to guarantee trust in the supply chain [13] and boost post-deployment IC/device security. However, security solutions often become obsolete with the emergence of novel attacks, while developing appropriate countermeasures requires extensive research effort, time, and expert resources.

To expedite the solution search process (against novel vulnerabilities) and to create a human-understandable knowledge base of the said vulnerability, we propose an automated framework, X-DFS (E**X**plainable - **D**esign **F**or **S**ecurity). X-DFS uses a heuristics-based search process to determine a large set of DFS candidate instances that might contribute towards the defense against a given vulnerability. These DFS candidates are then used to train an explainable AI model that is capable of: (1) Emitting DFS rules that can be used to efficiently mitigate the vulnerability; (2) Automatically apply these rules towards securing the design. Hence X-DFS not only secures the design, it can also help researchers obtain a deeper understanding of the vulnerability.

The proposed framework is highly generalized in nature and can be applied to a wide range of vulnerabilities (with minor tweaks), while most state-of-the-art DFS techniques are typically hand-crafted for mitigating a specific vulnerability or a small set of vulnerabilities. X-DFS can modify a design towards mitigating a given vulnerability and at the same time can generate human-understandable design transformation rules. Such capabilities do not exist in current state-of-the-art DFS techniques. X-DFS is highly flexible (parameterized) and extremely efficient in terms of computation cost, while most state-of-the-art DFS techniques are static in nature (not highly configurable) and fail to work for larger designs (inefficient).

We implement X-DFS as a highly parameterized robust framework and use it to perform a comprehensive effectiveness analysis for large-scale designs. We evaluate the X-DFS framework by testing it in the logic locking domain where X-DFS is used to automatically search for mitigation strategies (human-understandable) against three powerful logic locking attacks (SAIL [14], OMLA [15]), SWEEP [16]. X-DFS was able to learn how to defend against these attack models and at the same time extracted human-understandable rules that can be used by other logic locking frameworks (such as LeGO [17]) or human experts to carry out the locking process.

In particular, we make the following research contributions:

1) Formalize a general framework and the core mechanisms for automatic exploration of the design-for-security search space for countering novel attack vector(s).
2) Design a set of algorithms (for reverse engineering attacks) that leverages this knowledge regarding an attack vector(s) to build an X-DFS model that can modify a given target design to be resilient against the attacks.
3) Define a methodology to extract and understand the defense rules (human understandable) that are learned by the X-DFS models.
4) Implement the proposed algorithms as a highly parameterized and scalable tool.
5) Qualitatively and quantitatively verify the efficacy of the X-DFS framework/tool against different reverse engineering threats (SAIL, SWEEP, and OMLA).

II. Related Works & Motivations

Next, we provide the additional details about different DFS solutions and understand the motivations behind X-DFS.

A. Design for Security (DFS)

Design for security techniques are widely used to protect digital designs and ICs from diverse attack vectors such as hardware Trojans, reverse engineering, and side channel threats (see Fig. 2). These techniques typically involve the insertion of security elements in the design (like logic locking key gates) and strategic modification of existing logic. IC Metering techniques [1] modifies the original finite state machine towards creating a Boosted Finite State Machine (BFSM) that ensures that the IC is only in the power-up state when an unique ID is provided. Watermarking techniques often insert an watermark in unused configurable logic blocks (CLBs) outputs [3] to provide passive intellectual property theft protection. Techniques such as CamoPerturb [5] attempts to minimally perturb the design logic towards thwarting reverse engineering. Logic Locking (LL) involves inserting a set of additional gates into the design (connected to some key inputs) such that the modified design does not function correctly without the application of the right key values to these gates [18], [19]. This ensures functional security of the design (to stop reverse engineering), but guessing the correct key input is almost trivial by investigating the structure of the gates inserted. For example, the correct key input for an additional XOR gate inserted to lock a wire will always be 0. Hence, a set of structural changes are introduced by synthesizing the design to make key guessing difficult by observing the design structures (see Fig. 3). Karna [10] attempts to mitigate power side channel leakage by: (1) analyzing each $N \times N$ grid of the design; (2) obtaining the corresponding TVLA-scores; (3) reconfiguring the gates if a vulnerability is detected in the grid. Variable delay module insertion has also been proposed to provide defense against side channel analysis [11]. To prevent the insertion of hardware Trojans in a design, researchers have proposed the insertion of dummy logic in the design [12].

B. Background on Logic Locking

Several different types of logic locking techniques have been developed over the years. Combinational logic locking involves the insertion of combinational gates in the design towards corrupting the output if a wrong key value is applied [20], [21], [22], [23], [8]. Sequential locking techniques such as HARPOON [24] obfuscate the state space requiring a user to apply a sequence of correct key inputs towards unlocking the design. Trace logic locking (TLL) [25], Delay-based locking [26], Analog locking [27], and RTL locking techniques [28] have also been developed to address different requirements. The main focus of this work is combinational logic locking, and let us start by looking at its brief history.

1) First Generation of LL: The first set of techniques focused on inserting only XOR (key = 0) and XNOR (key = 1) gates randomly towards encrypting the digital design [18], [19]. However, this led to suboptimal output corruption and unpredictable security against attacks such as KSA [29] which leveraged several pitfalls in these locking schemes.

2) Heuristics Driven Locking: The next generation of locking schemes such as SLL [23], XOR/XNOR insertion based on fliprate [19], and logic cone based insertion (CS) [18] were designed to be more robust against vulnerabilities such as KSA [29]. However, these heuristics-driven techniques have been shown to be vulnerable to SAT-based attacks that could extract the key by pruning the key space [30], [31], [32].

3) Anti-SAT Locking Era: A plethora of logic-locking techniques were developed to defend against attacks like SAT (and later SMT). These techniques either attempted to increase the time it took for SAT to complete each iteration or simply increased the number of iterations that SAT required to perform before it is able to obtain the key. Prominent among these techniques are: Anti-SAT, Strong Anti-SAT, SFLL, and CAS-Lock [33], [34], [35], [36]. However, these techniques

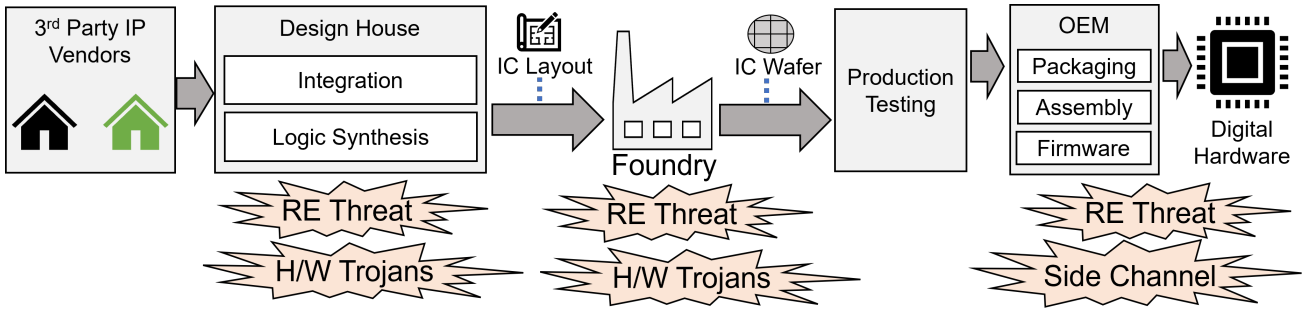


Fig. 1: Semiconductor supply chain security threats.

TABLE I: Notations and acronyms used in this article.

Notation	Description	Presented As	Denoted in
X-DFS	Explainable Design For Security	Framework	Entire Paper
XAI	Explainable AI	Concept	Entire Paper
design or D	Gate Level Digital Design	String Variable	Algorithm 1 and 2, Section III-A
A_t	Set of Attacks	Concept	Section III-A
C	Set of DFS Constructs	Concept	Section III-A
V	Vertices or Gates	Variable of Graph	Section III-A and Section III-G
E	Hyperedges or Wires	Variable of Graph	Section III-A and Section III-G
keySize or KL	Number of keys that needs to be inserted	Integer Variable	Algorithm 1 and 2
lockDict	Available logic gates to lock	String Variable	Algorithm 1 and 2
Th_{it}	Number of iterations to terminate Algorithm 1	Integer Variable	Algorithm 1
loc or lc	Size of locality	Integer Variable	Algorithm 1 and 2
Th_g	Threshold to measure goodness	Float Variable	Algorithm 2
A	Flag to turn on Ada X-DFS	Boolean Variable	Algorithm 2
U	Flag to lock only distinct wires	Boolean Variable	Algorithm 2
RN	Flag to turn on randomness in locking	Boolean Variable	Algorithm 2
M	Multiplier to choose number of good locking	Integer Variable	Algorithm 2

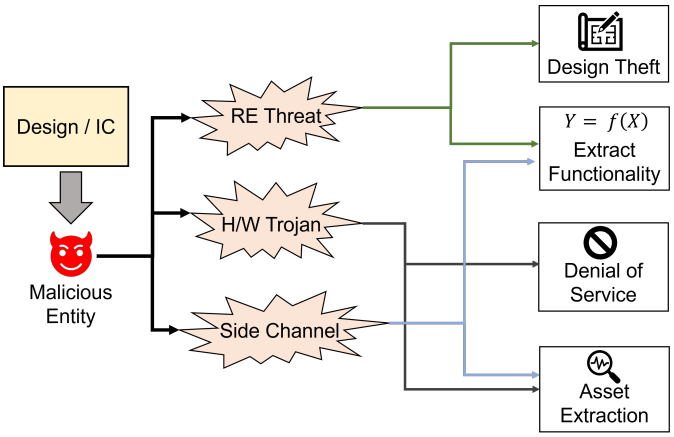


Fig. 2: Impacts of digital IC/design threats.

