# Hardware-Software Co-Design Based Obfuscation of Hardware Accelerators

Abhishek Chakraborty and Ankur Srivastava
Department of Electrical and Computer Engineering, University of Maryland, College Park
abhi1990@umd.edu, ankurs@umd.edu

***Abstract*—Existing logic obfuscation approaches aim to protect hardware design IPs from SAT attack by increasing query count and output corruptibility of a locked netlist. In this paper, we demonstrate the ineffectiveness of such techniques to obfuscate hardware accelerator platforms. Subsequently, we propose a Hardware/software co-design based Accelerator Obfuscation (HSCAO) scheme to provably safeguard the IP of such designs against SAT as well as removal/bypass type of attacks while still maintaining high output corruptability for applications. The attack resiliency of HSCAO scheme is manifested by using a sequence of keys to obfuscate instruction encoding for an application. Experimental evaluations utilizing an accelerator simulator demonstrate the effectiveness of our proposed countermeasure.**

## I. Introduction

Hardware designs are increasingly outsourced to offshore foundries in order to reduce the cost of fabrication. However, this trend has raised several security concerns related to reverse-engineering, Intellectual Property (IP) piracy, overproduction, and counterfeiting of designs [18]. Several *logic locking* approaches have been proposed in literature [16], [17], [18], [21], [24] to counter supply chain attacks on hardware designs by an untrusted foundry. In a typical combinational logic locking scheme, a design is obfuscated by inserting additional *key-gates* in the synthesized netlist and it exhibits the correct functionality only when a chip is *activated* by loading the correct key into an on-chip *tamper-proof* memory.

Customized hardware accelerators [12], [15], [1] have gained a lot of popularity with the increase in compute-intensive workloads as well as deep neural network (DNN) applications. To the best of our knowledge, there has been no study to analyze the security guarantees provided by existing logic locking schemes for protecting the design IP of domain-specific hardware accelerator platforms such as Google's Tensor Processing Unit (TPU) [15]. For the sake of illustration, in this work, we consider a Google TPU-like accelerator chip based on an open-source instruction set architecture (ISA) standard [9]. We obfuscate the control (decoder) logic as well as the computational module (matrix multiply unit) of such an accelerator using state-of-the-art stripped-functionality logic locking (SFLL) approach [25]. Then, we devise a SAT formulation based attack to completely deobfuscate the chip's decoder logic despite the existence of theoretical security guarantees of SFLL [25]. The computational module is also approximately deobfuscated to the point that applications running on the platform see very little to no impact on their functionality. For example, the experimental results of running a regression task on Boston Housing dataset on the approximately unlocked TPU-like design (simulated using a TPU simulator framework [8]) demonstrate the ineffectiveness of such a logic obfuscation approach to protect the IPs of hardware accelerator chips.

Subsequently, we present a Hardware/software co-design based Accelerator Obfuscation (HSCAO) approach which renders an unactivated hardware accelerator design completely useless for running any application correctly. Commercially available hardware accelerators come with *proprietary* software development kits (SDKs) [6] without which these accelerator chips cannot be used. Such an SDK is developed in the design house and its details are not exposed to an untrusted foundry. Moreover, state-of-the-art software obfuscation tools [3] can be used to protect an SDK against reverse-engineering. In our proposed countermeasure framework, the accelerator's *proprietary* SDK serves as the *root of trust* and it generates a sequence of keys to obfuscate instruction binary encoding of a compiled application. The instruction binaries are successfully deobfuscated on-chip during application run-time only if the accelerator hardware is properly *activated* post-fabrication. Unlike traditional logic locking approaches, this scheme partitions obfuscation/ deobfuscation procedures across software/ hardware portions of the hardware accelerator framework, thus creating a novel co-design based locking solution.

Our proposed HSCAO scheme augmented designs are provably resilient to state-of-the-art SAT formulation based attacks [20], [14], [19] as well as removal [26] or bypass type attacks [23]. The high security guarantees of the proposed HSCAO scheme against SAT attack is achieved due to the use of a sequence of keys (generated from an initial *secret key*) for locking the compiled instruction binaries. This technique ensures that **only** the *secret key* successfully deobfuscates the accelerator design. According to our theoretical analysis, the probability of SAT attack converging to the *single* correct key is exponentially small in terms of the key-size, hence, making it as impractical as brute force attack. In order to evaluate the effectiveness of our proposed technique, we augmented the proposed hardware-software co-design based obfuscation scheme to OpenTPU accelerator simulator [8] and studied the application-level error impact due to use of wrong keys for unlocking the accelerator. The outcomes of the experiments demonstrate that any wrong key leads to significant output corruptions, thus highlighting the effectiveness of our proposed HSCAO countermeasure.

## II. Related Work

Several existing logic locking techniques [16], [17], [18] have been shown to be vulnerable to SAT attack which successfully deobfusates a netlist within few hours [20]. The SAT formulation based attack strategy iteratively finds *distinguishing input-output* (DIO) pairs to eliminate unique subsets of wrong keys until none exists, thus converging to find a correct key. Point-function based schemes like Anti-SAT [21] and SARLock [24] were subsequently proposed which make the number of SAT iterations an exponential function in terms of the key-size, thus countering SAT attack in practice. However, such point-function based schemes were susceptible to AppSAT attack [19] which approximately unlocks a netlist. Even recently proposed Delay Locking approach [22] which obfuscates a circuit utilizing its timing profile has also been shown to be vulnerable to SAT formulation based attack [11]. In [25], the authors propose a provably secure logic locking approach called stripped-functionality logic locking (SFLL) which not only thwarts all known attacks but also provides a quantifiable trade-off among them. In $SFLL\text{-}HD^h$ technique, all *protected input cubes* are at same Hamming distance (HD) of $h$ from the correct key. If $k$ is the key-size, then the number of such protected inputs $|P|$ is $\binom{k}{h}$. Also, if we assume $n$ inputs to the netlist ($n \geq k$), then $SFLL\text{-}HD^h$ is:

- $(k - \lceil log_2\binom{k}{h} \rceil)$-secure against SAT attack
- $2^{(n-k)}\binom{k}{h}$-secure against removal attack

where, the notion of $\lambda$-secure and related theoretical derivations are provided in [25]. In summary, $h$ can be adjusted to trade resilience between the above attacks: $h \in \{0, k\}$ deliver highest resilience to SAT attack, whereas $h = k/2$ maximizes the resilience to removal attack. For any wrong key, there will be *at most* twice the number of error injection at the output as the number of protected input cubes. This implies that for a wrong key the maximum probability of erroneous output is $p_{SFLL}$=$2\binom{n}{h}/2^n$ (with $k = n$). In section III-D2, we utilize this expression for $p_{SFLL}$ to study the error injected by the SFLL scheme when running an application on a locked accelerator.

