Secure and Effective Logic Locking for Machine Learning Applications

Yuntao Liu, Yang Xie, Abhishek Charkraborty, and Ankur Srivastava

University of Maryland, College Park

Abstract. Logic locking has been proposed as a strong protection of intellectual property (IP) against security threats in the IC supply chain especially when the fabrication facility is untrusted. Various techniques have proposed circuit configurations which do not allow the untrusted fab to decipher the true functionality and/or produce usable versions of the chip without having access to the locking key. These techniques rely on using additional locking circuitry which injects incorrect behavior into the digital functionality when the key is incorrect. However, much of this conventional research focuses on locking individual modules (such as adders, ALUs etc.). While locking these modules is useful, the true test for any locking scheme should consider their impact on the application running on a processor with such modules. A locked module within a processor may or may not have a substantial impact at the application level thereby allowing the attacker (untrusted foundry or unauthorized user) to still get useful work out of the system despite not having access to the key details. In this work, we show that even when state of the art locking schemes are used to lock the modules within a processor, a large class of workloads derived from machine learning (ML) applications (which are increasingly becoming the most relevant ones) continue to function correctly. This has huge implications to the effectiveness of the current locking techniques. The main reason for this behavior is the inherent error resiliency of such applications. To counter this threat, we propose a novel secure and effective logic locking scheme, called *Strong Anti-SAT (SAS)*, to lock the entire processor and make sure that the ML applications undergo significant accuracy loss when any wrong key is applied. We provide two types of SAS, namely SAS-A and SAS-B. Experiments show that, for both types of SAS, 1) the application-level accuracy loss is significant (for ML applications) given any wrong key, 2) the attacker needs extremely long time to find a correct key, and 3) the hardware overhead is very small. Lastly, even though our techniques target machine learning type application workloads, the impact on conventional workloads will also be similar. Due to the inherent error resilience of ML, locking ML workloads is a harder problem to tackle.

Keywords: Logic Locking · SAT Attack · Machine Learning · Strong Anti-SAT

1 Introduction

Due to the increasingly high cost of building or maintaining an IC foundry and the relatively lower cost to access advanced fabrication technologies in off-shore foundries, many chip designers have chosen to become fabless and outsource the fabrication to such foundries. However, the designers usually have very little control over the foundries and thus the untrusted foundry issues have become a major security challenge to the semiconductor industry. Various kinds of attacks on the design can be launched in untrusted foundries, including hardware Trojan insertion, IP and IC piracy and counterfeiting, overbuilding, etc. [1–6].

In order to mitigate these security risks, various design-for-trust techniques have been studied, including logic locking [7–17], split manufacturing [18,19], and post-fabrication editing [20]. Among these techniques, logic locking has received the most attention from researchers. The locked circuit takes a key input in addition to the primary input and will produce the correct output if the key is the correct key. If the input key is a wrong key, however, the output will be wrong for some primary input values. The correct key is a secret kept by the chip designer and is not known by the foundry or any third party. The foundry receives the design of the locked circuit and manufactures it. After the locked circuit is manufactured and returned to the designer, a tamper-proof memory containing the correct key is connected to the key input of the circuit. The circuit with the correct key is called the *activated circuit*.

Conventional logic locking mechanisms [7,9,10,21,22] have been shown vulnerable to the Boolean satisfiability based attack, referred to as SAT attack. SAT is an oracle-guided attack: the attacker has (1) the netlist of the locked circuit and (2) an activated chip which can be obtained from the open market. Details of the SAT attack will be introduced in Section 2.2. In short, SAT prunes out wrong keys in an iterative manner. In each iteration, an input (called the differentiating input, or DI) is chosen by the SAT solver and all the wrong keys that corrupt the output of this DI are pruned out. All wrong keys are pruned out when no more DI can be found. SARLock [15] and Anti-SAT [12,13] force the number of SAT iterations to be exponential in the key size by pruning out only a very small number of wrong keys in each iteration. However, this is at the cost that there is only one input minterm (or a very small number of minterms) whose output is incorrect under the each wrong key. Hence the overall error rate of the locked circuit with an incorrect key is very small. This disadvantage is captured by approximate SAT attacks such as AppSAT [23] or DoubleDIP [24]. These attack schemes are able to find an *approximate key (approx-key)* which makes the locked circuit behave correctly for most (but not all) of the input values. If a processor has modules locked using such techniques, the AppSAT attack can be used to "mostly" unlock these modules. When an application with inherent error tolerance (such as most machine learning (ML) applications) is executed on such an approximately unlocked processors, the impact on the overall application functionality is minimum. We present evidence to this end in the subsequent sections. In essence, the locked processor is still mostly usable for all practical purposes thereby nullifying substantial attempts to lock the designs from unauthorized use by the foundry. Note that, the error-resilience property of ML workloads has been exploited to develop energy-efficient approximate computation methodologies for ML [25–32]. The same property adds a substantial challenge to the logic locking problem.

In this work, we address these challenges in locking processors for ML applications and propose the *Strong Anti-SAT (SAS)* scheme to protect processors against both exact and approximate SAT attacks. SAS is applicable to both general-purpose hardware (e.g. CPU and GPU) and specialized hardware for ML [33–41]. The objective of SAS is to ensure that, given any wrong (including approximate) key, the locked hardware will produce sufficient error so that the ML models would have significant loss of accuracy while still being provably resilient to SAT and AppSAT attacks. [1]

The contribution of this work is as follows.

1. We develop a framework for locking the processor and evaluating both the circuit-level error profile and its application-level impact. In our experiments, a MIPS processor which includes **both the controlpath and datapath** is synthesized and locked using state-of-the-art logic locking schemes [12,15]. We propose an attack methodology to investigate the vulnerability of the state-of-the-art locking techniques in securing ML applications. The proposed attack firstly utilizes the AppSAT attack

[1] In this work, we assume that the processor is only used for the inference, not training, of ML models. Our ML benchmarks are all for the purpose of classification. Classification accuracy means the percentage of input samples correctly classified by the ML model running on the processor chip.

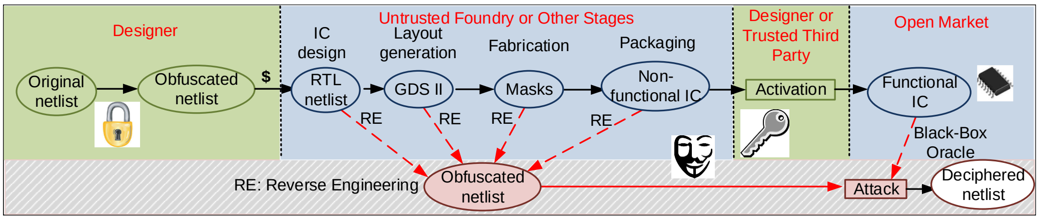


Figure 1: The targeted attack model of logic locking

to find an approx-key which results in an approx-unlocked processor. The error profile of the approx-unlocked processor is characterized and transferred to the simulation of ML applications. We show that executing ML applications on such processors results in very small reduction in ML classification accuracy.

2. To counter this attack, we propose the *Strong Anti-SAT (SAS)* scheme for ML applications which is based on the co-design of locking infrastructure and processor cores. We present two types of SAS: SAS-A and SAS-B.

   In SAS-A, we first propose an improved locking infrastructure (the SAS-A block) based on Anti-SAT [12]. We prove a lower-bound of error rate for the SAS-A block for any incorrect key. Thus, with an appropriate configuration of SAS-A, we guarantee a high error rate for any wrong key obtained by an approximate SAT attack (AppSAT). This results in significant loss of classification accuracy hence making the attack ineffective. We also show that the exact SAT attack still takes extremely long time to find the correct key of SAS-A.

3. In order to ensure exponential SAT solving time, we also propose SAS-B. In SAS-B, we identify a set of inputs which have higher impact on the accuracy of ML applications as *critical minterms* and increase the number of wrong keys that cause errors in their output. We introduce the locking infrastructure, the SAS-B block, and provide 2 different configurations of SAS-B. For both configurations, we derive the expected number of SAT iterations to find the correct key and show that this number is exponential in the size of the key.

4. Experimental results show that 1) the proposed locking schemes result in significant accuracy loss for the benchmark ML models, and 2) the exact SAT attack needs extremely long time and is practically infeasible. We also compare SAS-A and SAS-B with existing state-of-the-art logic locking approaches and show that our approaches have much higher application-level impact and lower hardware overhead.

In summary, our paper targets the realistic scenario of studying the impact of executing applications on locked processors. We illustrate that for ML workloads, conventional locking of modules within a processor is rather ineffective and propose new techniques to address the issue.

2 Background

2.1 Attack Model

The threat model considered in this work is illustrated in Fig. 1 and is consistent with the latest research articles on logic locking [7,12,15,17,23,24,42–45]. Only the chip designer has access to the proprietary design details. The untrusted foundry, which is usually considered as the attacker, is able to obtain the locked netlist of the chip by reverse-engineering [46].

The following components are considered available for the attacker:

1. *The locked gate-level netlist of the chip design.* As mentioned above, the designer outsources to the untrusted foundry the layout-level details of a design in the form of GDS-II files. Therefore, the attacker can reverse-engineer the GDS-II files to obtain the locked gate-level netlist.

2. *An activated chip.* An activated chip (*i.e.* the one with the correct key loaded) is considered available for the attacker since it can be purchased from the open market. The activated chip will serve as a black-box oracle to obtain the correct I/O behavior of the design. The attacker can query any input to this oracle and observe the correct response.

In general, logic locking techniques do not assume that the attacker can insert probes into the logic-locked circuit to observe the values at intermediate nodes. This is because protection techniques, such as analog shield [47], can be in place to counter probing attacks. If probing were feasible, however, the attack would become trivial since the attacker could simply probe the key values.

2.2 The Boolean Satisfiability-based (SAT) Attack Methodology

The functionality of any combinational digital circuit can be represented by a Boolean function $F : \vec{X} \rightarrow \vec{Y}$ where $\vec{X}$ is the primary input and $\vec{Y}$ is the primary output. Compared to the original circuit, the locked circuit $F_L$takes two inputs: one is the primary input and the other is the key input $\vec{K}$. If $\vec{K}$ is a correct key, then $\quad \vec{X}, F(\vec{X}) = F_L(\vec{X}, \vec{K})$. $F(\vec{X})$ may or may not be equal to $F_L(\vec{X}, \vec{K})$ if $\vec{K}$ is an incorrect key. The key is stored a the tamper-proof memory which is connected to the key input of the locked circuit.

