NORTHWESTERN UNIVERSITY

Enhancing Safety and Robustness for Mission-critical Systems with Formal Methods

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Engineering

By

You Li

EVANSTON, ILLINOIS

September 2023

ABSTRACT

Enhancing Safety and Robustness for Mission-critical Systems
with Formal Methods

You Li

Mission-critical systems are those imperative systems whose failures can result in catastrophic consequences. Traditional techniques, such as manual investigation and testing, cannot ensure the absence of errors and security vulnerabilities within these systems. This dissertation leverages formal methods to comprehensively examine several mission-critical systems and their essential components. For each of these systems, we either provide a rigorous mathematical proof of its correctness, or use concrete reasoning to identify the underlying security issues.

In the first part of this dissertation, we devise a method to prove that two systems are functionally equivalent modulo any timing differences. This method is based on symbolic model checking and induction, so it can generate a refinement mapping in addition to the verification result. In the second part, we propose an SAT-based attacking algorithm against sequential logic encryption, which does not require a working chip as the oracle. This study has established a new foundation for the security analysis of logic encryption.

The third part investigates whether logic encryption can resist I/O attacks on deep neural networks. Our findings suggest that existing defence mechanisms are susceptible to quantitative analysis techniques.

Acknowledgements

First and foremost, I would like to express my appreciation to my advisor, Prof. Hai Zhou. Prof. Zhou has always used his passion to inspire me to tackle research problems, his patience to support me during my Ph.D. journey, and his experience to encourage me to pursue an intellectual life.

Moreover, I would like to extend my gratitude to the professors in the EECS department at Northwestern University. I want to thank Prof. Yan Chen, who guided me in developing my research skills for network security. I want to thank Prof. Han Liu, who brought me into the field of statistical machine learning. I want to thank Prof. Qi Zhu, who enlightened me with his insights into the decision procedures of formal verification.

Furthermore, I am grateful to Kaiyu Hou and Guannan Zhao, my valuable collaborators and dear friends, for their pleasant companionship and productive teamwork over the years.

Meanwhile, I would like to thank all the members of the NuLogics group: Yunqi He, Shuyu Kong, Amin Rezaei, Yuanqi Shen, and Prof. Jia Wang. It has been a rewarding experience to work together to explore interesting research problems.

In addition, I appreciate Jingwen Chen, Prof. Tiansheng Lu, and Prof. Weikang Qian at Shanghai Jiao Tong University for encouraging me to apply for Ph.D. programs. I also appreciate the generous patrons who funded my Ph.D. study through fellowships.

Finally, I would like to thank my parents for their understanding and love.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."

- Edsger W. Dijkstra

Mission-critical systems are those imperative systems whose failures could lead to catastrophes [**41**]. Examples of mission-critical systems include but are not limited to autonomous vehicles, flight control systems, power grids, and data centers. The complexity of modern mission-critical systems is consistently growing. As a result, they are typically comprised of multiple key components, such as operating systems, network devices, databases, integrated circuits, and AI modules.

Traditionally, the *correctness* of mission-critical systems and their sub-modules are verified through manual investigation or testing. However, such methods are often inexhaustive on complex systems and thus cannot ensure the absence of errors. On the other hand, the *security* study on mission-critical systems has devolved into a perpetual cat-and-mouse chase between attackers and defenders. Due to the lack of formal security guarantees, research outcomes in this field are usually transitional and short-lived.

Formal methods refer to those mathematically rigorous reasoning techniques, such as model checking, constraint solving, mathematical deduction, and abstract interpretation. Given a formal specification of the underlying system and a set of correctness properties,

a formal verification algorithm typically returns a proof of correctness if the specification satisfies all the properties, or a counterexample if it violates any of the properties. In this dissertation, we apply formal methods to verify and analyze the correctness, safety, and robustness of mission-critical systems and their sub-modules. In each of the subsequent chapters, we concentrate on two common problems surrounding formal methods:

• How to improve the *scalability* of formal methods when applied to large and complex mission-critical systems;

• How to select appropriate *abstract domains* for the underlying mission-critical systems to improve the precision and effectiveness of formal methods.

In the following, we provide a brief overview of the subsequent chapters.

1.1. Sequential Equivalence Checking

In high-level design explorations, many useful optimizations transform a circuit into another with different operating cycles for a better trade-off between performance and resource usage. How to efficiently check their equivalence is critical and challenging since most existing equivalence checkers are designed for cycle-accurate circuits.

Chapter 2 presents SE3, an efficient sequential equivalence checker without assumption on cycle-accuracy, latch mapping, or I/O interface of the checked circuits. It proves the equivalence of two circuits by computing an equivalence relation between the states of the two circuits and utilizes syntax abstraction to accelerate this process. Experimental results show that SE3 is significantly faster than state-of-the-art sequential equivalence checking algorithms.

This exact checker can also be used to confirm the correctness of mission-critical systems. From the initial design specification to the final implementation, the development process of a complex system goes through multiple abstraction levels. SE3 can be adopted to ensure that functional equivalence is maintained at any refinement step between two consecutive abstraction levels. Additionally, SE3 generates an equivalence relation, or refinement mapping, alongside the verification result. This information can facilitate design and verification engineers to locate errors or gain insights into the design.

1.2. Oracle-less SAT-based Attack on Logic Encryption

Integrated circuits are indispensable components of most mission-critical systems. Logic encryption is a promising approach to protecting the intellectual properties of integrated circuits, preventing hardware trojans, and securing sensitive data. In Chapter 3, we thoroughly evaluate the security of sequential logic encryption using formal analysis. Our findings refute the belief that an oracle circuit is required to launch an I/O attack on sequential logic encryption.

Existing I/O attacks against logic encryption require query access to an oracle circuit. We presents OLSAT, an oracle-less logic decryption method. OLSAT only requires a very high-level specification of the victim chip, usually provided for marketing or technical support. An essential enabler of OLSAT is a synthesis-based sequential logic decryption algorithm called LIM. This algorithm introduces only a minimal overhead in every iteration and allows the flexibility to explore the error matrix along both dimensions. Experiments show that OLSAT can efficiently attack logic-encrypted benchmarks without

oracle access. Besides, LIM can solve 20% more ISCAS'89 benchmarks than state-of-the-art sequential logic decryption algorithms.

1.3. Security Analysis for Logic Encryption on Deep Neural Networks

Deep Neural Networks (DNNs) have achieved tremendous success in various applications. They are also integrated into numerous mission-critical systems. However, DNNs are susceptible to model piracy and adversarial attack if a malicious end-user has full access to the model parameters. Recently, a logic encryption scheme called HPNN has been proposed. HPNN utilizes hardware root-of-trust to prevent end-users from accessing the model parameters.

Chapter 4 exploits quantitative formal analysis to investigate whether logic encryption is secure on deep neural networks. Specifically, we present a systematic I/O attack that combines algebraic and learning-based approaches. Our attack incrementally extracts key values from the network to minimize sample complexity. Besides, it employs a validation and correction procedure to ensure the correctness of the extracted key values. Our experiments demonstrate the accuracy and efficiency of the presented attack on large networks and complex architectures. Consequently, we conclude that HPNN and a broad category of other logic encryption schemes are insecure on deep neural networks.

CHAPTER 2

Sequential Equivalence Checking for Non-Cycle-Accurate Design Transformations

2.1. Introduction

With the growing demand for high-performance integrated circuits, design engineers and optimization tools tend to perform more aggressive sequential transformations to meet the timing and throughput goals. For instance, retiming techniques move logic across flip-flops to meet timing constraints; pipelining techniques introduce additional pipeline stages, trading off latency for throughput; pre-computation techniques prepare the results earlier to remove their computations from the critical path. Other examples of sequential transformations include unrolling, resource reallocation, clock gating, and memory partitioning. A design after such transformations may no longer be cycle-accurate with the original one. Additionally, there may not exist a one-to-one latch mapping between the two circuits because these transformations can change the functionalities of the flip-flops.

Recently, adaptive pipelining [**23, 61**] is proposed to enable dynamic scheduling, *i.e.*, the period of each iteration is a variable depending on the inputs and the previous executions. Moreover, the latency-intensive (LI) design methodology has emerged to tolerate the arbitrary timing of individual hardware modules. Under the relaxed timing requirements, engineers can design highly customized hardware modules with *variable latencies*.

Sequential equivalence checking is the key enabler of sequential design transformations. It finds application in various stages of the IC design flow [**76**], for example, in checking if an RTL model conforms to the functional specification it seeks to implement or determining if a derivative RTL or gate-level model is functionally equivalent to a validated model. Co-simulation is also widely used in such scenarios but has limited functional coverage. In comparison, formal sequential equivalence checking is a comprehensive and rigorous approach that allows verification engineers to prove the consistency of two designs over any number of cycles.

Several formal sequential equivalence checking algorithms have been proposed [**17, 45, 7, 80, 64**], yet nearly all of them have restricted use cases. Some algorithms assume that a complete latch mapping is provided by the user [**17**]. Others assume that the two designs are structurally similar and that pairs of internal nodes exist that have identical functionalities [**45, 7, 80**]. Kairos [**64**] can handle almost all types of transformations, but it requires both designs under verification to follow a *valid-ready* interface protocol.

In this chapter, we propose SE3 (**S**yntax-**E**ncoded **S**tuttering **E**quivalence Checking for **SE**quential Circuits), a general and efficient algorithm based on symbolic model checking. SE3 formulates the sequential equivalence checking problem for any transformations [**65**] as checking whether the output sequences of the two designs under verification are alignable given a set of corresponding initial states and any input sequences. Nevertheless, the consequent formula involves an alternation of universal and existential quantifiers. Eliminating the internal existential quantifier can result in a formula that is exponential in size.

SE3 tackles this issue by searching for a reversed inductive invariant on a product machine of two designs, bypassing the need for quantifier elimination. Additionally, SE3 maintains a frame structure similar to IC3 [**10**]. This allows the algorithm to learn new clauses incrementally and enables high flexibility in expressing complex equivalence relations. Moreover, SE3 leverages syntax abstraction [**37**] to capture the equivalence relations among the internal nodes. Thus, the algorithm concentrates on high-level, coarse-grained relations at the beginning of the verification process and gradually shifts to finer-grained relations through iterative refinement.

We demonstrate the capability and efficiency of SE3 with a case study and a benchmark suite. SE3 is significantly faster than Kairos on equivalent test cases and scales well regarding word size. Besides, SE3 is capable of discovering concise and essential inductive invariants, which may guide design engineers to understand the nature of the transformations or locate errors in future transformations.

Our main contributions are:

• We devise a formal property to determine whether two designs are observational equivalent modulo stuttering;

• We utilize the reversed inductive invariant to bypass the quantifier alternation issue within the property;

• We adapt the IC3 symbolic model checking framework to achieve incremental verification;

• We embed syntax abstraction to the stuttering equivalence checking algorithm, so that SE3 can discover equivalence relations among state variables and internal nodes at various granularities;

Listing 1 Euclidean Algorithm A

```
procedure gcd_A(x, y)
    while (x − y) ≠ 0 do
        x ← (x − y) > 0 ? (x − y) : x
        y ← (y − x) > 0 ? (y − x) : y
    output x
```

Listing 2 Euclidean Algorithm B

```
procedure gcd_A(x, y)
    while (x − y) ≠ 0 do
        x ← (x − y) > 0 ? (x − y) : y
        y ← (x − y) > 0 ? y : x
    output x
```

• We evaluate SE3 against state-of-the-art sequential equivalence checking algorithms.

2.2. Background

2.2.1. A Running Example

We use the Euclidean algorithm as the running example throughout this chapter. To compute the greatest common divisor of two integers, the algorithm iteratively subtracts the smaller integer from the greater one until they become equal. Listing 1 shows an RTL implementation of the algorithm. Two subtractors are initiated in parallel, and only one integer is updated depending on the results. Suppose an engineer then decides to allocate only one subtractor and adjusts the implementation accordingly. As shown in Listing 2, only one subtractor is initiated in a clock cycle.

The two implementations are identical in function but different in timing. If x equals 6 and y equals 2 initially, both circuits will take 2 cycles to reach convergence. On the other hand, if the initial values of x and y are 6 and 10, gcd_A will take 3 cycles $\langle(6,10),(6,4),(2,4),(2,2)\rangle$, while gcd_B will take 6 cycles

$\langle(6,10),(10,6),(4,6),(6,4),(2,4),(4,2),(2,2)\rangle$. It can be seen that gcd_B has a variable latency relative to gcd_A, and there exists no latch mapping or equivalent internal nodes between the two designs.

2.2.2. Preliminaries

We consider standard first-order logic. A *term* is a variable or a *function* symbol. A *predicate* is an expression applied to a tuple of terms and evaluates to a Boolean value. An *atom* is a Boolean variable or a *predicate* symbol. The terms with non-Boolean values are also referred to as *words*. A *formula* is built over atoms with propositional logic. A *literal* is an atom or its negation. A *cube* is a conjunction of literals, while a *clause* is a disjunction of literals.

A transition system M is defined as a tuple $\langle X, I, T\rangle$, where X is a set of *state variables*, X' is the corresponding set of next-state variables, $I(X)$ is a formula representing the *initial condition*, and $T(X, X')$ is a formula representing the *transition relation*. It is a common practice to model input variables as additional state variables [**1, 19, 37**] such that $X = X_{state} \cup X_{in}$. The next-state variables X'_{in} are unconstrained or controlled by an external specification. A state s is a full assignment to all state variables. We write $s \models \phi$ if s satisfies a formula ϕ modulo the underlying theory, and s is a ϕ-state. A formula ψ *implies* another formula ϕ, $\psi \Rightarrow \phi$, if all state satisfying ψ also satisfies ϕ. A finite or infinite *path* is a state sequence such that the first state is an I-state and all consecutive steps satisfy $T(X, X')$. A path is a ϕ-path if all states along the path are ϕ-states. A reachable *lasso-shaped path* [**11**] (a *lasso*, in shorthand) is a finite run from an initial state followed by a loop.

A sequence σ *stutters* at step k if it keeps the same value for indices k and $k+1$. For instance, the sequence $\langle a, b, b, c, \cdots \rangle$ stutters at step 2. We define $\natural\sigma$ the *stutter-free sequence* of σ, which eliminates all stuttering steps in σ. We let $\sigma \simeq \rho$ mean $\natural\sigma = \natural\rho$ [**1**]. For instance, $\natural\langle a, b, b, c\rangle = \langle a, b, c\rangle$, and $\langle a, b, b, c\rangle \simeq \langle a, b, c, c\rangle$.

2.2.3. The IC3 Model Checking Algorithm

IC3 [**10**] is the state-of-the-art symbolic model checking algorithm for hardware verification. Given a safety property $P(X)$, IC3 checks whether $M \models P$, *i.e.*, all paths of M are P-paths. In this regard, it tries to find an *inductive invariant*, Inv, such that

$$(a)\ I \Rightarrow Inv,\ (b)\ Inv \wedge T \Rightarrow Inv',\ (c)\ Inv \Rightarrow P. \tag{2.1}$$

Once an Inv is found, IC3 completes the proof.

During its execution, IC3 maintains a sequence of *frames* $F_0(X),\ \cdots, F_k(X)$ such that

$$\forall i < k :\ (a)\ F_0 = I,\ (b)\ F_i \wedge T \Rightarrow F'_{i+1},\ (c)\ F_i \Rightarrow P. \tag{2.2}$$

Additionally, the algorithm ensures that each frame is a conjunction of clauses, and $clauses(F_i) \subseteq clauses(F_{i+1})$, where $clauses(F_i)$ denotes the set of all clauses that constitute F_i. The algorithm attempts to learn new clauses from the reachability information of the system and add them to the sequence of frames until one of the frames is proved to be an inductive invariant.

In the following, we give a high-level description of the algorithm. At the beginning of every iteration, k is incremented by 1, and a new frame $F_k = P$ is attached to the sequence. The algorithm queries

$$SAT?(F_k \wedge T \wedge \neg P') \tag{2.3}$$

for a new bad state $s \in F_k$. If (2.3) is satisfiable, a new *proof obligation* $\langle k, s \rangle$ is added to a priority queue. Whenever the priority queue is non-empty, IC3 pops the top element of the queue and queries

$$SAT?(F_i \wedge T \wedge s'). \tag{2.4}$$

If (2.4) is satisfiable, there must exist another bad state $t \in F_i$, which is a predecessor of s. A new obligation $\langle k-1, t \rangle$ is added to the queue. If (2.4) is unsatisfiable, s can be safely excluded from F_i without affecting the conditions of the frames. Thus, a new clause $c = \neg s$ is conjoined to F_{i+1}. The unsatisfiability of (2.4) guarantees that c is *relatively inductive* to F_i, *i.e.*, $F_i \wedge c \wedge T \Rightarrow c'$ is a tautology.

If (2.3) is unsatisfiable, all states in F_k must be more than 1 step away from bad states. Before IC3 starts a new iteration, it pushes every $c \in clauses(F_i)$ to $clauses(F_{i+1})$ if c is relatively inductive to F_{i+1}. During this process, if two consecutive frames F_i and F_{i+1} become identical, they must satisfy all the conditions of an inductive invariant, *i.e.*, $Inv = F_i$, and that finishes the algorithm.

2.2.4. Syntax Abstraction

Syntax abstraction [**37**] creates an abstract space using a subset of the terms present in the original model. An abstract state is a *partition assignment* which captures Boolean values of atoms and equality relations among the words of each sort. For example, in Listing 1, the concrete state $(6, 3)$ may correspond to the abstract state $((x-y) > 0) \wedge \neg((y-x) > 0) \wedge \{x \mid y, x-y \mid y-x\}$, where vertical bars divide terms into equivalence classes.

Syntax abstraction removes irrelevant bit-level details, thus facilitating the reasoning of equivalence relations at a coarse granularity. An abstract space it creates can be iteratively refined in a *counterexample-guided abstraction refinement* (CEGAR) fashion: once a spurious counterexample is found, new terms are introduced to eliminate it. Hence, syntax abstraction can be closely integrated with a model checking algorithm, where the former provides the domain for reasoning and the latter provides the guidance for refinement.

2.3. Problem Definition and Analysis

2.3.1. Problem Definition

Our objective is to check the observational equivalence of two systems, *i.e.*, whether their externally observable behaviors are always identical. More specifically, we aim to devise an efficient symbolic model checking algorithm for the following property: starting from any pair of corresponding initial states, the stutter-free output sequences produced by the two systems are identical given the same input sequence.

2.3.2. Product Machine for Stuttering

As the first step to solving the problem, we devise an automated reasoning mechanism that checks whether two output sequences are equivalent modulo stuttering. It is based on the observation that inserting stuttering steps to the faster sequence is the *dual* of eliminating stuttering steps from the slower sequence. Hence, we build a product machine for stuttering, $M_\times$, to mimic the process of inserting stuttering steps. The product machine converts the problem of checking alignability [**65**] to the problem of finding a feasible auxiliary input sequence. Denote the two systems under verification as M_A and M_B, where M_A runs no slower than M_B. The state space $S_\times$ of $M_\times$ is a Cartesian product $S_A \times S_B$, and every state $s \in S_\times$ is a pair (u, v) where $u \in S_A$ and $v \in S_B$. The transition relation $T_\times$ is composed of two branches, T_{syn} and T_{stu}, both of which are also product machines. T_{syn} specifies the behavior that M_A and M_B move *syn*chronously: $T_{syn}(u, v, u', v') \triangleq T_A(u, u') \wedge T_B(v, v')$, while T_{stu} specifies the behavior that M_A *stu*tters and M_B moves forward: $T_{stu}(u, v, u', v') \triangleq (u = u') \wedge T_B(v, v')$. $T_\times$ uses a dummy input, sel, to select its next state from the two branches[1]:

$$\begin{aligned} T_\times(u, v, sel, u', v') \triangleq \vee\ &((sel = 0) \wedge T_{syn}(u, v, u', v')) \\ &\vee ((sel = 1) \wedge T_{stu}(u, v, u', v')). \end{aligned} \tag{2.5}$$

We define the *observational equivalence property* to be $P_\times(u, v) \triangleq valid \Rightarrow (u_{out} = v_{out})$. The *valid* signal is only necessary when the output registers can turn into an unstable observable state. We use $(i_a, i_b) \in I_\times$ to denote that two initial states from both

[1]By adding a third branch, our method can be generalized to the case that both systems can be faster than the other.

systems are related by the *initial correspondence* $I_\times$. By default, $I_\times$ contains the pair of reset states. For some applications, it might be more *convenient* to directly initialize the corresponding pairs of state variables with the same input values. In our running example, this means setting $I_\times$ to the formula $(x_A = x_B) \wedge (y_A = y_B)$. Advanced users can also write customized specifications for $I_\times$ and $P_\times$.