## III. SAT Attack on Locked Accelerator

### A. Hardware accelerators

*1) Existing architectures:* The ever increasing demand for computational power by compute intensive applications such as speech recognition, computer vision, natural language processing, search ranking and other DNN applications have made architectural innovations crucial to achieve high performance and energy efficiency. GPUs as well as several domain specific hardware accelerators such as Diannao [12] and Google's Tensor Processing Unit (TPU) [15] have been developed which provide higher throughput while consuming much lower energy compared to general purpose processors. In this work, we study the effectiveness of existing logic locking schemes to protect the hardware accelerator IPs from an untrusted foundry. We use the TPU framework for the purpose of illustration, however our ideas are equally applicable to other accelerator designs.

*2) Open ISA:* The operation of any processor or accelerator hardware is guided by its instruction set architecture (ISA). In this work, we assume a hardware accelerator design which is based on an open-source ISA standard (i.e., instruction formats and opcodes are *known*). While some conventional ISAs have been proprietary, recent works have touted the benefits of making the ISA open source [9]. The industry would benefit by making the ISA free as it will enable affordable processor designs to expand the IoT framework. Moreover, open-source ISA doesn't imply that commercial proprietary processor designs cannot use such an ISA. This is due to the fact that the though the ISA is standardized, the chip designer decides the micro-architectural features to be implemented as well as the logical and physical design approaches [7]. For example, Intel 64 processors [5], Codix-Bk3 [2] use open ISAs.

### B. Threat Model

We consider that an attacker in the *untrusted foundry setting* has access to the following three components for analysis:

- An *activated* hardware accelerator chip bought from the open market, used to obtain the *correct* input-output responses.
- The gate-level netlist of the hardware accelerator chip reverse-engineered from layout level details available in GDS-II file.
- The hardware accelerator's software development kit (SDK) and ISA standard.

It is to be noted that availability of the first two components have been assumed in several related works [16], [17], [18], [20], [21]. In addition, we also consider that the adversary has access to the SDK of the hardware accelerator. This is a reasonable assumption as the SDKs of most commercially available hardware accelerators are freely available for download (e.g., Nvidia's CUDA SDK [6]). We assume that the attacker can use such an SDK to generate executable corresponding to some developed microbenchmark application. This enables her to observe *correct* input-output response pairs from the activated chip by mapping the instruction binaries to the corresponding register contents [10].

### C. Attack Framework

*1) Google TPU:* In this work, we use the Google TPU architecture [15] as a testbed for illustrating our ideas. In Google TPU framework, the host CPU sends instructions over PCIe bus to an instruction buffer for the TPU to execute rather than fetching them itself. The main computational component called the *Matrix Multiply Unit (MMU)* consists of 256X256 MACs which performs 8-bit multiply-and-adds on signed/unsigned integers. The inputs to the MMU are provided by *weight FIFO* and *unified buffer* (UB) components. The MMU outputs 16-bit products which are collected in the *accumulator unit*, which are then passed on to the *activation unit*. Finally, the results are written back to UB. A DMA controller transfers data between the CPU host memory (HM) and UB.

*2) Obfuscation Approach:* We consider locking both the instruction decoder logic and the matrix-multiply unit (MMU) of a TPU-like chip using state-of-the-art $SFLL\text{-}HD^h$ scheme [25]. As highlighted in section II, SFLL approach provides a quantifiable trade-off among all known types of attacks against logic locking techniques, including SAT based attacks [20], [19]. In order to study the application-level error impact of obfuscating accelerator designs, we used the OpenTPU simulator [8] and ran a regression task on Boston Housing dataset (more experimental details in section V).

### D. SAT Attack on locked TPU-like chip

*1) Decoder Deobfuscation:* To perform the experiments, we first synthesized a gate-level netlist of an 8-to-20 instruction decoder design (assuming there are 20 valid 8-bits opcodes in the ISA). Then, we locked the design following $SFLL\text{-}HD^0$ technique (i.e. h=0) to ensure maximum resilience to SAT attack (as much as point function schemes such as Anti-SAT [21] and SARLock [24]). Note that the number of valid opcodes (20 in our case) is much smaller compared to the size of input space of the decoder ($2^8$) and also known to the attacker (open ISA). We launched a modified SAT attack: in each iteration of conventional SAT formulation [20] we used a valid opcode as distinguishing input. The key returned by the SAT solver after all such opcodes are exhausted is guaranteed to retrieve the correct decoder functionality for the given ISA. In our experiments (second column of table I) we observe that only 2 iterations of SAT attack was sufficient to find a correct key with *known* opcode inputs.

In order to further study the effectiveness of this modified SAT attack strategy on the decoder design, we also augmented random (RLL) [18] and fault analysis based logic locking (FLL) [16] to the SFLL scheme for inserting additional key-gates (with gate overheads of $5\%, 10\%,$ and $20\%$) in the netlist. As evident from table I, such augmentations didn't help improve the security of locked decoder netlist and only 3 iterations of SAT attack was sufficient to deduce a correct key in all the test cases. In summary, despite using SAT attack tolerant locking schemes such as $SFLL\text{-}HD^0$, the SAT attack was highly effective to deobfuscate the decoder circuit.