Boolean satisfiability-based attack, a.k.a. *SAT attack* has been proven effective in finding the correct key of the locked circuit. The objective of the attacker is to find the correct key to the locked circuit. A SAT solver is a tool to determine if a Boolean expression is satisfiable, *i.e.* if there exists an assignment of the input bits such that the expression evaluates to TRUE. For example, the Boolean expression $a \quad b$ is satisfiable because the assignment of $a = 0,\ b = 1$ will make it evaluate to TRUE. In terms of SAT attack, we express the functionality of the locked circuit, *i.e.* $\vec{Y} = F_L(\vec{X}, \vec{K})$ in the *Conjunctive Normal Form (CNF)*: $C(\vec{X}, \vec{K}, \vec{Y})$. $C(\vec{X}, \vec{K}, \vec{Y})$ evaluates to TRUE if there exists an assignment of $\vec{X}$, $\vec{K}$, $\vec{Y}$ such that $\vec{Y} = F_L(\vec{X}, \vec{K})$ holds and evaluates to FALSE otherwise.

SAT attack is an iterative attack that narrows down the search space for the correct key iteration by iteration. The steps of the SAT attack is as follows.

1. In the initial iteration, the attacker looks for a primary input $\vec{X}_1$ and two keys $\vec{K}_\alpha$ and $\vec{K}_\beta$ such that the locked circuit produces two different outputs $\vec{Y}_\alpha$ and $\vec{Y}_\beta$:

$$C(\vec{X}_1, \vec{K}_\alpha, \vec{Y}_\alpha) \quad C(\vec{X}_1, \vec{K}_\beta, \vec{Y}_\beta) \quad (\vec{Y}_\alpha = \vec{Y}_\beta) \tag{1}$$

   $\vec{X}_1$ is called the *Distinguishing Input (DI).*

2. The DI $\vec{X}_1$ is applied to the activated circuit (the oracle) and the output $\vec{Y}_1$ is recorded. Note that $\vec{K}_\alpha$, $\vec{Y}_\alpha$, and $\vec{K}_\beta$, $\vec{Y}_\beta$ are not recorded. Only the DI and its correct output are carried over to the following iterations.

3. In the i^{th} iteration, a new DI and a pair of keys $\vec{K}_\alpha$ and $\vec{K}_\beta$ are found. The newly found $\vec{K}_\alpha$ and $\vec{K}_\beta$ should produce correct outputs for all the DIs found in previous iterations. To this end, we append a clause to (1):

$$\begin{gathered} C(\vec{X}_i, \vec{K}_\alpha, \vec{Y}_\alpha) \quad C(\vec{X}_1, \vec{K}_\beta, \vec{Y}_\beta) \quad (\vec{Y}_\alpha = \vec{Y}_\beta) \\ \bigwedge_{j=1}^{i-1} (C(\vec{X}_j, \vec{K}_\alpha, \vec{Y}_j) \quad C(\vec{X}_j, \vec{K}_\beta, \vec{Y}_j)) \end{gathered} \tag{2}$$

In this way, all the wrong keys that corrupt the output of previously found DIs (*i.e.* the output is different from that of the activated chip) are pruned out from the search space.

4. SAT solves (2) repeatedly until no more DI can be found, *i.e.* (2) is not satisfiable any more.

5. In this case, there is no more DI. The output of the SAT attack is a key $\vec{K}$ that produces the same output as the correct key to all the DIs, which can be expressed using the following CNF:

$$\bigwedge_{i=1}^{\lambda} C(\vec{X}_i, \vec{K}, \vec{Y}_i) \tag{3}$$

where λ is the total number of SAT iterations.

Theorem 1. *SAT is guaranteed to find a correct key $\vec{K}_c$ to the locked circuit.*

The proof is given in Appendix A. Note that there can be multiple correct keys: some keys are different from the actual key but functionally equivalent to the actual key.

2.3 Logic Locking Schemes against SAT Attacks

Multiple logic locking schemes have been proposed to thwart the SAT attack [12–15, 22]. There are two ways to mitigate the SAT attack: one is to increase the time for each SAT iteration and the other is to increase the number of SAT iterations. The former was implemented in [22] where one-way random functions, such as advanced encryption system (AES) blocks, were inserted into the locked circuit and it was shown that the time for each SAT iteration increased exponentially with the size of the key. The main drawback of this approach is its high area overhead: the AES block is too large to be practical for small circuits.

The other logic locking schemes that aim to thwart the SAT attack focus on forcing the number of SAT iterations to be exponential in the size of the key. Before introducing the details, we first define the *error rate* and the *corruptibility* for logic locking schemes.

Definition 1. We say that a key $\vec{K}$ corrupts a primary input minterm $\vec{X}$ if and only if the locked circuit produces a different output to $\vec{X}$ from the original circuit, *i.e.* $F_L(\vec{X}, \vec{K}) = F(\vec{X})$.

Definition 2. The error rate $\epsilon_{\vec{K}}$ of a key $\vec{K}$ is the portion of primary input minterms which are corrupted by the key $\vec{K}$.

Note that $\epsilon_{\vec{K}} = 0$ for any correct key. If the error rate is the same across all the *wrong* keys, we simply use ϵ to denote this error rate.

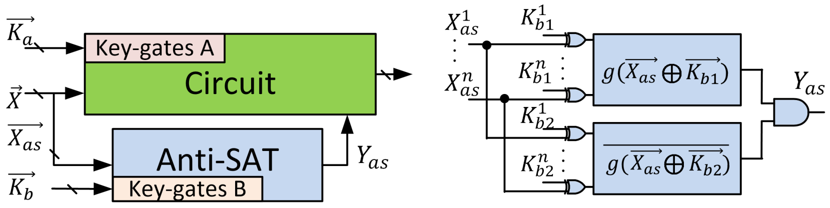
Definition 3. The corruptibility $\gamma_{\vec{X}}$ of a primary input minterm $\vec{X}$ is the portion of wrong keys that corrupts this minterm

Let $\mathsf{K}_{\vec{X}}$ be the set of wrong keys that corrupts the primary input minterm $\vec{X}$ and K^W be the set of wrong keys. Then,

$$\gamma_{\vec{X}} = \frac{|\mathsf{K}_{\vec{X}}|}{|\mathsf{K}^W|}$$

If the corruptibility is the same across all the primary input minterms, we simply use γ to denote this corruptibility.

Theorem 2. *If the error rate of any wrong key is the same and the corruptibility for all the primary input minterms is also the same, then the error rate equas the corruptibility,* i.e. $\epsilon = \gamma$



(a) (b)

Figure 2: Anti-SAT based logic locking: (a) overview; (b) Anti-SAT block

The proof is given in Appendix B. Let λ be the number of SAT iterations that any SAT attacker needs to perform to find the correct key.

Theorem 3. *If the corruptibility for all the primary input minterms equals γ, then the number of SAT iterations λ is lower bounded by $\frac{1}{\gamma}$.*

Proof. In each SAT iterations, the wrong keys pruned by the DI $\vec{X}$ is upper bounded by $|K_{\vec{X}}|$. This is an upper bound because some of the wrong keys may have already pruned out by DIs of previous iterations. Therefore,

$$\lambda \geq \frac{|K^W|}{|K_{\vec{X}}|} = \frac{1}{\gamma}$$

Hence proved. □

Theorems 2 and 3 explicitly quantifies the relationship among the error rate ϵ, the SAT iterations λ, and and corruptibility γ. In order to force λ to exponential, the error rate ϵ (and hence γ) have to be reduced exponentially. This strategy is the main idea of SARLock [15] and Anti-SAT [12, 13]. Due to the similarity of these two techniques, in this work, we focus on the Anti-SAT based logic locking. Here we describe the design of Anti-SAT and analyze its properties.

Fig. 2(a) shows the overview of Anti-SAT. The original circuit is locked with $\vec{K_a}$ (referred to as *conventional keys*) using conventional locking techniques such as [7, 9]. Besides, an Anti-SAT block is attached to the locked circuit. The Anti-SAT block has two inputs: $\vec{X}_{as} \quad \mathbb{Z}_2^n$ which is an n-bit subset of the bits in the primary input $\vec{X}$, and a key input $\vec{K}_b \quad \mathbb{Z}_2^{2n}$. The Anti-SAT output Y_{as} is connected to an internal wire of the original circuit using an XOR gate. Fig. 2(b) shows the detail of Anti-SAT block. It is composed of two logic blocks g and $\overline{g}$ which have complementary functionalities. The Anti-SAT keys $\vec{K}_b = (\vec{K}_{b1}, \vec{K}_{b2})$ are inserted at each input of g and $\overline{g}$. The outputs of g and $\overline{g}$ are fed into an AND gate to produce the Anti-SAT output Y_{as}, which makes $Y_{as} = g(\vec{X}_{as} \quad \vec{K}_{b1}) \quad \overline{g(\vec{X}_{as} \quad \vec{K}_{b2})}$. Given a wrong key, Y_{as} may output 1 and inject faults into the circuit. The logic block g in Anti-SAT has input-size n and on-set size p, where on-set size is the number of input patterns that can make function g output one.

Let K^C and K^W be the sets of correct and wrong keys, respectively. In the following discussion for Anti-SAT, we use $\vec{K}_1$ and $\vec{K}_2$ instead of $\vec{K}_{b1}$ and $\vec{K}_{b2}$ to refer to the Anti-SAT keys for simplicity. We also assume $n = n_p$ for simplicity, however, this analysis still holds when $n < n_p$.

Both $\vec{K}_1$ and $\vec{K}_2$ are n-bit long. For each $\vec{K}_1$, there is only one $\vec{K}_2$ such that $(\vec{K}_1, \vec{K}_2)$ K^C. In other words, $|K^C| = 2^n$ and $|K^W| = 2^n(2^n - 1)$. Let us analyze the case where $p = 1$, *e.g.* when the functionality of g is simply AND. In this case, there is only one assignment of $\vec{K}_1$ for the set of wrong keys that corrupts any input minterm $\vec{X}$ $K_{\vec{X}} = \{(\vec{K}_1, \vec{K}_2) | (\vec{K}_1, \vec{K}_2) \quad K^W\}$ due to $p = 1$. This effect is illustrated in Table 1 where each row stands for an input minterm and each column stands for the wrong keys with the same $\vec{K}_1$. A '•' indicates that the input minterm is corrupted by the wrong keys of

Table 1: The relationship between wrong keys and the minterms they corrupt in Anti-SAT [12] when $p = 1$. Each row stands for an input minterm and each column stands for the wrong keys with the same $\vec{K}_1$. A '•' indicates that the input minterm is corrupted by the wrong keys of that column.