From a specific initial state, the state sequence produced by M_A or M_B is either a finite or an infinite path. We convert all finite paths to infinite ones by adding a *self-loop* to the final states. Notice that final states can usually be distinguished symbolically with termination conditions, *valid* signals, or as the deadlock states in deadlock-free systems.

Facilitated by $M_\times$, we can check if two systems are *observational equivalent modulo stuttering* by checking the following condition:

Definition 1. *Two systems M_A and M_B are observational equivalent modulo stuttering if and only if there exists a $P_\times$-lasso on $M_\times$ from every pair $(u_1, v_1) \in I_\times$.*

Definition 1 suggests a naive solution. Given $M_\times$ and $P_\times$, one can check if they satisfy the correctness property:

$$\begin{aligned} &\forall (u_1, v_1) \in I_\times, \forall k > 0, \ \exists sel_1, \ \cdots, sel_k : \\ &\qquad T_\times(u_i, v_i, sel_i, u_{i+1}, v_{i+1}) \Rightarrow (u_{i+1}, v_{i+1}) \in P_\times. \end{aligned} \tag{2.6}$$

Intuitively, the property requires that for all pairs within the initial correspondence, there exists a fair path such that all pairs along the path satisfy observational equivalence. The existence of a lasso-shaped fair path can be verified by a fairness checking

algorithm [**20**]. Nevertheless, an exponential number of pairs may exist in $I_\times$, and enumerating all those pairs is computationally intractable even when the fairness checking algorithm is incremental. Another way to deal with the quantifier alternation problem is to eliminate all the internal existential quantifiers. However, an equisatisfiable formula without existential quantifiers can be exponential in size.

2.3.3. Inductive Invariant for Equivalence modulo Stuttering

A conventional *inductive invariant* proves that a transition system always satisfies a safety property P by showing $\neg P$-states are never reachable from the initial states. We devise a new type of inductive invariant to prove that two systems are observational equivalent modulo stuttering:

Definition 2. *$Inv_\times$ is an inductive invariant modulo stuttering for $M_\times$ and $P_\times$ if it satisfies all of the following conditions:*

- $I_\times \Rightarrow Inv_\times$, *(2.7a)*
- $\forall s \in Inv_\times : (s \wedge T_{syn} \Rightarrow Inv'_\times) \vee (s \wedge T_{stu} \Rightarrow Inv'_\times)$, *(2.7b)*
- $Inv_\times \Rightarrow P_\times$. *(2.7c)*

Lemma 1. *M_A and M_B are observational equivalent modulo stuttering if and only if there exists an $Inv_\times$ for $M_\times$ and $P_\times$.*

Proof. *Only-if part:* We show the existence constructively. From Definition 1, if the two systems are equivalent modulo stuttering, there exists a $P_\times$-lasso from every state $s_1 \in I_\times$. Let $Inv_\times$ be the formula that contains exactly all pairs along these lassos.

If part: Consider an arbitrary $s_1 \in I_\times$. From (2.7a), $s_1 \in Inv_\times$; recursively applying condition (2.7b) shows that there exists an $Inv_\times$-lasso from s_1; all pairs along the lasso are $P_\times$-pairs (2.7c). Because a $P_\times$-lasso exists for every pair in $I_\times$, M_A and M_B are equivalent modulo stuttering. □

Various methods can infer inductive invariants for a transition system [**10, 53**]. Nevertheless, due to the *disjunction operator* in (2.7b), these methods cannot be applied directly for the inference of $Inv_\times$. For example, a key procedure in IC3 is to check if a clause c is relatively inductive to a frame F_i, *i.e.*, $F_i \wedge c \wedge T \Rightarrow c'$ is a tautology. It can be verified that if two clauses c_1 and c_2 are both relatively inductive to F_i, their conjunction $c_1 \wedge c_2$ is also relatively inductive to F_i. This property allows IC3 to learn new clauses incrementally while maintaining the structure of the frame sequence. However, if two clauses both meet condition (2.7b), their conjunction may not meet the same condition. This limitation disallows us from adopting existing inductive invariant inference algorithms to our problem.

2.3.4. The Reversed Approach

To address the above issue, we tackle the problem from the opposite direction. In specific, we switch the roles of $I_\times$ and $\neg P_\times$. Additionally, we reverse the directions of both T_{syn} and T_{stu} by switching their current states and next states, yielding T°_{syn} and T°_{stu}. Thus, the reversed inductive invariant for observational equivalence modulo stuttering can be defined as follows:

Definition 3. *$Inv^\circ_\times$ is a reversed inductive invariant modulo stuttering for $M_\times$ and $P_\times$ if it satisfies all of the following conditions:*

- $\neg P_\times \Rightarrow Inv^\circ_\times$, *(2.8a)*
- $\forall s_a, s_b \in Inv^\circ_\times$:

$$(s_a \wedge T^\circ_{syn} \wedge s') \wedge (s_b \wedge T^\circ_{stu} \wedge s') \Rightarrow (s' \in Inv^{\circ\prime}_\times), \quad \textit{(2.8b)}$$

- $Inv^\circ_\times \Rightarrow \neg I_\times$. *(2.8c)*

Notice that there is no longer a disjunction operator in Definition 3. The next lemma states the correlation between $Inv_\times$ and $Inv^\circ_\times$.

Lemma 2. *When $Inv^\circ_\times = \neg Inv_\times$, $Inv_\times$ is an inductive invariant modulo stuttering if and only if $Inv^\circ_\times$ is a reversed inductive invariant modulo stuttering.*

Proof. (2.7a) (resp. (2.7c)) is the contrapositive statement of (2.8c) (resp. (2.8a)). The negation of (2.7b) is equisatisfiable to: $\exists s \in Inv_\times,\ s'_a, s'_b \in \neg Inv_\times : (s \wedge T_{syn} \wedge s'_a) \wedge (s \wedge T_{stu} \wedge s'_b)$, which in turn is equisatisfiable to the negation of (2.8b). Hence, (2.7) and (2.8) are equisatisfiable when $Inv^\circ_\times = \neg Inv_\times$. □

Theorem 3. *M_A and M_B are observational equivalent modulo stuttering if and only if there exists an $Inv^\circ_\times$ for $M_\times$ and $P_\times$.*

Proof. Readily follows from Lemma 1 and Lemma 2. □

Intuitively, the existence of $Inv^\circ_\times$ guarantees that a counterexample tree, whose leaf nodes are all $\neg P_\times$-states and whose root node is an $I_\times$-state, cannot exist.

Based on the above observations, we devise a method to search for a reversed inductive invariant modulo stuttering. We embed our method into IC3's general framework

(Section 2.2.3). In the remainder of this section, we highlight some key procedures we adapt from IC3. An overall description is left in the next section.

• Similar to (2.2), our method maintains a sequence of frames, where $F_0^\circ = \neg P_\times$ and $F_i^\circ \Rightarrow \neg I_\times$. One exception is (2.2b). If two states sharing the same parent state are both F_i°-states, that parent state must be an F_{i+1}°-state:

$$\begin{aligned} &\forall s_a, s_b \in F_i^\circ : \\ &\qquad (s_a \wedge T_{syn}^\circ \wedge s') \wedge (s_b \wedge T_{stu}^\circ \wedge s') \Rightarrow (s' \in {F_{i+1}^\circ}'). \end{aligned} \tag{2.9}$$

Hence, F_i° is an over-approximation of those states which are at most i steps away from $\neg P_\times$.

• Our method extracts new proof obligations through the query:

$$SAT?\ (F_k^\circ \wedge T_{syn}^\circ \wedge I_\times') \wedge (F_k^{\circ *} \wedge T_{stu}^\circ \wedge I_\times'). \tag{2.10}$$

Notice that F_k° and $F_k^{\circ *}$ represent two different sets of variables. If it is satisfiable, two new states $s \in F_k^\circ$ and $s^* \in F_k^{\circ *}$ are extracted and added to the queue of proof obligations.

• When the original IC3 discharges a proof obligation, if the query to (2.4) is satisfiable, a predecessor t of s is extracted and added to the priority queue. We mimic that procedure by querying

$$SAT?\ (F_i^\circ \wedge T_{syn}^\circ \wedge s') \wedge (F_i^{\circ *} \wedge T_{stu}^\circ \wedge s'). \tag{2.11}$$

This procedure allows our method to find candidate bad states that are more than 1 step away from $I_\times$, thus diversifying the clauses in the sequence of the frames.

• When a query to (2.11) is unsatisfiable, the newly produced clause $c = \neg s$ must be *relatively inductive modulo stuttering* to F_i, *i.e.*, for any state $\bar{s}$,

$$(F_i^\circ \wedge T_{syn}^\circ \wedge \bar{s}') \wedge (F_i^{\circ *} \wedge T_{stu}^\circ \wedge \bar{s}') \Rightarrow (\bar{s}' \in c'). \tag{2.12}$$

Hence, c can be conjoined to F_{i+1}° without violating any of the frame structure conditions including (2.9).

2.4. Algorithm

2.4.1. The SE3 Algorithm

The core procedures of SE3 are displayed in Algorithm 1. SE3 is designed to retain several desirable features of IC3: *a)* organizing clauses in a sequence of relatively inductive frames to enable fully incremental verification (Line 18; 30); *b)* generalizing clauses to accelerate convergence (Line 29); *c)* propagating clauses to increase the chance of finding an inductive invariant (Line 16-18); and *d)* being compatible with counterexample-guided abstraction refinement workflows (Line 9, 26, 11-12). From a high-level perspective, SE3 adapts the general framework of IC3 (Section 2.2.3) and combines it with the reversed invariant finding strategy [**74**] as well as the implicit abstraction technique [**18**]. Due to the similarity, we build our correctness proof on top of that in the IC3 paper [**10**]:

Theorem 4. *Upon termination,* CHECK *returns* `True` *if and only if the systems in* $M_\times$ *are observational equivalent modulo stuttering.*

Proof. *(Sketch)* According to Theorem 3, the decision of observational equivalence modulo stuttering is reduced to the searching of an $Inv_\times^\circ$. When both invocations of

Algorithm 1 SE3 Algorithm for Stuttering Equivalence Checking

```
1: procedure CHECK(I_×, T°_×, P_×) → bool:
2:     if SAT?(I_× ∧ ¬P_×) or SAT?(I'_× ∧ T°_× ∧ ¬P_× ∧ ¬P*_×) then
3:         return False                                   ▷ concrete counterexample found
4:     F°_0 ← ¬P_×
5:     k ← 1, F°_k ← ¬I_×
6:     while True do
7:         while SAT?(I'_× ∧ T°_× ∧ F°_k ∧ F°*_k) do
8:             ⟨s, s*⟩ ← the extracted states inside F°_k and F°*_k
9:             ⟨ŝ, ŝ*⟩ ← the abstraction of ⟨s, s*⟩ as partitions of terms
10:            if not (BLOCK(k, ŝ) or BLOCK(k, ŝ*)) then
11:                if the abstract counterexample is spurious then
12:                    refine the abstract space with more terms
13:                else return False                      ▷ concrete counterexample found
14:        k ← k + 1, F°_k ← ¬I_×
15:        for i ← 1 to k − 1 do
16:            for each clause c ∈ F°_i do                ▷ clause propagation
17:                if not SAT?(¬c' ∧ T°_× ∧ F°_i ∧ F°*_i) then
18:                    add c to F°_{i+1}
19:            if F°_i = F°_{i+1} then
20:                return True                            ▷ inductive invariant found, equivalence proved

21: ▷ Displayed as recursive function for simplicity; actual implementation uses priority queue. ◁
22: procedure BLOCK(i, ŝ) → bool:
23:     if i = 0 or ŝ ⇒ ¬P_× then return False            ▷ bad state reached
24:     while SAT?(ŝ' ∧ T°_× ∧ F°_{i−1} ∧ F°*_{i−1}) do
25:         ⟨t, t*⟩ ← the extracted states inside F°_{i−1} and F°*_{i−1}
26:         ⟨t̂, t̂*⟩ ← the abstraction of ⟨t, t*⟩ as partitions of terms
27:         if not (BLOCK(i − 1, t̂) or BLOCK(i − 1, t̂*)) then
28:             return False                               ▷ blocking fails on both branches
29:     c ← the clause generalized from ¬ŝ
30:     add c to F°_1, ··· , F°_i
31:     return True                                        ▷ blocking succeeds
```

BLOCK return `False` at Line 10, there is an abstract counterexample where *all* paths lead to $\neg P_\times$ states. If CHECK returns `False` at Line 3, a trivial counterexample is found; if it returns `False` at Line 13, a concrete counterexample corresponding to the abstract one is found. In both cases, a valid $Inv^\circ_\times$ cannot exist. On the other hand, if CHECK

returns `True`, Line 20 must have been reached. This implies that two consecutive frames are identical and a valid $Inv^{\circ}_{\times}$ is found. □

In general, if the word-level models under verification reside in an infinite space, there are no complexity bounds for a model checking algorithm. However, if all variables in the models are constant-sized bit vectors, SE3 will eventually terminate because both the abstraction refinement process and the frame sequence converging process are strictly monotonic.

2.4.2. Syntax-Guided Abstraction and Refinement

An appropriate abstract space can guide the model checking algorithm to find a concise and essential inductive invariant. Our insight is that an equivalence relation can usually be expressed by *terms* within either data-intensive or control-intensive models. Besides, the clause and frame structure of IC3 allows the expression of not only one-to-one mappings between terms but also the relations among terms described by logic formulas acrossing time domains.

The SE3 algorithm maintains a set of terms that are currently used to depict the abstract space. Once a counterexample is found, it is validated through SMT queries [**37**]. If it is confirmed to be spurious, SE3 investigates the unsat core and adds new terms to the set to eliminate the counterexample. SE3 prioritizes state variables and I/O variables over internal nodes and constants. If none of the above work, SE3 will also attempt to add primed variables, trying to capture the correlations between clock cycles. With this strategy, SE3 iteratively refines the granularity of its reasoning domain until an inductive invariant or a concrete counterexample is found.

2.5. Experimental Results

2.5.1. Experimental Setup

We implemented the SE3 algorithm in Python, using Boolector [**57**] as the backend SMT solver. Our implementation takes two RTL-level or gate-level Verilog designs as the inputs. It supports 2-branch (one design being no slower than the other) and 3-branch (no timing constraint) modes.

All evaluations are conducted on a Linux machine with a 3.2GHz CPU. Each instance runs on a single thread. We set a 4GiB memory limit and a 7,200-second timeout for all experiments.

In the first part of our evaluation, we compare the performance of SE3 and Kairos [**64**] and investigate their underlying mechanisms in a case study. We choose nuXmv [**18**] and AVR [**36**] as the backend model checkers for Kairos, because they won the first and the third place in the prestigious HWMCC'20 contest [**66**]. Both nuXmv and AVR are word-level safety property checkers based on implicit predicate abstraction and syntax abstraction, respectively.

For the case study, we manually write 4 RTL-level Verilog implementations of our running example. Fig. 2.1 shows their pseudo-code. Among those, (a) and (b) correspond to Listing 1 and Listing 2 respectively, while (c) is an alternative implementation of the running example. We intentionally add a fourth implementation, (d), which is not equivalent modulo stuttering to any of the rest implementations.

In the second part of our evaluation, we assess the capability and characteristics of SE3 in a realistic setting. We leverage a high-level synthesis (HLS) tool, Xilinx Vivado HLS, to

generate pairs of non-cycle-accurate RTL designs. The HLS workflow is a combination of software compilation and hardware optimization. Because a commercial HLS tool contains almost all kinds of sequential transformations, it is an excellent source to emulate realistic design transformations and generate a variety of designs with guaranteed correctness.

We select 7 high-level hardware specifications from the HLSynth benchmark suite [**59**] for HLS. Our selection is based on two rules: *i)* the benchmark is a standalone module, and *ii)* at least 6 different RTL-level implementations with different timing can be generated from the benchmark using Vivado HLS. Table 2.2 provides a summary of those benchmarks. We generate 3 specifications for each benchmark with the word size set to 8, 16, and 32 bits, respectively. For each specification, we generate 6 designs with different timing, thus yielding a total of 15 pairs of designs. We utilize HLS pragmas, including pipeline, initiation interval, resource allocation, latency, unroll, flatten, merge, partition, balance, etc., to control a design's timing. Eventually, we obtained a total of 315 pairs as our test cases. Because nuXmv is almost always faster than AVR when paired with Kairos, we only compare with *Kairos/nuXmv* in the second part.

2.5.2. Case Study

Table 2.1 compares the execution time *(s)* of Kairos and SE3 on the test cases shown in Fig. 2.1. The first two test cases check designs that are equivalent modulo stuttering. Even though Kairos uses word-level model checkers as its backend, its performance deteriorates quickly as the word size grows. On the contrary, the execution time of SE3 grows more slowly.

Test Case	Algorithm	3bit	4bit	5bit	6bit	8bit	16bit	32bit
(a) vs. (b) equiv.	*Kairos/nuXmv*	0.41	4.13	163.3	–	–	–	–
	Kairos/AVR	18.17	323.4	–	–	–	–	–
	SE3	0.04	0.05	0.06	0.06	0.07	0.19	0.51
(a) vs. (c) equiv.	*Kairos/nuXmv*	0.44	3.25	315.0	–	–	–	–
	Kairos/AVR	10.9	96.06	–	–	–	–	–
	SE3	0.23	0.29	0.44	0.80	1.34	4.42	13.63
(a) vs. (d) non-equiv.	*Kairos/nuXmv*	0.05	0.07	0.08	0.11	0.13	0.30	0.74
	Kairos/AVR	1.02	4.07	17.99	38.96	239.1	294.1	570.2
	SE3	1.32	1.77	2.53	3.27	4.77	8.24	36.75

Table 2.1. Comparing SE3 and Kairos for efficiency under different word sizes.

We believe that Kairos is over-conservative when aligning two sequences. As illustrated in Fig. 2.2(a), Kairos enforces both designs to execute when none of them or both of them reach a *valid* state; it stalls the faster design if exactly one of them reaches a *valid* state. Such an alignment pattern hinders the underlying model checker from finding a concise inductive invariant.

The last test case in Table 2.1 checks a pair of non-equivalent designs. *Kairos/nuXmv* turns out to be faster than SE3 in finding a counterexample. It is because Kairos prunes out all alignment patterns except one. In this regard, Kairos and SE3 are complementary to each other. A hybrid sequential equivalence checking engine can run the two algorithms in parallel to achieve optimal performance.