TABLE I: SAT attack results on locked decoder netlist

| *scheme* | SFLL | SFLL+RLL | | | SFLL+FLL | | |
|---|---|---|---|---|---|---|---|
| *% overhead* | – | 5 | 10 | 20 | 5 | 10 | 20 |
| *#SAT iterations* | 2 | 3 | 3 | 3 | 3 | 3 | 3 |
| *time* ($10^{-3}s$) | 4 | 8 | 8 | 8 | 8 | 8 | 8 |

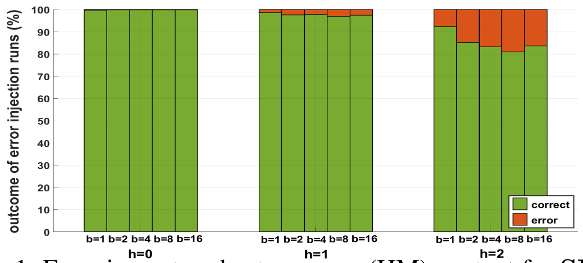


Fig. 1: Error impact on host memory (HM) content for $\mathbf{SFLL\text{-}HD^h}$ locked MMU design

*2) MMU Deobfuscation:* Next, we evaluate the effectiveness of $SFLL\text{-}HD^h$ scheme to lock the multiplier units in the MMU component of the accelerator chip. Each multiplier takes as input two 8-bits operands and outputs a 16-bits representation of the product. For our experiments, we set the key-size $k$ equal to the the multiplier's input size $n$, i.e., $k = n = 16$, and the $h$ value of $SFLL\text{-}HD^h$ scheme was varied between $h \in \{0, 1, 2\}$. As noted earlier, $h = 0$ represents the highest security to SAT attack but least output corruptability; while $h = 2$ has higher corruptability but substantially smaller SAT attack resilience. On this locked TPU-like chip, we run a regression application to evaluate the security offered by the obfuscation scheme considered. We term the outcome of an application run being *correct* if the host memory content matches exactly with the golden memory content (obtained without any error injection), else we term the outcome as being *erroneous*. For each type of $SFLL\text{-}HD^h$ lock, we injected appropriate amount of error in the multiplication outcomes as follows: First, a $b$ bit random number was generated and then, it was XORed with the correct multiplier output with a probability of $p_{SFLL}$=$2\binom{n}{h}/2^n$ (as outlined in section II). In addition, we varied the number of bits $b \in \{1, 2, 4, 8, 16\}$ to capture the error impact due to varying widths of *fanout cones* affecting the netlist output for a wrong key. In figure 1, we report the percentage of erroneous outcomes in the host memory contents of the OpenTPU simulator out of 1000 error injection runs for different values of $h$. As evident from the figure, for (i) $h = 0$: almost all (ii) $h = 1$: above 95% and (iii) $h = 2$: above 85% of the fault injection runs (out of 1000) are correct across all the values of $b$. There are two main reasons behind this outcome: First, the error injected by the locking schemes are not very high. Secondly, several machine learning applications (such as the regression task used for experiments) have inherent resiliency to errors. This strongly motivates the need of a new type of obfuscation framework which effectively safeguards the IP of hardware accelerator designs from an attacker in untrusted foundry.

## IV. Hardware-software Co-design based Accelerator Obfuscation

### A. Root of Trust

An overview of our proposed HSCAO framework is presented in figure 2 (details in section IV-B). HSCAO relies on partitioning the obfuscation/deobfuscation task between the accelerator's SDK and the hardware. Conventional locking approaches [18], [16], [25] rely exclusively on hardware keys to obfuscate a design. However, most hardware accelerators comprise of *proprietary* SDKs [6] without which these accelerator chips cannot be used. These SDKs represent substantial software development efforts and are developed by the design house (generally not exposed to the untrusted fab). The details of an SDK implementation can be easily hidden from users/fabs using software obfuscation techniques [13]. For example, DexGuard tool [3] provides state-of-the-art software obfuscation features to protect an SDK against reverse-engineering. As per our threat model, the attacker can only use such an SDK as a *black box* without having access to its internal details. It is quite reasonable to assume that the attacker doesn't have the capability to develop a substitute SDK utilizing the GDS-II file of a chip. This is because several architecture-level design specifications/protocols of accelerator designs are not publicly available [4], [6], [1].

### B. Proposed HSCAO Framework

HSCAO framework consists of following three components:

- **Key sequencer:** It generates a pseudo-random key sequence using a secret key $K_{seed}$ as initial seed value.
- **Software-level Obfuscation:** The *control bits* of instructions consisting of opcode and flag bits are locked by proprietary SDK and then communicated to the accelerator hardware design.
- **Hardware-level Deobfuscation:** Subsequently, the *control bits* are unlocked on-chip using a hardware-level deobfuscation module before further processing the instructions in other modules.

The overall approach is to share the obfuscation/deobfuscation processes between the software and hardware components of an accelerator. The software portion obfuscates the instructions with *dynamic* keys generated using the key sequencer algorithm. The hardware portion replicates the key sequencer on-chip and is fully synchronized with its software counterpart to deobfuscate the locked instructions. The secret key $K_{seed}$ is shared by the hardware and software counterparts of HSCAO framework: it resides in the SDK (root of trust) and in an on-chip tamper-proof memory (TPM). It is to be noted that such a hardware-software co-design based obfuscation approach aims to protect the design IP of an accelerator, not the information content of an user application running on it. Next, we present the details of different components in HSCAO framework.

*1) Key sequencer:* The key sequencer utilizes the secret key $K_{seed}$ to generate a pseudo-random sequence of keys $K_{seq}$ for locking/unlocking of instructions in software/hardware counterparts. According to the threat model considered (see section III-B), the attacker has knowledge of the accelerator's instruction format: we consider that an instruction consists of $n$ bits of opcode and control flags, referred to as *control bits*. The remaining bits of the instruction consists of data handling and memory access related information, referred to as *non-control bits*. We lock the functionality of overall hardware accelerator by only obfuscating the *control bits* of instructions in an application. The *control bits* of the $i^{th}$ instruction is locked by XORing it bit-wise with $n$-bits of $K_i$ which is the $i^{th}$ key in $K_{seq}$. The software/ hardware counterparts initializes their key sequencer implementations with the **same** $K_{seed}$, thus generating identical $K_{seq}$ for locking/unlocking instructions.