$\vec{K}_{b1}$ of wrong keys	$\vec{k}_1$	$\vec{k}_2$	$\vec{k}_3$	$\vec{k}_4$	$\vec{k}_5$	$\vec{k}_6$	$\cdots$	$\vec{k}_{2^n}$
$\vec{t}_1$	•							
$\vec{t}_2$		•						
$\vec{t}_3$			•					
$\vec{t}_4$				•				
$\vec{t}_5$					•			
$\vec{t}_6$						•		
$\cdots$							$\ddots$	
$\vec{t}_{2^n}$								•

that column. According to this relationship, we can derive the corruptibility of any input minterm as

$$\gamma_{\vec{X}} = \frac{|K_{\vec{X}}|}{|K^W|} = \frac{2^n - 1}{2^n(2^n - 1)} = \frac{1}{2^n} \quad \vec{X} \quad \mathsf{Z}_2^n$$

And by Theorem 2, $\epsilon_{\vec{K}} = \gamma = \frac{1}{2^n}$ $\vec{K}$ K^W. Therefore, the attacker can easily find an approximate key that produces the correct output for most of the input minterms. This is effectively captured by AppSAT [23]. Basically, the AppSAT attack enhances the SAT attack by adding an early termination condition to avoid taking an exponential number of iterations to find a correct key. When the early termination condition is satisfied, the AppSAT terminates and outputs the approx-key which can match all already found distinguishing inputs to their correct outputs. As shown in [23], after a few iterations, the AppSAT attack can decipher the conventional keys $\vec{K}_a$ but not the Anti-SAT keys $\vec{K}_b$, because the error induced by the former is much higher than that of the latter. Thus, as iteration progresses, $\vec{K}_a$ is gradually learned, thereby leaving a circuit which is only locked using the Anti-SAT block with keys $\vec{K}_b$. As analyzed in Sec. 2.3, the error rate of such locking scheme is $\epsilon = \frac{1}{2^n}$. For typical n, this error rate may be very small thereby resulting in an approx-unlocked circuit which is correct for most inputs.

3 Ineffectiveness of Existing Logic Locking for Machine Learning Applications

In this section, we investigate the security of processors running ML applications when the processor is locked with state-of-the-art logic locking [12]. As discussed in Sec. 2.3, such locking utilizes a combination of conventional logic locking [7,9] (with conventional keys $\vec{K}_a$) and Anti-SAT block (with Anti-SAT keys $\vec{K}_b$) which represents among the strongest defenses to SAT attack. Such locking schemes would render the SAT attack which attempts to learn the correct key ineffective.

However, as we show later in this section, such locking schemes is not secure for ML applications under AppSAT-type attacks [23]. In comparison with conventional applications, the ML applications are normally error-tolerant. By exploiting the error-tolerant nature, an attacker only needs to obtain an approx-key which can unlock most (but not all) of the correct functionality for the processors. This relaxed requirement makes attacks such as AppSAT [23] applicable and practical. *In this paper we focus exclusively on neural network type applications which represent the most popular ML applications. However, our techniques and arguments are valid of any ML application.* Five neural network models (as listed in Table 2) are used to evaluate the performance of an approx-unlocked processor.

Processor locking strategy was studied in [53] and it was shown to be vulnerable to AppSAT attack. We consider a similar experimental set-up to obtain the circuit-level error of a processor design. In our experiments, the circuit under attack is a MIPS processor

Table 2: ML benchmarks

Benchmark	#Label	#Test Data	Model
MNIST [48]	10	10000	LeNet
SVHN [49]	10	10000	CIFAR10_Full
CIFAR10 [50]	10	10000	CIFAR10_Quick
ILSVRC-2012 [51]	1000	500	CaffeNet
Oxford102 [52]	102	1000	CaffeNet

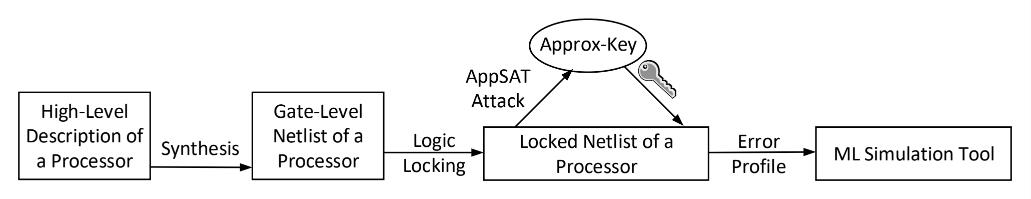

High-Level Description of a Processor
Synthesis
Gate-Level Netlist of a Processor
Logic Locking
Locked Netlist of a Processor
AppSAT Attack
Approx-Key
Error Profile
ML Simulation Tool

Figure 3: Our Experimental Framework

that includes the controller logic circuitry (controlpath) and the arithmetic logic unit (datapath). **Both the controlpath and datapath** of the synthesized processor netlist is locked using state-of-the-art logic locking schemes as follows: (i) first, the circuit is locked with fault-impact based keys $\vec{K}_a$ which has key-size $|\vec{K}_a| = 5\%$ key-gate overhead (ii) Next, a 64-input obfuscated Anti-SAT block (with $p = 1$) was attached to the above locked netlist. Such a combined locking scheme can cost the SAT attack exponential time to find the correct key as demonstrated in [12].

In Fig. 3, we explain how we transfer the error from the gate-level netlist to the application level. After obtaining an approx-key $\vec{K}_{App}$, we test it on the locked netlist using randomly generated imputs to estimate its error rate ϵ. Then, we transfer this error rate ϵ to the neural network simulation tool, Ristretto [54], where the computation in hardware is emulated. We XOR the correct result of every operation bit-wise with a random number with probability ϵ. In this way, the application-level error is obtained. This framework is used throughout the experiments in this paper.

We utilize the AppSAT attack to find an approx-key to de-obfuscate most of the correct functionality. Fig. 4 shows the error rate of the approx-key as AppSAT attack progresses. As seen, the error rate ϵ starts at 100% for a random key. However, ϵ drops dramatically during the first 30 iterations and it continues to decrease gradually as the attack proceeds. These results show the efficiency of AppSAT attack on finding an approx-key which can achieve low error rate. In this experiment, the AppSAT attack is terminated at 5000 iterations and it outputs an approx-key denoted as $\vec{K}_{App}$. With this *approximate key*, **we tried 10^{10} randomly generated instructions and find that there is not even a single fault** in the processor output response, *i.e.* the circuit-level error rate is upper bounded by 10^{-10}. Then, we use this error rate upper bound in application-level simulation. Table 3 shows the accuracy of 5 models running on the locked processor with a correct key $\vec{K}_C$ and the approx-key $\vec{K}_{App}$. We can see that the accuracy loss of an approx-unlocked processor is 0% for all the 5 benchmarks, despite the error rate is already an upper bound.

These results demonstrate that an attacker can approx-unlock a processor to get very low loss in classification accuracy of ML workloads using the AppSAT attack. Therefore, in the rest of this paper, we propose locking methods that is resilient to this type of attack.

4 Secure Locking for Machine Learning Applications

The attack results in Sec. 3 illustrate that AppSAT can easily decipher an approx-key to obtain an approx-unlocked circuit without significant application-level impact. *However, if we arbitrarily increase the error rate ϵ of wrong keys (and hence the corruptibility γ of*

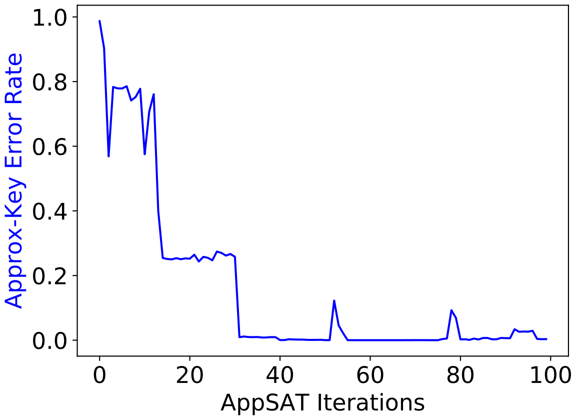


Figure 4: Error rate of the MIPS processor v.s. AppSAT attack iterations. The error rate is estimated using 10000 random input patterns.

Table 3: Accuracy of neural models deployed on a processor that is unlocked with a correct key $\vec{K}_C$ and an approx-key $\vec{K}_{App}$

Benchmark	**With $\vec{K}_C$**	**With $\vec{K}_{App}$**	
	Accuracy	**Accuracy**	**Accuracy Loss**
MNIST	99.00%	99.00%	0.00%
SVHN	93.51%	93.51%	0.00%
CIFAR10	89.40%	89.40%	0.00%
ILSVRC-2012	51.60%	51.60%	0.00%
Oxford102	93.28%	93.28%	0.00%

any input minterm), the complexity of exact SAT attack for finding a correct key would surely come down as stated in Theorem 3. Hence there are two competing objectives:

1. Objective 1: The approx-key deciphered by approximate SAT attacks should have a high error rate such that the ML application when executed on the approx-unlocked chip undergoes substantial errors.

2. Objective 2: The complexity of exact SAT attack to determine the correct key should still be very high.

In this section, we propose the *Strong Anti-SAT (SAS)* logic locking scheme which aims at achieving these two objectives simultaneously. The proposed locking technique is based on a co-design of the locking infrastructure and the processor. We provide two versions of SAS: SAS-A and SAS-B. In SAS-A, the corruptibility γ is uniform across all the input minterms. In SAS-B, we select a subset of input minterms which would have higher corruptibility than others. Both SAS-A and SAS-B have very high impact on the application-level accuracy of ML models given any approx-key and guarantee an extremely long solving time for exact SAT attack.

4.1 SAS-A: Higher Corruptibility for all Input Minterms

To achieve the first objective mentioned above, we propose a modified Anti-SAT block, called the *SAS-A* block, which makes two modifications to existing Anti-SAT block. Firstly, the SAS-A block decomposes the logic block g (in Fig. 2(b)) into n_0-input mini-blocks g_0, as shown in Fig. 5 and each g function contains n/n_0 mini-blocks. The mini-blocks have the following properties:

1. The on-set size of each mini-block p_0 satisfies $1 \leq p_0 \leq 2^{n_0} - 1$.

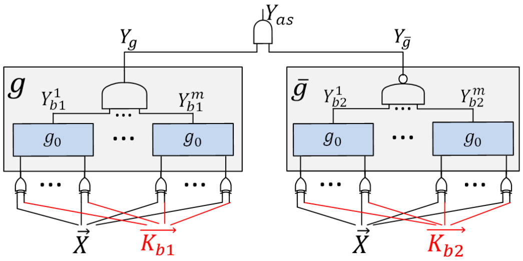


Figure 5: An n-input SAS-A block. Each mini-block g_0 has n_0 inputs and on-set size p_0.