Fig. 2.2(c) and 2.2(d) showcase the inductive invariants for gcd_A vs. gcd_B and gcd_A vs. gcd_C, respectively. Because gcd_A vs. gcd_C is harder to prove, SE3 introduced primed terms during refinements to capture the relations between the current and the next state variables.

```
if x > y then
    x ← x − y
else if y > x then
    y ← y − x
```

(a) gcd_A

```
if x > y then
    x ← x − y
else
    swap x and y
```

(b) gcd_B

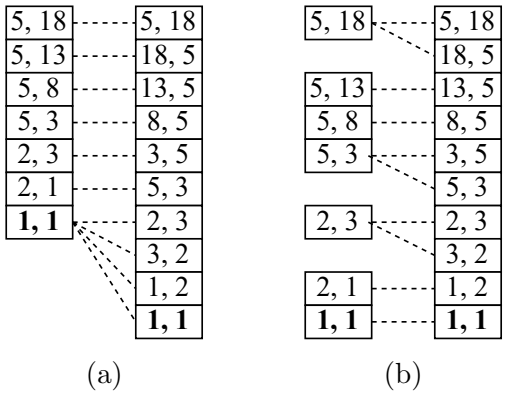
```
msb ←
 the highest bit of y
if msb = 0 and
 x > y ∗ 2 then
    x ← x − y ∗ 2
else if x > y then
    x ← x − y
else swap x and y
```

(c) gcd_C

```
▷ Possible over-
  flow in y ∗ 2 ◁
if x > y ∗ 2 then
    x ← x − y ∗ 2
else if x > y then
    x ← x − y
else
    swap x and y
```

(d) gcd_D (incorrect)

Figure 2.1. Loop bodies of 4 implementations of the *gcd* running example.



$$\vee\ (P_\times \wedge\ \{x_A, y_B\}\ \wedge \{y_A, x_B\})$$
$$\vee\ (P_\times \wedge\ \{x_A, x_B\}\ \wedge \{y_A, y_B\})$$

(c)

$$\vee\ (P_\times \wedge \{x_A, y_C\} \wedge \{y_A, x_C\})$$
$$\vee\ (P_\times \wedge \{x_A, x_C \mid y_A, y_C\})$$
$$\vee\ (P_\times \wedge \{x'_A, y'_C \mid y'_A, x'_C\})$$
$$\vee\ (P_\times \wedge \{x'_A, x'_C \mid y'_A, y'_C\})$$
$$\vee\ (P_\times \wedge \{x'_A, x'_C, y'_A, y'_C\})$$
$$\vee\ (P_\times \wedge \{x_A, y'_C \mid y_A, x'_C, y_C\})$$
$$\vee\ (P_\times \wedge \{x_A, y'_C \mid y_A, x'_C \mid y_C\})$$

(d)

Figure 2.2. Analysis of equivalence relations found by SE3 and Kairos. Alignments of state sequences starting from $(5, 18)$ for gcd_A vs. gcd_B by (a) Kairos and (b) SE3 (**bold** states are the *valid* states); the inductive invariants in partition assignment representation found by SE3 for (c) gcd_A vs. gcd_B and (d) gcd_A vs. gcd_C.

2.5.3. Experimental Results

The outcomes of our experiments are shown in Fig. 2.3. Either in the 3-branch mode or the 2-branch mode, SE3 can solve more equivalent test cases than Kairos within any period. We also observe that both the 2-branch and the 3-branch modes can solve some extra test cases than the other modes within the same period. Specifically, the 2-branch

Name	Description	# Nodes	# Regs
gcd	GCD Algorithm	60-115	6-11
euclid	Alternative GCD Algorithm	58-109	6-11
counter	Bidirectional Counter with Limit	96-164	6-14
diffeq	Differential Equation Solver	86-170	6-15
barcode	Barcode Reader	93-187	8-18
ellipf	Fifth Order Elliptical Wave Filter	135-210	11-21
kalman	Kalman Filter	157-229	8-26

Table 2.2. A summary of high-level benchmarks. The number of nodes and registers are counted by terms.

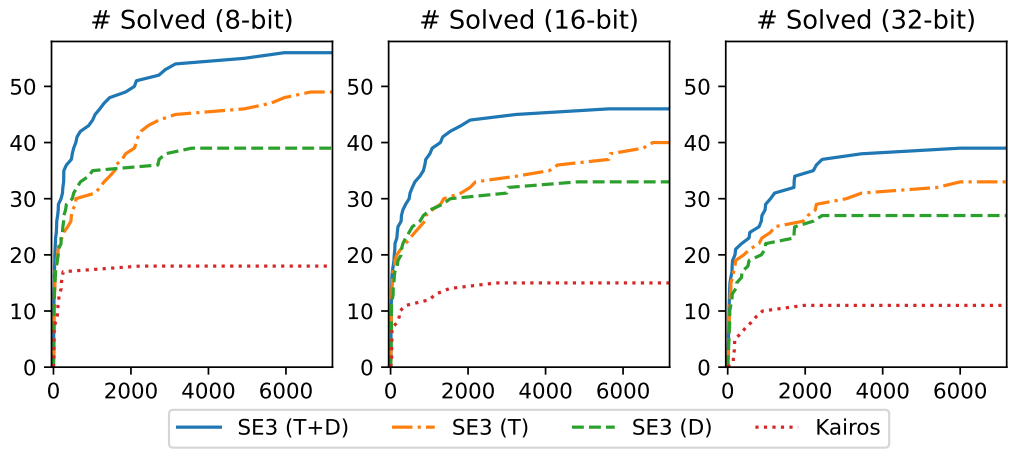


Figure 2.3. The number of solved instances over time (seconds) under different word sizes. T refers to the 3-branch mode and D refers to the 2-branch mode.

mode is more efficient when a design is always faster, while the 3-branch mode can cope with the general situation. Hence, a combination of the two modes running in parallel can solve the largest number of test cases.

We notice that Kairos is closer to SE3 in terms of execution time on two benchmarks: *barcode* and *ellipf*. These benchmarks produce outputs periodically, triggering the *valid* signal frequently. Kairos can leverage this additional information to accelerate its computation.

Finally, we analyze the statistics during SE3's execution (Fig. 2.4). Thanks to the close integration of the model checking algorithm and syntax abstraction, the number of total frames and SMT queries are almost unchanged with respect to the word size. The average time spent per SMT query moderately increases as the word size grows, and this accounts for the trend shown in Table 2.1.

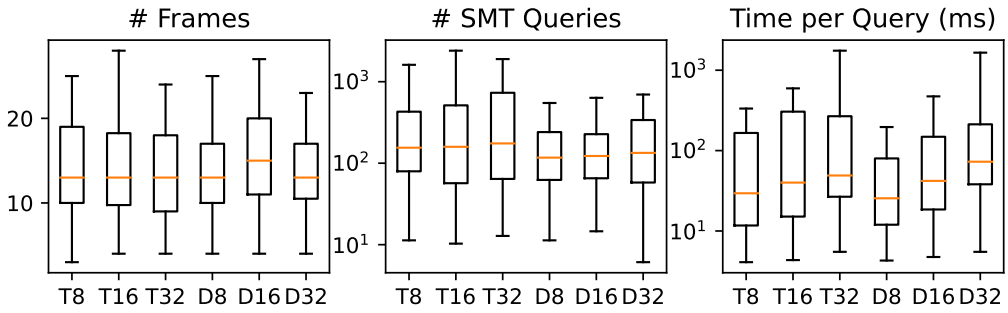


Figure 2.4. Statistics of SE3 under different word sizes.

2.6. Related Work

Mitering is the standard technique to check if two designs are combinationally equivalent. A miter composes the designs under verification by connecting their corresponding inputs. Then an SAT/SMT solver is queried to check if there exists an input pattern for the two designs to produce different output patterns. Chauhan *et. al.* attempts to reduce sequential equivalence checking problems to combinational ones [**17**]. They iteratively unroll both designs until their periods are both 1. The validity of this approach is based on two assumptions: both designs have fixed periods, and a latch mapping is provided by the user. These assumptions may not hold after significant transformations.

Most algorithms on sequential equivalence checking [**45, 7, 80**] utilize structural similarities between the designs to verify their equivalence. They either use SAT/BDD sweeping to identify and compress equivalent internal nodes and combine them into equivalent classes, or use random simulation and counterexample-guided refinement to partition the classes. The algorithms terminate when all corresponding output nodes from both designs are proven to be equivalent. Otherwise, they launch a sequence of rewriting and retiming steps and repeat the whole process. However, a fixed one-to-one mapping of internal nodes from both designs may not exist. Moreover, rewriting and retiming rely heavily on heuristics, which are incomplete methods. In comparison, our proposed method enables high flexibility to express the correlations between internal signals. Additionally, it is based on model checking and guarantees soundness and completeness.

2.7. Conclusion

In this chapter, we propose SE3, an efficient algorithm for non-cycle-accurate equivalence checking. SE3 first reduces the sequential equivalence checking problem to the problem of checking the alignablity of observable sequences. It then solves the problem by finding reversed inductive invariants on a syntax abstracted space. Experimental results confirm the effectiveness and correctness of SE3.

CHAPTER 3

SAT-based Sequential Logic Decryption without an Oracle

3.1. Introduction and Motivation

3.1.1. Adversarial Model of Logic Encryption

Since proposed in 2015, the SAT attack [**78**] has established the status quo for logic encryption. The algorithm of the SAT attack is both rigorous and exhaustive. It provides solid guarantees on both termination and the correctness of the returned key. Although there exist defense mechanisms thwarting this attack [**87, 83**], it is pointed out that there are inherent trade-offs among SAT resilience, structural robustness, and locking efficiency [**90**]. Therefore, the SAT attack still poses a significant and realistic threat to logic encryption.

To launch the SAT attack, an adversary needs to simultaneously access *i)* an encrypted netlist of the circuit and *ii)* a working chip as the oracle circuit. In reality, an adversary can acquire the encrypted netlist through a rogue insider within the design house or recover it from a physical layout obtained from an offshore foundry or an assembly facility [**88**]. On the other hand, it is challenging for the adversary to acquire a working chip in many

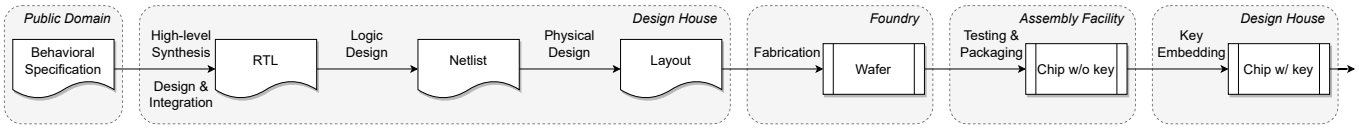


Figure 3.1. IC design flow with logic encryption.

scenarios. For example, *a)* a chip may be designed for mission-critical applications or is fully customized, so a working chip cannot be purchased from the open market; *b)* it may be too late to start reverse engineering after a chip is on the market, because the market opportunity has already expired; and *c)* if the adversary targets IP blocks or sub-modules, a corresponding working chip does not present.

For this sake, researchers have devised various oracle-less attacks on logic encryption, including synthesis-based attacks[**3, 49**], testing-based attacks [**49, 30**], machine-learning-based attacks [**15, 4**] and structural attacks [**89, 44**]. However, these methods are either limited to specific use cases and locking schemes, or cannot guarantee termination and the correctness of extracted keys. Does there exist a logic decryption method that enjoys the same advantages as the SAT attack, while it does not require an oracle circuit?

3.1.2. Leveraging Behavioral Specification for Logic Decryption

Figure 3.1 depicts the general IC design flow with logic encryption. The design house introduces key-controlled protection logic into the circuit during logic or physical design. Once the fabricated chip is returned to the design house, they are activated by applying the pre-selected key. Without knowing a correct key, an adversary cannot fully recover the original functionalities of the chip.

Logic encryption can effectively protect design efforts, particularly those within the logic and physical design stages. On the other hand, it is a common situation that the behavioral specification of the chip lies in the public domain or is publicly accessible. For instance, *a)* the functionalities of various families of chips are common knowledge. These families include but are not limited to data converter IC, interface and protocol IC,

driver and controller IC, and signal processing IC. *b)* To facilitate their customers to build products, it is a common practice for design houses to release data sheets and development guides as well as to provide application and technical support. A behavior model can be reconstructed from this information. *c)* High-level synthesis is widely adopted by custom ASIC designs nowadays [**73**]. The high-level specifications in C++/SystemC are usually online accessible, especially those used for machine learning, data processing, and hardware acceleration. *d)* Leading IP vendors, including Synopsys and ARM, release transaction-level models (TLM) in accompany with their IP blocks[**79**]. These behavioral models enable fast prototyping, simulation, and verification for existing and potential customers.

At a glance, an adversary can treat such a behavioral specification as an oracle to launch the SAT attack. Nevertheless, most of these specifications are either un-timed or non-cycle-accurate to the corresponding working chip. Even worse, some specifications have variable periods due to the high-level languages they use. Hence, neither combinational nor sequential SAT attacks can utilize them as oracles.

3.1.3. Overview

This chapter presents OLSAT, a novel and rigorous sequential logic decryption method based on formal methods. OLSAT assumes that the adversary has access to an encrypted netlist and a corresponding behavioral specification, which is not necessarily cycle-accurate to the original design. OLSAT first utilizes high-level and logic synthesis tools to automatically convert the behavioral specification to a netlist automatically. Timing and logic variations are permitted in this process. Subsequently, a clock cycle

alignment between the encrypted and the synthesized netlist is automatically generated on the fly. Thus, an SAT or SMT solver can be applied to find a correct key.

A key enabler of OLSAT is a novel parameter synthesis mechanism called LIM (Less-Is-More). For each DIP, the original sequential SAT attack duplicates the whole unrolled netlist and adds it to the solver as an I/O constraint to key values. LIM instead distills just a single cube of wrong keys from each I/O constraint. It improves the scalability of sequential SAT attacks on large circuits that require deep unrolling. Moreover, this simple and flexible mechanism ensures the feasibility of OLSAT.

Our main contributions are:

• We discover that logic encryption is still vulnerable to I/O attack even when the adversary cannot obtain a working chip;

• We propose LIM, a flexible and efficient sequential logic decryption algorithm;

• We devise OLSAT, an oracle-less SAT attack that can automatically align two non-cycle-accurate designs on the fly;

• We conduct extensive experiments to evaluate the feasibility and performance of LIM and OLSAT.

3.2. Background

3.2.1. Preliminaries

A *combinational* encrypted netlist C_e is defined as a tuple $\langle X, K, Y \rangle$, where X represents the primary inputs, K the key inputs, and Y the primary outputs. We use C_o to denote the oracle circuit corresponding to C_e. If a correct key K_c is inserted, C_e and C_o should exhibit the same behavior given any input sequence.

A variable or its negation is called a *literal*. A conjunction of literals is called a *cube*, and a disjunction of literals is called a *clause*. A clause is the negation of a cube and vise versa. A *pattern* is an assignment to all corresponding variables and it can be described as a cube. A *sequence* is a series of patterns. A Boolean formula is in conjunctive normal form (CNF) if it is a conjunction of clauses. A formula F *implies* another formula G, written as $F \Rightarrow G$, if all assignments satisfying F also satisfy G. I is an *implicant* of a formula F if I is a cube and $I \Rightarrow F$. I_p is a *prime implicant* if it is *minimal*, *i.e.,* dropping any literals from I_p will result in a non-implicant.

A sequence σ *stutters* at step k if it keeps the same value for indices k and $k+1$. For instance, the sequence $\langle a, b, c, c, \cdots \rangle$ stutters at step 3. Two sequences are equivalent modulo stuttering if they become identical after all stuttering steps are eliminated.

3.2.2. Problem Definition

Before we start formulating the sequential decryption problem, we first give a formal definition of whether two *sequential* circuits are *observationally equivalent* [**1**]:

Definition 4 (Observational equivalence)**.** *Two circuits are observationally equivalent if and only if their output sequences are equivalent modulo stuttering for every input sequence.*

The problem of non-cycle-accurate sequential logic decryption is thus defined as follows:

Problem 1 (Non-cycle-accurate sequential logic decryption)**.** *Given an encrypted netlist C_e and a corresponding reference circuit C_b, find a correct key K_c such that $C_e(K_c)$ and C_b are observationally equivalent.*

3.2.3. SAT Interface

We use $SAT[\psi]$ to denote a SAT query to formula ψ. It returns *satisfiable* if there exists an assignment to all variables in F such that ψ evaluates to true, and *unsatisfiable* otherwise. Furthermore, we use $SAT[\psi].model(X)$ to denote the value of variable X within a satisfiable assignment.

Many modern SAT solvers support unsat core extraction. We write an unsat core query as $SAT[\psi, \gamma]$, where ψ is a CNF formula and γ is a set of *assumption clauses*. If $\psi \wedge \gamma$ is satisfiable, both $SAT[\psi, \gamma]$ and the normal SAT query $SAT[\psi \wedge \gamma]$ will return a satisfiable assignment. However, if $\psi \wedge \gamma$ is unsatisfiable, $SAT[\psi, \gamma]$ will return an unsat core β in addition to the unsatisfiable result. β is a subset of γ such that $\psi \wedge \beta$ is still unsatisfiable. Some SAT solvers can ensure that β is minimal for most of the time [**27**].

3.2.4. The SAT Attack Algorithm

Algorithm 2 The SAT Attack Algorithm [**78**]

Input: encrypted netlist C_e, oracle circuit C_o
Output: a correct key pattern K_c
$M \leftarrow C_e(X, K_1, Y_1) \wedge C_e(X, K_2, Y_2)$
while $SAT[M \wedge (Y_1 \neq Y_2)]$ **do**
    $X_d \leftarrow SAT[M \wedge (Y_1 \neq Y_2)].model(X)$
    $Y_d \leftarrow C_o(X_d)$
    $M \leftarrow M \wedge C_e(X_d, K_1, Y_d) \wedge C_e(X_d, K_2, Y_d)$
$K_c \leftarrow SAT[M].model(K_1)$

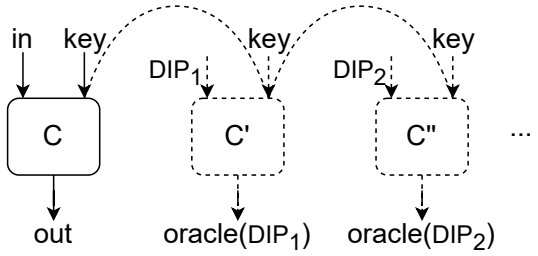


Figure 3.2. Redundancy in SAT attack. In every iteration, the whole netlist is duplicated and added to the solver.

The SAT attack algorithm (Algorithm 2) requires an *encrypted netlist* C_e and a corresponding running *oracle* circuit C_o. It first constructs a *product machine* M with two copies of the encrypted netlist (line 3). In each iteration, it queries a SAT solver on the product machine for a differentiating input pattern (DIP) X_d: given such an input pattern, there exists two different key patterns K_1 and K_2 that produce two different output patterns Y_1 and Y_2 (line 5). At least one of these key patterns must be incorrect. The algorithm then queries the oracle for the corresponding correct output Y_d (line 6). X_d and Y_d together with a fresh copy of the encrypted netlist form a new *I/O constraint* on key pattern. This I/O constraint is conjoined to both copies in the product machine (line 7), and the algorithm starts a new iteration. The algorithm terminates when no more DIPs can be found (line 4). Then a correct key K_c can be extracted through another SAT query (line 8).

3.3. LIM: Synthesis-based Logic Decryption

3.3.1. Redundancy in SAT Attack

In the SAT attack, each DIP duplicates the whole encrypted netlist to form a new I/O constraint (Figure 3.2). As a consequence, clauses pile up quickly within the SAT solver.

Although modern SAT solvers are equipped with clause learning and clause deletion mechanisms [**52**], a growth of such a magnitude will eventually overwhelm them. It is observed that the execution time of each SAT query increases super-linearly with respect to the number of DIPs [**75**].

The SAT attack is essentially a covering process: it terminates when I/O constraints eliminate exactly all wrong keys. From this perspective, the constraint mechanism of the SAT attack is highly redundant. For example, *a)* there is redundancy within a single I/O constraint, because with fixed X_d and Y_d, it is unnecessary to keep the original structure of the encrypted netlist. *b)* There is redundancy across multiple I/O constraints, because they may use identical internal nodes or cover the same prime implicant multiple times.

All the observations above suggest that constraints should not be multiplied beyond necessity. KC2 [**75**] is a pioneering work in this direction. It applies a set of sweeping and reduction techniques to compress I/O constraints. While KC2 can reduce up to 80% of the clauses for each I/O constraint, it still suffers from the drawbacks caused by netlist duplication.