Now, we describe the process of generating the pseudo-random key sequence $K_{seq}$ from the secret key $K_{seed}$. In our design we use $N$ cyclic shift registers (each $n$ bits in length) as shown in figure 3. Both $N$ and $n$ are design parameters. These $N$ shift registers are initialized with $K_{seed}$ of size $n \times N$ bits. Figure 3 illustrates the state of the key sequencer for generating the first key $K_1$ in the sequence from the secret key $K_{seed}$. The $m^{th}$ bit, $m \in \{1, 2, \ldots, n\}$ , of $K_1$ (denoted by $K_{1,m}$) is obtained by XORing the $m^{th}$ bits of all the $N$ shift registers. For generating the next key $K_2$ in the sequence, all the shift registers are cyclically shifted by certain number of bits as specified in a *shift vector* $\vec{S} = [s_1, s_2, \ldots, s_N]$ where

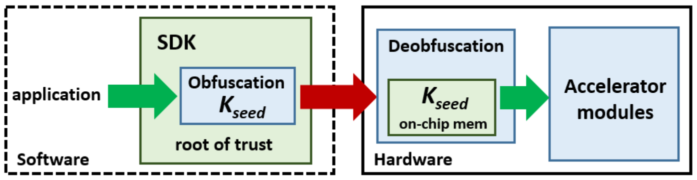


Fig. 2: Hardware-software Co-design based obfuscation

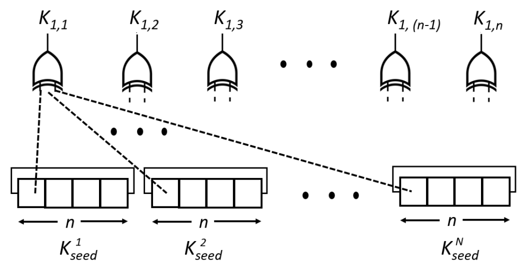


Fig. 3: Cyclic shift register based key sequencer

$s_j$, $j \in \{1, 2, \ldots, N\}$, corresponds to the number of bits the $j^{th}$ register is to be shifted. Now, as before, the contents of all the registers are bit-wise XORed together to generate the key $K_2$. This process is repeated to generate the subsequent keys in $K_{seq}$. It is to be noted that the *shift vector* is randomly generated at run-time in the accelerator SDK for software-level obfuscation of an instruction, and its contents are **not known** to an attacker. For generating the same $K_{seq}$, not only $K_{seed}$ must be common, but also these *shift vectors* needs to be shared between the software and hardware counterparts in the HSCAO framework.

*2) Software-level Obfuscation:* The accelerator's SDK (root of trust) generates $K_{seq}$ by implementing the above key sequencer algorithm in software. Each key in $K_{seq}$ is XORed bit-wise with the *control bits* of an instruction to obfuscate it. Thus, the application binary is locked in software as a function of secret key $K_{seed}$ and *shift vectors* (dynamically generated per instruction). The SDK also locks the *shift vector* $\vec{S}$ to generate *locked shift vector* $\vec{S'} = [s'_1, s'_2, \ldots, s'_N]$ for every instruction, where the mapping from $\vec{S}$ to $\vec{S'}$ is obtained using a *secret* look-up table (*shift LUT*). This *shift LUT* is not available to an attacker who uses the SDK as a *black-box*. Note that, like $K_{seed}$, the contents of *shift LUT* is also shared between the software and hardware counterparts of HSCAO framework. For generating the same $K_{seq}$ in hardware, the *locked shift vectors* are communicated to the chip using following ISA extensions:

- **INITK**: Initiate key instruction resets the $N$ registers (each of length $n$ bits) with the $n \times N$ bits of $K_{seed}$.
- **CSHFT**: Cyclic shift instruction shifts the contents of $N$ registers as per the corresponding elements in *shift vector* $\vec{S} = [s_1, s_2, \ldots, s_N]$ only if the *shift LUT* is correctly configured on-chip, else a faulty mapping from $\vec{S'}$ to $\vec{S}$ will result in wrong operations. The CSHFT instruction format is as follows:

  $$\text{CSHFT} \quad [s'_1, s'_2, \ldots, s'_N]$$

  The CSHFT instruction consists of an array of length $N$, where each element $s'_i$ corresponds to a random number between $-r_{max}$ and $+r_{max}$.

In section IV-C, we show that an adversary will be practically unable to reconstruct the key sequence $K_{seq}$ without the knowledge of $K_{seed}$ and *shift LUT* contents. Thus in effect, the software component of HSCOA successfully locks the functionality of accelerator design.

*3) Hardware-level Deobfuscation:* The hardware-level deobfuscation module serves as a counterpart of the software-level obfuscation module. It replicates the key sequencer design on-chip to unlock the *control bits* of software-obfuscated instructions using the *same* secret key $K_{seed}$ and *shift LUT* information (by bit-wise XORing keys with the locked control bits). On encountering an INITK instruction from the software interface, the hardware key sequencer design resets the cyclic shift registers with $K_{seed}$ content. If a CSHFT instruction is encountered, first the content of the *shift vector* $\vec{S}$ is retrieved from the *locked shift vector* $\vec{S'}$ (using *shift LUT* stored in on-chip TPM) and then, all $N$ registers are cyclically shifted according to $\vec{S}$. This process allows perfect synchronization between the key sequencers in software and hardware counterparts of the HSCOA framework. Therefore, both the modules generate the *same* $K_{seq}$ for performing instruction obfuscation/deobfuscation operations. As highlighted in figure 2, the above deobfuscation process is carried out on-chip before performing any application-specific computations in accelerator modules. Note that as both $K_{seed}$ and *shift LUT* are configured by the designer post-fabrication in on-chip TPM, these are not known to an untrusted foundry.

*4) Overall process:* In summary, the proposed HSCOA framework obfuscates the functionality of a hardware accelerator chip as follows: The proprietary SDK locks the encoding of instructions and sends them to the accelerator chip, where they are deobfuscated using the shared $K_{seed}$ and *shift LUT* information. By default, the software initially sends an INITK instruction to reset the on-chip shift registers in the hardware key sequencer module. The very first instruction of an application is locked using the key $K_1$ which depends on the state of the registers initialized with $K_{seed}$. The obfuscation of subsequent instructions using keys $\{K_2, K_3, \ldots, K_I\}$ in $K_{seq}$ is governed by the *shift vectors* which are randomly generated at run-time in secure SDK. This information is communicated to the accelerator chip using CHSFT type instructions. Note that one does not need to obfuscate each of the subsequent instructions with a separate key in $K_{seq}$. The designer can choose to lock blocks of instructions with common keys or lock a few randomly selected instructions, thereby *reducing the locking overhead*. In section IV-C, we provide theoretical analysis to demonstrate the resiliency of HSCOA framework against SAT formulation based attack. We also describe how this framework is also immune to removal [26] or bypass [23] types of attacks. We assume that the sizes (design parameters) of $K_{seed}$ and *shift LUT* are large enough so that the attacker cannot devise any brute force based attack strategy. Our proposed hardware-software based obfuscation approach can also be seamlessly integrated with conventional logic obfuscation schemes [18], [17], [25] to lock other components (like MMU) of an accelerator chip to further enhance the security-level.