2. $\quad \vec K_1$ and $\vec K_2$ where $\vec K_1 = \vec K_2$, there exists $\vec X$ such that $g(\vec X \quad \vec K_1) = g(\vec X \quad \vec K_2)$.

These are important premises of Theorem 4. In Theorem 4, we will derive an error rate lower-bound ϵ_0 of any wrong key of the SAS-A block. Such ϵ_0 holds for any wrong key and it can be tuned to a large value, so AppSAT type attacks will not be able to decipher a "good" enough approx-key.

Theorem 4. *The error rate of any wrong key of an n-input SAS-A block (in Fig. 5) is* $\epsilon \geq \epsilon_0 = p_0^{(n/n_0-1)}/2^n$ *for any wrong key.*

Proof. The proof contains the following steps.

1. We first show that a key for Anti-SAT $\vec K_b = (\vec K_{b1}, \vec K_{b2})$ is a wrong key if $\vec K_{b1} \quad \vec K_{b2} = \vec 0$. In other words, a key is a wrong key if $\quad i \quad [1, n]$ such that the i-th bit of $\vec K_{b1}$ is different from the i-th bit of $\vec K_{b2}$. This is a proven result from [12].

2. We want to get a lower bound of the number of corrupted input minterms for all such wrong keys. As shown in Fig. 5, to make $Y_{as} = 1$, the Anti-SAT block should have both $Y_g = 1$ and $Y_{\bar g} = 1$, which requires that

$$i \quad [1, n/n_0], Y_{b1}^i = 1 \quad \text{and} \quad j \quad [1, n/n_0], Y_{b2}^j = 0 \tag{4}$$

   The corrupted input patterns must satisfy the conditions in (4).

3. Let $\vec X^i$, $\vec K_{b1}^i$ and $\vec K_{b2}^i$ be the portion of the bits in $\vec X$, $\vec K_{b1}$, and $\vec K_{b2}$ that goes into the i^{th} mini-blocks of g and $\bar g$, respectively. Without loss of generality, we assume that $\vec K_{b1}^1 = \vec K_{b2}^1$. By the properties of g, since $\vec K_{b1}^1 = \vec K_{b2}^1$, there must exist at least one assignment to $\vec X^1$ such that $g(\vec X^1 \quad \vec K_{b1}^1) = g(\vec X^1 \quad \vec K_{b2}^1)$. This has two possibilities: $g(\vec X^1 \quad \vec K_{b1}^1) = 1$ and $g(\vec X^1 \quad \vec K_{b2}^1) = 0$, which satisfies (4), or $g(\vec X^1 \quad \vec K_{b1}^1) = 0$ and $g(\vec X^1 \quad \vec K_{b2}^1) = 1$. Suppose the latter is the case, then we let $\vec X^{1\square} = \vec X^1 \quad \vec K_{b1}^1 \quad \vec K_{b2}^1$ and this will satisfy $g(\vec X^{1\square} \quad \vec K_{b1}^1) = 1$ and $g(\vec X^{1\square} \quad \vec K_{b2}^1) = 0$.

4. For the remaining $\vec X^i, i \quad [2, n/n_0]$,since g_0 has on-set size p_0, there exists p_0 possible assignments to each $\vec X^i$ such that $Y_{b1}^i = 1$.

5. Based on steps 3 and 4, the total number of ways to construct a corrupted input pattern would be p_0^{n/n_0-1}.

6. Since the total number of input patterns is 2^n, the error rate for any wrong key is

$$\epsilon \geq \epsilon_0 = p_0^{(n/n_0-1)}/2^n \tag{5}$$

Hence proved. □

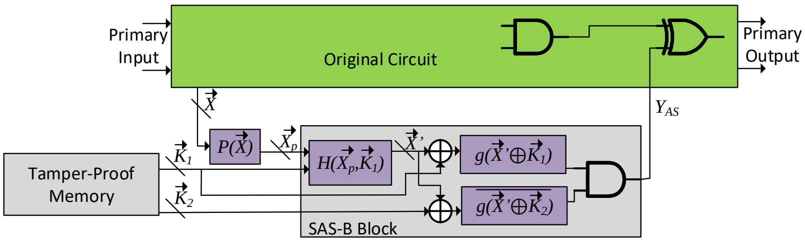


Figure 6: The Architecture of SAS-B Configuration 1 with the Details of the SAS-B Block

Theorem 4 provides a rigorous error rate lower-bound ϵ_0 for any wrong key of SAS-A. Based on Theorem 4, we can design an n-input SAS-A block with a guarantee of high error rate by tuning n_0 and p_0. Hence AppSAT will never be able to find a key whose error rate is smaller.

The corruptibility γ is uniform across all input minterms. This is because, given any input minterm, the number of wrong keys that corrupt this minterm is: $\vec{X} \quad Z_2^n$, $|K_{\vec{X}}| = p_0^{(n/n_0)}(2^n - p_0^{(n/n_0)})$ which is determined by p_0 and n_0. By Theorem 3, the number of SAT iterations of attacking SAS-A is lower bounded by

$$\lambda \geq \lambda_0 = \frac{1}{\gamma} = \frac{|K^W|}{|K_{\vec{X}}|} = \frac{2^{2n} - 2^n}{p_0^{(n/n_0)}(2^n - p_0^{(n/n_0)})} = \Theta\left(\frac{2^n}{p_0^{(n/n_0)}}\right) \tag{6}$$

As can be seen in Eq. (5) and Eq. (6), for $p_0 = 1$, the error rate lower bound $\epsilon_0 = \frac{1}{2^n}$ and the SAT iterations lower bound $\lambda_0 = 2^n$ are exactly those for Anti-SAT, which means that Anti-SAT is a special case of SAS-A when $p_0 = 1$.

4.2 SAS-B: SAS with non-Uniform Corruptibility

In order to get desirable application-level accuracy loss using SAS-A, the circuit-level corruptibility γ of minterms must be *uniformly increased* by tuning the parameters p_0 and n_0. However, this will inevitably reduce the number of SAT iterations below the order of 2^n, as shown in (6), and therefore SAS-A does not guarantee exponential SAT solving time.

SAS-B guarantees an exponential average SAT solving time while still having a large impact on the accuracy of ML applications. In SAS-B, instead of uniformly distributing the error across all possible inputs, we identify certain input patterns which potentially have a higher impact on the overall application-level error. We call these inputs **critical minterms**. Any incorrect key corrupts at least 1 critical minterm. For the other minterms, the corruptibility is the same as Anti-SAT.

In this section, we start with introducing the locking infrastructure, *i.e.* the SAS-B block. Then, we explain the two configurations of SAS-B. Last but not the least, the strategy to choose critical minterms based on the ML applications is presented.

4.2.1 The SAS-B Block

Typical ML applications contain a large number of multiply-and-accumulate (MAC) operations where an input vector is multiplied by a weight matrix and the products are summed up [48–52]. As the weight values are fixed and independent of data, we select a subset of weight values as critical minterms to have higher corruptibility. The strategy of choosing critical minterms is detailed in Sec. 4.2.4. Let M be the set of critical minterms and $m = |M|$ be the number of critical minterms. For the ease of implementation, we always choose m to be a power of 2.

Table 4: Illustration of the Partition of Wrong Key Space of a SAS-B Block with m Critical Minterms

$\vec{K}_1$ of wrong keys		$\vec{k}_1$	$\cdots$	$\vec{k}_{\frac{2^n}{m}}$	$\vec{k}_{\frac{2^n}{m}+1}$	$\cdots$	$\vec{k}_{2\frac{2^n}{m}}$	$\cdots$	$\vec{k}_{2^n}$
critical minterms	$\vec{X}_1$	•	•	•					
	$\vec{X}_2$				•	•	•		
	$\cdots$						$\cdots$		
	$\vec{X}_m$							•	•
non-critical minterms	$\vec{X}_{m+1}$	•							
	$\vec{X}_{m+2}$		•						
	$\cdots$					$\ddots$			
	$\vec{X}_{2^n}$								•

The basic locking infrastructure is the *SAS-B* block which is illustrated in Fig. 6. In order to describe the mechanism of the SAS-B locking scheme clearly, we use a reverse order and start our illustration from the output side:

- Y_{AS} is the output of the SAS-B block. If $Y_{AS} = 1$, a fault will be injected into the original circuit. g and $\bar{g}$ is the same as those in Anti-SAT. g has an on-set of size 1. Up to here, SAS-B is exactly the same as Anti-SAT. Therefore, the set of correct and wrong keys are also the same as Anti-SAT.

- The SAS-B block is designed to increase corruptibility of the **critical minterms**. In Configuration 1 of SAS-B, there is only one SAS-B block and hence the SAS-B block will inject a high amount of error for every critical minterms. In Configuration 2, however, as there are multiple SAS-B blocks, the critical minterms are distributed among the SAS-B blocks.

- A function block $\vec{X}\ = H(\vec{X}_p, \vec{K}_1)$ is inserted before the Anti-SAT and it works as follows. If $\vec{X}_p$ is not a critical minterm, then $\vec{X}\ = \vec{X}_p$. In this case, SAS-B works in the same way as Anti-SAT and $\vec{X}_p$ has a low corruptibility. If $\vec{X}_p$ is a critical minterm, then for a higher portion of $\vec{K}_1$ (where the portion equals the corruptibility of critical minterms), $\vec{X}_p$ is adjusted according to $\vec{K}_1$ to obtain $\vec{X}$ such that $g(\vec{X}\ , \vec{K}_1) = 1$ and hence the corruptibility is increased. $\vec{X}\ = H(\vec{X}_p, \vec{K}_1)$ further ensures that the wrong keys that corrupts each critical minterm are mutually exclusive and evenly partition the wrong keys space. More specifically, as the partitioning is based on the $\vec{K}_1$ part of the key, we have the following. Let $\mathsf{K}^1_{\vec{X}} = \{\vec{K}_1 |\quad \vec{K}_2$ such that $(\vec{K}_1, \vec{K}_2)$ K^W, $(\vec{K}_1, \vec{K}_2) \quad \mathsf{K}_{\vec{X}}\}$. Then we have

$$\vec{X}_1, \vec{X}_2 \quad \mathsf{M},\ |\mathsf{K}^1_{\vec{X}_1}| = |\mathsf{K}^1_{\vec{X}_2}|,\ \mathsf{K}^1_{\vec{X}_1} \quad \mathsf{K}^1_{\vec{X}_2} = \quad , \text{ and } \bigcup_{\vec{X} \quad \mathsf{M}} \mathsf{K}^1_{\vec{X}} = \mathsf{Z}_2^n \tag{7}$$

  where n is the number of bits in $\vec{X}$, $\vec{K}_1$, and $\vec{K}_2$. This effect is illustrated in Table 4.