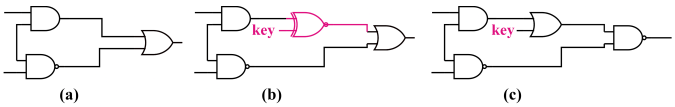


Fig. 3: Logic locking: (a) original netlist, (b) obfuscated netlist, and (c) synthesized netlist (structurally changed).

have vulnerabilities, including susceptibility to removal attacks and having low output corruption. Full-Lock [37] is designed to exponentially increase the time required for each iteration of the SAT attack by making the underlying SAT problem harder to solve. It uses logarithmic self-routing networks to create SAT-hard instances by increasing the complexity of recursive calls. To enhance security, Full-Lock [37] introduces key-configurable inverters and replaces certain gates with small Spin Transfer Torque (STT) based Look-Up Tables (LUTs). Another novel work, LoPher [38] is the first which combines hardware security with well-established cryptographic primitives like block ciphers. LoPher[38] embeds combinational logic into a block cipher, using the S-Box and diffusion layers to perform gate-level operations.

4) Rise of Structural Analysis: SAIL [39], [14] was the first machine learning-guided structural analysis attack that was designed to identify the lack of structural obfuscation during logic locking. Due to poor structural security, one could carry out removal attacks (to even undermine SAT security), perform key guessing, and carry out reverse engineering. Following SAIL, a large range of structural analysis-based logic locking attacks emerged such as OMLA [15], Snapshot [40], SCOPE [41], and SWEEP [16]. These attacks are highly scalable because they typically perform local analysis-based prediction, making their computations scale only with the size of key inputs (not the design size). Different defense techniques have been proposed to counter these attacks (e.g., UNSAIL [42]).

C. Why should we use XAI for DFS?

Most DFS methods follow two steps: (1) Determine where in the design modifications are required and (2) Determine the correct nature of these modifications. Assuming that there

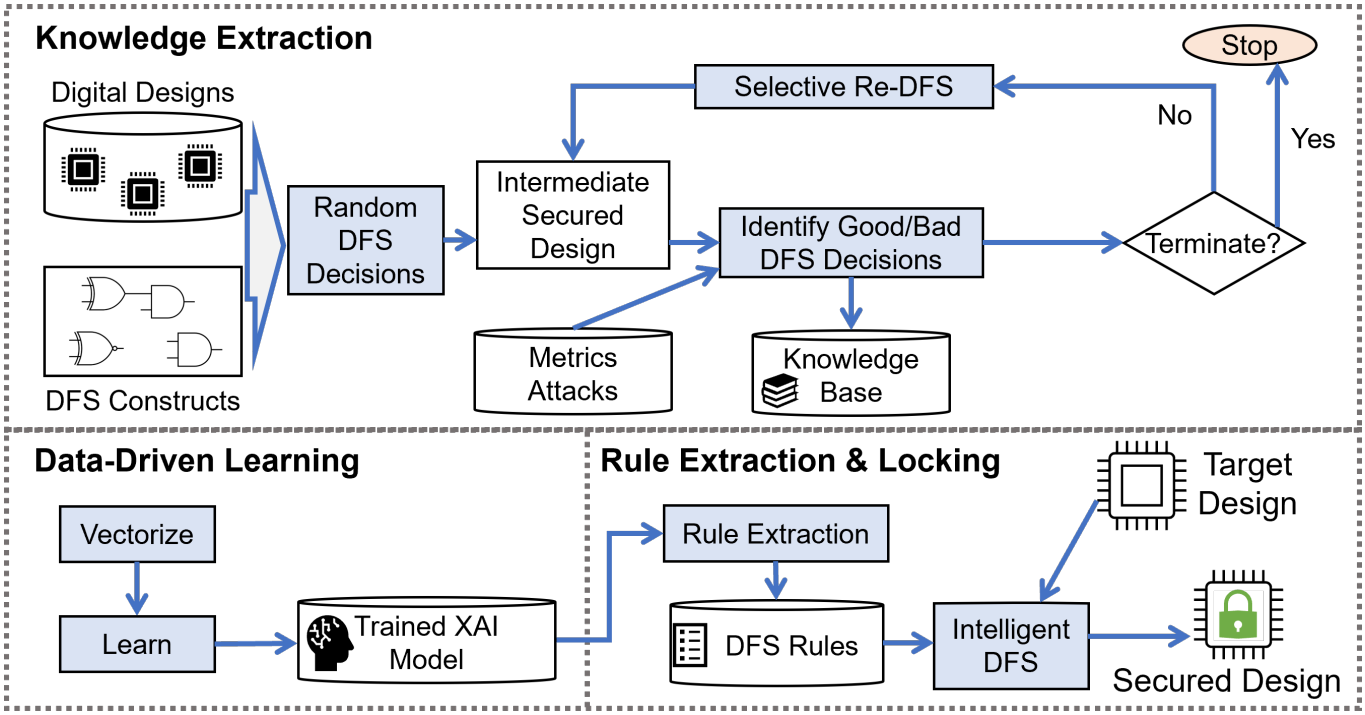


Fig. 4: Overview of the proposed X-DFS methodology.

are n valid regions in the design where modifications can be made for security and also let us assume that m different types of modifications are possible. Then we have a total of $(m+1)^n$ possible design modification options (+1 to consider no-change). For example, if (1) we are performing logic locking on a design with $60,000$ gates; (2) we are allowed to insert XOR, XNOR, and AND gates; (3) we are to insert a 256 bit key, then we have a total of $\binom{60,000}{256} \times 3^{256}$ choices. Hence, the DFS search space is vast and navigating this requires expert crafted heuristics driven algorithms. Designing these algorithms can be time-consuming and will most likely require the involvement of domain experts. Hence, using an automated algorithm to the search the solution space can save significant amount of time and cost. DFS strategies get obsolete with the emergence of novel attacks requiring a tweak to the existing DFS strategy or the development of a new DFS strategy. Hence, an automated framework can be instrumental in rapidly dealing with these novel attacks. To build this automated framework, we propose to use Artificial Intelligence (AI) due to its wide spread success in automating many other tasks in diverse domains. Furthermore, we would like to incorporate explainability (hence, XAI) into the framework to enable the extraction of human-understandable DFS rules that are being learned by the AI model for dealing with a given attack(s).

D. Related Works

Isolated DFS solutions have been developed to mitigate reverse engineering [23], [19], [18], [43], side channel [10], [11], and hardware Trojan [12] threats but there exists no systematic way to automatically devise defense strategies against novel attack vectors. Due to the rapidly growing microelectronic threat space, it has become extremely difficult for the research community and the industry to keep up with developing defense strategies against these novel attacks. These concerns are not addressed by current state-of-the-art DFS schemes. Moreover, techniques such as LeGO [17] are designed to apply already known defense rules to mitigate threats and are not designed to generate new defense strategies. Our proposed framework, X-DFS promises to address these concerns making it distinct and highly impactful.

III. X-DFS Methodology

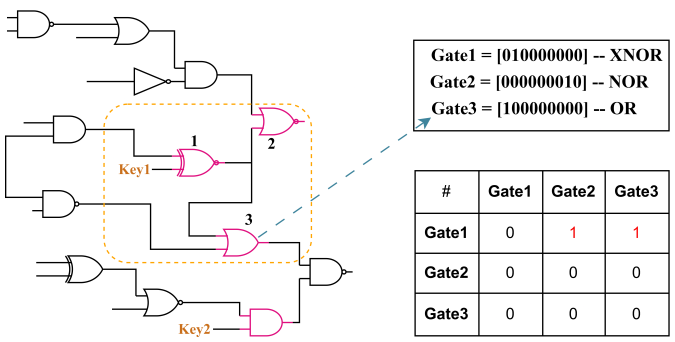
The overall framework is depicted in Fig. 4. Next, we formally describe how X-DFS can be used to defend against different microelectronic threats and do a deeper dive into the methodology, particularly for reverse engineering threats.

A. A Formal & Generalized approach to X-DFS

To formalize the X-DFS process, let us assume that:

1) We have a digital design ($D = \{V, E\}$). V = vertices/gates and E = hyperedges among elements in V.
2) There is a set of metrics/attacks ($A_t = \{a_1, a_2, ..., a_n\}$) that can evaluate the vulnerabilities of the design (D).
3) There are a set of DFS constructs ($C = \{c_1, c_2, ..., c_k\}$) that might lead to the design being resilient to the said attacks (A_t) if inserted or utilized in the right way.

With these assumptions, X-DFS first executes a knowledge extraction process by: (1) Randomly selecting c_i from C; (2) Randomly selecting a location ($gate \in V$ or $wire \in E$) in D to insert c_i; (3) Inserting c_i and obtaining a modified design (D'); (4) Validating the effectiveness of such a modification (D') by utilizing A_t and tracking this information in the knowledge base (K); (5) Fully or partially reverting the design



#	Gate1	Gate2	Gate3
Gate1	0	1	1
Gate2	0	0	0
Gate3	0	0	0

Fig. 5: Encoding a 3-gates sub-design structural features.