## *C. Security Analysis of HSCAO*

*1) Resiliency to SAT attack:* First we illustrate how the process of determining $K_{seed}$ is computationally infeasible through SAT formulation based attack [20]. According to our threat model, an attacker has access to the netlist of the key sequencer design. Her objective is to find the key sequence $K_{seq}$ for unlocking the software obfuscated instructions using SAT attack. Note that the very first instruction is obfuscated using $K_1$, which is derived from the state of the key sequencer initialized with $K_{seed}$ (using default INITK instruction). Unlike subsequent keys in $K_{seq}$ which are dependent on *shift vectors* generated at run-time (and thus varies from one application run to another), $K_1$ is run-time independent. The attacker can deduce $K_1$ as follows: At first, she develops a microbenchmark application having knowledge of all the instruction types. Then, she finds $K_1$ by simply XORing bit-wise the locked *control bits* of first instruction with the correct opcode bits (known from ISA). This is because $(a \oplus b) \oplus a = b$, where $a$ represents the opcode bits, $b = K_1$ and $\oplus$ denotes bit-wise XOR operation.

Note that $K_1$ is derived from $K_{seed}$ using the key sequencer algorithm whose functionality is known to the attacker. Hence, she can use SAT solver (or any other Boolean solver) to find a key $K_{eqv}$ belonging to the *equivalence class* of all keys that result in $K_1$. Note this $K_{eqv}$ may or may not be equal to secret key $K_{seed}$. Although $K_{eqv}$ correctly determines $K_1$, the entire key sequence generated assuming $K_{eqv}$ was the initial seed of the key sequencer may not be the same as the actual key sequence $K_{seq}$. This is because the sequence of keys generated by the key sequencer design with separate initialization seeds (producing same $K_1$) will not be the same. Hence, finding a key $K_{eqv}$ is not sufficient and the attacker needs to find the exact key $K_{seed}$. We show that the probability of $K_{eqv}$ equals $K_{seed}$ is exponentially small in terms of size of the secret key $K_{seed}$, thus making SAT attack (or other Boolean logic based attacks) against HSCOA as impractical as a brute force attack.

*Theorem 1:* The probability of finding $K_{seed}$ using the above SAT attack approach is $1/2^{(N-1)n}$, where $N$ is the number and $n$ is the size of the cyclic shift registers.

*Proof:* Let $v_i^j$ denote the value of $i^{th}$ bit, $i \in \{1, n\}$, of the $j^{th}$, $j \in \{1, N\}$, cyclic shift register. Also, let $S_i$ denote the set of all $v_i^j$, i.e., $S_i = \{v_i^1, v_i^2, \ldots, v_i^N\}$. Without loss of generality let us assume $N$ is odd (similar arguments hold for $N$ being even). As per the key sequencer design, the $i^{th}$ key-bit of the first key $K1$ (denoted by $K_{1,i}$) is obtained by XORing all the elements of $S_i$. The value of $K_{1,i}$ is 0 whenever there are even number of ones in $S_i$, while $K_{1,i}$ is 1 when there are odd number of ones in $S_i$. Therefore, the number of possible combinations $Q_i^0$ of values of elements in $S_i$ which result in $K_{1,i} = 0$ can be expressed as follows:

$$Q_i^0 = \binom{N}{0} + \binom{N}{2} + \binom{N}{4} + \ldots + \binom{N}{N-1} \tag{1}$$

Similarly, the number of possible combinations $Q_i^1$ of values of elements in $S_i$ which result in $K_{1,i} = 1$ is as follows:

$$Q_i^1 = \binom{N}{1} + \binom{N}{3} + \binom{N}{5} + \ldots + \binom{N}{N} \tag{2}$$

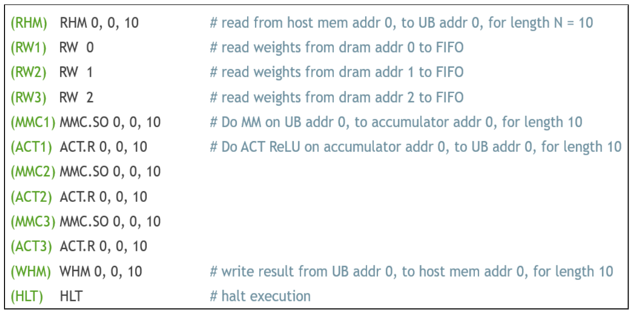
Since $\binom{N}{k} = \binom{N}{N-k}$, $k \in \{0, N\}$, from equations (1) and (2) we get $Q_i^0 = Q_i^1 = Q_i = 2^N/2 = 2^{N-1}$. As the cyclic shift registers are initialized with $K_{seed}$ and any two bits in a shift register are independent of eachother, any two bits of the key $K_1$ are also *independent* of each other. Therefore, the number of possible values of key $K_{eqv}$ which results in the same $K_1$ (of size $n$ bits) is $2^{(N-1)n}$. Essentially, the set of keys which result in the same $K_1$ has a size of $2^{(N-1)n}$. Only **one** of these keys is $K_{seed}$. Hence, the probability of finding $K_{seed}$ from $K_1$ using SAT attack based approach is $1/2^{(N-1)n}$. ■

From the above theorem, we see that the probability of finding the secret key $K_{seed}$ is exponentially small in terms of the key-size. Note that using the *shift LUT* we end up hiding the randomly generated *shift vectors* as well. This adds to the security guarantee even further as both $K_{seed}$ and the contents of *shift LUT* needs to be determined correctly to break the HSCOA framework.

*2) Resiliency to other attacks:* Our proposed HSCAO scheme is inherently secure to other types of attack on logic locking schemes, like removal attack [26] and bypass attack [23]. The underlying principle of such approaches is to either remove or bypass the protection circuitry to retrieve the netlist exhibiting correct functionality. Though the hardware-level deobfuscation logic of our proposed HSCAO scheme can be structurally identified, the removal/bypass of it won't neutralize the effect of software-level obfuscation performed by the proprietary accelerator SDK (root of trust). Also, as highlighted in section IV-A, the attacker doesn't have the capability to develop a substitute SDK using the netlist information.