- With the H function described above, any approx-key deciphered by any approximate SAT attack will corrupt one critical minterm. However, this also leads to a trivial lower bound of the iterations of exact SAT attacks: $\lambda_0 = m$. This is because, by selecting the m critical minterms as the first m DIs, all the wrong keys will be pruned out. In practice, we find that SAT can indeed find these critical minterms in a very small number of iterations. The reason can is that there are logic expressions in the H function block that are only satisfied by the critical minterms, which makes the critical minterms more likely to be chosen by the SAT solver.

- In order to counter this effect, we insert a pseudorandom permutation (PRP) function $P : X \rightarrow X_p$. Our implementation of the PRP is a reduced-bits, fixed-key, and reduced-rounds AES block. The property of AES ensures that its output is a one-to-one mapping of the input but the input-output mapping looks completely random. This makes it very hard for the SAT solver to solve for the critical minterms which is the input to the PRP function given the logic it observes from the H function. Instead, the SAT solver will choose a random DI and our goal to force the SAT iterations to be exponential can be achieved. Note that the attacker cannot formulate a SAT attack between $\vec{X}_p$ and the output to bypass the PRP module since this requires probing into the circuit which is considered in the attack model, as explained in Sec. 2.1.

We treat the PRP as a separate part of the SAS-B block since in Configuration 2 of SAS-B, there are multiple SAS-B blocks and the output of the PRP is shared by all the SAS-B blocks.

The 2 configurations of SAS-B will be introduced in the rest of this section.

4.2.2 Configuration 1: SAS-B with One SAS-B Block

This configuration is illustrated in Fig. 6. In this configuration, there is one SAS-B block. As the critical minterms evenly partition the set of wrong keys, the corruptibility of each critical minterm is $\gamma_c = \frac{1}{m}$.

Below we derive the expected SAT iterations of this configuration. Due to the functionality of the PRP block, we assume that the SAT solver chooses a DI uniformly at random in each iteration. The resilience of SAS-B to the SAT attack is quantified using the expected number of SAT iterations $E[\lambda]$. To start with, we give 2 useful lemmas.

Lemma 1. *Let D^i be the set of DIs that the SAT solver has chosen in the first i iterations and $\vec{X}$ be a primary input minterm. If $\mathsf{K}_{\vec{X}} \quad \bigcup_{\vec{X}' \quad \mathsf{D}^i} \mathsf{K}_{\vec{X}'}$, then $\vec{X}$ cannot be the DI of any SAT iteration beyond i.*

The proof is given in Appendix C.

Lemma 2. *For SAS-B Configuration 1, any critical minterm must exist in the set of DIs when SAT finishes: $\vec{X} \quad \mathsf{D}^\lambda \quad \vec{X} \quad \mathsf{M}$, where λ is the total number of SAT iterations and D^λ is the set of all the DIs.*

The proof is given in Appendix D.

Theorem 5. *The expected number of SAT iterations of SAS-B Configuration 1 is*

$$E[\lambda] = \frac{2^n + m}{2} \tag{8}$$

Proof. By Lemma 2, all the critical minterms must count toward the total number of SAT iterations. Therefore, we only need to find the expected number of *non-critical minterms* that are chosen as DIs.

As is a property of the SAS-B block and illustrated in Table 4, $\vec{X} \;\; / \; \mathsf{M}$, exactly one $\vec{X} \quad \mathsf{M}$ such that $\mathsf{K}_{\vec{X}'} \quad \mathsf{K}_{\vec{X}}$. By Lemma 1, if this $\vec{X}$ is chosen as DI before $\vec{X}$, then $\vec{X}$ cannot be chosen in further iterations any more. In other words, $\vec{X}$ will count towards the total number of iterations only when it is chosen before the critical minterm $\vec{X}$. By our assumption that the DI is chosen uniformly at random in each iteration, $\vec{X}$ has a probability of $\frac{1}{2}$ to be chosen as DI before $\vec{X}$ is chosen. As this is true for any non-critical minterm, the expected number of SAT iterations is $E[\lambda] = \frac{1}{2}(2^n - m) + m = \frac{2^n+m}{2}$. $\square$

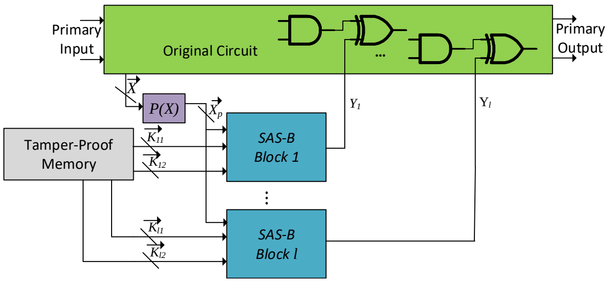


Figure 7: Illustration of the Configurations with Multiple SAS-B Blocks

4.2.3 Configuration 2: SAS-B with Multiple SAS-B Blocks

In this configuration, we have l SAS-B blocks as illustrated in Fig. 7. Each SAS-B block has n-bit primary input $\vec{X}_p$, which is the output of the PRP function and shared among all the SAS-B blocks, and $2n$-bit key input. The output of each SAS-B block is XOR'ed with a wire in the original circuit. Therefore, a fault is injected into the original circuit if at least 1 SAS-B block has output 1. Let M^j be the set of critical minterms for the j^{th} SAS-B block , $j = 1, 2, \ldots, l$. For the ease of implementation, we choose l also to be a power of 2 and $l \leq m$. The relationship between M^j and the total set of critical minterms M is that $M^1, M^2, \ldots, M^l$ *are mutually exclusive and evenly partition* M*, i.e.*

$$|M^1| = |M^2| = \cdots = |M^l|, \quad M^i \cap M^j = \qquad i = j, \text{ and } \bigcup_{k=1}^{l} M^k = M \tag{9}$$

In orther words, each SAS-B block has $\frac{m}{l}$ critical minterms and each critical minterm receives high corruptibility from exactly one SAS-B block. The corruptibility of any critical minterm is hence $\gamma_c = \frac{l}{m}$.

Lemma 3. *For SAS-B Configuration 2, any critical minterm must exist in the set of DIs when SAT finishes:* $\vec{X} \quad D^{\lambda} \quad \vec{X} \quad M$*, where* λ *is the total number of SAT iterations and* D^{λ} *is the set of all the DIs.*

The proof is given in Appendix E. Below, we will analyze the SAT attack resilience of this configuration by deriving the expected number of SAT iterations.

Theorem 6. *The expected number of SAT iterations of SAS-B Configuration 2 with* l *SAS-B blocks and* m *critical minterms is*

$$E[\lambda] = \frac{l \cdot 2^n + m}{l + 1} \tag{10}$$

Proof. By Lemma 3, every critical minterm must count toward the total number of SAT iterations. Therefore, we only need to derive the expected number of non-critical minterms that are chosen as DIs.

For any non-critical minterm $\vec{X} \; / \; M$, in the i^{th} SAS-B block, there exists exactly only one critical minterm $\vec{X}_i$ such that the set of wrong keys that corrupt $\vec{X}$ in this SAS-B block, $K_{i,\vec{X}\square}$, is a subset of the set of wrong keys that corrupt $\vec{X}_i$, $K_{i,\vec{X}_i}$. *i.e.* $K_{i,\vec{X}\square} \quad K_{i,\vec{X}_i}$. As the construction of Configuration 2 makes sure that this is true for any $i = 1, 2, \ldots, l$ and the critical minterms for each SAS-B block is mutually exclusive, there are a total of l such critical minterms. When these l critical minterms are all chosen as DI, they will cover the entire set of wrong keys that corrupt $\vec{X}$. Therefore, by Lemma 1, $\vec{X}$ must be chosen as DI at least before one of such critical minterms in order to count toward the total number of SAT iterations. This holds for any non-critical minterm.

By our assumption that the DIs are chosen uniformly at random in each SAT iteration, for each non-critical minterm, the probability that it will be chosen as DI is $\frac{l}{l+1}$. Therefore, the expected number of SAT iterations is $E[\lambda] = \frac{l}{l+1}(2^n - m) + m = \frac{l \cdot 2^n + m}{l+1}$. □

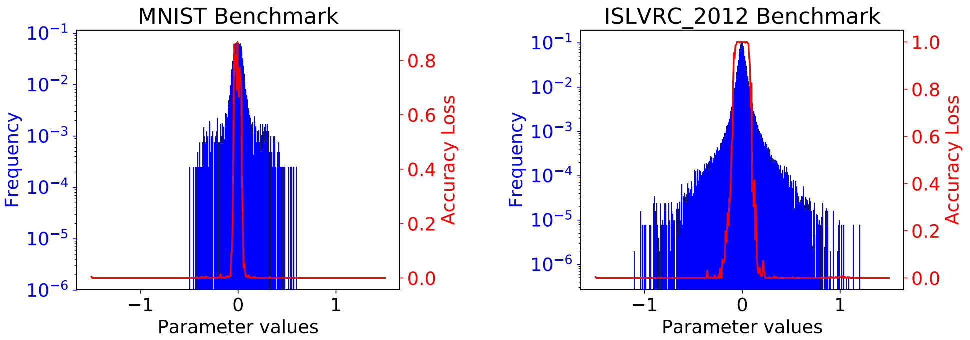


Figure 8: Weight Distribution (Histogram, Left Y Axis) and Application-Level Accuracy Loss (Right Y Axis) of LeNet trained on MNIST (Left) and CaffeNet Trained with ISLVRC-2012 Dataset (Right)

4.2.4 Choosing Critical Minterms

The critical minterms for injecting large errors should be selected judiciouly. These are the minterms which happen more often than others for the target ML workload. A careful analysis of the ML workload would help identify these typical minterms. Generally these minterms would be very few as compared to the overall input space of the functional modules.