(D') to its original state (D). After repeating steps (1-5) for a certain number of times we use the knowledge base (K) to craft potent DFS rules (R) that utilizes C to defend designs beyond just D against A_t.

X-DFS can apply to the most microelectronic security threats if we can define A_t and C for that specific threat as shown in Table II. The success of X-DFS and the layout of this approach depends on the the hypothesis: Randomly selecting elements in C and inserting them in random locations of D will lead to at least some positive outcomes in terms of defense against A_t. In this paper, we empirically validate this hypothesis using our reverse engineering case study results. Future works will attempt to extensively validate this hypothesis for other microelectronic threats. We also intend to augment this random search with heuristics from existing human-generated knowledge-base to speed up the search process. For example, for Trojans, we will attempt to focus our search around rarely trigger nodes since that is where Trojans are typically inserted.

B. Understanding the Feature Set

Training and using an AI/XAI model will require us to first come up with a set of features that can capture diverse properties of a design or a sub-design (locality). For training and using X-DFS, we propose to use local structural features of design regions similar to SAIL [14], [39] and functional features (Static and Transition probabilities) of nearby wires based on methods proposed in [53]. The structural features of a wire captures information about the placement and connectivity of its locality (different types of gates and their interconnections). For a sub-design graph, we encode the connectivity as an adjacency matrix and represent the gate types using a one-hot encoding scheme. The structural feature extraction process is depicted in Fig. 5 and more details can be found in the SAIL article [14]. The gates are named ($G1$, $G2$, etc.) based on a breadth first search (BFS) algorithm search-order starting from the originating gate of the wire being considered for locking.

The functional features of a given wire are computed using the formulas presented in Table III and Table IV [53]. For a given signal or net in Table III, P_A and P_B are static probabilities (also known as signal probability) which indicate the fraction of time that the state is predicted to be at logic-1 or logic-0. If the net has a static probability of 0.4, then 40% of the time it is anticipated to be at logic-1. In Table IV, A and B represent the transition probability which is the number of jumps from logical level 0 to 1 or 1 to 0. The formulas in both of these tables are restricted to only the logic gates used to represent our benchmarks [54]. However, this list can be extended to support more complex gates (MUX, AOI, OAI) for future works. For computing functional features (during logic locking case study), the input wires are assumed to have a static and transition probability of 0.5.

Although we limit our current study to these specific features, the designed X-DFS framework can easily support additional structural and functional features of the design in the future. For example, it is feasible to incorporate features obtained from a graph neural network [55], as well as other functional features such as fan-in, fan-out, clustering coefficient and centrality measurement [56].

TABLE III: Static probability computation formulas.

Gate	Signal Probability of 1
NOT	1 - P_A
AND	$P_A * P_B$
OR	$P_A + P_B - P_A * P_B$

TABLE IV: Transition probability computation formulas.

Gate	$P_{0to1} = P_{out=0} * P_{out=1}$
NOT	1 - $A * A$
AND	(1 - $A * B$) * ($A * B$)
OR	(1 - A) * (1 - B) * (1 - (1 - A) * (1 - B))
NAND	($A * B$) * (1 - $A * B$)
NOR	(1 - (1 - A) * (1 - B)) * (1 - A) * (1 - B)
XOR	(1 - (A + B - 2 * $A * B$)) * (A + B - 2 * $A * B$)

C. Automatic Knowledge Extraction

As seen in Fig. 4, we start with a set of reference digital designs, and a set of DFS constructs. In case of logic locking, these constructs are gates such as XOR, AND, etc. Next, we randomly insert a set of DFS constructs into the design, apply the selected attack vector, and identify which insertions were successfully attacked and which insertions remain secure. If the attack model bypasses/removes an inserted construct, we label this insertion (choice of both DFS construct + region of insertion) as 0, a bad label. If not, we label it 1, a good label. These labels and associated features are stored in the knowledge base. After that, we remove bad insertions, randomly add new DFS constructs, and continue the knowledge extraction process until a termination condition is met (based on runtime, iterations, or label ratio). The amount of randomly added new constructs will be equal to the number of removal of bad insertions.

The essential consistent notations used in this paper are outlined in Table I. We have thoroughly discussed those notations when necessary. Algo. 1, provides greater details of the knowledge extraction process specifically for logic locking. Algo. 1 receives inputs such as the design ($design$) itself, possible locking gates ($lockDict$), number of keys to lock the design ($keySize$), locality size (loc) and number of iterations (Th_{it}) to terminate the algorithm. In line 1, we parse the

TABLE II: X-DFS Generalization.

	A_t (Attack, Metrics)	C (DFS Constructs)
Reverse Engineering	SAIL [14], OMLA [15], SWEEP [16]	Logic Locking Constructs (XOR, XNOR, AND) [18], [19]
Hardware Trojan	TRIT [44], MIMIC [45], TRIT-DS [46]	Dummy Logic, Control Points [12]
Side Channel	TVLA [47], SVF [48]	Gate Resizing [10], Masking Gates [49]
Fault Analysis	SOLOMON [50], FaultDroid [51]	Speed-up, Slow-down Constructs [52]

input design file and capture the information using an internal graph data structure. In line 2, we pre-compute the functional features. In line 5, we initialize the run variable that is used to track the number of iterations of the knowledge extraction process. In lines 6-23, we lock the design, perform the selected attack, extract the locking knowledge, relock the design, and repeat until a termination condition (no vulnerability is found or certain iterations by user's input are met, line 23) is reached. Inside this block, in line 7, we perform a random locking of the design by inserting an additional $keySize$ bits worth of key gates, where the locking constructs are also randomly sampled from the locking dictionary ($lockDict$). Next in line 8, we extract all the vulnerable and non-vulnerable key wires based on the $attack$ model. Next for each key wire region, we extract a set of structural (line 10) and functional (line 11) features. Next, the locking type associated with the specific key wire region is encoded as a vector (line 12 and line 13). If the vulnerability is found for that key insertion, then it is marked as 0 (the attack is successful for this region), lines 14-15. Otherwise, we set the label as 1 (considered a good locking against this attack), lines 16-17. The features {S,F,L} are inserted into $Train\_Data$ and the labels are inserted inside $Train\_Label$. The vulnerable key wires are removed (line 20) and the $keySize$ is reduced to the total number of vulnerable insertions ($count$) in line 21. In line 22, we increment the run variable. We terminate this iterative knowledge extraction process when the $Vuln$ list is empty or when we have reached the user-specified maximum iteration threshold (Th_{it}). The dataset ($Train\_Data$ & $Train\_Label$) is returned in line 24.

Algorithm 1: Knowledge Extraction

Input: $[lockDict, attack, design, keySize, loc, Th_{it}]$
Output: $[Train\_Data, Train\_Label]$

1 $G \leftarrow extract\_Graph(design)$
2 $FnFeat \leftarrow compute\_Functional\_Feat(G)$
3 $Train\_Data \leftarrow []$
4 $Train\_Label \leftarrow []$
5 $run \leftarrow 0$
6 **do**
7 $\quad G \leftarrow random\_Lock(G, keySize, lockDict)$
8 $\quad Vuln, nonVuln \leftarrow attack(G)$
9 $\quad$ **for** i ***in*** $(Vuln + nonVuln)$ **do**
10 $\quad\quad S \leftarrow ext\_Structural\_Feat(G, i, loc)$
11 $\quad\quad F \leftarrow FnFeat[i]$
12 $\quad\quad lockType \leftarrow find\_Type(G, i)$
13 $\quad\quad L \leftarrow encode\_Locking\_Type(lockType)$
14 $\quad\quad$ **if** i *in* $Vuln$ **then**
15 $\quad\quad\quad label \leftarrow 0$
16 $\quad\quad$ **else**
17 $\quad\quad\quad label \leftarrow 1$
18 $\quad\quad Train\_Data.append([S, F, L])$
19 $\quad\quad Train\_Label.append(label)$
20 $\quad [G, count] \leftarrow remove\_Vuln\_Key(G, Vuln)$
21 $\quad keySize \leftarrow count$
22 $\quad run++$
23 **while** $Vuln! = empty$ *and* $run \leq Th_{it}$
24 **return** $[Train\_Data, Train\_Label]$

D. Building the X-DFS Model

Next, we train an XAI model to capture the DFS experimentation knowledge base. All the $Train\_Data$ and $Train\_Label$ obtained from each design using Algo. 1 are merged to form $Train\_Data\_Merged$ and $Train\_Label\_Merged$. This dataset is used to fit a wide range of AI models that are either inherently explainable (such as Decision Tree) or can be explained using other techniques such as SHAP (SHapley Additive exPlanations) [58], LIME (Local Interpretable Model-agnostic Explanations) [59], and Anchors [60]. We also experiment with different data preprocessing techniques (e.g., standard scalar, min/max scaling [61]). Although the Decision Tree (DT) and Random Forest (RF) model are by default explainable, we also obtain results with Ensembling (Decision Tree, Random Forest, and Adaboost). When considering the ensemble results, we assign equal importance to each model. Due to the random nature of the development of the dataset, if we encounter any data imbalance, we employ the SMOTE [62] technique to

TABLE V: SAIL accuracy for different X-DFS models. Algo. 2 parameters: KL = 10% of Design Size, $A = FALSE$, $RN = FALSE$, $U = TRUE$, $LC = 3$.