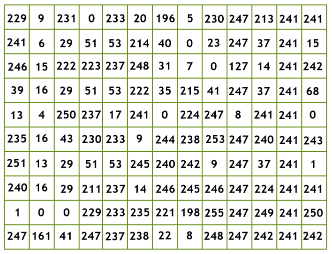
## V. Experimental Results

For our experiments, we used OpenTPU simulator [8] which is an open-source re-implementation of Google's TPU chip [15]. We considered a Tensorflow based implementation of Multi-layer Perceptron (MLP) regressor on the Boston Housing dataset [8]. In figure 4, we present the assembly-level program of such an application with detailed description of each instruction type. To augment the proposed HSCAO scheme with the OpenTPU simulator, we designed a *key sequencer* with $N$=9 cyclic shift registers (each $n = 16$ bits in length) to generate key sequence $K_{seq}$ initialized with a randomly selected $K_{seed}$



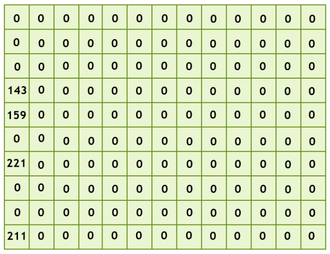
```
(RHM)   RHM 0, 0, 10        # read from host mem addr 0, to UB addr 0, for length N = 10
(RW1)   RW 0                # read weights from dram addr 0 to FIFO
(RW2)   RW 1                # read weights from dram addr 1 to FIFO
(RW3)   RW 2                # read weights from dram addr 2 to FIFO
(MMC1)  MMC.SO 0, 0, 10     # Do MM on UB addr 0, to accumulator addr 0, for length 10
(ACT1)  ACT.R 0, 0, 10      # Do ACT ReLU on accumulator addr 0, to UB addr 0, for length 10
(MMC2)  MMC.SO 0, 0, 10
(ACT2)  ACT.R 0, 0, 10
(MMC3)  MMC.SO 0, 0, 10
(ACT3)  ACT.R 0, 0, 10
(WHM)   WHM 0, 0, 10        # write result from UB addr 0, to host mem addr 0, for length 10
(HLT)   HLT                 # halt execution
```

Fig. 4: Assembly-level of MLP regression application



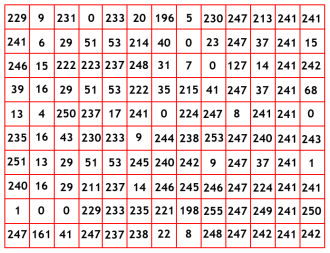
| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 229 | 9 | 231 | 0 | 233 | 20 | 196 | 5 | 230 | 247 | 213 | 241 | 241 |
| 241 | 6 | 29 | 51 | 53 | 214 | 40 | 0 | 23 | 247 | 37 | 241 | 15 |
| 246 | 15 | 222 | 223 | 237 | 248 | 31 | 7 | 0 | 127 | 14 | 241 | 242 |
| 39 | 16 | 29 | 51 | 53 | 222 | 35 | 215 | 41 | 247 | 37 | 241 | 68 |
| 13 | 4 | 250 | 237 | 17 | 241 | 0 | 224 | 247 | 8 | 241 | 241 | 0 |
| 235 | 16 | 43 | 230 | 233 | 9 | 244 | 238 | 253 | 247 | 240 | 241 | 243 |
| 251 | 13 | 29 | 51 | 53 | 245 | 240 | 242 | 9 | 247 | 37 | 241 | 1 |
| 240 | 16 | 29 | 211 | 237 | 14 | 246 | 245 | 246 | 247 | 224 | 241 | 241 |
| 1 | 0 | 0 | 229 | 233 | 235 | 221 | 198 | 255 | 247 | 249 | 241 | 250 |
| 247 | 161 | 41 | 247 | 237 | 238 | 22 | 8 | 248 | 247 | 242 | 241 | 242 |

(a) Initial HM



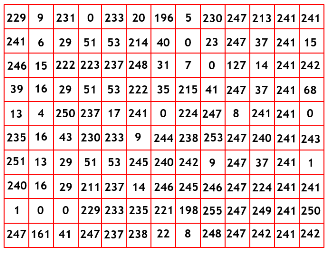
| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 143 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 159 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 221 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 211 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

(b) Final HM with $K_{seed}$



| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 229 | 9 | 231 | 0 | 233 | 20 | 196 | 5 | 230 | 247 | 213 | 241 | 241 |
| 241 | 6 | 29 | 51 | 53 | 214 | 40 | 0 | 23 | 247 | 37 | 241 | 15 |
| 246 | 15 | 222 | 223 | 237 | 248 | 31 | 7 | 0 | 127 | 14 | 241 | 242 |
| 39 | 16 | 29 | 51 | 53 | 222 | 35 | 215 | 41 | 247 | 37 | 241 | 68 |
| 13 | 4 | 250 | 237 | 17 | 241 | 0 | 224 | 247 | 8 | 241 | 241 | 0 |
| 235 | 16 | 43 | 230 | 233 | 9 | 244 | 238 | 253 | 247 | 240 | 241 | 243 |
| 251 | 13 | 29 | 51 | 53 | 245 | 240 | 242 | 9 | 247 | 37 | 241 | 1 |
| 240 | 16 | 29 | 211 | 237 | 14 | 246 | 245 | 246 | 247 | 224 | 241 | 241 |
| 1 | 0 | 0 | 229 | 233 | 235 | 221 | 198 | 255 | 247 | 249 | 241 | 250 |
| 247 | 161 | 41 | 247 | 237 | 238 | 22 | 8 | 248 | 247 | 242 | 241 | 242 |

(c) Final HM with $K_{eqv}^1$ (cause: exception raised)



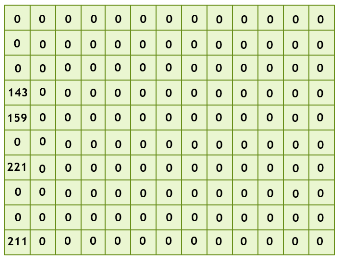
| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 229 | 9 | 231 | 0 | 233 | 20 | 196 | 5 | 230 | 247 | 213 | 241 | 241 |
| 241 | 6 | 29 | 51 | 53 | 214 | 40 | 0 | 23 | 247 | 37 | 241 | 15 |
| 246 | 15 | 222 | 223 | 237 | 248 | 31 | 7 | 0 | 127 | 14 | 241 | 242 |
| 39 | 16 | 29 | 51 | 53 | 222 | 35 | 215 | 41 | 247 | 37 | 241 | 68 |
| 13 | 4 | 250 | 237 | 17 | 241 | 0 | 224 | 247 | 8 | 241 | 241 | 0 |
| 235 | 16 | 43 | 230 | 233 | 9 | 244 | 238 | 253 | 247 | 240 | 241 | 243 |
| 251 | 13 | 29 | 51 | 53 | 245 | 240 | 242 | 9 | 247 | 37 | 241 | 1 |
| 240 | 16 | 29 | 211 | 237 | 14 | 246 | 245 | 246 | 247 | 224 | 241 | 241 |
| 1 | 0 | 0 | 229 | 233 | 235 | 221 | 198 | 255 | 247 | 249 | 241 | 250 |
| 247 | 161 | 41 | 247 | 237 | 238 | 22 | 8 | 248 | 247 | 242 | 241 | 242 |