3.3.2. Parameter Synthesis

Table 3.1. The advance of symbolic verification and logic decryption.

	SAT	SMT	BMC	UMC	ParamSynth
Symbolic Verification	DPLL (1960)	Nelson (1979)	Unrolling (2001)	ITP (2003)	IC3-based (2015)
Logic Decryption	[**78**] (2015)	[**6**] (2019)	[**33**] (2017)	[**75, 70**] (2019)	**LIM**

The last decade has witnessed the rapid advance of logic decryption. We noticed that almost every symbolic formal verification method induces a corresponding logic decryption

method. Table 3.1 shows representative methods from both categories and the earliest year they were proposed. Initially, symbolic verification problems are encoded as SAT [**24**] and SMT [**56**] problems. Clarke *et al.* [**21**] proposed bounded model checking to verify a system within a limited number of time steps. Unbounded model checking algorithms [**53, 32, 10**] were later proposed to remove such a limit.

Recently, a new formal verification method called parameter synthesis has emerged. Given a parameterized transition system and a safety property in temporal logic, parameter synthesis aims to find the set of all correct patterns of the parameters, under which the system satisfies the property. Existing solutions [**19, 8**] utilize IC3, an unbounded symbolic model checking algorithm to deal with infinite state transition systems. SAT-based sum-of-product logic synthesis [**63, 72**] is a special case of parameter synthesis, where the property is a user-specified Boolean formula and the parameters are the Boolean variables within the formula.

The SAT attack can extract a correct key only after it finds out and excludes all incorrect keys. The problem of finding all *incorrect* keys is similar to parameter synthesis: the key variables can be considered as parameters, while the behavioral equivalence to the oracle circuit can be treated as the safety property. However, the adversary in the SAT attack has only *query access* to the oracle. Hence, parameter synthesis cannot be directly applied to logic decryption.

3.3.3. The LIM Algorithm

The pseudo-code of the LIM algorithm is shown in Algorithm 3. The algorithm maintains a CNF formula W, which blocks all wrong keys discovered so far (line 3). In every

Algorithm 3 The LIM Algorithm

1: **Input:** encrypted netlist C_e, oracle circuit C_o
2: **Output:** a correct key pattern K_c
3: $W \leftarrow \texttt{True}$
4: $M \leftarrow C_e(X, K_1, Y_1) \wedge C_e(X, K_2, Y_2)$
5: **while** $SAT[W \wedge M \wedge (Y_1 \neq Y_2)]$ **do**
6: $\quad X_d \leftarrow SAT[W \wedge M \wedge (Y_1 \neq Y_2)].model(X)$
7: $\quad Y_d \leftarrow C_o(X_d)$
8: $\quad K_w \leftarrow SAT[W \wedge C_e(X_d, K, Y) \wedge (Y \neq Y_d)].model(K)$
9: $\quad K_w^\circ \leftarrow SAT[W \wedge C_e(X_d, K, Y_d), \{K_w\}]$
10: $\quad W \leftarrow W \wedge \neg K_w^\circ$
11: $K_c \leftarrow SAT[W].model(K)$

iteration, it finds a DIP in a similar way as the original SAT attack (line 6). Then the algorithm finds a wrong key K_w that is not yet excluded by the key condition W and falsifies the current I/O constraint (line 8).

Afterward, K_w is generalised to a cube to cover a large number of wrong keys (line 9). In specific, $\{K_w\}$ is a set of unit clauses: each unit clause corresponds to a literal in K_w. With $\{K_w\}$ enforced to K, the SAT query in line 9 must be unsatisfiable. Moreover, a SAT solver as described in Section 3.2.3 should return a minimal unsat core K_w°, which is a prime implicant of the formula $\neg W \wedge \neg C_e(X_d, K_2, Y_d)$. In other words, K_w° is an irreducible cube distilled from both the current I/O constraint and all the previously generalised cubes. For those SAT solvers that cannot ensure K_w° is minimal, we iteratively drop literals from it and check unsatisfiability until no literals can be dropped anymore. At the end of an iteration, the key condition W is updated by the negation of K_w°. Finally, when no more DIPs can be found, a correct key is extracted from the remaining keys within W (line 11).

Lemma 5. *Upon termination, formula W excludes exactly all incorrect keys.*

Proof. *Soundness part:* A K_w produced in line 8 must be an incorrect key, because the output pattern Y generated by X_d and K is in consistent with the reference output Y_d. By the same reasoning, all K_w° cubes contain only incorrect keys.
Completeness part: If there exists an incorrect key not yet excluded by W, the SAT query on line 5 will be satisfiable. In this case, the algorithm is not terminated, which conflicts with the condition. □

Theorem 6. *The LIM algorithm will eventually terminate and return a correct key.*

Proof. *Termination part:* In every iteration, at least one incorrect key will be discovered and conjoined to W. Hence, the algorithm will eventually terminate.
Correctness part: From Lemma 5, the algorithm must return a correct key if there exists such a key for C_e and C_o. □

3.3.4. Simplifying Key Conditions

LIM periodically execute the following two procedures to ensure that all cubes are *irreducible* and *irredundant*, *i.e.,* no literals can be further dropped without eliminating a correct key, and no generalised cubes can be dropped without uncovering a wrong key. With a little abuse of terminology, we say that a clause is irreducible (resp. irredundant) in W, if the cube formed by its negation is irreducible (resp. irredundant) in $\neg W$.

Syntactic Subsumption: Because W gets stronger after every iteration, more recently produced clauses are likely to subsume earlier produced ones. A clause c_1 is removed from W if there exists another clause c_2, such that all literals in c_2 also present in c_1.

Essential Minterm: A clause is irredundant if it excluded at least one wrong key that is not excluded by any other clauses. We eliminate all redundant clauses from W in a

fully incremental approach by leveraging the assumption mechanism of the SAT solver. Specifically, for each clause $c_i = l_{i,1} \vee \cdots \vee l_{i,n}$ in W, we reserve an auxiliary variable a_i and add the new clause $c'_i = a_i \vee l_{i,1} \vee \cdots \vee l_{i,n}$ instead of the original clause to the SAT solver. Notice that if a unit clause a_i is within the assumption clauses, the corresponding clause c'_i is voided. On the other hand, if $\neg a_i$ is within the assumption clauses, c'_i is reduced to c_i.

To check if a clause c_p is irredundant, LIM queries a SAT solver for the following problem:

$$SAT[\bigwedge_{i=1}^{|W|} c'_i, \bigwedge_{i=1}^{p-1} \neg a_i \wedge a_p \wedge \bigwedge_{i=p+1}^{|W|} \neg a_i \wedge \bigwedge_{j=1}^{|c_p|} \neg l_{p,j}]. \tag{3.1}$$

In other words, LIM checks whether there exists a key within the cube $\neg c_p$ ($\bigwedge_{j=1}^{|c_p|} \neg l_{p,j}$) that is not excluded by any other clauses ($\bigwedge_{i=1}^{p-1} c_i \wedge \bigwedge_{i=p+1}^{|W|} c_i$). LIM iterates through all clauses in W to check if they are irredundant. If any query to (3.1) is unsatisfiable, a unit clause a_p is added to the SAT solver to permanently disable the corresponding clause c'_p.

Theorem 7. *All clauses in W are irreducible and irredundant.*

Proof. *Irreducible part:* The generalisation procedure in line 9 ensures that the most recent clause is irreducible. The syntactic subsumption procedure ensures that all previous clauses are also irreducible.

Irredundant part: The essential minterm procedure ensures that all the clauses are irredundant. □

3.3.5. Extension to Sequential Logic Decryption

The original SAT attack assumes the adversary to have access to all flip-flops through the scan chain. However, such access can be disabled. For example, the design house can request the assembly facility to blow down the anti-fuses on the scan chain [**22**]. Because the adversary cannot read or write data to the flip-flops, the observability and the controllability over the victim oracle circuit are reduced. On the other hand, sequential logic encryption schemes [**16, 29**] embed additional finite-state machines to the original circuit. To activate an encrypted circuit, a user must apply a correct input sequence to the primary inputs cycle-by-cycle. Starting from the *reset state*, this input sequence leads the circuit to reach the *protected initial state*, from which the circuit resumes its original functionality.

The standard approach for sequential SAT attack is through unrolling [**33, 70**]. Suppose that the *unrolling diameter* is k. Then the encrypted netlist is duplicated for k times, and each secondary output (the input signals of the flip-flops) of a copy is merged with the corresponding secondary input of the next copy (the output signals of the flip-flops). This unrolling process can also be conducted implicitly with a bounded model checker. For the disabled scan chain scenario, the key is fixed across all clock cycles. For the sequential encryption schemes, the adversary can formulate the protected initial state as the key. In particular, the adversary can modify the encrypted netlist, such that the key is loaded to the flip-flops upon initialization. Notice that in both cases, the key space remains unchanged regardless of the unrolling diameter. The unrolling process is terminated when either of the three conditions can be satisfied [**33**]: *i) Unique Key:* under the current key constraints, only 1 key is remaining; *ii) Combinational Equivalence:* all the

remaining keys are within the same equivalent class even if the circuit was combinational, *i.e.,* the secondary inputs and outputs are treated as primary ones; *iii) Unbounded Model Checking:* an unbounded model checker cannot find an differentiable input sequence given any of the remaining keys.

The sequential LIM algorithm adopts a similar unrolling scheme. It has two main advantages over the naive sequential SAT attack. For one, it fosters information exchange across different unrolling diameters, because the current cube can learn from all the previous cubes (line 9). For another, the overhead introduced by a new key constraint is insensitive to the unrolling diameter. In comparison, the size of a key constraint in the naive sequential SAT attack is proportional to the unrolling diameter. Finally, sequential LIM is orthogonal to the unrolling optimization techniques [**42**], *i.e.,* any advance of those techniques can also benefit our method.

3.4. OLSAT: Oracle-less SAT Attack

Figure 3.3 shows the general workflow of OLSAT. As Section 3.1.3 mentions, OLSAT assumes that the adversary cannot acquire an oracle circuit. Instead, the adversary is able to obtain a behavioral specification of the victim circuit. This specification is automatically synthesized to an RTL design with a high-level synthesis tool (*e.g.,* Xilinx Vivado and Cadence Stratus). Afterward, the RTL design is mapped to logic gates with a logic synthesis tool. In the remainder of this paper, we refer to the synthesized netlist as C_b and the expected behavior of the victim circuit as $C_e(K_c)$.

The behavioral specification could be untimed or have only limited timing information. Additionally, the synthesis process can introduce uncertainties in timing. As a result, C_b

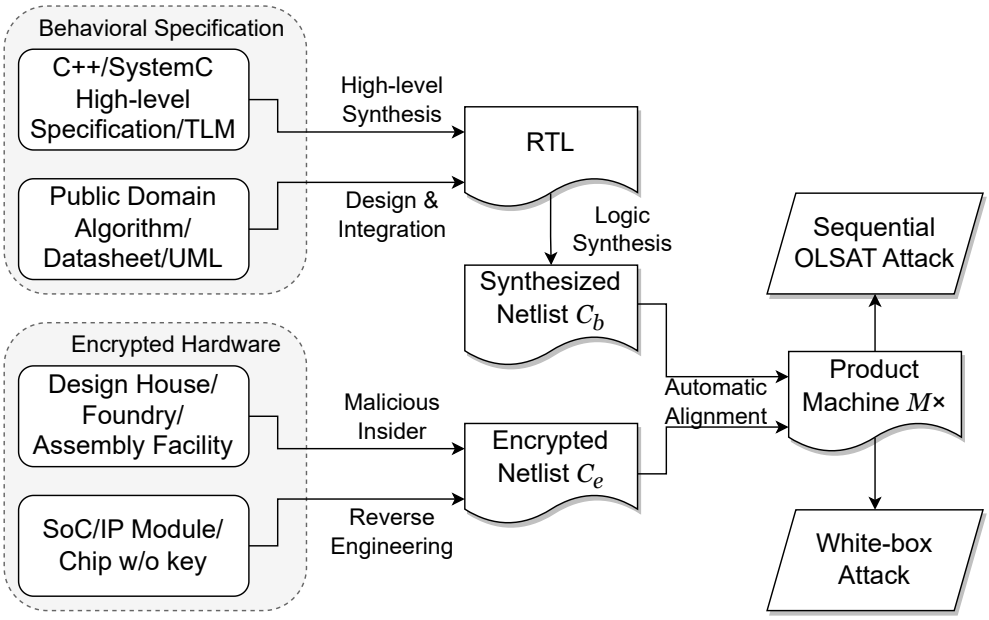


Figure 3.3. The general workflow of OLSAT.

and $C_e(K_c)$ are likely to be non-cycle-accurate. Furthermore, C_b could have a variable period with respect to $C_e(K_c)$, because a high-level synthesis tool can create a control logic that is fundamentally different from the one within C_e.

The OLSAT algorithm brings up two main components to tackle the timing challenge. The first component is an automatic alignment mechanism. It attempts to insert stuttering steps on the fly to maintain the observational equivalence between C_b and $C_e(K_c)$. We formalize the observational equivalence constraints as CNF formulas so that they are compatible with SAT solvers. The second component is the LIM key condition synthesis algorithm. With LIM, the size of the key condition is agnostic to the unrolling diameter, while the format of the key condition is irrelevant to the observational equivalence constraints. These benefits ensure feasibility and scalability for the OLSAT algorithm.

3.4.1. Automatic Alignment of Clock Cycles

Definition 4 states a criteria to determine whether two circuits are observationally equivalent: given any particular input sequence, the output sequences of the two circuits are identical after removing all stuttering steps. We observe that removing a stuttering step from one sequence is the *dual* of inserting a stuttering step to the other sequence at the corresponding location. For simplicity of presentation, we first assume that the behavioral specification C_b is always no slower than the implementation $C_e(K_c)$. In other words, for any input sequence, the period of the implementation is always no shorter than that of the specification. This assumption is realistic because the former is not limited by resource constraints.

From the above discussions, we can derive the following lemmas:

Lemma 8. *Consider two observationally equivalent circuits C_1 and C_2, where C_1 is always no slower than C_2 and they are both deadlock-free. For every specific input sequence, there exists a way to halt C_1 during its execution, such that the output sequences of both circuits are identical.*

Proof. We prove the existence by construction. Consider the following strategy:

Automatic Alignment Strategy: Each time C_1's output pattern has changed, halt C_1; resume C_1 once C_2's output pattern is also changed.

To show that the output sequences of the two circuits are identical, we split the output sequence of C_2 into segments: the first step in a segment is a *critical step* (whose output pattern is different from the previous step), and the rest of the steps in that segment are *stuttering steps* (whose output pattern is the same as the previous step).

The above strategy is to process the segments of C_1's output sequence successively. For every segment, it attaches dummy stuttering steps until it matches the corresponding segment of C_2. □

Lemma 9. *C_1 and C_2 are observationally equivalent if and only if the automatic alignment strategy presented in Lemma 8 produces identical output sequences for any input sequences.*

Proof. *Only If part:* From Lemma 8, if the two circuits are observationally equivalent, they will always produce identical output sequences when the automatic alignment strategy is adopted.
If part: After removing all stuttering steps, two initially identical sequences are still identical. Hence, by Definition 4, two circuits are observationally equivalent if their output sequences are always identical given any input sequences. □

Lemma 8 and Lemma 9 suggest a method to check whether two circuits are observationally equivalent. Specifically, we build a *product machine* consisting of two branches, C_1 and C_2. The behavior of the product machine is determined by a *controller*, which checks the states of the product machine for the following condition:

$$(C_1.out' \neq C_1.out) \wedge (C_2.out' = C_2.out), \tag{3.2}$$

where $C_i.out$ and $C_i.out'$ represent the output patterns of C_i in the current and the next clock cycle, respectively. The controller halts C_1 only when the condition is satisfied. In our implementation, halting is realized by adding a multiplexer and a feedback path

for every flip-flop of C_1. When Equation 3.2 returns `True`, the multiplexers selects the feedback path, so that C_1 remains in its current state. Furthermore, we add a *miter* to the product machine to detect whether the output sequences of C_1 and C_2 are identical. The miter outputs `True` if C_1 and C_2 output distinct patterns at any clock cycle. Finally, we add a bounded fairness constraint [**28**], which enforces both C_1 and C_2 to move forward at least once in every k consecutive steps. To check whether the two circuits are observational equivalent, one can call a model checker on the product machine, and use the miter output as the safety property.

The above discussions assume that one circuit is always no slower than the other. In fact, the same method can be extended to the general case that both circuits can be faster than one another at anytime. Particularly, the controller need to maintain another condition that is symmetric to Equation 3.2, and halt C_2 according to its return value.

The idea of checking observational equivalence through automatic synchronization is first proposed by Kairos [**64**]. Our method is different from Kairos in the following aspects: *a)* Kairos requires both circuits to follow a *valid-ready* interface protocol. It utilizes the *valid* signal for synchronization and equivalence checking. In comparison, our method does not make such an assumption[1]. *b)* Kairos cannot deal with the situation where one circuit is trapped in a deadlock state. In that situation, even though the output sequences are indeed different, Kairos cannot detect it. Our construction does not have the same issue. *c)* Kairos relies on clock gating for synchronization. Consequently, the resulting hardware cannot be encoded as CNF formulas. On the contrary, our construction is fully synchronous and can be directly checked by a SAT solver.

[1]It still requires C_1 and C_2 to decide when to consume a new input pattern.

3.4.2. The OLSAT Algorithm

The automatic alignment mechanism bridges the timing gap between C_b and $C_e(K_c)$. According to Lemma 9, when the miter's output is enforced `True`, the two netlists are guaranteed to be observationally equivalent. To put differently, by controlling the miter's output, the adversary obtains *white-box access* to the oracle circuit. At a glance, this allows the adversary to immediately extract the correct key. Nevertheless, due to the universal quantifier in the problem ($\exists K_c$ *s.t.* $\forall$ *input sequences: miter's output* = `True`), it is still necessary to cover all incorrect keys before a correct one can be extracted.

The working principle of the OLSAT algorithm (Algorithm 4) is similar to that of the sequential LIM algorithm. The algorithm maintains a global formula of key condition W throughout its execution. It builds a product machine modulo stuttering M_s with the automatic alignment mechanism. Initially, the unrolled product machine M_u has a diameter of 1, so it is identical to M_s.

During every iteration of the inner loop (line 6 - line 10), W is strengthened until it blocks exactly all incorrect keys activating the miter ($Y_1 \neq Y_2$) up to the current unrolling diameter. Detailed explanations for this procedure can be found in Section 3.3.3. OLSAT then checks whether termination has reached. As described in Section 3.3.5, this is equivalent to check whether the miter will always output a `False` regardless of the unrolling diameter. However, checking this condition directly is very time-consuming [**42**]. In this regard, we exploit a trial-and-error (TE) strategy to improve its efficiency. Specifically, OLSAT first extracts a potentially correct key K_p from W (line 11). It then queries an unbounded model checker for the above-mentioned termination condition on just K_p (line 12). Because the searching space is reduced to a single key, the model checking

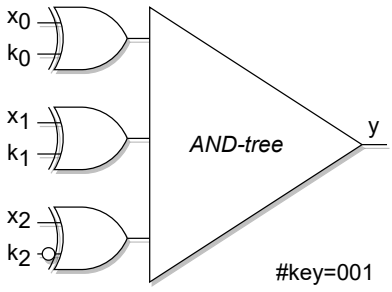
Algorithm 4 The OLSAT Algorithm

Input: encrypted netlist C_e, synthesized netlist C_d
Output: a correct key pattern K_c
$W \leftarrow$ `True`
$M_s, M_u \leftarrow C_e(X, K, Y_1) \times C_d(X, Y_2)$ ▷ *Construct product machine with automatic alignment*
while `True` **do**
    while $SAT[W \wedge M_u \wedge (Y_1 \neq Y_2)]$ **do**
        $X_d \leftarrow SAT[W \wedge M_u \wedge (Y_1 \neq Y_2)].model(X)$
        $K_w \leftarrow SAT[W \wedge M_u \wedge (X = X_d) \wedge (Y_1 \neq Y_2)].model(K)$
        $K_w^\circ \leftarrow SAT[W \wedge M_u \wedge (X = X_d) \wedge (Y_1 = Y_2), \{K_w\}]$
        $W \leftarrow W \wedge \neg K_w^\circ$
    $K_p \leftarrow SAT[W].model(K)$
    if $UMC[M_s \wedge (K = K_p) \wedge (Y_1 \neq Y_2)] =$ `False` **then**
        $K_c \leftarrow K_p$; **break**
    $M_u \leftarrow M_u \| M_s$ ▷ *unrolling*
return K_c

efficiency is significantly improved. Notice that *TE* is not applicable to the sequential SAT attack, because a model checker cannot decide the correctness of a key with only query access to the oracle circuit.