Designs	DT	ENSM	RF	ADB
sqrt	40.83	22.09	17.41	2.10
sin	52.32	15.40	12.20	1.47
div	36.3	15.72	29.79	20.14
arbiter	33.53	6.93	11.13	0.24
memctrl	47.14	26.3	28.17	13.03
log2	42.83	13.33	12.72	0.74
square	33.31	20.93	20.79	13.56
multiplier	40.13	19.97	12.77	0.69
voter	61.70	30.78	28.39	24.77
Average	**43.12**	**19.05**	**19.20**	**8.52**

TABLE VI: Comparison of X-DFS with other prominent Logic Locking algorithms. Values indicate SAIL attack accuracy. Algo. 2 parameters: $KL = 128$, $A = FALSE$, $RN = FALSE$, $U = TRUE$, $LC = 3$.

Designs	Random	SFLL_Point [19], [57]	Anti-SAT [19], [33]	XORProb [19]	AOR [19]	X-DFS [ADB]
sqrt	54.69	42.96	46.09	56.25	50.78	0
sin	53.12	52.34	(-)	47.65	55.46	4.69
div	46.88	50	32.03	44.53	53.13	0
arbiter	53.90	48.44	43.75	42.19	57.03	0
memctrl	62.50	55.47	44.53	48.43	63.28	3.13
log2	49.21	44.53	(-)	51.56	50	0
square	49.21	50.78	(-)	42.96	52.34	7.03
multiplier	51.56	45.31	56.25	48.43	47.65	1.56
voter	53.13	46.88	55.46	53.90	54.68	0
Average	**52.69**	**48.52**	**46.35**	**48.43**	**53.81**	**1.82**

TABLE VII: X-DFS against OMLA attack. Values indicate OMLA attack accuracy. Algo. 2 parameters: $KL = 64$, $A = FALSE$, $RN = FALSE$, $U = TRUE$, $LC = 3$.

Designs	Random	X-DFS [RF]
sqrt	95.31	9.37
sin	67.19	9.37
div	67.19	4.68
arbiter	70.31	43.75
memctrl	95.31	48.44
log2	93.75	23.44
square	78.13	4.68
multiplier	89.06	6.25
voter	70.31	23.44
Average	**80.73**	**19.27**

TABLE VIII: X-DFS against SWEEP. Values indicate SWEEP accuracy. Algo. 2 parameters: KL = 10% of Design Size, $A = FALSE$, $RN = FALSE$, $U = TRUE$, $LC = 3$.

Designs	Random	X-DFS [RF]
sqrt	72.95	15.73
sin	77.45	6.74
div	79.95	8.83
arbiter	69.87	28.36
memctrl	75.59	18.77
log2	78.75	25.08
square	85.83	10.85
multiplier	73.19	28.32
voter	82.60	15.89
Average	**77.35**	**17.62**

address the imbalance. SMOTE, unlike basic oversampling methods that simply replicate instances from the minority class, produces new synthetic examples. This technique mitigates overfitting by increasing diversity and addressing class imbalance in classification problems. For all machine learning models, we utilize the default settings (as specified in [61]) except for the Adaboost learning rate (set to 0.01).

E. Extraction of Design Transformation Rules

We leverage different explainability algorithms towards visualizing the internal learning and decision making process of X-DFS models. Complex models perform best; however, they are not inherently human-understandable. Hence, we utilize the SHAP (SHapley Additive exPlanations) algorithms, specifically the Kernel Explainer (model-agnostic) towards

Algorithm 2: X-DFS_Lock

Input: $[D, KL, Model, lockDict, A, U, RN, Th_g, lc, M]$
Output: D_Locked

$G \leftarrow extract\_Graph(D)$
$FnFeat \leftarrow compute\_Functional\_Feat(G)$
$options \leftarrow []$
$Good \leftarrow []$
$breakFlag \leftarrow FALSE$
for $i \leftarrow 0$ ***to*** $len(G.wires)$ **do**
  for $j \leftarrow 0$ ***to*** $len(lockDict)$ **do**
    $S \leftarrow ext\_Structural\_Feat(G, G.wires[i], lc)$
    $F \leftarrow FnFeat[G.wires[i]]$
    $L \leftarrow encode\_Locking\_Type(lockDict[j])$
    $P \leftarrow Model([S, F, L])$
    $options.append([G.wires[i], lockDict[j], P])$
    if $A == TRUE$ **then**
      $Good \leftarrow find\_Good(options, Th_g)$
      if $len(Good) \geq M \times KL$ **then**
        $breakFlag \leftarrow TRUE$
        $break$
  if $breakFlag == TRUE$ **then**
    $break$
if $A == TRUE$ *and* $RN == FALSE$ **then**
  $Good \leftarrow Good.sort\_Descend\_ByP()$
if $A == TRUE$ *and* $RN == TRUE$ **then**
  $Good \leftarrow Good.sort\_Descend\_ByP()$
  $Good \leftarrow Good.shuffle()$
if $RN == TRUE$ *and* $A == FALSE$ **then**
  $Good \leftarrow find\_Good(options, Th_g)$
  $Good \leftarrow Good.shuffle()$
if $RN == FALSE$ *and* $A == FALSE$ **then**
  $Good \leftarrow options.sort\_Descend\_ByP()$
if $U == TRUE$ **then**
  $Good \leftarrow unique\_Wire(Good)$
$G \leftarrow lock(G, Good[0 : KL])$
$D\_Locked \leftarrow write\_To\_File(G)$
return D_Locked

understanding the feature importance [58]. SHAP uses a game theory methodology to determine which features contribute the most towards making the final prediction. For a given prediction, the Shapley value of a feature is its expected (weighted by probability) marginal contribution. The Shapley value of a feature i, ϕ_i is computed as shown in equation 1. Here we assume that there are p features and hence $p!$ is the total number of ways to form a coalition of p features (players). Assume that g is a given coalition. Then $\frac{|g|!(p-|g|-1)!}{p!}$ is the weight term and $val(g \cup \{i\}) - val(g)$ is the marginal contribution of the feature i to a given coalition g.

$$\phi_i = \sum_{g \subseteq \{1,\ldots,p\}\{i\}} \frac{|g|!(p-|g|-1)!}{p!}[val(g \cup \{i\}) - val(g)] \tag{1}$$

This Shapley value equation is derived based on four axioms: (1) Efficiency, (2) Symmetry, (3) Null Player, and (4) Additivity. The additivity axiom conveys that: if we combine two games (predictions), then the sum of a feature's contribution is its total contribution (assuming each prediction is independent). For more details, refer to [63], [58]. For an X-DFS model, understanding the importance of each feature for different samples will allow us to understand what the X-DFS model has actually learned and distil that understanding into human-understandable DFS rules.

F. DFS using the XAI Model

With a trained $X-DFS\_Model$, we can also directly use it (and the inherent DFS rules) to transform a target design. This process is described in Algo. 2 specifically for logic locking. Algo. 2 requires the design (D), number of key length (KL), the X-DFS model ($Model$), possible gates to lock ($lockDict$), participation of Ada X-DFS (A), participation of unique wires (U), participation of RN X-DFS (RN), threshold to determine the goodness of locking (Th_g), locality size (lc), integer value to filter Ada X-DFS (M) as inputs. In line 1, we parse the input design (D) and capture it in our graph data structure as G. In line 2, we pre-compute the functional features of every wire. Then for each wires in G and for each entry in the locking dictionary $lockDict$, we obtain a real-valued prediction (0 to 1) using $X-DFS\_Model$ (lines 6-19). Next the specific design wire ($G.wires[i]$), the locking construct ($lockDict[j]$), and the prediction (P) are appended to $options$ (line 12). To speed up the locking process (and avoid unnecessary computation), we terminate this exhaustive evaluation process only when the A (X-DFS_Ada) flag is set to 'TRUE' (line 13) and at least $M \times KL$ good locking options are discovered (lines 14-17). Here M is a multiplier integer value provided as input by the user and KL is the target key length for the locking operation. The goodness of a locking option is determined using a threshold (Th_g) imposed on the prediction values. Based on the goodness threshold (Th_g), the best $options$ are filtered into $Good$ in line 14. If A (X-DFS_Ada) flag is set to 'TRUE' and RN (X-DFS_RN) is 'FALSE', we use prediction value in descending order to sort the $Good$ candidates in lines 20-21. To allow flexibility and some variability (to divert attackers), we also introduce some controlled randomness if RN (X-DFS_RN) is set to 'TRUE' and if A is set to 'FALSE'. This is achieved by finding the top options ($Good$) using Th_g (line 26), and then shuffling the list in line 27. To introduce randomness in the X-DFS_Ada both A and RN need to be 'TRUE' (shown in line 22-24). If both RN and A are set to 'FALSE', then we simply utilize prediction value to sort the $options$ in descending order to create $Good$ (keeping only the top locking options) in lines 28-29. Locking can also be done for distinct wires by setting U is 'TRUE' (unique nets) in line 30-31. Finally, the design is locked with the top KL elements in $Good$ and the resulting design graph is converted to the design format (D_Locked) before being returned.