Here we describe how to select the critical minterms. It turns out that the weight values of most trained ML models follow a similar distribution: they are more frequent near 0 and the frequency decreases as the parameter value moves away from 0. For example, Figure 8 shows the distribution of parameters of the LeNet model trained with MNIST dataset and CaffeNet model trained with the ISLVRC-2012 dataset. These two are the smallest benchmark and the largest benchmark, respectively, among the benchmarks used in this paper.

We select a subset of weights values to be critical minterms based on considerations on application-lavel corruptibility and hardware overhead. On one hand, the selected critical minterms should cause significant accuracy loss at the application-level. In Fig. 8, we also show the accuracy loss of the ML model in the following experiment: for each input minterm, we measure the accuracy loss of the ML model when every multiplication involving this minterm is corrupted with a probability of $\frac{1}{4}$ (while **no other** minterm is corrupted). We choose this probability because we choose $m = 4$ in the following experiments, and each critical minterm will be corrupted by $\frac{1}{m}$ of the wrong keys in Configuration 1 and higher in Configuration 2.

On the other hand, we explore to utilize hardware parallelism to reduce the overall hardware overhead. In most cases, ML applications are run on processors with very large number of parallel cores such as GPUs and specialized ML processors [33–41]. In our experiments, we only lock 1% of the cores using SAS-B and use the following strategy to schedule multiplications to the cores: 1) If the multiplication involves a critical minterm, then it will be scheduled to a core locked with SAS-B unless all the locked cores are busy, in which case it will be scheduled to a non-locked core. 2) If it does not involve a critical minterm, it will be scheduled to a non-locked core. Note that a similar compilation strategy was used in [55] where the multiplications involving certain values are assigned to specific multipliers.

Targeting high error for these small number of critical minterms implies that we do not need to have a SAS-A type approach where all minterms are being uniformly increased in there corruptibility level. Hence the higher degradation in SAT complexity can be avoided. Since critical minterms are injected with large amounts of error, the ML application also

Table 5: Properties of the 2 Configurations of SAS-B

Configuration	l	γ_c	$E[\lambda]$
1	1	$\frac{1}{m}$	$\frac{2^n+m}{2}$
2	$1 \le l \le m$	$\frac{l}{m}$	$\frac{l2^n+m}{l+1}$

Table 6: Error rate ϵ and # SAT iterations λ of a 16-input SAS-A block with different (n_0, p_0). ϵ_0 and λ_0 are the analytical lower-bounds. ϵ is the experimental error rate obtained by simulating all 2^{16} input patterns. λ is the experimental # iterations required by SAT attack.

n=16						
(n_0, p_0)	(2,1)	(2,3)	(4,7)	(4,8)	(4,13)	(4,15)
ϵ_0	1.53E-05	3.34E-02	5.23E-03	7.81E-03	3.35E-02	5.15E-02
ϵ	1.53E-05	9.13E-02	3.57E-02	3.66E-02	2.16E-01	1.86E-01
λ_0	65536	12	29	18	5	6
λ	65536	364	656	1334	344	187

experiences higher rates of classification failures.

4.3 Summary

In this section, we provide two mitigation techniques against SAT attack.

In SAS-A, we modify the Anti-SAT block design to achieve a higher error rate of wrong keys and higher corruptibility for any input minterm. Although the SAT solving time is no longer exponential, as will be shown in the next section, by properly choosing the configuration parameters n, n_0, and p_0 for SAS-A, an extremely long SAT solving time can still be achieved.

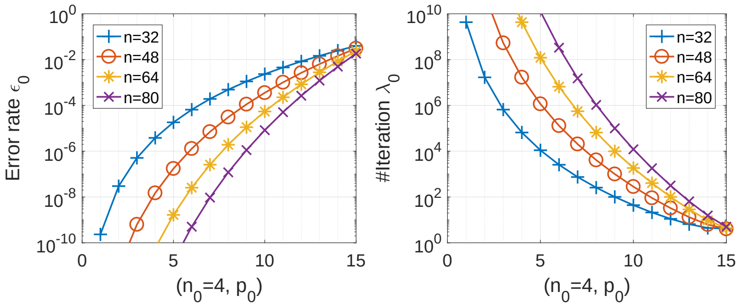
For SAS-B, in order to achieve high accuracy loss for the neural model, a strategy is provided to choose a small set of weight values as critical minterms. These critical minterms are those inputs which happen more frequently, and hence have higher impact on the accuracy, for a class of ML workloads. The SAS-B blocks ensure that the corruptibility of these critical minterms is very high. The pseudorandom permutation function forces SAT solver to choose DIs uniformly at random and hence will force the number of SAT iterations to be exponential in the input size n. The properties of both configurations of SAS-B are summarized in Table 5.

5 Experiments and Results

This section shows the experimental results of our proposed secure locking techniques for machine learning applications. Recall that our experimental setup is illustrated in Fig. 3 in Sec. 3. We obtain the gate-level netlists of a MIPS processor by synthesizing the high-level description using Cadence RTL Compiler. Then we lock the netlist using SAS-A and SAS-B in respective experiments. The application-level accuracy is emulated using Ristretto [54], a tool that emulates the neural network computation in hardware, by injecting error into the operations according to the error profile of the locking scheme.

5.1 Experiments and Results for SAS-A

In Sec. 4.1, we discuss the configuration of a SAS-A and analyze the lower-bounds for error rate ϵ_0 and # SAT iterations λ_0 as shown in Eq. (5) and Eq. (6). To validate the correctness of two analytical lower-bounds, we design a 16-input SAS-A block ($n = 16$) with different (n_0, p_0) and test their actual error rate and SAT iterations. The result is shown in Table 6. As seen, for different (n_0, p_0), we always have $\epsilon \ge \epsilon_0$ and $\lambda \ge \lambda_0$, which validates the correctness of our analysis for the lower-bounds ϵ_0 and λ_0.



(a) (b)

Figure 9: Lower-bounds of (a) error rate ϵ_0; (b) # SAT iterations λ_0 for different Strong Anti-SAT configurations (n, n_0, p_0).

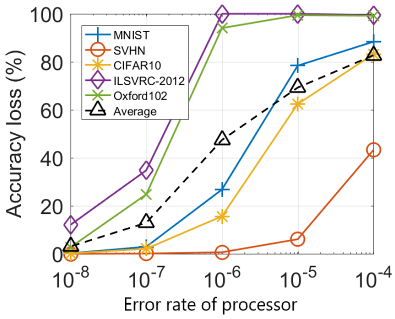
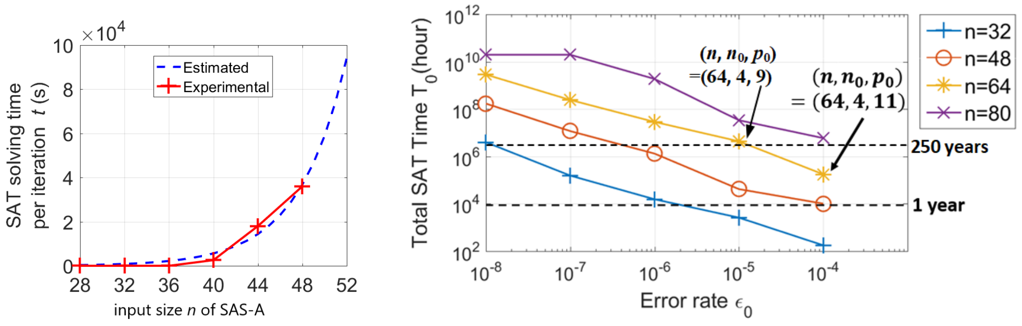


Figure 10: Accuracy loss v.s. error rate of approx-key for 5 benchmarks run on the approx-unlocked processor

In Fig. 9, we plot ϵ_0 and λ_0 for an n-input SAS-A block with different (n, n_0, p_0). For each n $\{32, 48, 64, 80\}$, we set $n_0 = 4$ and increase p_0 from 1 to 15. As seen in Fig. 9(a), by increasing p_0, the ϵ_0 will increases substantially. For different n, we can always find a configuration (n_0, p_0) such that the ϵ_0 is larger than certain desired error-rate. A desired ϵ_0 can be estimated to a value such that ML applications of interest is guaranteed to have a high error rate. Fig. 10 plots the relationship between accuracy loss of 5 ML models and error rate of a processor. Using this plot we can estimate the desired error rate for the processors which can then be used to design the SAS-A block.

Increasing ϵ_0, however, will inevitably decreases λ_0. This is validated in Fig. 9(b), which shows that λ_0 decreases as p_0 increases. Although there is a decrease in the number of SAT iterations, we show that given the very long SAT solving time per iteration, we still guarantee an extremely long total SAT solving time. Let us denote the SAT solving time per iteration as t. Fig. 11a shows the t for different n-bit data input processors locked with n-bit SAS-A. (Note that the arithmetic logic in the processor is adjusted with n.) t is computed by running the SAT attack for 10 hours and dividing this time by the number of iterations that the attack has progressed. As seen, as n increases, t increases exponentially. To validate the extrapolated exponential increase, we run the SAT attack on a processor locked with a SAS-A block of $n = 56$ and find that it can only process 1 iteration in 2.41E+05 seconds (about 67 hours), which is consistent with the estimated value based on the exponential curve. Hence such a predictive approach can be used to estimate the actual per-iteration SAT solving time of the real 32-bit MIPS processor where $n = 64$. Based on λ_0 in Fig. 9b and t in Fig. 11a, we can compute the lower-bound of total SAT solving time $T_0 = t \times \lambda_0$ for each tuple (n, n_0, p_0). We relate the SAT solving time to the application-level impact by observing the relationship between T_0 and ϵ_0. Based on Fig. 9a, we first determine the (n, n_0, p_0) which can just achieve certain ϵ_0. Then, we



(a) SAT solving time per iteration for n-input SAS-A locked processor

(b) Total SAT solving time v.s. error rate

Figure 11: Per-iteration and Total SAT Solving Time

Table 7: Classification Accuracy of ML Benchmarks Running on a Processor Locked with SAS-B

Benchmark	Configuration 1	Configuration 2		Activated Chip
		$l = 2$	$l = 4$	
MNIST	0.18	0.11	0.11	0.99
CIFAR10	0.10	0.10	0.10	0.89
SVHN	0.20	0.20	0.20	0.94
ISLVRC-2012	0.002	0.002	0.002	0.52
Oxford102	0.01	0.01	0.01	0.93

compute T_0 for each tuple and show the relationship between T_0 and ϵ_0 in Fig. 11b.