Theorem 10. *The OLSAT algorithm will eventually terminate and return a correct key.*

Proof. *Termination part:* In every iteration, at least one incorrect key will be discovered and conjoined to W. Hence, the algorithm will eventually terminate.
Correctness part: From Lemma 9 and Theorem 16, the algorithm must return a correct key if a K_c exists such that C_b and $C_e(K_c)$ are observationally equivalent. □



(a)

x \ k	000	001	010	011	100	101	110	111
000	0	0	0	0	0	0	1	0
001	0	0	0	0	0	0	0	1
010	0	0	0	0	1	0	0	0
011	0	0	0	0	0	1	0	0
100	0	0	1	0	0	0	0	0
101	0	0	0	1	0	0	0	0
110	1	0	0	0	0	0	0	0
111	0	1	0	0	0	0	0	0

(b)

Figure 3.4. A demonstration of white-box attack against logic encryption. (a) A 3-input AND-tree protected by input-XOR encryption [**90**]. (b) The corresponding error matrix. Shadowed entries with underscores have incorrect outputs.

3.4.3. White-box Attack Against Logic Encryption

The error matrix is a powerful tool to analyze the complexity of an attacking algorithm against a logic encryption scheme. Each row in the error matrix represents an input pattern, each column represents a key pattern, while each entry represents whether the output pattern of the encrypted netlist deviates from the correct output pattern. In the original SAT attack, every query to the oracle circuit can reveal only one row of the error matrix. Existing logic encryption schemes [**87, 83**] exploit this limitation to thwart the SAT attack. They ensure that the error rate is exponentially small in the number of input patterns. Formally speaking, for every column representing an incorrect key, there is only a limited number of incorrect entries.

OLSAT gives the adversary another degree of freedom when traversing the error matrix. We use a case study to demonstrate the capability of the white-box attack. Figure 3.4 (b) plots the error matrix of an AND-tree protected by input-XOR encryption. If the adversary can discover the *dominant row* ($X = 111$), it will immediately cover all incorrect keys. Facilitated by OLSAT, the white-box attack exploits SAT-based *model enumeration*

to locate the dominant row quickly. Specifically, it randomly chooses a K^* and queries $SAT[M_s \wedge (K = K^*) \wedge (Y_1 \neq Y_2)].model(X)$ for a new assignment X^*. If such an assignment does not exist, K^* must be correct. Otherwise, it conjoins $\neg X^*$ to the formula and repeats the query. Because the number of satisfiable rows for a given K^* is exponentially smaller than the number of feasible DIPs, the white-box attack is exponentially faster than the SAT attack in this example. The white-box attack can be adapted to defeat other encryption schemes.

3.5. Experiments

We built a prototype for LIM and OLSAT in Python. We chose Boolector [**57**] and Z3 [**25**] as the backend SMT solvers, because we observed that they are more efficient on circuit benchmarks in terms of SAT solving and unsat core extraction, respectively. We employed the property-directed reachability command [**31**] in Berkeley ABC for unbounded model checking. All experiments were conducted on a Linux machine with a 3.2GHz CPU, and every instance was executed on a single thread. We set a memory limit of 4GiB and a timeout limit of 3,600 seconds for all experiments.

3.5.1. Evaluating LIM for Sequential Logic Decryption

We compared LIM against the state-of-the-art sequential logic decryption algorithms, including the KC2 command [**75**] in NEOS and the RANE decryption suite [**70**]. We used the default settings of the oracle-guided sequential SAT attack for both algorithms. Since neither of them selects *UMC* as a termination condition (see Section 3.3.5 for

Table 3.2. Statistics of ISCAS'89 sequential benchmark circuits.

Circuit	#PI	#PO	#FF	#Gate	Circuit	#PI	#PO	#FF	#Gate
s208	11	2	8	96	s713	35	23	19	393
s298	3	6	14	119	s820	18	19	5	289
s344	9	11	15	160	s832	18	19	5	287
s349	9	11	15	161	s838	34	1	32	446
s382	3	6	21	158	s953	16	23	29	395
s386	7	7	6	159	s1196	14	14	18	529
s400	4	6	21	164	s1238	14	14	18	508
s444	3	6	21	181	s1423	17	5	74	657
s510	19	7	6	211	s1488	8	19	6	653
s526	3	6	21	193	s1494	8	19	6	647
s526n	3	6	21	194	s5378	35	49	179	2779
s641	35	24	19	379	s9234	19	22	228	5597

details), we also disabled *UMC* for LIM to foster fair comparisons. We incremented the rolling diameter by 1 in each step and set an upper limit of 100 steps for all algorithms.

Table 3.2 lists the ISCAS'89 [**12**] benchmark circuits we used for performance evaluation. We encrypted these circuits with two prevalent logic locking schemes: *i)* randomly inserting key-controlled *XOR* gates to the combinational part of a circuit [**78**], and *ii)* the HARPOON sequential logic locking scheme [**16**], which requires an unlocking input sequence to steer a circuit from its reset state to its actual initial state. Prior study [**70**] has discovered that case *ii)* can be reduced to case *i)* by treating the initial state as the key. We constructed three encrypted instances with different encryption overheads for each benchmark circuit. In particular, we used the default CMOS cells library in Yosys [**82**] to measure the overheads after technology mapping.

Table 3.3 compares the execution time of the three algorithms on *XOR*-encrypted instances. It can be seen that the overall performances of the algorithms are similar. On the other hand, Figure 3.5 compares the three algorithms by the number of decrypted

Table 3.3. A comparison of execution time (seconds) for KC2, RANE, and LIM on *XOR*-encrypted sequential benchmark circuits.

Overhead	5%			10%			15%		
Method	KC2	RANE	LIM	KC2	RANE	LIM	KC2	RANE	LIM
s208	–	–	–	–	0.6	1.8	–	–	–
s298	0.1	0.7	1.6	–	–	–	1.2	2.4	4.7
s344	0.4	0.6	1.6	0.3	0.9	2.7	0.5	1.1	3.6
s349	0.1	0.4	1.1	0.1	0.5	1.5	1.0	1.1	4.1
s382	106	120	47.1	104	63.1	34.2	–	–	–
s386	0.1	0.8	1.0	0.2	0.9	2.3	0.3	0.7	4.1
s400	88.6	62.6	27.0	–	–	278	–	–	2772
s444	–	3507	–	–	–	–	–	–	–
s510	1.8	3.6	12.1	0.7	1.4	13.4	39.1	34.7	861
s526	–	–	–	–	1451	2839	–	–	–
s526n	80.8	–	–	–	92.9	2014	–	–	–
s641	0.5	1.2	3.1	0.4	–	4.9	–	–	–
s713	0.4	2.6	3.2	1.5	3.3	7.1	–	–	–
s820	3.1	4.7	8.7	6.8	6.9	12.4	16.6	10.8	22.0
s832	1.8	2.9	7.2	7.3	6.6	15.3	12.5	9.7	27.3
s838	–	–	–	–	–	–	–	–	–
s953	–	–	12.8	–	–	18.4	–	–	57.6
s1196	0.5	3.7	3.7	1.1	–	6.7	6.0	5.2	15.8
s1238	0.5	1.8	3.5	1.3	3.8	11.8	2.0	4.1	13.3
s1423	–	1912	575	–	–	1884	–	–	–
s1488	8.1	10.1	36.0	22.2	14.1	99.2	187	44.9	229
s1494	4.0	5.5	21.8	94.4	28.2	125	91.8	19.9	187
s5378	–	–	–	–	–	–	–	–	–
s9234	–	–	–	–	–	–	–	–	–

instances over time. It displays how many instances out of 72 can be decrypted (the vertical axis) by each algorithm if all instances are assigned the same timing budget (the horizontal axis). LIM is initially slower than the other two but eventually stands out. After 3,600 seconds, it decrypts 23% and 17% more instances than KC2 and RANE, respectively. Because it displays a similar trend, we omit the table of the HARPOON

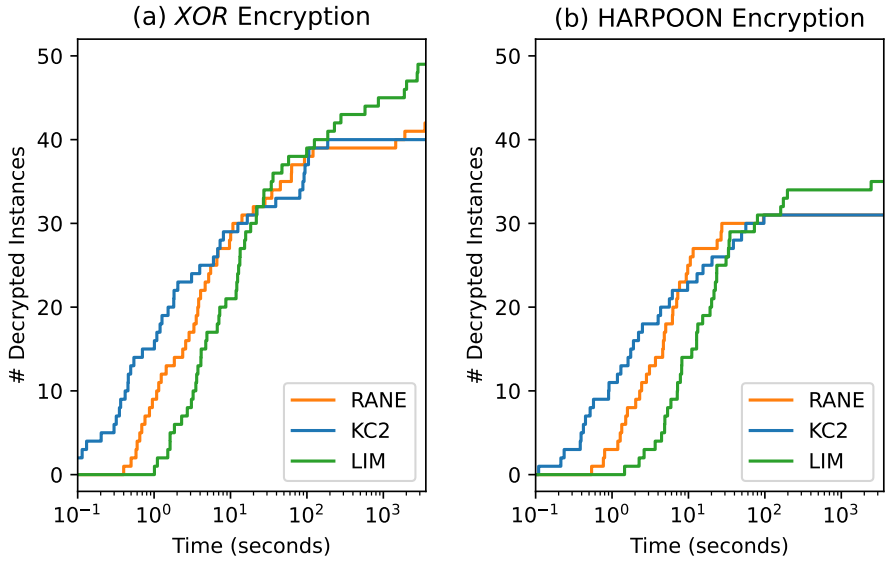


Figure 3.5. Comparing the number of decrypted instances over time.

Table 3.4. Execution time (seconds) of OLSAT on *XOR*-encrypted sequential benchmark circuits.

Overhead	5%	10%	15%	Overhead	5%	10%	15%
s208	2.4	31.9	605	s713	62.0	998	820
s298	0.9	18.3	901	s820	13.8	77.2	706
s344	1.0	6.3	683	s832	17.8	43.0	409
s349	0.9	10.8	1566	s838	–	–	–
s382	365	–	1079	s953	38.8	1350	–
s386	0.6	114	131	s1196	619	16.4	601
s400	296	1170	–	s1238	5.0	11.4	706
s444	155	–	–	s1423	–	–	–
s510	207	174	–	s1488	46.0	146	–
s526	–	–	–	s1494	339	804	–
s526n	–	–	–	s5378	–	–	–
s641	3.9	634	669	s9234	–	–	–

experiments. LIM can decrypt 13% more instances than either of the rest algorithms within 3,600 seconds on HARPOON-encrypted instances.

We further investigated why LIM is faster on hard instances. Figure 3.6 visualizes the decryption progresses of 4 instances that LIM performs better than the other algorithms.

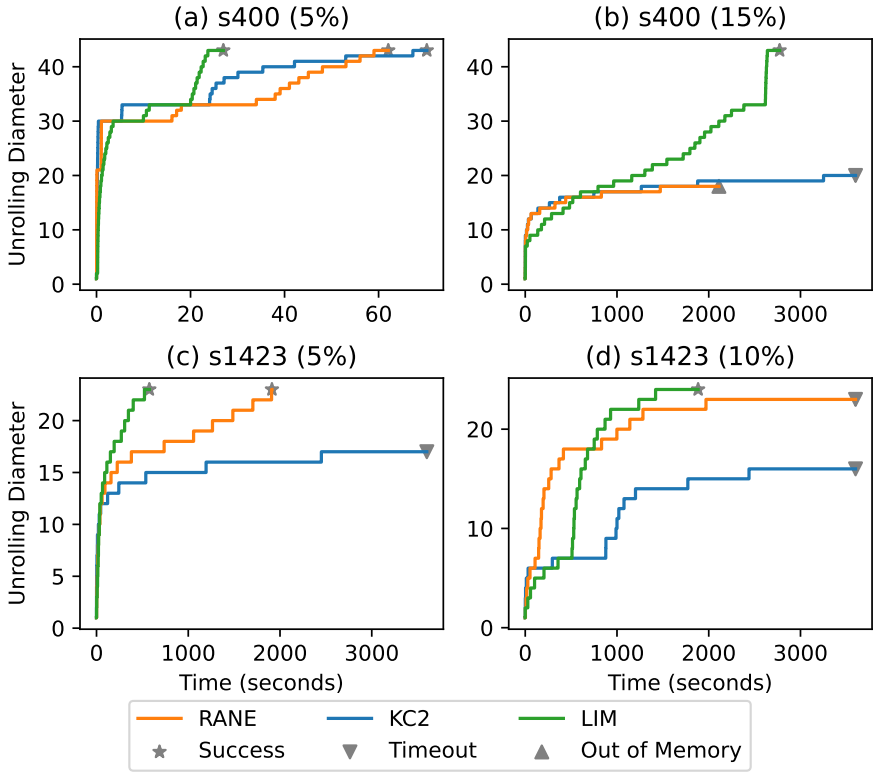


Figure 3.6. Decryption progress (unrollong diameter) *vs.* execution time for 4 selected instances.

It displays how long (the horizontal axis) each algorithm takes to eliminate all incorrect keys within a certain unrolling diameter (the vertical axis). LIM is initially slower as it can only eliminate a small cube of incorrect keys in one iteration. On the contrary, KC2 and RANE can rule out many incorrect keys with a single I/O constraint. However, since KC2 and RANE must duplicate the whole unrolled circuit for each I/O constraint, clauses pile up quickly within the SAT solver. As the number of steps grows, LIM takes significantly less time than the other algorithms for every iteration. In conclusion, LIM is more sustainable on hard instances which demand many iterations to solve.

Table 3.5. Statistics of high-level synthesized benchmark circuits.

Circuit	#PI	#PO	#FF	#Gate
euclid	8	5	11 - 22	231 - 315
gcd	8	5	11 - 37	255 - 766
barcode	9	10	17 - 35	279 - 591
counter	6	5	17 - 34	357 - 469
numeric	12	13	15 - 30	506 - 623
fuzzy	13	5	33 - 51	669 - 963
diffeq	20	13	27 - 42	864 - 1073
ellipf	32	33	33 - 63	1339 - 1433
kalman	22	9	14 - 69	1507 - 1909
wavef	32	33	72 - 108	1788 - 2183

3.5.2. Evaluating OLSAT for Oracle-less Attack

This section evaluates the capability of OLSAT on a variety of benchmarks. We used the same settings as Section 3.5.1 except that we restored *UMC* as a termination condition.

For every instance in Table 3.3, we inserted a stalling logic to extend its period by a constant. Existing I/O attacks cannot be applied to this scenario without knowing this constant exactly, because the oracle circuit C_o and the encrypted netlist C_e are different in timing. Table 3.4 summarizes the results of OLSAT on these instances. OLSAT can decrypt 64% of the instances within 3,600 seconds, and the average execution time of decrypted instances is 694.5 seconds. Meanwhile, as shown in Table 3.3, LIM can decrypt 68% of the instances within 3,600 seconds, and the average execution time of decrypted instances is 41.6 seconds. Although the results in Table 3.4 are not directly comparable with those in Table 3.3, we emphasize that most instances solvable by LIM are also solvable by OLSAT. We also observed that the *TE* strategy significantly reduces the execution time of *UMC*. This improvement is vital because *UMC* can dominate the total execution time on those instances which require a large unrolling diameter.

Table 3.6. Execution time (seconds) of OLSAT on high-level synthesized sequential benchmark circuits.

Latency	Small			Medium			Large		
Overhead	3%	5%	10%	3%	5%	10%	3%	5%	10%
euclid	0.7	97.8	6.3	1.6	8.9	–	19.9	1547	–
gcd	23.0	–	–	504	–	1.2	11.7	3569	–
barcode	2.0	907	–	7.0	42.4	181	–	5.1	–
counter	191	13.1	7.8	3.4	14.2	6.0	2.0	43.7	3.2
numeric	2.9	1437	48.7	2.0	–	203	3.6	1.3	–
fuzzy	34.7	–	–	1.8	–	–	–	45.8	–
diffeq	2.3	61.4	140	2.6	3.1	–	–	–	–
ellipf	12.9	13.3	257	640	–	–	–	5.5	5.7
kalman	130	8.6	–	–	9.1	–	–	–	–
wavef	–	–	–	239	219	–	–	–	142

We designed another experiment to assess OLSAT in a more realistic setting. Concretely, we used a high-level synthesis tool, Xilinx Vivado HLS, to synthesize a set of behavioral designs from the HLSynth benchmark suite [**59**] to RTL designs. We utilized HLS pragmas, including pipeline, initiation interval, resource allocation, latency, unroll, flatten, partition, balance, *etc.*, to control an RTL design's timing. For each behavioral design, we generated 4 RTL designs with distinct latencies. Afterwards, we applied Yosys [**82**] to map them to gate-level netlists. We treated the netlist with the smallest latency as the reference netlist C_d. Each of the remaining netlists of the same behavioral design is encrypted by the *XOR* encryption scheme with different encryption overheads. As such, we constructed 9 instances of the encrypted netlist C_e for every behavioral design. Table 3.6 shows the execution time of OLSAT on the realistic test cases. Given a timing budget of 3,600 seconds, OLSAT successfully decrypts 73%, 67%, and 40% of the instances encrypted with 3%, 5%, and 10% of overheads, respectively.

3.6. Conclusion

This chapter presents LIM, a synthesis-based attack against sequential logic encryption. Facilitated by LIM, it further presents OLSAT, an SAT-based I/O attack that does not require oracle access. Experiment results confirm that both LIM and OLSAT are realistically applicable. The chapter suggests a new method to explore the error matrix of an encrypted circuit. It also necessitates new defense techniques for logic encryption.

CHAPTER 4

Evaluating the Security of Logic Encryption on Deep Neural Networks

4.1. Introduction

The past decade has witnessed the broad deployment of Deep Neural Networks (DNNs) on hardware systems. Despite their successes, DNN models are suffering from IP piracy as well as security and privacy threats. Adversaries are motivated to extract the parameters of a DNN model for two main reasons. First, training a new DNN model is expensive. *a)* Deep learning tasks are data-hungry, while the data can be proprietary or requires enormous effort to collect. *b)* Performing model training involves high expertise, including selecting model architectures, training algorithms, and hyper-parameters. *c)* The final training process demands substantial computational resources and a long time. Second, the exposure of the model parameters poses severe threats to the security and confidentiality of the model, even if that model is deployed elsewhere by other end-users. With access to a leaked model, an adversary can launch *a)* the evasion attack [**9**] and the data poisoning attack [**77**] to abuse or deceive the victim DNN model, or *b)* the model inversion attack [**34**] to reconstruct sensitive training examples.