G. Algorithms Complexity Analysis

For Algo. 1, let us assume that the termination condition (line 23) is based on a set number of iterations (I), there are V gates in the design, there are E wires in the design, and the time complexity of the attack/metric is $\mathcal{O}(A_t)$ (varies). Then the computational time complexity of line 1 is $\mathcal{O}(V+E)$, line 2 is $\mathcal{O}(V+E)$, line 7 (for each outer iteration) is $\mathcal{O}(KeySize)$, line 8 (for each outer iteration) is $\mathcal{O}(A_t)$, line 9-19 (for each outer iteration) is $\mathcal{O}(KeySize)$, line 20 (for each outer iteration) is $\mathcal{O}(KeySize)$, and line 21-22 (for each outer iteration) is $\mathcal{O}(Const)$. Then the total computational time complexity considering the outer loop, line 23 (I) is: $\mathcal{O}(V+E+V+E+I*KeySize+I*A_t*Const+I*KeySize+I*KeySize+I)$ = $\mathcal{O}(V+E+I*KeySize+I*A_t)$. Both I and $KeySize$ will be small values and will not necessarily scale across applications. Hence, the complexity can be further simplified to $\mathcal{O}(V+E+A_t)$. For attacks such as SAIL where the computation required is $\mathcal{O}(KeySize)$, the time complexity becomes $\mathcal{O}(V+E)$ (considering $KeySize$ is small).

For Algo. 2, let us also assume that the locking dictionary ($lockDict$) size is L. Then the computational time complexity of line 1 is $\mathcal{O}(V+E)$, line 2 is $\mathcal{O}(V+E)$, lines 8-12 (for each iteration) is $\mathcal{O}(Const)$ where $Const$ is constant, line 14 (for each iteration) is $\mathcal{O}(EL)$, line 21 is $\mathcal{O}(EL*Log(EL))$, lines 23-24 is $\mathcal{O}(EL*Log(EL)+EL)$, lines 26-27 is $\mathcal{O}(EL)$, line 29 is $\mathcal{O}(EL*Log(EL))$, line 31 is $\mathcal{O}(EL)$, line 32 is $\mathcal{O}(KeySize)$, and line 33 is $\mathcal{O}(V+E)$. The total computational time complexity considering the outer loops (EL) and upon simplification is: $\mathcal{O}(E^2L^2)$. L will typically be a small value making the time complexity $\mathcal{O}(E^2)$.

IV. Results & Analysis

In this section, we analyze the effectiveness of the proposed framework (X-DFS) in creating DFS strategies to counter reverse engineering attacks (SAIL [14], [39], OMLA [15], SWEEP [16]). We do not consider the SAT [64] attack for this study because: (1) It can be countered with well-known techniques such as Anti-SAT [34], Full-Lock [37], and LoPher [38]; (2) It does not perform well for large designs and large key sizes due to its reliance on solving an NP-Hard problem in the backend; (3) It requires an oracle or unlocked design making it less practical.

TABLE IX: Ada X-DFS results for different values of M. We observe significant run-time improvement. Algo. 2 parameters: KL = 10% of Design Size, $A = TRUE$, $RN = FALSE$, $U = FALSE$, $Th_g = 0.9$, $LC = 3$.

Designs	Ada X-DFS [$M = 3$]		Ada X-DFS [$M = 7$]		Ada X-DFS [$M = 11$]		General Settings	
#	SAIL Acc	Lock Time (Sec)	SAIL Acc	Lock Time (Sec)	SAIL Acc	Lock Time (Sec)	SAIL Acc	Lock Time (Sec)
sqrt	29.33	289.78	17.44	424.72	5.40	575.19	2.10	1310.51
sin	14.33	43.62	7.96	73.92	4.02	105.03	1.47	194.34
div	26.65	520.41	19.04	818.16	19.99	1120.37	20.14	1678.42
arbiter	23.71	104.50	0.24	181.71	0.24	183.35	0.24	183.35
memctrl	30.20	655.21	15.34	985.47	15.18	1355.06	13.03	1620.57
log2	28.93	366.53	12.84	884.01	11.10	1184.59	0.74	1784.23
square	28.10	366.38	22.43	592	13.43	855.71	13.56	1083.20
multiplier	27.27	571.11	11.94	807.53	23.76	1156.57	0.69	1695.14
voter	49.32	170.23	38.47	314.22	25.49	451.90	24.77	583.12
Average	**28.64**	**343.03**	**16.19**	**564.63**	**13.17**	**776.42**	**8.52**	**1128.06**

TABLE X: RN X-DFS can produce diverse (low similarity) SAIL-resilient locked designs. Algo. 2 parameters: KL = 10% of Design Size, $A = FALSE$, $RN = TRUE$, $U = TRUE$, $Th_g = 0.9$, $LC = 3$.

Designs	Round 1		Round 2		Round 3		Round 4	
#	SAIL	Sim	SAIL	Sim	SAIL	Sim	SAIL	Sim
sqrt	6.47	0.132	7.78	0.130	7.62	0.136	8.11	0.132
sin	7.11	0.151	6.23	0.157	6.71	0.150	6.48	0.140
div	7.91	0.228	8.7	0.221	10.3	0.230	9.39	0.226
arbiter	14.1	0.410	12.93	0.423	13.58	0.473	14.06	0.435
memctrl	10.8	0.251	10.50	0.265	11.35	0.241	11.86	0.260
log2	8.84	0.153	10.70	0.171	9.55	0.167	9.43	0.161
square	9.18	0.210	8.22	0.220	8.81	0.217	8.70	0.227
multiplier	12.43	0.193	12.48	0.194	13.65	0.192	13.59	0.192
voter	23.30	0.198	24.05	0.186	20.93	0.184	20.70	0.230
Average	**11.12**	**0.214**	**11.34**	**0.218**	**11.39**	**0.221**	**11.37**	**0.225**

A. Experimental Setup

We implement the above described algorithms towards creating a highly parameterized framework for extensive evaluations. Algo. 1 was used to extract training data from a set of small designs from the ISCAS-85 benchmark suit (c1355, c1908, c2670, c3540, c5315, c6288, c7552 [54]) to speed up the learning process. However, during the evaluation process, we utilize large scale designs (sqrt, sin, div, arbiter, memctrl, log2, square, multiplier, voter [54]) following a transfer learning approach to better understand the scalability of the proposed framework. The locking dictionary ($lockDict$) has the following logic locking gate constructs: XOR, $XNOR$, OR, AND. We choose SAIL, SWEEP, and OMLA for our evaluation because: (1) They represent the large set of structural analysis-based attacks; (2) These attacks are cutting edge and strongest framework in structural attacks domain; (3) Each of them is highly scalable for large designs; and (4) Other functional attacks such as SAT attack can be mitigated using locking constructs such as Anti-SAT [34], Full-Lock [37], LoPher [38].

The SAIL and SWEEP attack models are trained on the following designs: c1355, c1908, c2670, c3540, c5315, c6288, c7552, sqrt, sin, div, arbiter, voter [54].

On the contrary, we train the OMLA attack using the designs presented in [55]. We have located 1,000 designs for each of c1355, c1908, c2670, and c3540, totaling 4,000 designs. All designs have been utilized to train OMLA. The total number of gates defines the size of a design.

$keySize$ for Algo 1 are always set to 128. Th_{it} for Algo. 1 is set to 80. KL for Algo. 2 has been altered in various ways for different experimental types, and we have specified all those key sizes in the subsequent descriptions. Algo. 1 creates the dataset in just 1.5 hours with this settings. All experiments are run on: Intel Core i9-11900, 8 cores and 16 threads, 32GB RAM.

B. X-DFS for Defending Against SAIL

As shown in Table V, we experiment utilizing different AI techniques for creating the X-DFS model to counter the SAIL attack. Among them, Adaboost outperforms all. 10% of each design size is the total amount (large quantity) of locking gates that X-DFS uses to lock the designs for this table. For instance, if a design has 30,000 gates, then 3,000 locking gates are inserted (10%). Each value in this Table V indicates SAIL's accuracy. This SAIL accuracy is the appropriate proportion of keys that the SAIL attack model identifies correctly. It appears that the div and the voter designs are inherently more vulnerable to SAIL, probably due to uniform structural patterns (similar to c6288 from ISCAS-85 benchmark suit as demonstrated in the original SAIL work [14]). Table V compares the effectiveness of different AI models for mitigating the SAIL attack using the X-DFS framework.

C. Comparison Against Other LL Techniques

In Table VI, we compare X-DFS with other popular logic locking algorithms, including SFLL_Point [19], [36]. Here, the random locking (column $Random$) uses the same set of locking dictionary as X-DFS without the added learning/intelligence. X-DFS clearly outperforms all these techniques in terms of building up SAIL resiliency. Also note that Anti-SAT fails to lock some of the designs (ran for > 20 hours) probably due to their large size and other algorithm specific constraints (indicated with $-$, inside the table). For the experiments in Table VI, these big designs are locked using 128 bit keys. This is because some logic locking methods are unable to lock designs with larger keys (> 128 can not be handled). This is, however, not a limitation for X-DFS, which is highly scalable in this regard.