As seen, we can always select a configuration (n, n_0, p_0) so as to satisfy a desired ϵ_0 and T_0 simultaneously using Fig. 11b. For example, if the designer requires that $\epsilon_0 > 10^{-4}$ and $T_0 > 1$ year, we can choose $(n, n_0, p_0) = (64, 4, 11)$ as the configuration. If the requirements is $\epsilon_0 > 10^{-5}$ and $T_0 > 250$ years, we can choose $(n, n_0, p_0) = (64, 4, 9)$. Such configurations would result in 80% and 60% accuracy loss for most ML models running on approx-unlocked processors, respectively, as shown in Fig. 10.

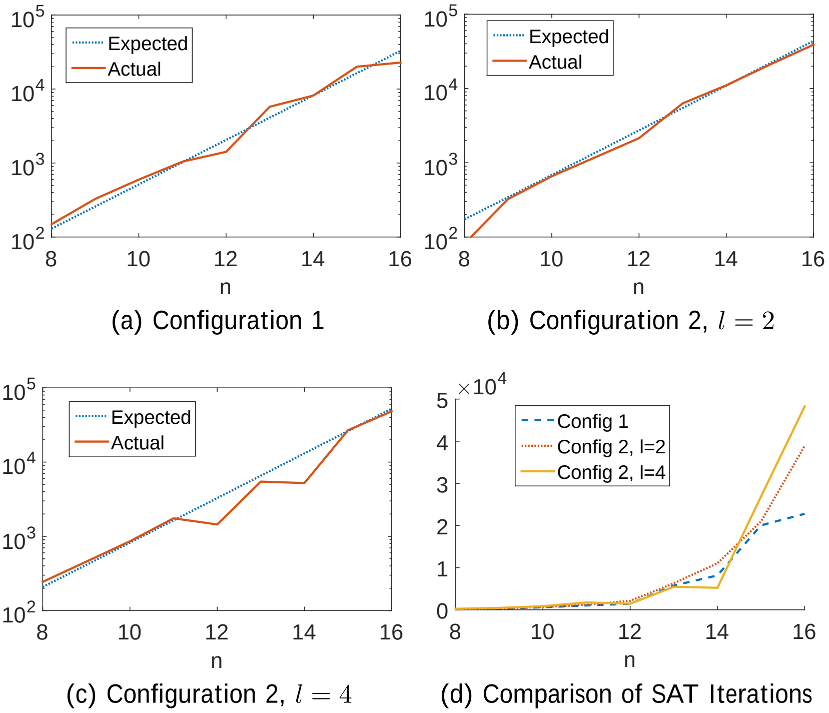
5.2 Experiments and Results for SAS-B

In this subsection, we measure the effectiveness of SAS-B as well as its SAT resilience.

From Fig. 8 we can observe that the accuracy loss of the ML model increases drastically as we lock those minterms which occur at greater frequency. This is quite intuitive since more multiplication results are corrupted when we lock the minterms of higher frequency. It is also noteworthy that the number of exclusive minterms that need to be locked to degrade the ML classification accuracy are very few.

We conduct the following experiment to obtain the application-level accuracy loss of ML workloads. Four minterms are chosen as critical minterms (*i.e.* $m = 4$) among the weight values that have high impact on application-level accuracy (see Fig. 8). With these minterms, we lock the processor using SAS-B with $n = 64$ and $l = 1, 2, 4$.

The majority of ML application nowadays are executed on processors that contain hundreds of processing cores [53] including Graphic Processing Units (GPUs) and specialized ML processors. The processor used in the experiment of SAS-B contains 100 32-bit MIPS cores among which only 1 core is locked using SAS-B (and the others are not locked). In order to emulate the application-level impact,*for every 100 multiplications in an ML model, we corrupt the result of the first multiplication with the probability of* $\frac{l}{m}$, *the corruptibility of critical minterms* γ_c, *by XOR'ing it with a random number, and corrupt all the other operations with the probability of* 2^{-n}. The scheduling is done at compile time and does



(a) Configuration 1

(b) Configuration 2, $l = 2$

(c) Configuration 2, $l = 4$

(d) Comparison of SAT Iterations

Figure 12: The Expected vs. Actual SAT Iterations for the 3 experimented SAS-B Configurations and their Compaison

not cause run time overhead or hardware overhead.

The classification accuracy of 5 ML benchmarks is shown in Table 7. Both Configuration 1 ($l = 1$) and Configuration 2 ($l = 2, 4$) are simulated. It can be observed that no matter which configuration of SAS-B is used, the loss of classification accuracy is very significant. Note that the critical minterms are the same across the experiments on all the ML models. The high accuracy loss of these models demonstrates that SAS-B can generalize to a broad class of ML models.

The expected vs. actual SAT iterations of SAS-B of different configurations are given in Fig. 12 where they are consistent with each other and grow at an exponential pace. Fig. 12d compares the SAT iterations of different configurations. It can be observed that, especially for larger n, a larger l comes with more SAT iterations as expected. As increasing l has a very small impact on the size of the locking module but significantly increases the SAT iterations, we would recommend choosing Configuration 2 and having a relatively large l.

5.3 Comparison with Existing Logic Locking Schemes

Now that we have demonstrated the effectiveness and SAT-resilience of both SAS-A and SAS-B, we evaluate their hardware overhead and compare with existing logic locking methods. The hardware (*i.e.* chip area) overhead is estimated using the number of gates. The baseline case is a processor with 100 32-bit MIPS cores without any logic locking. The details of each compared approach are as follows.

- *The existing logic locking scheme*: **every core** is locked using a 64-bit Anti-SAT block + 5% key-gates inserted into the processor circuitry using fault impact based analysis. This is the state-of-the-art logic locking approach proposed in [12] and used in Sec. 3.

Table 8: Comparison among Various Logic Locking Techniques

Locking Scheme	Application-Level Impact	SAT Solving Time	HW Overhead
existing logic locking approach [12]	very low after AppSAT	exponential	7.2%
SAS-A	high	not exponential but very long	2.9%
SAS-B	high	exponential	0.83%

- *SAS-A*: **every core** in the processor is locked with a SAS-A block with $(n = 64, n_0 = 4, p_0 = 9)$. This configuration is shown to guarantee an extremely long solving time for exact SAT attack in Sec. 5.1.

- *SAS-B*: **only 1 out of 100 cores** is locked using a 64-bit SAS-B block with $m = 4$ and $l = 4$. This configuration is shown to have both high application-level impact and exponential SAT solving time in Sec. 5.2.

The comparison between proposed SAS approaches with existing logic locking schemes are shown in Table 8. As seen, both SAS-A and SAS-B have higher application-level accuracy impact and lower hardware overhead than the existing approach. SAS-B has exponential SAT resilience and lower hardware overhead compared to SAS-A, although the critical minterms for SAS-B is only applicable for ML applications. For conventional applications, one can use SAS-A to lock the processor since these applications have much lower error resilience than ML applications. To build a multi-core processor that is secure for both conventional and ML applications, we suggest to lock a small portion of cores using SAS-B and all the other cores with SAS-A. This diversified approach will ensure both high accuracy impact on any application and exponential SAT solving complexity.

6 Conclusion

In this work, we investigate defense methodologies for ML workloads on locked processors. We motivate our work by exploiting the AppSAT attack on processors locked with the state-of-the-art logic locking scheme and showing that this scheme does not have any application-level impact on ML workloads after the processor is approximately unlocked. To counter this attack, we propose the Strong Anti-SAT (SAS) scheme to lock the processors and provide two types of SAS: SAS-A and SAS-B. SAS-A uniformly increases the corruptibility of every input minterm while SAS-B increases that of only a very small number of critical minterms that have high application-level impact. Experimental results show that both schemes effectively secure processors against the SAT and AppSAT attacks by ensuring extremely long SAT solving time and high application-level accuracy loss given any wrong key at the same time.

We also evaluate the hardware overhead of proposed approaches as opposed to existing locking schemes and show that both SAS-A and SAS-B have low hardware overheads. SAS-B has higher asymptotic SAT resilience and lower hardware overhead than SAS-A, although SAS-B is only applicable when the targeted application is a neural network (the majority of machine learning models). In order to build a secure processor for all applications, We suggest to lock a small portion of cores using SAS-B and all the other cores with SAS-A hence ensuring both high accuracy impact on any application and exponential SAT solving complexity.

References

[1] C. Helfmeier, D. Nedospasov, C. Tarnovsky, J. S. Krissler, C. Boit, and J.-P. Seifert, "Breaking and entering through the silicon," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security.* ACM, 2013, pp. 733–744.

[2] M. Kammerstetter, M. Muellner, D. Burian, C. Platzer, and W. Kastner, "Breaking integrated circuit device security through test mode silicon reverse engineering," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 549–557.

[3] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware trojans," *Computer*, vol. 43, no. 10, pp. 39–46, 2010.

[4] M. Rostami and et al., "A primer on hardware security: models, methods, and metrics," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1283–1295, 2014.

[5] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik, "Reverse engineering digital circuits using functional analysis," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2013, pp. 1277–1280.

[6] M. M. Tehranipoor, U. Guin, and S. Bhunia, "Invasion of the hardware snatchers," *IEEE Spectrum*, vol. 54, no. 5, pp. 36–41, 2017.

[7] J. Rajendran and et al., "Security analysis of logic obfuscation," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 83–89.

[8] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, "Logic encryption: A fault analysis perspective," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2012, pp. 953–958.

[9] J. Rajendran and et al., "Fault analysis-based logic encryption," *Computers, IEEE Transactions on*, vol. 64, no. 2, pp. 410–424, 2015.

[10] J. A. Roy and et al., "Epic: Ending piracy of integrated circuits," in *Proceedings of the conference on Design, Automation and Test in Europe*. ACM, 2008, pp. 1069–1074.

[11] X. Xu, B. Shakya, M. M. Tehranipoor, and D. Forte, "Novel bypass attack and bdd-based tradeoff analysis against all known logic locking attacks," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 189–210.

[12] Y. Xie and et al., "Mitigating sat attack on logic locking," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2016, pp. 127–146.

[13] Y. Xie and A. Srivastava, "Anti-sat: Mitigating sat attack on logic locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

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

[15] M. Yasin and et al., "Sarlock: Sat attack resistant logic locking," in *Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 236–241.

[16] M. Yasin, A. Sengupta, B. C. Schafer, Y. Makris, O. Sinanoglu, and J. J. Rajendran, "What to lock?: Functional and parametric locking," in *Proceedings of the on Great Lakes Symposium on VLSI 2017*. ACM, 2017, pp. 351–356.

[17] Y. Xie and A. Srivastava, "Delay locking: Security enhancement of logic locking against ic counterfeiting and overproduction," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 9.