Logic encryption [**71, 78**] is a well-established technique to protect the intellectual property of an integrated circuit. It embeds *binary* key bits to the circuit netlist (a network of logic gates). The original functionality of a logic-encrypted circuit can be fully

restored only when a correct key is inserted. Logic encryption achieves four objectives simultaneously [**90**]: *i) locking robustness*, which ensures that decryption is prohibitively expensive; *ii) structural security*, which ensures that recovering the original functionality by removing or bypassing a submodule of the encrypted network is impossible; *iii) encryption efficiency*, which ensures that only a small number of key bits (in the magnitude of hundreds) and minimum overheads are required for encryption; and *iv) end-to-end protection*, which ensures that even a malicious insider or an end-user who possesses a working chip cannot duplicate the circuit without authorization.

Various techniques are proposed to protect the intellectual property of DNNs. Unfortunately, most of them cannot be applied to hardware accelerators (*e.g.,* TPU, NPU, GPGPU, and dedicated AI acceleration chips) without sacrificing security and efficiency. For example, DNN watermarking [**2, 39**] cannot prevent illegal private usage because it does not alter the functionality of the network. Input, output, and convolutional kernel obfuscations [**58, 38**] are susceptible to removal attack or unauthorized reproduction by the end-user, because the key bits are not entangled with learnable parameters. Public-key encryptions [**85**] will incur large overheads on hardware accelerators.

Recently, Chakraborty *et. al.* proposed Hardware Protected Neural Network (HPNN) [**14**], which employs logic encryption to protect the intellectual property of a DNN model. HPNN selects a small subset of neurons in the hidden layers of the DNN as the *key-protected neurons*. A key bit is associated with every such neuron, controlling whether to *flip* the sign of the pre-activation value. A DNN model is trained as a function of a pre-selected key pattern. In this way, model parameters and key bits are closely entangled. The key is stored on hardware in a tamper-proof memory [**5**] or a Trusted Platform

Module (TPM) [**62**]. A combination of a hardware accelerator, a key storage module, and a correct key serves as a license to access the service provided by the IP owner. A DNN model and its future parameter updates can be released publicly on cloud platforms in an MLaaS fashion. Subscribers who have acquired valid licenses can fully restore the functionality of the DNN model. On the contrary, a model's prediction accuracy will be severely degraded without a correct key.

This chapter studies whether logic encryption is secure on DNNs. Our investigations reveal that logic-encrypted DNNs are still vulnerable to I/O attacks. Specifically, a malicious end-user can query a working network for a set of selected input samples and utilize the observed outputs to infer the key values. In this way, it can replicate the DNN model without the permission of the IP owner.

We propose a systematic attack algorithm for all feasible DNN logic encryption schemes we can foresee. Our attack is based on two observations: *i)* reversing a ReLU activation function over the y-coordinate does not change the location of its critical point; *ii)* the location of a critical point is determined collectively by the key values of the preceding layers.

The attack algorithm leverages divide-and-conquer to minimize sample complexity. Starting from the first hidden layer, it targets one layer in every iteration. For each hidden layer, the algorithm attempts to infer the value of a key bit by moving along the coordinate in the hidden space. If such an attempt fails, it initiates a learning-based attack to predict the key value. Once all the key values are extracted for a hidden layer, the algorithm executes a rigorous validation and correction procedure to fix errors. According to our

experiments, the attack algorithm can scale to large DNNs and generalize to complex network architectures.

Our main contributions are:

• We establish a theoretical framework for security analysis of logic-encrypted deep ReLU networks;

• We develop an algebraic approach to infer the value of a single key bit when the network is contractive;

• We propose a learning-based approach as well as a validation and correction procedure to extract the key values of a hidden layer when the network is expansive;

• We evaluate accuracy, scalability, and generality for the attack algorithm on practical networks.

4.2. Background

4.2.1. Preliminaries

A deep neural network can be represented as a function $f : \mathcal{X} \to \mathcal{Y}$, which takes inputs from the input space $\mathcal{X} \subseteq \mathbb{R}^P$ and returns outputs to the output space $\mathcal{Y} \subseteq \mathbb{R}^Q$. A k-layer deep neural network [**13**] f is an alternating sequence of linear transformations and non-linear activation functions:

$$f = f_{k+1} \circ \sigma \circ f_k \circ \cdots \circ \sigma \circ f_1. \tag{4.1}$$

In the above equation, the i-th *hidden layer* is given by a linear transformation f_i followed by an element-wise ReLU activation function σ. Specifically, $f_i(x_{i-1}) = A^{(i)}x_{i-1}+$

Table 4.1. Notations used throughout this chapter.

Notation	Description
$f \circ g$	function composition
$M_1 M_2$	matrix multiplication
$c \cdot M$	scalar multiplication
$M_1 * M_2$	element-wise multiplication
$f : \mathcal{X} \to \mathcal{Y}$	a deep neural network
$\mathcal{X} \subseteq \mathbb{R}^P$	input space of f
$\mathcal{Y} \subseteq \mathbb{R}^Q$	output space of f
k	total number of hidden layers of f
$f_1, \cdots, f_k$	hidden layers of f
f_{k+1}	output layer of f
ϕ	ReLU activation function
σ	element-wise ReLU activation function
d_i	dimension of the i-th hidden layer
$A^{(i)} \in \mathbb{R}^{d_i \times d_{i-1}}$	weight matrix of the i-th hidden layer
$b^{(i)} \in \mathbb{R}^{d_i}$	bias vector of the i-th hidden layer
$m^{(i)} \in \{0,1\}^{d_i}$	activation pattern vector of the i-th hidden layer
$z_i \in \mathbb{R}^{d_i}$	pre-activation hidden state of the i-th hidden layer
$x_i \in \mathbb{R}^{d_i}$	post-activation hidden state of the i-th hidden layer
$\eta_{i,j}$	j-th neuron on the i-th hidden layer
$h_{i,j}$	the (bent) hyperplane induced by $\eta_{i,j}$
$K_{i,j} \in \{0,1\}$	key value of $\eta_{i,j}$ (if it is key-protected)
$\hat{A}^{(i)}$	product weight matrix of a layer-i linear region
$\hat{b}^{(i)}$	product bias vector of a layer-i linear region
$\mathcal{O}$	the oracle network corresponding to f
K^*	a correct key within the equivalent class
$K'_{i,j} \in [-1,1]$	floating-point version of K
θ	(quantization) error tolerance

$b^{(i)}$ is an affine transformation, in which the *post-activation hidden state* $x_{i-1} \in \mathbb{R}^{d_{i-1}}$ is a d_{i-1}-dimensional vector, the *weights* $A^{(i)} \in \mathbb{R}^{d_i \times d_{i-1}}$ is a d_i by d_{i-1} matrix, and the *biases* $b_i \in \mathbb{R}^{d_i}$ is a d_i-dimensional vector. Notice that the final *output layer* f_{k+1} is not followed by activation functions. All parameters consisting $A^{(i)}$ and $b^{(i)}$, $i \in 1, \cdots, k+1$, are learnable parameters that can be updated during training.

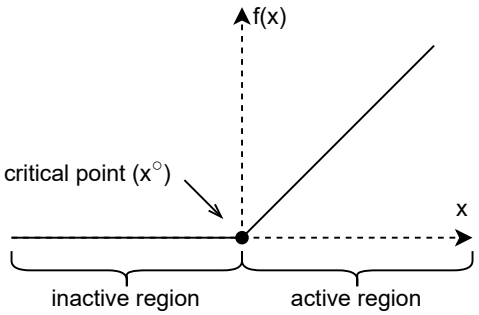


Figure 4.1. The Rectified Linear Unit (ReLU) activation function. The function is at the critical point if its input equals 0.

The above formulation assumes that all the hidden layers are *fully connected layers*. In reality, a convolutional neural network can have *convolutional layers* and *maximum pooling layers*, but they can also be represented as linear transformations locally. Additionally, a *softmax layer*, $softmax(x_{k+1}) \in \mathbb{R}^Q \times \mathbb{R}^Q$, can be attached to the output layer. In this case, given an input example $x \in \mathcal{X}$, we call x_{k+1} the *logits* of $f(x)$ and $y = softmax(x_{k+1})$ the output vector of $f(x)$.

Each component of σ is a ReLU activation function [**55**] defined as $\phi(z) = max(z, 0)$. ReLU has established itself as the default choice for deep learning because DNNs with ReLUs can be optimized more easily [**67**]. As shown in Figure 4.1, a ReLU is a piecewise linear function. We denote the j-th *neuron* in the i-th hidden layer as $\eta_{i,j}$. It computes $x_{i,j} = \phi(A_j^{(i)} x_{i-1} + b_j^{(i)})$, where $A_j^{(i)}$ is the j-th row of $A^{(i)}$ and $b_j^{(i)}$ is the j-th element of $b^{(i)}$. Particularly, we refer to $z_{i,j} = A_j^{(i)} x_{i-1} + b_j^{(i)}$ as the *pre-activation value* of $\eta_{i,j}$. We say that $\eta_{i,j}$ is at its *critical point* if $z_{i,j} = 0$. Moreover, a neuron is *inactive* if $z_{i,j} \leq 0$ and is *active* if $z_{i,j} > 0$.

We summarize the notations used in this chapter in Table 4.1.

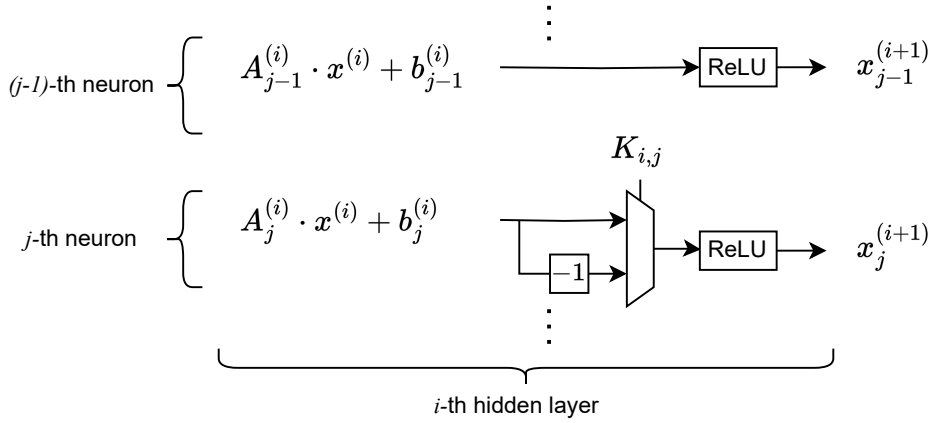


Figure 4.2. The implementation of HPNN on a hardware accelerator. A flipping unit is embedded to the j-th neuron of the i-th hidden layer. The pre-activation value of the protected neuron is negated when the key value equals 1.

4.2.2. HPNN Implementation

The actual implementation of HPNN is depicted in Figure 4.2. HPNN embeds several flipping units into the hardware accelerator, with each unit governing the functionality of an associated neuron in a hidden layer. A flipping unit takes as inputs *i)* the pre-activation value of the associated neuron, and *ii)* a key bit that determines whether to negate the pre-activation value. Throughout the training phase, the weights and the biases of the deep neural network are updated as functions of not just the training data but also the key values.

4.2.3. Adversary Model

We use the standard adversary model for logic encryption [**78, 90**]. The adversary is an end-user who can either download the model architecture and all the parameters from a cloud platform or receive them from the IP owner. The adversary cannot read or probe the key from an instance of the hardware accelerator because it is stored in a tamper-proof

memory or a trusted platform module. Despite this, once the model is installed onto the hardware, the adversary can query the model with arbitrary inputs a reasonable number of times. It can then observe the logits or the output vector produced by the model.

Compared to the assumptions in the HPNN paper [**14**], our adversarial model is even weaker, given that we do not require the adversary to have access to a subset of labeled data.

4.2.4. Adversarial Goal

The adversary aims to obtain a correct key, denoted as K^*. When such a key is embedded into the hardware, the encrypted DNN model should be *functionally equivalent* to the original model. Once the adversary acquires such a key, it can replicate and distribute the DNN model without permission from the IP owner. Additionally, the adversary can compromise a remote mission-critical system that uses the same DNN model by launching an adversarial attack on the local model.

4.3. Mathematical Investigations

In this section, we establish a theoretical framework for DNN security analysis. Some of the notions used in this section are adapted from explainable AI [**35, 84, 40**] and model extraction [**43, 13, 69**] studies.

4.3.1. Formal Analysis of HPNN

Consider a key-protected neuron in a DNN encrypted by HPNN. The following lemma elaborates how the key bit modifies the behaviour of that neuron:

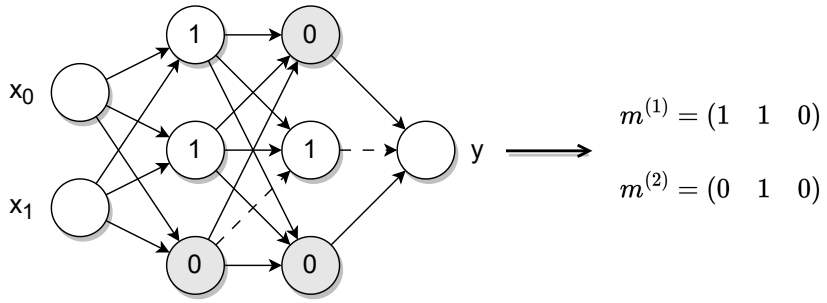


Figure 4.3. Activation patterns of deep ReLU networks. Left: statuses of ReLU activation functions of a 2-layer DNN. Right: the corresponding activation patterns. The dashed path is a sensitizable path from an intermediate node to the output node.

Lemma 11. *Flipping a key bit* $K_{i,j} \in \{0, 1\}$ *has the same effect as negating* $A_j^{(i)}$ *and* $b_j^{(i)}$ *while keeping the remaining parameters unchanged.*

Proof. Let us focus on the neuron $\eta_{i,j}$, whose pre-activation value $z_{i,j} = (-1)^{K_{i,j}}(A_j^{(i)} x_{i-1} + b_j^{(i)})$. This value equals $A_j^{(i)} x_{i-1} + b_j^{(i)}$ when $K_{i,j} = 0$, or $-A_j^{(i)} x_{i-1} - b_j^{(i)}$ when $K_{i,j} = 1$. In contrast, $K_{i,j}$ affects none of the other neurons (and thus no other parameters) in the same layer. □

In other words, a key bit determines whether to reverse the corresponding *row signs* of the associated matrices. An adversary can thus recover the correct key value by querying the oracle model with carefully selected input examples. We discuss the detailed approaches in Section 4.4.

4.3.2. Formal Analysis of Deep ReLU Networks

Consider a DNN $f : \mathbb{R}^P \to \mathbb{R}^Q$. Each neuron $\eta_{i,j}$ on a hidden layer of f induces a $(P-1)$-dimensional bent *hyperplane* in the input space. A bent hyperplane $h_{i,j}$ consists of such points in the input space that the ReLU activation function within $\eta_{i,j}$ is at its critical

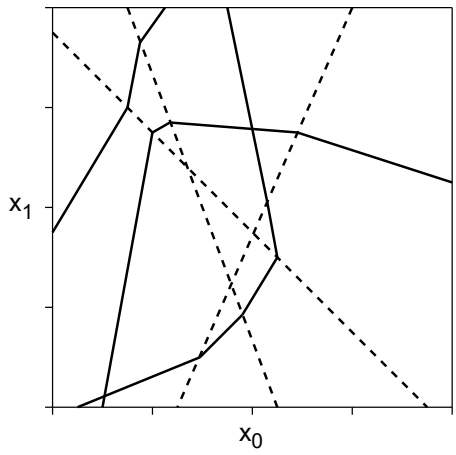


Figure 4.4. Geometric view of hyperplanes of the 2-layer DNN in Figure 4.3. Each dashed line (resp. bent solid line) is induced by a ReLU in the first (resp. second) hidden layer.

point. A hyperplane induced by a neuron in the first hidden layer is indeed "flat". In seek of conciseness, we refer to both flat and bent hyperplanes as "hyperplanes" in the rest of this chapter. Figure 4.4 shows the hyperplanes of a 2-layer DNN.

The hyperplanes of a DNN splits the input space into disjoint *linear regions*. For an input example $x_0 \in \mathcal{X}$, one can compute the pre-activation values for all neurons through a single forward pass. We use an *activation pattern* vector $m^{(i)} \in \{0,1\}^{d_i}$ to represent the activation statuses for all the neurons in the i-th hidden layer (Figure 4.3). The j-th element of that vector, $m_j^{(i)}$, is 1 if $A_j^{(i)} x_{i-1} + b_j^{(i)} > 0$, otherwise it is 0. Two input examples are within the same linear region if they share the same activation patterns across all hidden layers.

Each linear region is associated with a unique affine transformation from the input space. Given the activation patterns of a linear region, one can recursively compute the weights and the biases of the transformation, layer by layer:

$$M^{(i)}_{j,:} = m^{(i)}_j,\ \hat{A}^{(1)} = A^{(1)},\ \hat{b}^{(1)} = b^{(1)}, \tag{4.2}$$

$$\hat{A}^{(i)} = A^{(i)}(\hat{A}^{(i-1)} * M^{(i-1)}), \tag{4.3}$$

$$\hat{b}^{(i)} = A^{(i)}(\hat{b}^{(i-1)} * m^{(i-1)}) + b^{(i)}. \tag{4.4}$$

In the above equations, $*$ denotes element-wise multiplication between matrices, and $M^{(i)}$ is the mask matrix obtained by broadcasting $m^{(i)}$ to all columns of $\hat{A}^{(i)}$. Intuitively, according to the activation patterns, the above recursive formulas select either the inactive or the active region for all ReLU activation functions. We refer to $\hat{A}^{(i)}$ (resp. $\hat{b}^{(i)}$) as the *product weight matrix* (resp. *product bias vector*) of a level-i linear region.

It can be seen that the linear regions exhibit a hierarchical structure: a linear region in one layer depends upon its predecessor regions in the previous layers. Hence, we can derive the following lemma:

Lemma 12. *The hyperplane induced by $\eta_{i,j}$ is exclusively determined by $A^{(1)}, \cdots, A^{(i-1)}$, $b^{(1)}, \cdots, b^{(i-1)}$ as well as $A^{(i)}_j$ and $b^{(i)}_j$ sans the row sign associated to $A^{(i)}_j$ and $b^{(i)}_j$.*

Proof. The hyperplane induced by $\eta_{i,j}$ consists of those points x_0 in the input space such that $A^{(i)}_j x_{i-1} + b^{(i)}_j = 0$, where $x_{i-1} = (\hat{A}^{(i-1)}x_0 + \hat{b}^{(i-1)}) * m^{(i-1)}$ if $i > 1$. According to the definition of $m^{(i)}$ and Formulas 4.2-4.4, x_{i-1} only depends on x_0 itself and the weights and biases of layers 1 through $i-1$. Moreover, reversing the signs of $A^{(i)}_j$ and $b^{(i)}_j$ simultaneously does not change the above equations. □

Lemma 12 suggests a method to validate the key values of layers 1 through $i-1$. Recall that a key bit of HPNN on layer i modifies the weights and biases of the same layer by reversing a row sign, thus conserving all hyperplanes on layer i. Hence, without information of the key bits on layer i, the adversary can still confirm that all the key bits of the previous layers are correct if every hyperplane computed from the white-box network exactly matches a hyperplane inferred from the oracle network.

Furthermore, even without the information from the white-box network, the adversary can infer which layer a hyperplane belongs to:

Lemma 13. *Let two hyperplanes $h_{p,m}$, $h_{q,n}$ be induced by neurons $\eta_{p,m}$ and $\eta_{q,n}$, respectively. The fact that $h_{q,n}$ bends at the intersection of the two hyperplanes while $h_{p,m}$ remains flat implies $p < q$. That neither hyperplane bends at the intersection implies $p = q$.*

Lemma 13 has been discovered and proved in prior literature [**69**]. With a little abuse of terminology, we refer to every flat piece of a bent hyperplane as a *boundary*.