(d) Final HM with $K_{eqv}^2$ (cause: early termination)

Fig. 5: Error impact on host memory (HM) due to wrong instruction deobfuscation for different equivalent keys

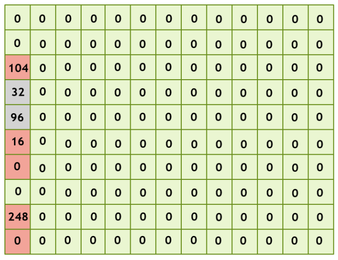
of size 144 bits. Each key belonging to $K_{seq}$ was bit-wise XORed with the opcode bits (as specified in OpenTPU ISA) for performing obfuscation/ deobfuscation of an instruction in the software/ hardware counterparts of HSCAO framework. In figures 5a and 5b, we present the initial host memory content and the final host memory content (after running the application) of an unlocked TPU-like chip (activated using the correct key $K_{seed}$). Each small green square contains the correct value of a memory location.

To study the application-level error impact due to the use of an equivalent first round key for unlocking the TPU-like chip, we used two such keys $K_{eqv}^1$ and $K_{eqv}^2$ as initial seeds and ran the regression application (see figure 4). Note that for performing these experiments, we considered that the *shift LUT* is configured correctly. But in practice, the attacker will face additional challenge to determine the *shift LUT* contents. In figures 5c and 5d, we present the final host memory contents for using $K_{eqv}^1$ and $K_{eqv}^2$ respectively. With $K_{eqv}^1$, the application terminated with an exception that a new matrix multiply (MMC) type instruction cannot be dispatched while a previous instruction is still being issued, thus resulting in no memory update (as denoted by red squares). Similarly, with $K_{eqv}^2$ also there was no memory update as well due to early application termination (no exception raised). We observed that the reason behind this early termination being one of the keys (in the sequence generated by $K_{eqv}^2$) when bit-wise XORed with the corresponding obfuscated instruction opcode incorrectly resulted in the opcode for exit/halt condition (HLT). These results highlight that use of such equivalent keys fail to



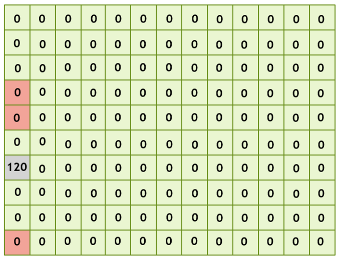
| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 143 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 159 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 221 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 211 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

(a) No error (with $K_{seed}$)



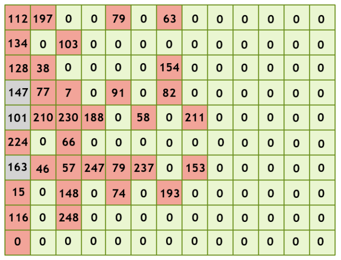
| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 104 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 32 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 96 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 248 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

(b) Locked RW1



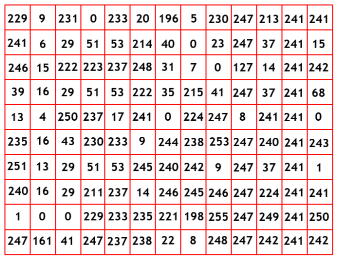
| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 120 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

(c) Locked RW2



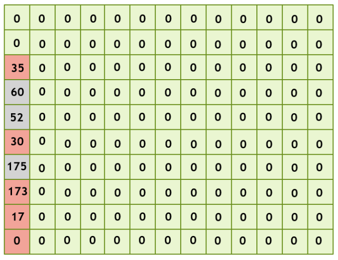
| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 112 | 197 | 0 | 0 | 79 | 0 | 63 | 0 | 0 | 0 | 0 | 0 | 0 |
| 134 | 0 | 103 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 128 | 38 | 0 | 0 | 0 | 0 | 154 | 0 | 0 | 0 | 0 | 0 | 0 |
| 147 | 77 | 7 | 0 | 91 | 0 | 82 | 0 | 0 | 0 | 0 | 0 | 0 |
| 101 | 210 | 230 | 188 | 0 | 58 | 0 | 211 | 0 | 0 | 0 | 0 | 0 |
| 224 | 0 | 66 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 163 | 46 | 57 | 247 | 79 | 237 | 0 | 153 | 0 | 0 | 0 | 0 | 0 |
| 15 | 0 | 148 | 0 | 74 | 0 | 193 | 0 | 0 | 0 | 0 | 0 | 0 |
| 116 | 0 | 248 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

(d) Locked RW3



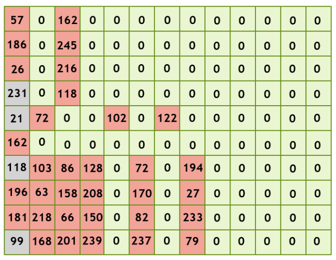
| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 229 | 9 | 231 | 0 | 233 | 20 | 196 | 5 | 230 | 247 | 213 | 241 | 241 |
| 241 | 6 | 29 | 51 | 53 | 214 | 40 | 0 | 23 | 247 | 37 | 241 | 15 |
| 246 | 15 | 222 | 223 | 237 | 248 | 31 | 7 | 0 | 127 | 14 | 241 | 242 |
| 39 | 16 | 29 | 51 | 53 | 222 | 35 | 215 | 41 | 247 | 37 | 241 | 68 |
| 13 | 4 | 250 | 237 | 17 | 241 | 0 | 224 | 247 | 8 | 241 | 241 | 0 |
| 235 | 16 | 43 | 230 | 233 | 9 | 244 | 238 | 253 | 247 | 240 | 241 | 243 |
| 251 | 13 | 29 | 51 | 53 | 245 | 240 | 242 | 9 | 247 | 37 | 241 | 1 |
| 240 | 16 | 29 | 211 | 237 | 14 | 246 | 245 | 246 | 247 | 224 | 241 | 241 |
| 1 | 0 | 0 | 229 | 233 | 235 | 221 | 198 | 255 | 247 | 249 | 241 | 250 |
| 247 | 161 | 41 | 247 | 237 | 238 | 22 | 8 | 248 | 247 | 242 | 241 | 242 |