TABLE XI: Ada X-DFS for varying thresholds (Th_g). Algo. 2 parameters: KL = 10% of Design Size, $A = TRUE$, $RN = FALSE$, $U = FALSE$, $M = 11$, $LC = 3$.

Designs	Ada X-DFS [$Th_g > 0.50$]		Ada X-DFS [$Th_g > 0.65$]		Ada X-DFS [$Th_g > 0.80$]	
#	SAIL Acc	Lock Time (Sec)	SAIL Acc	Lock Time (Sec)	SAIL Acc	Lock Time (Sec)
sqrt	13.98	342.17	10.57	367.39	7.18	571.31
sin	18.63	72.32	11.17	83.41	6.67	97.94
div	27.82	998.07	18.69	1035.44	18.05	1117.22
arbiter	11.45	67.33	9.21	126.09	1.89	155.07
memctrl	18.32	845.49	14.16	920.13	15.91	1239.39
log2	23.2	823.11	17.4	897.27	13.47	1106.4
square	18.43	619.54	19.11	834.39	13.22	843.15
multiplier	24.05	778.29	18.29	907.52	24.01	1078.71
voter	26.59	243.02	21.83	330.02	25.11	398.26
Average	**20.27**	**532.15**	**15.6**	**611.3**	**13.95**	**734.16**

TABLE XII: X-DFS for different LC. Values indicate SAIL attack accuracy. Algo. 2 parameters: KL = 10% of Design Size, $A = FALSE$, $RN = FALSE$, $U = TRUE$.

Designs	Random	$LC = 5$	$LC = 7$	$LC = 10$
sqrt	55.89	6.89	5.46	3.94
sin	53.9	5.62	4.43	2.5
div	57.08	8.67	10.95	4.37
arbiter	53.75	3.3	3.91	3.79
memctrl	57.97	1.12	0.9	0.86
log2	53.73	2.19	3.58	1.94
square	53.17	4.81	2.12	4.4
multiplier	55.52	4.98	5.24	4.65
voter	53.58	5.1	15.18	8.32
Average	**54.96**	**4.74**	**5.75**	**3.86**

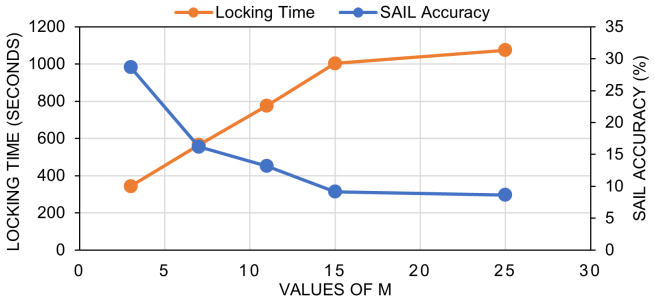


Fig. 6: Trade off between X-DFS speed and efficiency. Algo. 2 parameters same as Table IX.

D. X-DFS for Countering OMLA and SWEEP

In Table VII and Table VIII we illustrate the effectiveness of X-DFS against OMLA [15] and SWEEP [16] respectively. We utilize the transfer learning approach for these experiments by using the X-DFS model already trained for mitigating SAIL. As seen in Table VII and Table VIII, X-DFS shows great performance in terms of mitigating OMLA and SWEEP even when it is not trained on those specific data. This ability to provide protecting against unknown attacks is one of the most crucial aspects of building a logic locking (or DFS) framework, since new attacks will inevitably arise. X-DFS is quite effective in accomplishing this goal. This demonstrates the practicality and robustness of X-DFS. Results shown in Table VII are provided for 64 bit keys because the publicly available trained model and the dataset for OMLA [55] utilize 64 bit keys. For SWEEP and OMLA, we observed the best performance when utilizing Random Forest (among other machine learning algorithms).

E. Effectiveness of X-DFS for Different Locality Sizes

In Table XII we present the X-DFS results for countering SAIL attack while using structural features with locality sizes 5, 7, and 10. X-DFS generally performs better as we increase the locality size. Analyzing a larger locality provides more context for the AI model in terms of determining the optimal locking choices. All the results in this Table XI are obtained using the Adaboost model.

F. Improving the Runtime: Ada X-DFS

The X-DFS locking algorithm (Algo 2) has a time complexity of $\mathcal{O}(E^2)$ which is already very scalable (analysis provided above). However, to further speed things up, we introduce an approximation using the adaptive flag (A) as shown in Algo 2. We set $A = TRUE$, $RN = FALSE$, and vary $M = \{3, 7, 11\}$ (M selects the least number of possibilities, mentioned in Algo. 2) toward obtaining the results in Table IX. For this table, $Good$ (in Algo 2) is measured using $Th_g \geq 0.90$. Since synchronization with other locking approaches is not required, larger size keys (10% of design size) are utilized to track how long X-DFS takes to produce resilient designs. It shows a clear trade-off between run-time and SAIL resilience in Fig. 6. As M increases, we observe that both the execution time and SAIL resilience improve. This is because higher values of M result in a larger search space, allowing for the identification of better DFS/locking candidates. The *General Settings* column of Table IX showcases the results of X-DFS when $A = FALSE$ (non Adaptive).

G. Ada X-DFS with different thresholds

Next we examine the behavior of the framework for different threshold (Th_g) values, with M being held constant at 11. These results are shown in Table XI. When the threshold (Th_g) is increased to values above 0.50, we observed that the locking duration increases and the accuracy of the SAIL attack decreases. This is because a higher threshold allows the X-DFS to search for more optimal locking candidates.

TABLE XIII: Directly locking with extracted rules using LeGO. Values indicate SAIL attack accuracy. KeySize, KL = 10% of Design Size is used.

Designs	Random	X-DFS Rules + LeGO
sqrt	55.89	16.11
sin	53.90	14.89
div	57.08	18.13
arbiter	53.75	11.06
memctrl	57.97	17.67
log2	53.73	19.90
square	53.17	20.01
multiplier	55.52	15.62
voter	53.58	29.29
Average	**54.96**	**18.07**

H. Introducing Randomness: RN X-DFS

X-DFS is an informed (through learning) locking technique, but some degree of randomness is necessary for different designing scenarios (for obfuscations or variant generations). Hence, we have allowed for some controlled randomness using the flag RN in Algo 2. We set $A = FALSE$, $RN = TRUE$, towards obtaining the results in Table X for multiple instances of X-DFS locking. We also show through a cross-instance averaged cosine similarity metric (Sim column in Table X) in several rounds that the generated designs are indeed locked differently while maintaining a high degree of SAIL resilience ($SAIL$ column in Table X).

I. Explainability: Which Features are Important for X-DFS?

The top sub-figures in Fig. 7(a) shows the feature importance summary plot for the X-DFS_RF_Model (on 100 random data points). The feature names are mentioned on the Y-Axis ranked in descending order of significance. The position on the X-axis (with respect to the neural point 0.0) determines the amount of positive (right side, towards label 1) or negative (left side, towards label 0) impact. Each dot is a data point which represents a row of data from the original dataset. Each point on the graph is assigned a color that corresponds to the value of the related feature. High values are represented by the color red, while low values are represented by the color blue. $LockType$ refers to a specific type of locking gate. G1 (Gate1), G2 (Gate2), and G3 (Gate3) are the three gates in the structural feature locality (in Fig. 5) and the value next to them indicates their gate type. The naming of these gates are based on their breadth-first-search order as detailed in Section III-B. Static_Prob is referring to static probability (Table III) and Transition_Prob is referring to transition probability (Table IV). We can observe that locking with gate types OR and $XNOR$ is preferred by the model, whereas locking with gate types AND and XOR is avoided. Selecting a wire with higher static probability for locking is preferred. It is also evident that design regions with AND gates are avoided during locking. SHAP [58] utilizes a game-theory-based approach and some of the features do not participate strongly enough (dropped from these plots).

J. Explainability: Can we Extract Good Locking Rules?

Using SHAP algorithms, it is also possible to understand the decision process for each sample by observing the waterfall graphs (b,c,d,e,f,g in Fig. 7). The waterfall plot represents the values of the X-axis corresponding to the target variable, which in this case is the probability of good locking. x represents the selected observation, f(x) is the predicted value of the model for input x, and E[f(x)] represents the expected value of the target variable, which is essentially the average of all predictions (in our case, 100 random points). The SHAP value for each feature is indicated by the length of the bar (effect on prediction). For example, in Fig. 7(c) locking type $XNOR$ is pushing the prediction towards 1.

From these plots, we can extract 'rules' that are being implicitly used by the X-DFS_Models for locking a given design to counter SAIL, OMLA, and SWEEP attacks. We consider probability values below 0.3 as low, 0.3 to 0.5 as moderate, and beyond 0.5 as high. Rules are shown in Table XIV. These rules (extracted via X-DFS) can also be used to directly lock designs using manual methods or frameworks like LeGO [17]. This knowledge can be used to understand the strengths and weaknesses of a given attack vector and thereby assist researchers/engineers in mitigating similar attacks/variants.

K. Directly Locking with Extracted Rules: LeGO

The human-understandable rules generated from X-DFS can be directly utilized by the LeGO [17] framework to lock a design. In Table XIII, we show the effectiveness of using the X-DFS rules (Table XIV) directly with the help of the LeGO framework for mitigating SAIL attack.