As stated in Section 4.2.3, an adversary can only access the output nodes of a DNN. Intuitively, the existence of a boundary can be revealed by observing the non-linear changes from the output nodes. This is not the case if the boundary is covered by subsequent layers. The following lemma discusses when a boundary is sensitizable to an output:

Lemma 14. *Consider a set of compatible activation patterns and the corresponding linear region. In addition, consider a boundary of the hyperplane induced by $\eta_{i,j}$ on the*

linear region. This boundary is sensitizable to an output node y, if there exists a consecutive neural path from $\eta_{i,j}$ (exclusive) to y such that all neurons along the path are active.

Proof. The neuron $\eta_{i,j}$ must be active on one side of the boundary and inactive on the other. On its active side, a change in its pre-activation value $z_{i,j}$ can be propagated to y along the assumed path. Propagation along the same path is blocked on its inactive side. This difference causes non-linearity across the boundary, which can be observable from y. □

A set of activation patterns are compatible if the linear region associated with them is not vacant. In the example shown in Figure 4.3, the last neuron in the first hidden layer is observable from y, because there exists a consecutive 1-path to y from this neuron. In a modern DNN architecture [**51**], most neurons in hidden layers have multiple successor neurons. Consequently, it is very unlikely that an intermediate neuron has no sensitizable path to any output nodes.

4.4. Algorithm

4.4.1. Overview

We present a logic decryption algorithm for deep ReLU networks with guaranteed correctness. Our attack is launched consecutively on every hidden layer of a network. Once it completes the extraction and validation for all key bits in a hidden layer, the attack is repeated on the next layer.

We construct two approaches to determine the value of a key bit. The first approach infers $K_{i,j}$ by moving along the coordinate of $\eta_{i,j}$ in the hidden state space (Section 4.4.3).

The second approach extracts key values through a learning process (Section 4.4.4). Because the first approach is more accurate and lightweight, the second approach is initiated only when necessary. Extracted key values of a hidden layer are confirmed when they altogether pass a rigorous validation process (Section 4.4.5). As such, the final key returned by the algorithm is guaranteed to be correct.

4.4.2. Finding Critical Points of a Neuron

Algorithm 5 The Critical Point Searching Function

Input: white-box network f, a neuron $\eta_{i,j}$,
decrypted key values of preceding layers $K_1^*, \cdots, K_{i-1}^*$
Output: a witness $x^\circ(\eta_{i,j}) \in \mathcal{X}$ of the hyperplane
induced by $\eta_{i,j}$
Collect points in the input space along a straight line until for two neighboring points x_l and x_r,
$sign(z_{i,j}(x_l)) \neq sign(z_{i,j}(x_r))$ ▷ *pre-activation value*
$x_m \leftarrow$ the middle point of x_l and x_r
while $|z_{i,j}(x_m)| > \delta$ **do** ▷ *precision*
    $x_l \leftarrow x_l$ **if** $sign(z_{i,j}(x_l)) = sign(z_{i,j}(x_m))$ **else** x_m
    $x_r \leftarrow x_r$ **if** $sign(z_{i,j}(x_r)) = sign(z_{i,j}(x_m))$ **else** x_m
    $x_m \leftarrow$ the middle point of x_l and x_r
return x_m

Our attack relies on an essential utility function, whose goal is to find a witness $x^\circ \in \mathcal{X}$ to a designated hyperplane. Recall that a hyperplane generally has $P-1$ dimensions. As a result, a 1-dimensional line is likely to intersect with the hyperplane at least once [**86**]. In addition, as stated in Lemma 12, the hyperplane of a neuron is exclusively determined by the key bits in the preceding layers. Because our algorithm proceeds layer by layer, the current information in the white-box network f is sufficient for the goal.

The critical point search function is displayed in pseudo-code in Algorithm 5. It starts by randomly selecting a straight line in the input space. Afterwards, it draws random samples along the line and tracks the pre-activation values $z_{i,j}$ of the associated neuron $\eta_{i,j}$. Once it detects two consecutive samples that yield opposite signs for $z_{i,j}$, it performs a standard binary search on the line segment between the two samples (Line 5-8). Upon completion, it finds a sample on the designated hyperplane.

4.4.3. Key Inference with Basis Vector

Algorithm 6 The Key Value Inference Function

1: **Input:** white-box network f, oracle network $\mathcal{O}$, key-protected neuron $\eta_{i,j}$, decrypted key values of preceding layers $K_1^*, \cdots, K_{i-1}^*$
2: **Output:** $K_{i,j}^*$
3: $x^\circ \leftarrow$ `search_critical_point`$(\eta_{i,j})$
4: Observe $m^{(1)}, \cdots, m^{(i-1)}$ with a forward pass from x°
5: Compute $\hat{A}^{(i)}$ according to Formulas 4.2-4.3
6: $e_{i,j} \leftarrow$ the j-th standard basis vector in $\mathbb{R}^{d_i}$
7: Find a $v_{i,j}$ using least squares *s.t.* $\hat{A}^{(i)} v_{i,j} = e_{i,j}$
8: **if** $v_{i,j}$ does not exist **then**
9: ⌊ **return** $\varnothing$
10: **if** $\mathcal{O}(x^\circ) \neq \mathcal{O}(x^\circ + \epsilon \cdot v_{i,j})$ **then**
11: ⌊ **return** 0
12: **if** $\mathcal{O}(x^\circ) \neq \mathcal{O}(x^\circ - \epsilon \cdot v_{i,j})$ **then**
13: ⌊ **return** 1
14: **return** $\varnothing$

Consider the pre-activation hidden space of the i-th hidden layer, $\mathbb{R}^{d_i}$. Let $e_{i,j}$ denote the j-th *standard basis vector* of $\mathbb{R}^{d_i}$ and let $z_{i,j}^\circ \in \mathbb{R}^{d_i}$ denote an intermediate critical point of $\eta_{i,j}$. With a sufficiently small ϵ [**13**], it is guaranteed that the ϵ-neighborhood of $z_{i,j}^\circ$ does not intersect with any hyperplanes induced by other neurons. In other words, all the points in this neighborhood are within the same linear region. Moving along $e_{i,j}$

within the neighborhood can only change $z_{i,j}$ but not any other pre-activation values of the same hidden layer because, by definition, $e_{i,j}$ is parallel to the j-th coordinate of $\mathbb{R}^{d_i}$ and orthogonal to any other coordinates. Therefore, we can derive the following lemma:

Lemma 15. *Let $e_{i,j}$ represent the j-th standard basis vector of $\mathbb{R}^{d_i}$, $v_{i,j} \in \mathbb{R}^P$ represent a pre-image of $e_{i,j}$, and $x^\circ(\eta_{i,j}) \in \mathbb{R}^P$ represent a critical point of $\eta_{i,j}$. Then $K^*_{i,j} = 0$ implies that $\mathcal{O}(x^\circ(\eta_{i,j}) - \epsilon \cdot v_{i,j}) = \mathcal{O}(x^\circ(\eta_{i,j}))$, and $K^*_{i,j} = 1$ implies that $\mathcal{O}(x^\circ(\eta_{i,j}) + \epsilon \cdot v_{i,j}) = \mathcal{O}(x^\circ(\eta_{i,j}))$.*

Proof. If $K^*_{i,j} = 0$, the pre-activation value $z_{i,j}(x)$ is within the inactive region of the ReLU activation function (Figure 4.1) for both $x_1 = x^\circ(\eta_{i,j})$ and $x_2 = x^\circ(\eta_{i,j}) - \epsilon \cdot v_{i,j}$. Besides, moving along $v_{i,j}$ does not change any other pre-activation values of the same hidden layer, as $e_{i,j}$ is orthogonal to other coordinates in the hidden space. Because all elements of z_i remain unchanged, the oracle network must produce identical outputs for both input samples.

The same reasoning applies to the $K^*_{i,j} = 1$ case. □

Algorithm 6 illustrates how we implement the key value inference procedure. It starts by finding a critical point x° of the targeted neuron (Line 3). Then it computes the product weight matrix associated with the level-i linear region where x° is located (Line 4-5). Given the product weight matrix $\hat{A}^{(i)}$, we are able to compute the pre-image vector $v_{i,j} \in \mathbb{R}^P$ for $e_{i,j}$. In practice, we find the pre-image with *least squares* (Line 7), which is a built-in function provided by statistics and deep learning frameworks such as SciPy [**81**] and PyTorch [**60**]. Finally, the algorithm determines the key value through queries to the oracle network (Line 10-13). Notice that the statements on Line 10 and Line 12

are contra-positive to their counterparts in Lemma 15. In some rare cases (*e.g.*, when the condition of Lemma 14 does not hold), all the three values obtained from the oracle queries are close to each other. In that circumstance, we attempt to find another x° and start over the entire procedure.

However, there is still a caveat in the above discussions: a $v_{i,j}$ may not exist for a specific x° (Line 8). First, a deep neural network can be *expansive* [**68**] in certain locations. In a nutshell, if $d_l < d_i$ for some $l < i$, then $v_{i,j}$ does not exist for every $e_{i,j}$ because $\hat{A}^{(i)}$ is not an onto mapping. Second, inactive neurons reduce the chance of finding a $v_{i,j}$. For a randomly initialized network, about half of the neurons on a hidden layer are inactive for a given input example. The situation worsens when the network encounters a "dying ReLU" problem [**50**]. Hence, we need a complementary approach to address these issues.

4.4.4. Learning-based Attack

Suppose that K_i^* contains $\varnothing$ elements, *i.e.*, the key bits on the current hidden layer are not settled by the key inference algorithm. We perform a supervised learning attack on these remaining key bits.

To convert an HPNN-encrypted network model to a continuous function, we substitute every flipping unit (Figure 4.2) with a scalar multiplication operator. This operator multiplies the pre-activation value $z_{i,j}$ with a floating point number $K'_{i,j} \in [-1, 1]$ before the ReLU activation function. Then we create a training dataset using the oracle network. Specifically, we randomly generate a set of unlabeled input examples $x_1 \cdots x_n \in \mathcal{X}$ and query the oracle for the corresponding outputs. During the training process, we fix all

the weights and biases of the white-box network. Moreover, for all the key bits in the preceding layers and the non-$\varnothing$ key bits in the i-th layer, we enforce $K'_{i,j}$ to be -1 or 1 if the original $K_{i,j}$ is 1 or 0, respectively. Upon termination, we replace a $\varnothing$ with 0 if $K'_{i,j}$ is a positive number and with 1 otherwise.

The learning-based approach cannot guarantee absolute correctness for the extracted key values. Meanwhile, a single bit of error is devastating to our attack on subsequent layers. In this regard, we must validate the correctness of K_i^* before proceeding to the next layer.

4.4.5. Validating Extracted Key Values

Algorithm 7 The Key Vector Validation Function

Input: white-box network f, oracle network $\mathcal{O}$, decrypted key values of the current layer K_i^*, decrypted key values of preceding layers $K_1^*, \cdots, K_{i-1}^*$
Output: `true` or `false` ▷ *pass or fail the validation*
overlaps $\leftarrow 0$
for $1 \leq j \leq d_{i+1}$ **do**
    $x^\circ(\eta_{i+1,j}) \leftarrow$ `search_critical_point`$(\eta_{i+1,j})$
    if `non_linearity`$(\mathcal{O}(x^\circ(\eta_{i+1,j})), \delta)$ **then** ▷ *the oracle network encounters a hyperplane at the same place*
        overlaps $=$ *overlaps* $+ 1$
return *overlaps*$/d_{i+1} > \theta$ ▷ *reach the threshold*

From Lemma 12, the hyperplane induced by $\eta_{i+1,j}$ is uniquely determined by the key values on layers 1 through i. Moreover, Lemma 14 proves that portions of hyperplanes are nearly always sensitizable to the outputs. Leveraging these facts, we can devise an algorithm to check the correctness of K_i^*. Intuitively, if K_i^* is correct, for a level-$(i+1)$ hyperplane of the white-box network, we can almost always find the exact hyperplane of the oracle network at the same location in the input space. On the contrary, if K_i^* is

incorrect, it is almost impossible to find any hyperplanes with the oracle network at that location, given the high dimensionality of the input space.

Algorithm 7 displays the key vector validation algorithm. For each neuron in the $(i+1)$-th layer, the algorithm searches a witness $x^\circ \in \mathcal{X}$ to its induced hyperplane (Line 5). It then samples a set of points within the δ-neighborhood of $x^\circ(\eta_{i+1,j})$, queries the oracle for the corresponding outputs, and verifies whether all the outputs are on the same linear surface (Line 6). There must exist a hyperplane crossing the δ-neighborhood if those points belong to more than one linear region.

The number of overlaps can be over-estimated when *i)* a boundary of the white-box network overlaps with an unrelated boundary of the oracle network, and can be under-estimated when *ii)* a boundary is not sensitizable to the outputs. We tolerate these uncertainties with a threshold θ. However, the exact value of θ is unimportant as we are unlikely to encounter either of the problems on a real network. In practice, it is sufficient to conclude that K_i^* is correct if we find overlaps for a few neurons. Validating the key vector for the last hidden layer requires special treatment, as its next layer (the output layer) has no ReLU activation functions. However, we can directly compare the outputs of the two networks for a set of input samples. It is possible because all remaining key bits are already determined at the time.

4.4.6. The DNN Decryption Algorithm

Algorithm 8 summarizes our main procedure, the DNN decryption algorithm. Starting from the first hidden layer, the algorithm sequentially iterates through all key-protected layers. For every layer, it traverses all key-protected neurons in an attempt to infer its key

Algorithm 8 The DNN Decryption Algorithm

1: **Input:** white-box network f, oracle network $\mathcal{O}$
2: **Output:** a correct key K^*
3: Let K^* be a 2-D array with k rows
4: **for** $1 \leq i \leq k$ **do** ▷ *hidden layers*
5: **for** $1 \leq j \leq d_i$ **do** ▷ *neurons*
6: **if** $\eta_{i,j}$ is key-protected **then**
7: $K^*_{i,j} \leftarrow$ `key_value_inference`$(\eta_{i,j})$
8: $K^*_i \leftarrow$ `learning_attack`(K^*_i)
9: **while not** `key_vector_validation`(K^*_i) **do**
10: $K^*_i \leftarrow$ `error_correction`(K^*_i)
11: **return** K^*

value (Line 7). Nevertheless, such an attempt may fail for a subset of neurons on some hidden layers. In that case, the algorithm initiates the learning-based attack to extract the key values of these neurons (Line 8).

The algorithm checks the key values K^*_i before it moves to the next hidden layer (Line 9). If K^*_i cannot pass the validation, the algorithm enters an error correction procedure (Line 10). Concretely, it first computes the confidence level for each key bit, which is defined as the absolute value of $K'_{i,j}$. A higher confidence level implies that the learned key value is more likely to be correct. Throughout its execution, the procedure maintains a counter of Hamming Distance, whose initial value is 1. In ascending order of the confidence level, the procedure attempts to flip each bit within the limit of Hamming Distance. The counter is incremented by 1 if all attempts fail for the current Hamming Distance.

The next theorem proves the correctness of the DNN decryption algorithm.

Theorem 16. *The DNN decryption algorithm will eventually terminate. Upon termination, it will always return a correct key.*

Proof. *Termination:* The `key_vector_validation` procedure can execute at most $2^{|K_i|}$ times for the i-th hidden layer, because the `error_correction` procedure eliminates one incorrect assignment to K_i each time. Other procedures can execute a finite number of times.

Correctness: Each row of K^* can be confirmed only when it passes the rigorous `key_vector_validation` process. This mechanism ensures that the final K^* is correct. □

In the following, we briefly analyze the sample complexity of the proposed algorithm. The overall complexity is dominated by the `key_value_inference` and the `key_vector_validation` procedures. The former procedure is called for a total of $|K|$ times. The latter can be executed for at most $\Sigma_i 2^{|K_i|}$ times. In reality, the `key_value_inference` is likely to settle all key bits on a hidden layer. In cases where `key_value_inference` does not work well, the `learning_attack` procedure can recover a large proportion of key bits. Combined with the `error_correction` procedure, our algorithm can usually find a correct assignment to the remaining key bits in a small number of attempts. On the other hand, the utility function `search_critical_point` requires $O(log(dist/\delta))$ forward passes on the white-box network, where $dist$ represents the distance between the initial points and δ is the error threshold. The `learning_attack` is initiated once for every hidden layer. This attack takes a far smaller dataset than the original learning task of the DNN, because it only targets a subset of K as opposed to all the elements within the $A^{(i)}$ and $b^{(i)}$ matrices. Overall, the sample complexity of the DNN decryption algorithm is $O(log(dist/\delta) \cdot \Sigma_i 2^{|K_i|})$ in the worst case, and

$O(log(dist/\delta) \cdot (|K| + |K|^{c|K|}))$ in the average case. Here $c \ll 1$ is a small constant representing the average probability that either `key_value_inference` or `learning_attack` fail on a key bit.

4.4.7. Discussions

Our attack can be generalized to a broad category of logic encryption methods on DNNs. As far as we can foresee, a defender can also *a)* modify a single element within the weight or bias matrices instead of the pre-activation value; or *b)* embed key bits to convolutional or max-pooling layers rather than fully connected layers; or *c)* choose another arithmetic operator instead of the negation operator. Unlike HPNN, none of these methods ensures that all key bits in a hidden layer are mutually dependent. Therefore, they are less robust than HPNN.

Case a): Modifying elements of $A_j^{(i)}$ or $b_j^{(i)}$ changes the geometry of $h_{i,j}$. On the contrary, such a modification does not change the geometry of $h_{i,k},\ k \neq j$. As a result, an adversary can leverage a divide-and-conquer strategy on each neuron to significantly reduce the sample complexity.

Case b): Similarly, the adversary can tackle one kernel in a convolutional layer or one patch in a maximum pooling layer at a time.

Case c): Suppose that the defender chooses to multiply the pre-activation value by a positive number when the key value is 1. Due to the distributive law of multiplication and the property of the ReLU function, an adversary can propagate a positive operand to all the successor neurons in the next hidden layer. It can then focus on a single neuron in the next hidden layer to decrypt the key bits within the fan-in cone.

Table 4.2. Configurations of the DNNs. (Conv: convolutional layer, MP: maximum pooling layer, FC: fully connected layer.)

DNN	Network Architecture	# Neurons	# Learnable Parameters
ProtoNet	FC^3	330	218,058
*LeNet**	Conv-MP-Conv-MP-FC^3	8,094	61,706
AlexNet	Conv-MP-Conv-MP-$Conv^3$-MP-FC^3	716,554	44,428,106

Our attack assumes that the underlying DNN uses double precision (FP64) or single precision (FP32) floating-point numbers. However, there is a growing trend to use FP16 or even FP8 in industry [**54**]. The employment of low-precision numbers can adversely impact the accuracy and reliability of the `key_value_inference` and `key_vector_validation` procedures. We leave this problem for future investigation.

4.5. Experiments

4.5.1. Implementation

We implement the DNN decryption algorithm with PyTorch [**60**]. We exploit parallelism to accelerate the speed of our attack. Specifically, we initiate multiple instances of `search_critical_point` and `key_value_inference` procedures for multiple neurons on the same hidden layer simultaneously. Regarding the `error_correction` procedure, we make several guesses of the key vector at a time and execute `key_vector_validation` to validate them in parallel. Besides, we use the built-in Jacobian matrix for any computations related to the product weight matrix $\hat{A}^{(i)}$ to achieve high efficiency.

As for the learning-based attack, we substitute each $K'_{i,j}$ with a *sigmoid* function. The sigmoid function has a $[-1, 1]$ range, which coincides with that of $K'_{i,j}$. We choose the mean squared error between the output vectors of the encrypted network and the oracle network as the loss function. The initial values of the sigmoid functions are set

Table 4.3. Experiment results of attacks against logic encryption on DNNs.