[18] Y. Xie, C. Bao, and A. Srivastava, "Security-aware design flow for 2.5 d ic technology," in *Proceedings of the 5th International Workshop on Trustworthy Embedded Devices*. ACM, 2015, pp. 31–38.

[19] K. Vaidyanathan, B. P. Das, E. Sumbul, R. Liu, and L. Pileggi, "Building trusted ics using split fabrication," in *Hardware-Oriented Security and Trust (HOST), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 1–6.

[20] B. Shakya, N. Asadizanjani, D. Forte, and M. Tehranipoor, "Chip editor: leveraging circuit edit for logic obfuscation and trusted fabrication," in *Proceedings of the 35th International Conference on Computer-Aided Design*. ACM, 2016, p. 30.

[21] A. Baumgarten and et al., "Preventing IC piracy using reconfigurable logic barriers," *IEEE Design & Test of Computers*, 2010.

[22] M. Yasin, J. J. Rajendran, O. Sinanoglu, and R. Karri, "On improving the security of logic locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 9, pp. 1411–1424, 2016.

[23] K. Shamsi and et al., "Appsat: Approximately deobfuscating integrated circuits," in *Hardware Oriented Security and Trust (HOST), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 95–100.

[24] Y. Shen and H. Zhou, "Double dip: Re-evaluating security of logic encryption algorithms," in *Proceedings of the on Great Lakes Symposium on VLSI 2017*. ACM, 2017, pp. 179–184.

[25] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu, "Approxann: an approximate computing framework for artificial neural network," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 701–706.

[26] Z. Du, A. Lingamneni, Y. Chen, K. Palem, O. Temam, and C. Wu, "Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators," in *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*. IEEE, 2014, pp. 201–206.

[27] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 449–460.

[28] Y. Kim, Y. Zhang, and P. Li, "An energy efficient approximate adder with carry skip for error resilient neuromorphic vlsi systems," in *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press, 2013, pp. 130–137.

[29] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "Axnn: energy-efficient neuromorphic systems using approximate computing," in *Proceedings of the 2014 international symposium on Low power electronics and design*. ACM, 2014, pp. 27–32.

[30] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 356–367, 2012.

[31] A. Mondal and A. Srivastava, "Power optimizations in mtj-based neural networks through stochastic computing," in *Low Power Electronics and Design (ISLPED, 2017 IEEE/ACM International Symposium on.* IEEE, 2017, pp. 1–6.

[32] ——, "In-situ stochastic training of mtj crossbar based neural networks," in *Proceedings of the International Symposium on Low Power Electronics and Design.* ACM, 2018, p. 51.

[33] T. Chen and et al., "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, vol. 49, no. 4. ACM, 2014, pp. 269–284.

[34] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 92–104.

[35] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1. ACM, 2015, pp. 369–381.

[36] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 2014, pp. 609–622.

[37] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *Proceedings of the 43rd International Symposium on Computer Architecture.* IEEE Press, 2016, pp. 393–405.

[38] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on.* IEEE, 2016, pp. 1–12.

[39] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," *arXiv preprint arXiv:1704.04760*, 2017.

[40] S. Han and et al., "Eie: efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture.* IEEE Press, 2016, pp. 243–254.

[41] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *Proceedings of the 44th Annual International Symposium on Computer Architecture.* ACM, 2017, pp. 548–560.

[42] M. Yasin, B. Mazumdar, S. S. Ali, and O. Sinanoglu, "Security analysis of logic encryption against the most effective side-channel attack: Dpa," in *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS), 2015 IEEE International Symposium on.* IEEE, 2015, pp. 97–102.

[43] A. Chakraborty, Y. Xie, and A. Srivastava, "Template attack based deobfuscation of integrated circuits," in *Computer Design (ICCD), 2017 IEEE International Conference on.* IEEE, 2017, pp. 41–44.

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

[45] Y. Xie, C. Bao, and A. Srivastava, “Security-aware 2.5 d integrated circuit design flow against hardware ip piracy,” *Computer*, no. 5, pp. 62–71, 2017.

[46] R. Torrance and D. James, “The state-of-the-art in ic reverse engineering,” in *Cryptographic Hardware and Embedded Systems-CHES 2009*. Springer, 2009, pp. 363–381.

[47] X. T. Ngo, J.-L. Danger, S. Guilley, T. Graba, Y. Mathieu, Z. Najm, and S. Bhasin, “Cryptographically secure shield for security ips protection,” *IEEE Transactions on Computers*, vol. 66, no. 2, pp. 354–360, 2017.

[48] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[49] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, “Reading digits in natural images with unsupervised feature learning,” in *NIPS workshop on deep learning and unsupervised feature learning*, vol. 2011, no. 2, 2011, p. 5.

[50] A. Krizhevsky and G. Hinton, “Learning multiple layers of features from tiny images,” 2009.

[51] J. Deng, A. Berg, S. Satheesh, H. Su, A. Khosla, and L. Fei-Fei, “Ilsvrc-2012, 2012,” *URL http://www. image-net. org/challenges/LSVRC*, 2012.

[52] M.-E. Nilsback and A. Zisserman, “Automated flower classification over a large number of classes,” in *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing*, Dec 2008.

[53] A. Chakraborty, Y. Xie, and A. Srivastava, “Gpu obfuscation: attack and defense strategies,” in *Proceedings of the 55th Annual Design Automation Conference*. ACM, 2018, p. 122.

[54] P. Gysel and et al., “Hardware-oriented approximation of convolutional neural networks,” *arXiv preprint arXiv:1604.03168*, 2016.

[55] Z. Yang and A. Srivastava, “Value-driven synthesis for neural network asics,” in *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM, 2018, p. 1.

A Proof of Theorem 1

Proof. This can be proved by contradiction: suppose the key returned by the last step of SAT attack is a wrong key. This implies that there must exist a primary input $\vec{X}$ such that

$$C(\vec{X}, \vec{K}_c, \vec{Y}_c) \quad C(\vec{X}, \vec{K}, \vec{Y}) \quad (\vec{Y}_c = \vec{Y})$$

where $\vec{K}$ is the actual key, $\vec{Y}_c$ is the output with returned key $\vec{K}_c$ and $\vec{Y}$ is the correct output according to the actual key $\vec{K}$. $\vec{X}$ cannot be a previously found DI because otherwise $\vec{K}_c$ will not satisfy (3). We can see that $\vec{X}$ qualifies for a DI: just assign $\vec{K}_\alpha = \vec{K}_c$ and $\vec{K}_\beta = \vec{K}$. This means that (2) is still satisfiable and contradicts the criteria that no more DI can be found before the SAT attack goes to the final step.

Hence proved. □

B Proof of Theorem 2

Proof. Let us call the pair of a primary input minterm and a wrong key that corrupts this minterm a *input-wrong key pair (IWP)*.

The total number of IWPs will be equal to the total number of wrong keys times the number of input minterms corrupted by a wrong key, *i.e.* $|\mathsf{K}^W| \cdot |\mathsf{M}_{\vec{K}}|$ $\vec{K}$ K^W.

This number is also equal to the number of input minterms times the number of wrong keys that corrupts this minterm, *i.e.* $2^{n_p} \cdot |\mathsf{K}_{\vec{X}}|$ $\vec{X}$ $\mathsf{Z}_2^{2^{n_p}}$.

Therefore, we have

$$|\mathsf{K}^W| \cdot |\mathsf{M}_{\vec{K}}| = 2^{n_p} \cdot |\mathsf{K}_{\vec{X}}| \qquad \vec{X} \qquad \mathsf{Z}_2^{2^{n_p}} \text{ and } \vec{K} \qquad \mathsf{K}^W$$

which means

$$\frac{|\mathsf{K}_{\vec{X}}|}{|\mathsf{K}^W|} = \frac{|\mathsf{M}_{\vec{K}}|}{2^{n_p}} \qquad \vec{X} \qquad \mathsf{Z}_2^{2^{n_p}} \text{ and } \vec{K} \qquad \mathsf{K}^W$$

i.e. $\gamma = \epsilon$. Hence proved. □

C Proof of Lemma 1

Proof. Recall that Equation (2) gives the SAT formula for each SAT iteration:

$$C(\vec{X}_i, \vec{K}_\alpha, \vec{Y}_\alpha) \quad C(\vec{X}_1, \vec{K}_\beta, \vec{Y}_\beta) \quad (\vec{Y}_\alpha = \vec{Y}_\beta)$$

$$\bigwedge_{j=1}^{i-1}(C(\vec{X}_j, \vec{K}_\alpha, \vec{Y}_j) \quad C(\vec{X}_j, \vec{K}_\beta, \vec{Y}_j))$$

To satisfy the first line, at one of $\vec{K}_\alpha$ and $\vec{K}_\beta$ must be a wrong key that corrupts $\vec{X}$. However, since any wrong key that corrupts $\vec{X}$ also corrupts at least 1 previously found DI, this wrong key cannot satisfy the second line. Hence such $\vec{X}$ cannot be the DI in future iterations. □

D Proof of Lemma 2

Proof. Recall that g has on-set size 1. Let $\vec{P}$ be the very input that makes $g(\vec{P}) = 1$. $\vec{X}$ M, let $\vec{K}_1 = \vec{X}$ $\vec{P}$. Then, any $\vec{K} = (\vec{K}_1, \vec{K}_2)$ K^W is a wrong key that only corrupts $\vec{X}$. Therefore, $\vec{X}$ has to be chosen as a DI to prune out this wrong key. □

E Proof of Lemma 3

Proof. This is a natural extension to Lemma 2. Let $\vec{X}$ be a critical minterm and $\vec{X}$ M^j. Recall that g has on-set size 1. Let $\vec{P}$ be the very input that makes $g(\vec{P}) = 1$. $\vec{X}$ M^j, let $\vec{k} = \vec{X}$ $\vec{P}$. Then, let us consider the following wrong key $\vec{K} = (\vec{K}^1, \vec{K}^2, \ldots, \vec{K}^l)$ K^W which is composed as follows: $\vec{K}^j = (\vec{k}, \vec{K}_2^j)$ K_j^W where K_j^W is the set of wrong keys for the j^{th} SAS-B block. For any $i = 1, 2, \ldots, l$ that $i = j$, $\vec{K}^i$ K_i^C where K_i^C is the set of correct keys for the i^{th} SAS-B block. Such a key $\vec{K}$ is a wrong key that only corrupts $\vec{X}$. Therefore, $\vec{X}$ has to be chosen as a DI to prune out this wrong key. □