V. Conclusion

Through this work, we have developed an explainable artificial intelligence guided framework (X-DFS) that can automatically navigate the design-for-security search space for generating a set of mitigation rules for a given novel attack vector. We have explained the inner-workings of X-DFS using detailed algorithms and implemented the framework towards creating a highly parameterized tool (specifically for reverse engineering attacks). Using the implantation, we have demonstrated the effectiveness of the framework for mitigating logic locking attacks such as SAIL, SWEEP, and OMLA in diverse settings. To boost the speed of X-DFS, we introduce X-DFS_Ada which can operate on large designs with up to 60,000 gates within 15 minutes and still ensure attack resiliency. Another variant, X-DFS_RN, was also introduced for injecting controlled randomness in the process towards diverting certain type of bias-driven attacks. X-DFS has outperformed most existing popular logic locking DFS algorithms including SFLL_Point. We have also demonstrated how X-DFS can be used to understand the decision process of AI models towards creating human-understandable X-DFS rules for countering target attacks. Future works will focus on exploring the effectiveness of X-DFS against other attack vectors (e.g., hardware Trojans, fault injection, side channel) and developing more sophisticated explainable algorithms.

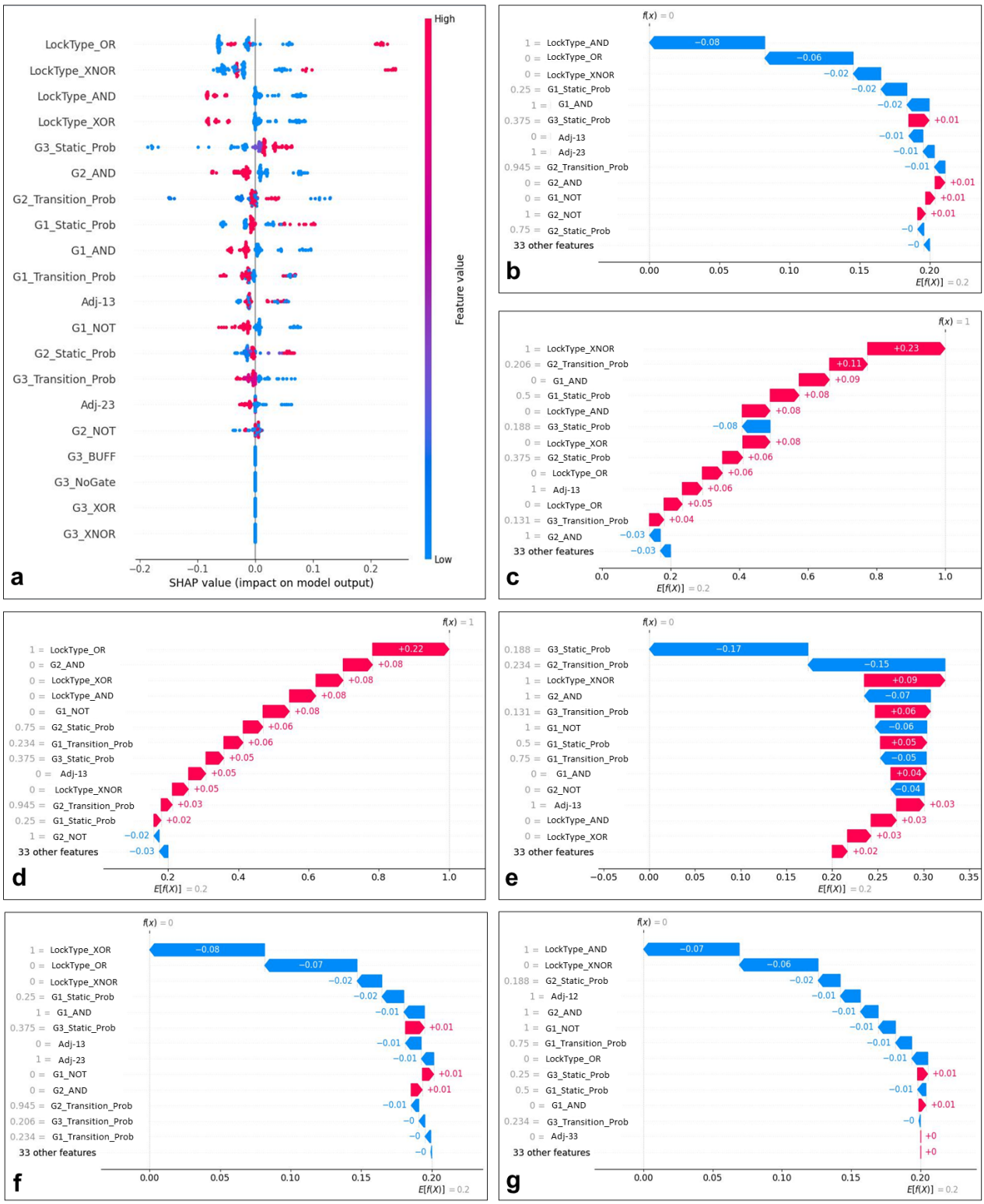


Fig. 7: Explainability of X-DFS_RF Model. From top to bottom: (a) SHAP-based feature summary extracted from 100 random data points; (b-g) SHAP waterfall plots for different samples.

VI. Acknowledgment

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 2350363 and Grant No. 2316399.

References

[1] Y. M. Alkabani and F. Koushanfar, "Active hardware metering for intellectual property protection and security," *Proceedings of the 16th USENIX Security Symposium*, 2007.

[2] F. Koushanfar, "Provably secure active ic metering techniques for

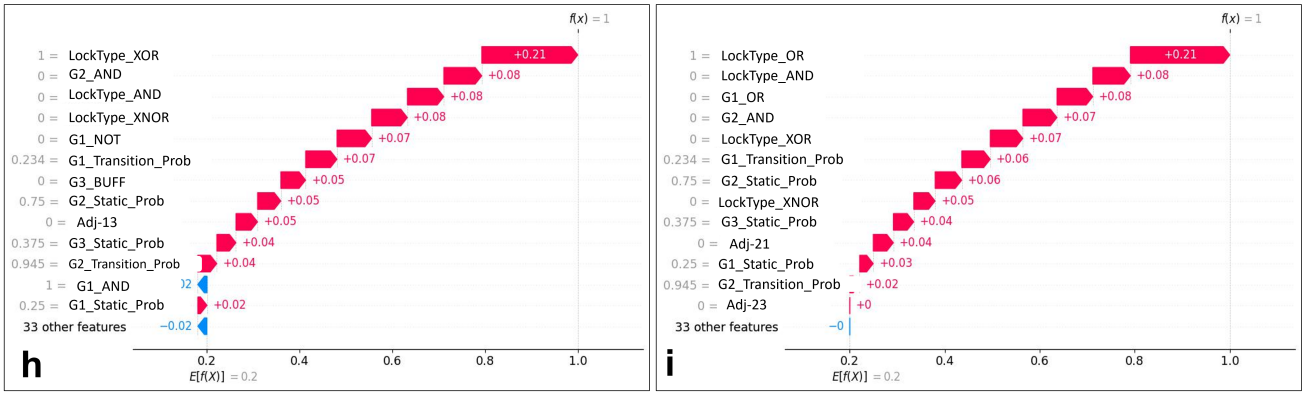


Fig. 8: (h-i) Explainability of X-DFS_RF Model: Additional SHAP waterfall plots.

TABLE XIV: Human-readable rules from X-DFS: Logic Locking Case Study.

	IF	Action
b	G1 = AND && Static probability of G1 = low	Do not lock with AND/XOR
c	G1 != AND && Transition probability of G2 = low && Static probability of G1 = moderate	Lock with XNOR
d	G2 != AND && G1 != NOT && Static Probability of G2 = high	Lock with OR
e	Transition probability of G2 = low && G2 = AND && G1 = NOT	Do not lock with any gate
f	Static probability of G1 = low && G1 = AND && G2 and G3 connected	Do not lock with XOR
g	Static probability of G2 is low && G2 == AND && a NOT gate is connected to an AND gate	Do not lock with AND
h	G1 != NOT && Static probability of G1 = high && G2 != AND && Transition probability of G2 = low && G3 != BUFF && Static probability of G3 = moderate	Lock with XOR
i	Transition probability of G1 = low && Static probability of G1 = low && Static probability of G2 = high && Transition probability of G2 = high && G1 != OR && G2 != AND	Lock with OR

piracy avoidance and digital rights management," *IEEE Transactions on Information Forensics and Security*, 2012.

[3] A. B. Kahng, J. Lach, W. H. Mangione-Smith, S. Mantik, I. L. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe, "Watermarking techniques for intellectual property protection," in *Proceedings of the 35th annual Design Automation Conference*, 1998, pp. 776–781.

[4] J. Rajendran, O. Sinanoglu, and R. Karri, "Is split manufacturing secure?" in *Proceedings -Design, Automation and Test in Europe, DATE*, 2013.

[5] M. Yasin, B. Mazumdar, O. Sinanoglu, and J. Rajendran, "Camoperturb: Secure ic camouflaging for minterm protection," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2016, pp. 1–8.

[6] R. W. Jarvis and M. G. Mcintyre, "Split manufacturing method for advanced semiconductor circuits," Mar. 27 2007, uS Patent 7,195,931.