(e) Locked MMC1/ WHM



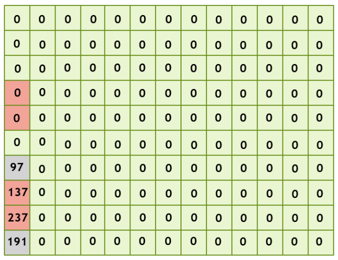
| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 35 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 60 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 52 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 30 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 175 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 173 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 17 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

(f) Locked ACT1



| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 57 | 0 | 162 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 186 | 0 | 245 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 26 | 0 | 216 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 231 | 0 | 118 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 21 | 72 | 0 | 0 | 102 | 0 | 122 | 0 | 0 | 0 | 0 | 0 | 0 |
| 162 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 118 | 103 | 86 | 128 | 0 | 72 | 0 | 194 | 0 | 0 | 0 | 0 | 0 |
| 196 | 63 | 158 | 208 | 0 | 170 | 0 | 27 | 0 | 0 | 0 | 0 | 0 |
| 181 | 218 | 66 | 150 | 0 | 82 | 0 | 233 | 0 | 0 | 0 | 0 | 0 |
| 99 | 168 | 201 | 239 | 0 | 237 | 0 | 79 | 0 | 0 | 0 | 0 | 0 |

(g) Locked MMC2/MMC3/ACT3



| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 97 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 137 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 237 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 191 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

(h) Locked ACT2

🟩 correct data update 🟥 wrong data update ▨ corrupted data □ initial content □ no update

Fig. 6: Error impact on host memory (HM) due to single locked instruction

unlock the accelerator obfuscated using HSCOA framework.

The above approach of obfuscating every instruction, though effective, may incur significant delay for running applications due to updates of cyclic shift registers (depending on *shift vector* contents) per instruction. Therefore, we **locked only a single instruction** in the entire application assembly (apart from the first RHM instruction which is locked by run-time independent key $K_1$) and observed the resulting *corruption* in final memory contents. The outcomes of such experimental runs are presented in figure 6, where each subfigure shows the final host memory content for a particular locked instruction in the regression application. The states of memory locations are classified into 4 categories: (i) correct data update (ii) wrong data update which signifies data update in a faulty memory location (iii) corrupted data update where the memory update location is correct but the data content is wrong and (iv) no data update from initial data content. As observed from the figures, even locking a single instruction leads to significant errors in the application outcomes, highlighting the strength of our proposed HSCAO countermeasure to protect the IP of an accelerator chip design.

## VI. Conclusion

In this paper, we first show the ineffectiveness of state-of-the-art locking scheme to protect the IP of hardware accelerators. Subsequently, we propose a hardware-software co-design based obfuscation approach to render an unactivated accelerator chip functionally useless. Our proposed HSCOA scheme uses proprietary SDK as the root of trust for generating locked program binary which is subsequently deobfuscated in the hardware. The experimental results obtained by running a regression application on OpenTPU simulator demonstrate the effectiveness of such an obfuscation framework.

## References

[1] AI chip. https://www.gyrfalcontech.ai/solutions/2801s/.

[2] Codix-Bk3. https://www.codasip.com/risc-v-processors.

[3] DexGuard. https://www.guardsquare.com/en/products/dexguard.

[4] Google Edge TPU. https://cloud.google.com/edge-tpu/.

[5] Intel ISA. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.

[6] NVIDIA CUDA. http://www.nvidia.com/cuda/.

[7] Open ISA based processor. https://www.codasip.com/2017/08/08/does-risc-v-mean-open-source-processors/.

[8] OpenTPU. https://github.com/UCSBarchlab/OpenTPU.

[9] K. Asanović and D. A. Patterson. Instruction sets should be free: The case for risc-v. *EECS, UCB, Tech. Rep. UCB/EECS-2014-146*, 2014.

[10] A. Chakraborty et al. GPU obfuscation: attack and defense strategies. In *55th Annual Design Automation Conference*, page 122. ACM, 2018.

[11] A. Chakraborty et al. Timingsat: timing profile embedded sat attack. In *International Conference on Computer-Aided Design*. ACM, 2018.

[12] T. Chen et al. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *Sigplan Notices*, 49(4):269–284, 2014.

[13] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, The University of Auckland, 1997.

[14] M. El Massad et al. Integrated circuit (ic) decamouflaging: Reverse engineering camouflaged ics within minutes. In *NDSS*, 2015.

[15] N. P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 1–12. IEEE, 2017.

[16] J. Rajendran et al. Security analysis of logic obfuscation. In *Proceedings of 49th Annual Design Automation Conference*, pages 83–89, 2012.

[17] J. Rajendran et al. Fault analysis-based logic encryption. *Computers, IEEE Transactions on*, 64(2):410–424, 2015.

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

[19] K. Shamsi et al. Appsat: Approximately deobfuscating integrated circuits. In *IEEE Symp. Hardware-Oriented Security and Trust*, 2017.

[20] P. Subramanyan et al. Evaluating the security of logic encryption algorithms. In *Hardware Oriented Security and Trust, 2015*, 2015.

[21] Y. Xie and A. Srivastava. Mitigating sat attack on logic locking. In *CHES 2016*, pages 127–146. Springer, 2016.

[22] Y. Xie et al. Delay locking: Security enhancement of logic locking against ic counterfeiting and overproduction. In *54th Annual Design Automation Conference*, 2017.

[23] X. Xu et al. 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.

[24] M. Yasin er al. Sarlock: Sat attack resistant logic locking. In *Hardware Oriented Security and Trust (HOST), 2016*, pages 236–241. IEEE, 2016.

[25] M. Yasin et al. Provably-secure logic locking: From theory to practice. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1601–1618. ACM, 2017.

[26] M. Yasin et al. Removal attacks on logic locking and camouflaging techniques. *IEEE Transactions on Emerging Topics in Computing*, 2017.