DNN	Key *(bits)*	Original Accuracy	Baseline Accuracy	Monolithic Learning-based Attack				DNN Decryption Attack			
				Accuracy	Fidelity	Time*(s)*	#Queries	Accuracy	Fidelity	Time*(s)*	#Queries
ProtoNet	32	98.1%	27.6%	98.1%	100.0%	2.70	1,000	98.1%	100.0%	0.18	156
	64	98.1%	10.4%	78.6%	93.8%	5.26	1,000	98.1%	100.0%	0.25	252
	128	98.1%	7.5%	58.5%	87.6%	5.31	1,000	98.1%	100.0%	0.35	444
*LeNet**	32	99.0%	86.7%	99.0%	100.0%	7.38	1,000	99.0%	100.0%	6.48	1,166
	64	99.0%	49.9%	98.9%	96.9%	9.37	1,000	99.0%	100.0%	7.49	1,262
	128	99.0%	16.4%	78.5%	86.7%	9.70	1,000	99.0%	100.0%	10.02	1,464
AlexNet	64	91.7%	91.7%	91.7%	100.0%	55.45	1,000	91.7%	100.0%	118.47	2,232
	128	91.7%	91.5%	91.7%	99.2%	77.22	1,000	91.7%	100.0%	179.24	2,754
	256	91.7%	91.0%	91.7%	97.2%	77.72	1,000	91.7%	100.0%	826.71	15,258

to 0. During training, we periodically settle down those key bits that have reached the confidence threshold.

4.5.2. Experimental Setup

We assess the proposed algorithm on the following DNN architectures: *a) ProtoNet*, a multilayer perceptron trained on the MNIST [**26**] dataset; *b) LeNet**, a ReLU variant of LeNet-5 [**48**] trained on the MNIST dataset; and *c)* an *AlexNet* [**47**] trained on the CIFAR-10 [**46**] dataset. Detailed configurations of these networks are shown in Table 4.2. We conduct all experiments on a Linux workstation with a 2.4GHz CPU and an Nvidia RTX A6000 graphics card.

We apply HPNN to encrypt the abovementioned DNNs. Given a specific key size, we *i)* equally distribute the key bits to all designated hidden layers, *ii)* embed key bits to a random subset of neurons for every hidden layer, and *iii)* assign a value to every key bit uniformly at random. After that, we train the DNN models as functions of the keys until they converge. We launch two types of attacks on the resulting DNN models: *a)* a monolithic learning-based attack, which only applies the method described in Section 4.4.4; and *b)* the proposed comprehensive DNN decryption algorithm (Algorithm 8).

We use four metrics to measure the effectiveness and efficiency of the attacks: *a) accuracy*, which is the percentage of correct predictions on the testing dataset; *b) fidelity* [**43**], which is the percentage of exactly recovered key bits; *c) execution time*; and *d) query complexity*, which is the total number of queries made to the oracle network. Extracting a model with high accuracy could facilitate IP piracy, whereas extracting a model with high fidelity also enables adversarial attacks. Table 4.3 shows our evaluation results.

Table 4.4. Breakdown of the execution time (seconds) of the DNN decryption algorithm. (`KVI`: `key_value_inference`, `LA`: `learning_attack`, `V/EC`: the `key_vector_validation` / `error_correction` loop.) A parenthesized value indicates that the attempt is failed or does not pass the validation.

DNN	Key	First FC Layer			Second FC Layer		
	(bits)	`KVI`	`LA`	`V/EC`	`KVI`	`LA`	`V/EC`
ProtoNet	32	0.13	–	0.031	0.023	–	0.001
	64	0.13	–	0.086	0.030	–	0.002
	128	0.23	–	0.089	0.033	–	0.002
*LeNet**	32	0.053	–	0.036	(0.049)	6.34	0.004
	64	0.053	–	0.036	(0.045)	7.35	0.004
	128	0.058	–	0.037	(0.043)	(9.86)	0.018
AlexNet	64	(12.81)	54.59	1.62	(4.27)	45.10	0.080
	128	(12.96)	(77.85)	25.44	(4.43)	58.48	0.080
	256	(13.39)	(77.65)	648.38	(4.92)	(77.43)	4.94

4.5.3. The Monolithic Learning-based Attack

For every network model, this attack first generates a set of input examples and then queries the oracle network for corresponding output vectors. It terminates when either *i)* all key bits have reached the confidence threshold, or *ii)* the limit of the training epoch is reached.

Table 4.3 shows the experiment results of the two attacks on DNN logic encryption. The original accuracy is measured with the correct key inserted. As for the baseline accuracy, we randomly generate 16 incorrect keys for every network model and then compute the average of their accuracy.

The monolithic learning-based attack alone is already sufficiently powerful when the key size is small. However, as the key size grows, this attack becomes less effective. Interestingly, we observe that the baseline accuracies of *AlexNet* are relatively high, meaning

the network can still make accurate predictions even when the randomly generated incorrect keys are inserted. We believe this is because *AlexNet* uses the *dropout* technique to reduce redundancy and improve generalization. Such techniques can enhance the robustness and resist the impacts of incorrect keys.

We tried to *i)* increase the limit for the training epoch and *ii)* increase the number of input examples. Nevertheless, none of them improves the performance of the monolithic learning-based attack on those instances with relatively large key sizes. We resort to the DNN decryption algorithm for large key sizes.

4.5.4. The DNN Decryption Attack

With the orchestration of the key value inference procedure, the learning-based attack, the key vector validation procedure and the correction loop, our proposed algorithm succeeds on all the instances with a 100% fidelity.

Table 4.4 breaks down the total execution time. As shown in the table, the portion of total execution time consumed by each procedure depends on the network architecture. In particular, as *ProtoNet* is highly contractive (784 input nodes and 256/64 neurons for the first/second fully connected (FC) layer), `key_value_inference` can easily decrypt all of the three instances. On the other hand, both *LeNet** and *AlexNet* contain expansive convolutional layers. However, this problem does not prevent `key_value_inference` because the FC layers are still contractive with respect to the size of the input space. As a result, `key_value_inference` can still recover the key bits in the first hidden layer of *LeNet**. Nevertheless, this procedure cannot fully recover the key bits in the other FC layers due to inactive neurons.

We also notice that the learning-based attack is more reliable when integrated into the DNN decryption algorithm because it only targets a single hidden layer at a time. The learning-based attack makes at most 3 bits of an error on any hidden layers. Hence, it leaves a small search space for `error_correction`.

Given that the convolutional layers are deep and expansive, why not encrypt those layers instead of the FC layers? First, Section 4.4.7 points out that encryptions on the convolutional layers are vulnerable to the divide-and-conquer strategy. Second, we observe that it is less key-efficient to encrypt the convolutional layers. For instance, embedding 64 key bits to the convolutional layers of *LeNet** reduces the accuracy by merely 4.7%, whereas embedding the same number of key bits to the FC layers reduces the accuracy by 49.1%.

4.6. Related Work

4.6.1. Logic Decryption for Integrated Circuits

The most powerful I/O attack against conventional logic encryption is the SAT attack [**78**] and its SMT variant [**6**]. In every iteration, it *i)* calls a SAT solver for a distinguishing input pattern, *ii)* queries the oracle circuit for the corresponding output pattern, and *iii)* adds the I/O pair and a fresh copy of the circuit to the SAT solver as a new constraint. The attack terminates when no more distinguishing input patterns exist, and a correct key can be extracted from the solver afterwards. At a glance, the same method can be applied to our problem. Unfortunately, a deep ReLU network should be encoded as mixed-integer linear programming formulas rather than Boolean formulas. Due to this gap, it is unrealistic to attack a DNN monolithically.

4.6.2. Model Extraction for Deep ReLU Networks

Differential attacks [**43, 13, 69**] consistently query an oracle network to recover the weights and biases of the original network. While these studies provide valuable insights into deep ReLU networks, even the state-of-the-art implementation [**13**] requires more than 2^{21} queries to partially reconstruct a DNN with less than 1,000 neurons. They are also struggling with DNNs which have more than three hidden layers.

4.7. Conclusion

This paper presents the first attack on logic-encrypted DNN. It combines algebraic and learning-based approaches to extract a correct key of a victim DNN. Experimental results show that the attack can scale to large DNNs and generalize to complex network architectures. Therefore, the hardware security research community needs to develop new techniques to protect DNNs with hardware root-of-trust.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] Yossi Adi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. In *USENIX Security 2018*, pages 1615–1631.

[3] Abdulrahman Alaql, Domenic Forte, and Swarup Bhunia. Sweep to the secret: A constant propagation attack on logic locking. In *AsianHOST 2019*, pages 1–6.

[4] Lilas Alrahis, Satwik Patnaik, Muhammad Shafique, and Ozgur Sinanoglu. Omla: An oracle-less machine learning-based attack on logic locking. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 69(3):1602–1606, 2021.

[5] Ross Anderson. *Physical Tamper Resistance*. John Wiley & Sons, 2020.

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

[7] Jason Baumgartner et al. Scalable sequential equivalence checking across arbitrary design transformations. In *ICCAD 2006*, pages 259–266.

[8] Shoham Ben-David, Baruch Sterin, Joanne M Atlee, and Sandy Beidu. Symbolic model checking of product-line requirements using sat-based methods. In *ICSE 2015*, volume 1, pages 189–199.

[9] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *ECML PKDD 2013*, pages 387–402. Springer.

[10] Aaron R Bradley. Sat-based model checking without unrolling. In *VMCAI 2011*, pages 70–87.

[11] Aaron R Bradley et al. An incremental approach to model checking progress properties. In *FMCAD 2011*, pages 144–153.

[12] Franc Brglez, David Bryan, and Krzysztof Kozminski. Combinational profiles of sequential benchmark circuits. In *ISCAS 1989*, pages 1929–1934.

[13] Nicholas Carlini, Matthew Jagielski, and Ilya Mironov. Cryptanalytic extraction of neural network models. In *CRYPTO 2020*, pages 189–218.

[14] Abhishek Chakraborty, Ankit Mondai, and Ankur Srivastava. Hardware-assisted intellectual property protection of deep learning models. In *DAC 2020*, pages 1–6.

[15] Prabuddha Chakraborty, Jonathan Cruz, and Swarup Bhunia. Sail: Machine learning guided structural analysis attack on hardware obfuscation. In *AsianHOST 2018*, pages 56–61.

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

[17] Pankaj Chauhan et al. Non-cycle-accurate sequential equivalence checking. In *DAC 2009*, pages 460–465.

[18] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Ic3 modulo theories via implicit predicate abstraction. In *TACAS 2014*, pages 46–61.

[19] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Parameter synthesis with ic3. In *FMCAD 2013*, pages 165–168.

[20] Koen Claessen and Niklas Sörensson. A liveness checking algorithm that counts. In *FMCAD 2012*, pages 52–59.

[21] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal methods in system design*, 19(1):7–34, 2001.

[22] Jean Da Rolt, Giorgio Di Natale, Marie-Lise Flottes, and Bruno Rouzeyre. New security threats against chips containing scan chain structures. In *HOST 2011*, pages 110–110.

[23] Steve Dai, Gai Liu, Ritchie Zhao, and Zhiru Zhang. Enabling adaptive loop pipelining in high-level synthesis. In *ACSSC 2017*, pages 131–135. IEEE.

[24] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[25] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS 2008*, pages 337–340.

[26] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[27] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *SAT 2006*, pages 36–41.

[28] Nachum Dershowitz, DN Jayasimha, and Seungjoon Park. Bounded fairness. *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, pages 304–317, 2003.

[29] Avinash R Desai, Michael S Hsiao, Chao Wang, Leyla Nazhandali, and Simin Hall. Interlocking obfuscation for anti-tamper hardware. In *Proceedings of the eighth annual cyber security and information intelligence research workshop*, pages 1–4, 2013.

[30] Danielle Duvalsaint, Xiaoxiao Jin, Benjamin Niewenhuis, and RD Blanton. Characterization of locked combinational circuits via atpg. In *ITC 2019*, pages 1–10.

[31] Niklas Eén, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *FMCAD 2011*, pages 125–134.

[32] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[33] Mohamed El Massad, Siddharth Garg, and Mahesh Tripunitara. Reverse engineering camouflaged sequential circuits without scan access. In *ICCAD 2017*, pages 33–40.

[34] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *CCS 2015*, pages 1322–1333.

[35] Matteo Gamba, Adrian Chmielewski-Anders, Josephine Sullivan, Hossein Azizpour, and Marten Bjorkman. Are all linear regions created equal? In *International Conference on Artificial Intelligence and Statistics*, pages 6573–6590. PMLR, 2022.

[36] Aman Goel and Karem Sakallah. Avr: abstractly verifying reachability. In *TACAS 2020*, pages 413–422.

[37] Aman Goel and Karem Sakallah. Model checking of verilog rtl using ic3 with syntax-guided abstraction. In *NASA Formal Methods Symposium*, pages 166–185. Springer, 2019.

[38] Brunno F Goldstein, Vinay C Patil, Victor C Ferreira, Alexandre S Nery, Felipe MG França, and Sandip Kundu. Preventing dnn model ip theft via hardware obfuscation. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 11(2):267–277, 2021.

[39] Jia Guo and Miodrag Potkonjak. Watermarking deep neural networks for embedded systems. In *ICCAD 2018*, pages 1–8.

[40] Boris Hanin and David Rolnick. Deep relu networks have surprisingly few activation patterns. *NIPS 2019*, pages 361–370.

[41] Yalin Hu and Robert C Armstrong. A survey of formal verification in mission-critical high-consequence applications. Technical report, Sandia National Lab, 2011.

[42] Yinghua Hu, Yuke Zhang, Kaixin Yang, Dake Chen, Peter A Beerel, and Pierluigi Nuzzo. Fun-sat: Functional corruptibility-guided sat-based attack on sequential logic encryption. In *HOST 2021*, pages 281–291.

[43] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. High accuracy and high fidelity extraction of neural networks. In *USENIX Security 2020*, pages 1345–1362.

[44] Ayush Jain, Ziqi Zhou, and Ujjwal Guin. Taal: tampering attack on any key-based logic locked circuits. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 26(4):1–22, 2021.

[45] Alfred Koelbl, Reily Jacoby, Himanshu Jain, and Carl Pixley. Solver technology for system-level to rtl equivalence checking. In *DATE 2009*, pages 196–201.

[46] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The cifar-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 55(5), 2014.

[47] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[48] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[49] Leon Li and Alex Orailoglu. Piercing logic locking keys through redundancy identification. In *DATE 2019*, pages 540–545.

[50] Lu Lu, Yeonjong Shin, Yanhui Su, and George Em Karniadakis. Dying relu and initialization: Theory and numerical examples. *arXiv:1903.06733*, 2019.

[51] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. *NIPS 2017*.

[52] Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In *Handbook of satisfiability*, pages 133–182. 2021.

[53] Kenneth L McMillan. Interpolation and sat-based model checking. In *CAV 2003*, pages 1–13.

[54] Paulius Micikevicius et al. Mixed precision training. *ICLR 2018*, OpenReview.net.

[55] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML 2010*, pages 807–814.

[56] Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[57] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2, btormc and boolector 3.0. In *ICCAD 2018*, pages 587–595.

[58] Brooks Olney and Robert Karam. Protecting deep neural network intellectual property with architecture-agnostic input obfuscation. In *GLSVLSI 2022*, pages 111–115.

[59] Preeti R Panda and Nikil D Dutt. 1995 high level synthesis design repository. In *ISSS 1995*, pages 170–174.

[60] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, et al. Pytorch: An imperative style, high-performance deep learning library. *NIPS 2019*.

[61] Hongwu Peng et al. A length adaptive algorithm-hardware co-design of transformer on fpga through sparse attention and dynamic pipelining. In *DAC 2022*, pages 1135–1140.

[62] Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. vtpm: virtualizing the trusted platform module. In *USENIX Security 2006*, pages 305–320.

[63] Ana Petkovska, Alan Mishchenko, David Novo, Muhsen Owaida, and Paolo Ienne. Progressive generation of canonical sum of products using a sat solver. In *Proceedings of the 25th International Workshop on Logic and Synthesis, Austin, Tex*, 2016.

[64] Luca Piccolboni et al. Kairos: Incremental verification in high-level synthesis through latency-insensitive design. In *FMCAD 2019*, pages 105–109.

[65] Carl Pixley. Introduction to a computational theory and implementation of sequential hardware equivalence. In *ICCAD 1990*, pages 54–64.

[66] Mathias Preiner, Armin Biere, and Nils Froleyks. Hardware model checking competition 2020. 2020.

[67] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

[68] Stefano Recanatesi, Matthew Farrell, Madhu Advani, Timothy Moore, Guillaume Lajoie, and Eric Shea-Brown. Dimensionality compression and expansion in deep neural networks. *arXiv:1906.00443*, 2019.

[69] David Rolnick and Konrad Kording. Reverse-engineering deep relu networks. In *ICML 2020*, pages 8178–8187.

[70] Shervin Roshanisefat, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. Rane: An open-source formal de-obfuscation attack for reverse engineering of logic encrypted circuits. In *GLSVLSI 2021*, pages 221–228.

[71] Jarrod A Roy, Farinaz Koushanfar, and Igor L Markov. Epic: Ending piracy of integrated circuits. In *DATE 2008*, pages 1069–1074.

[72] Samir Sapra, Michael Theobald, and Edmund Clarke. Sat-based algorithms for logic minimization. In *ICCD 2003*, pages 510–517.

[73] Benjamin Carrion Schafer and Zi Wang. High-level synthesis design space exploration: Past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2628–2639, 2019.

[74] Tobias Seufert, Christoph Scholl, D Groe, and R Drechsler. Sequential verification using reverse pdr. In *MBMV 2017*, pages 79–90.

[75] Kaveh Shamsi, Meng Li, David Z Pan, and Yier Jin. Kc2: Key-condition crunching for fast sequential circuit deobfuscation. In *DATE 2019*, pages 534–539.

[76] Nikhil Sharma, Gagan Hasteer, and Venkat Krishnaswamy. Sequential equivalence checking for rtl models. eetimes now, 2006.

[77] Jacob Steinhardt, Pang Wei W Koh, and Percy S Liang. Certified defenses for data poisoning attacks. *NIPS 2017*, 30.

[78] Pramod Subramanyan, Sayak Ray, and Sharad Malik. Evaluating the security of logic encryption algorithms. In *HOST 2015*, pages 137–143.

[79] Synopsys. Systemc tlm models broadest portfolio of systemc transaction-level models, 2023.

[80] CAJ Van Eijk. Sequential equivalence checking without state space traversal. In *DATE 1998*, pages 618–623.

[81] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, , et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

[82] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[83] Yang Xie and Ankur Srivastava. Anti-sat: Mitigating sat attack on logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(2):199–207, 2018.

[84] Shaojie Xu, Joel Vaughan, Jie Chen, Aijun Zhang, and Agus Sudjianto. Traversing the local polytopes of relu neural networks. In *The AAAI-22 Workshop on Adversarial Machine Learning and Beyond*, 2022.

[85] Mingfu Xue, Yushu Zhang, Jian Wang, and Weiqiang Liu. Intellectual property protection for deep learning models: Taxonomy, methods, attacks, and evaluations. *IEEE Transactions on Artificial Intelligence*, 3(6):908–923, 2021.

[86] Xiaodong Yang, Taylor T Johnson, Hoang-Dung Tran, Tomoya Yamaguchi, Bardh Hoxha, and Danil V Prokhorov. Reachability analysis of deep relu neural networks using facet-vertex incidence. In *HSCC 2021*, pages 19–21.

[87] Muhammad Yasin, Bodhisatwa Mazumdar, Jeyavijayan JV Rajendran, and Ozgur Sinanoglu. Sarlock: Sat attack resistant logic locking. In *HOST 2016*, pages 236–241.

[88] Kimia Zamiri Azar, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. Threats on logic locking: A decade later. In *GLSVLSI 2019*, pages 471–476.

[89] Yuqiao Zhang, Pinchen Cui, Ziqi Zhou, and Ujjwal Guin. Tga: An oracle-less and topology-guided attack on logic locking. In *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop*, pages 75–83, 2019.

[90] Hai Zhou, Amin Rezaei, and Yuanqi Shen. Resolving the trilemma in logic encryption. In *ICCAD 2019*, pages 1–8.