[7] F. Imeson, A. Emtenan, S. Garg, and M. Tripunitara, "Securing computer hardware using 3d integrated circuit (IC) technology and split manufacturing for obfuscation," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 495–510.

[8] J. A. Roy, F. Koushanfar, I. L. Markov, J. Roy A., and I. Markov L., "Epic: Ending piracy of integrated circuits," *2008 Design, Automation and Test in Europe*, 2008.

[9] D. Sisejkovic and R. Leupers, *Logic Locking: A Practical Approach to Secure Hardware*. Springer Nature, 2022.

[10] P. Slpsk, P. K. Vairam, C. Rebeiro, and V. Kamakoti, "Karna: A gate-sizing based security aware eda flow for improved power side-channel attack protection," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.

[11] V. Samadi Bokharaie and A. Jahanian, "Power side-channel leakage assessment and locating the exact sources of leakage at the early stages of asic design process," *The Journal of Supercomputing*, pp. 1–26, 2022.

[12] B. Khaleghi, A. Ahari, H. Asadi, and S. Bayat-Sarmadi, "Fpga-based protection scheme against hardware trojan horse insertion using dummy logic," *IEEE Embedded Systems Letters*, vol. 7, no. 2, pp. 46–50, 2015.

[13] A. Raj, N. Avula, P. Das, D. Sisejkovic, F. Merchant, and A. Acharyya, "Deepattack: A deep learning based oracle-less attack on logic locking," in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2023, pp. 1–5.

[14] P. Chakraborty, J. Cruz, A. Alaql, and S. Bhunia, "Sail: Analyzing structural artifacts of logic locking using machine learning," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3828–3842, 2021.

[15] L. Alrahis, S. Patnaik, M. Shafique, and O. Sinanoglu, "Omla: An oracle-less machine learning-based attack on logic locking," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2021.

[16] A. Alaql, D. Forte, and S. Bhunia, "Sweep to the secret: A constant propagation attack on logic locking," in *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 2019, pp. 1–6.

[17] A. Alaql, S. Chattopadhyay, P. Chakraborty, T. Hoque, and S. Bhunia, "Lego: A learning-guided obfuscation framework for hardware ip protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[18] S. Amir, B. Shakya, X. Xu, Y. Jin, S. Bhunia, M. Tehranipoor, and D. Forte, "Development and evaluation of hardware obfuscation benchmarks," *Journal of Hardware and Systems Security*, 2018.

[19] K. Shamsi and Y. Jin, "Neos." [Online]. Available: https://bitbucket.org/kavehshm/neos/src/master/

[20] J. Rajendran, H. Zhang, C. Zhang, G. S. Rose, Y. Pino, O. Sinanoglu, and R. Karri, "Fault analysis-based logic encryption," *IEEE Transactions on Computers*, 2015.

[21] A. Baumgarten, A. Tyagi, and J. Zambreno, "Preventing ic piracy using reconfigurable logic barriers," *IEEE Design and Test of Computers*, 2010.

[22] S. Dupuis, P. S. Ba, G. Di Natale, M. L. Flottes, and B. Rouzeyre, "A novel hardware logic encryption technique for thwarting illegal overproduction and hardware trojans," in *Proceedings of the 2014 IEEE 20th International On-Line Testing Symposium, IOLTS 2014*, 2014.

[23] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, "Security analysis of logic obfuscation," in *DAC '12 Proceedings of the 49th Annual Design Automation Conference*, 2012.

[24] R. S. Chakraborty and S. Bhunia, "HARPOON: An obfuscation-based SoC design methodology for hardware protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2009.

[25] M. Zuzak, Y. Liu, and A. Srivastava, "Trace logic locking: Improving the parametric space of logic locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 8, pp. 1531–1544, 2020.

[26] Y. Xie and A. Srivastava, "Delay locking: Security enhancement of logic locking against ic counterfeiting and overproduction," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[27] N. G. Jayasankaran, A. Sanabria-Borbón, A. Abuellil, E. Sánchez-Sinencio, J. Hu, and J. Rajendran, "Breaking analog locking techniques," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 10, pp. 2157–2170, 2020.

[28] G. Takhar, R. Karri, C. Pilato, and S. Roy, "Holl: Program synthesis for higher order logic locking," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022, pp. 3–24.

[29] 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*, 2016.

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

[31] K. Z. Azar, H. M. Kamali, H. Homayoun, and A. Sasan, "SMT attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the SAT attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019.

[32] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin, "Appsat: Approximately deobfuscating integrated circuits," in *Proceedings of the 2017 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2017*, 2017.

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

[34] Y. Liu, M. Zuzak, Y. Xie, A. Chakraborty, and A. Srivastava, "Strong anti-sat: Secure and effective logic locking," in *2020 21st International Symposium on Quality Electronic Design (ISQED)*.

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

[36] M. Yasin, C. Zhao, and J. J. Rajendran, "Sfll-hls: Stripped-functionality logic locking meets high-level synthesis," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–4.

[37] 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 *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[38] A. Saha, S. Saha, S. Chowdhury, D. Mukhopadhyay, and B. B. Bhattacharya, "Lopher: Sat-hardened logic embedding on block ciphers," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[39] P. Chakraborty, J. Cruz, and S. Bhunia, "SAIL: Machine learning guided structural analysis attack on hardware obfuscation," in *Proceedings of the 2018 Asian Hardware Oriented Security and Trust Symposium, AsianHOST 2018*, 2019.

[40] D. Sisejkovic, F. Merchant, L. M. Reimann, H. Srivastava, A. Hallawa, and R. Leupers, "Challenging the security of logic locking schemes in the era of deep learning: A neuroevolutionary approach," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 17, no. 3, pp. 1–26, 2021.

[41] A. Alaql, M. M. Rahman, and S. Bhunia, "Scope: Synthesis-based constant propagation attack on logic locking," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 8, pp. 1529–1542, 2021.

[42] L. Alrahis, S. Patnaik, J. Knechtel, H. H. Saleh, B. Mohammad, M. Al-Qutayri, and O. Sinanoglu, "Unsail: Thwarting oracle-less machine learning attacks on logic locking," *IEEE Transactions on Information Forensics Security*, vol. 16, no. 2, pp. 2508–2523, 2021.

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

[44] J. Cruz, Y. Huang, P. Mishra, and S. Bhunia, "An automated configurable trojan insertion framework for dynamic trust benchmarks," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1598–1603.

[45] J. Cruz, P. Gaikwad, A. Nair, P. Chakraborty, and S. Bhunia, "A machine learning based automatic hardware trojan attack space exploration and benchmarking framework," in *2022 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 2022, pp. 1–6.

[46] J. Cruz, C. Posada, N. V. R. Masna, P. Chakraborty, P. Gaikwad, and S. Bhunia, "A framework for automated exploration of trojan attack space in fpga netlists," *IEEE Transactions on Computers*, vol. 72, no. 10, pp. 2740–2751, 2023.

[47] T. Schneider and A. Moradi, "Leakage assessment methodology: Extended version," *Journal of Cryptographic Engineering*, vol. 6, pp. 85–99, 2016.

[48] M. A. KF, V. Ganesan, R. Bodduna, and C. Rebeiro, "Param: A microprocessor hardened for power side-channel attack resistance," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2020, pp. 23–34.

[49] E. Trichina, "Combinational logic design for aes subbyte transformation on masked data," *Cryptology ePrint Archive*, 2003.

[50] M. Srivastava, P. Slpsk, I. Roy, C. Rebeiro, A. Hazra, and S. Bhunia, "Solomon: An automated framework for detecting fault attack vulnerabilities in hardware," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 310–313.

[51] I. Roy, C. Rebeiro, A. Hazra, and S. Bhunia, "Faultdroid: an algorithmic approach for fault-induced information leakage analysis," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 26, no. 1, pp. 1–27, 2020.

[52] P. B. Roy, P. Slpsk, and C. Rebeiro, "Avatar: Reinforcing fault attack countermeasures in eda with fault transformations," in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2022, pp. 417–422.

[53] P. Gaikwad, J. Cruz, P. Chakraborty, S. Bhunia, and T. Hoque, "Hardware ip assurance against trojan attacks with machine learning and post-processing," *ACM Journal on Emerging Technologies in Computing Systems*.

[54] "designs." [Online]. Available: https://github.com/jpsety/verilog_benchmark_circuits

[55] "Omla source code." [Online]. Available: https://github.com/DfX-NYUAD/OMLA

[56] T. Hoque, J. Cruz, P. Chakraborty, and S. Bhunia, "Hardware ip trust validation: Learn (the untrustworthy), and verify," in *2018 IEEE International Test Conference (ITC)*. IEEE, 2018, pp. 1–10.

[57] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. Rajendran, and O. Sinanoglu, "Provably-secure logic locking: From theory to practice," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1601–1618.

[58] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 4765–4774. [Online]. Available: http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf

[59] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should I trust you?": Explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*.

[60] ——, "Anchors: High-precision model-agnostic explanations," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[61] "scikit-learn." [Online]. Available: https://scikit-learn.org/stable/

[62] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[63] L. S. Shapley, "Notes on the n-person game—ii: The value of an n-person game," 1951.

[64] P. Subramanyan, S. Ray, and S. Malik, "Evaluating the security of logic encryption algorithms," in *Proceedings of the 2015 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2015*, 2015.