Securator: A Fast and Secure Neural Processing Unit

Nivedita Shrivastava
Electrical Engineering Department
Indian Institute of Technology, Delhi, India
Email: nivedita.shrivastava@ee.iitd.ac.in

Smruti Ranjan Sarangi
Electrical Engineering Department
Indian Institute of Technology, Delhi, India
Email: srsarangi@cse.iitd.ac.in

***Abstract*—Securing deep neural networks (DNNs) is a problem of significant interest since an ML model incorporates high-quality intellectual property, features of data sets painstakingly collated by mechanical turks, and novel methods of training on large cluster computers. Sadly, attacks to extract model parameters are on the rise, and thus designers are being forced to create architectures for securing such models. State-of-the-art proposals in this field take the deterministic memory access patterns of such networks into cognizance (albeit partially), group a set of memory blocks into a *tile*, and maintain state at the level of tiles (to reduce storage space). For providing integrity guarantees (tamper avoidance), they don't propose any significant optimizations, and still maintain block-level state.**

We observe that it is possible to exploit the deterministic memory access patterns of DNNs even further, and maintain state information for only the current tile and current layer, which may comprise a large number of tiles. This reduces the storage space, reduces the number of memory accesses, increases performance, and simplifies the design without sacrificing any security guarantees. The key techniques in our proposed accelerator architecture, Securator, are to encode memory access patterns to create a small HW-based tile version number generator for a given layer, and to store layer-level MACs. We completely eliminate the need for having a MAC cache and a tile version number store (as used in related work). We show that using intelligently-designed mathematical operations, these structures are not required. By reducing such overheads, we show a speedup of 20.56% over the closest competing work.

I. Introduction

The AI hardware (Neural Processing Unit (NPU)) market was valued at 8 billion USD in 2020 and is expected to grow to 84 billion USD by 2028 [1] (CAGR of 34.15%). Hardware-based AI chips are expected to play a major role in telecommunications [8], mobile vision [39], edge computing [7], augmented/virtual reality [2], health care [17], and autonomous driving [47]. Similar to code for general-purpose processors, ML models incorporate a lot of high-value intellectual property (IP) that takes a lot of time and effort to collate and develop. For example, even for a mid-size AI project, just collecting the raw data using mechanical turks takes upwards of $100K USD [32]; the know-how for the model and training methodology can be worth millions of dollars, and finally it may take many weeks to finally train the model using large cluster computers. Along with these conventional arguments, many a time we don't realize that even getting access to the training data can be quite challenging with numerous legal hurdles. Therefore, protecting an ML model is of paramount importance, which sadly also makes it an attractive target for hackers.

There are three broad approaches for protecting models, as shown in Figure 1. All of them rely on secure CPUs as the baseline technology. To secure CPUs, we rely on Trusted Execution Environments (TEEs) such as Intel SGX [5], AMD SEV [18], and ARM Trustzone [29]. In all these TEEs, the CPU is assumed to be secure; it is within the *TCB* (Trusted Computing Base). The main contribution of the most elaborate scheme, Intel SGX, is in protecting the off-chip memory and providing confidentiality, integrity, and freshness. The key insight is ensuring that every time a block is written to main memory, it is encrypted with a different key. Since, storing a key for every memory block is too expensive, a more efficient mechanism is used based on AES counter-mode encryption. Every page is associated with a major counter, and every block uses a minor counter (a combination of both guides the encryption/decryption). The counters themselves need to be protected; this is achieved using a Merkle tree, where the root of the tree is guaranteed to be in the TCB. Let us refer to this version of SGX as *SGX-Client*. Because of the Merkle tree and associated overheads of maintaining counters, the maximum size of the protected memory region is limited to 128-256 MB (same problem with ARM TrustZone). Most ML models and datasets as of today are much larger. As a result, *SGX-Client* is not the best choice for them.

Keeping in mind these issues, Intel recently discontinued support for *SGX-Client* in its 11^{th} and 12^{th} generation processors. It replaced it with another version of SGX (referred to as *SGX-Server*) that simply encrypts memory and foregoes the integrity and freshness guarantees [33] (on the lines of AMD SEV). It can provide up to 512 GB of encrypted memory. We should bear in mind that *SGX-Server* is far weaker than *SGX-Client* in terms of the security guarantees that are provided, namely integrity and freshness.

ML architecture designers have traditionally opted to use versions of *SGX-Client* to secure their systems (refer to Figure 1). Several early approaches either proposed optimizations to *SGX-Client* [5], [33] to reduce its overhead or partition an ML algorithm into a secure portion and unsecure portion that ran on fast hardware such as GPUs [40]. However, these approaches have been superseded by a newer family of approaches that leverage the stable data communication patterns of ML workloads to design bespoke NoCs. Two custom accelerators stand out in this space: TNPU [22] and GuardNN [15]. Their basic ideas are similar: group a set of contiguous memory blocks into *tiles* and provide freshness guarantees at the level of a tile. Both use a dedicated software module running on the host CPU for managing version numbers (VNs); they are used to encrypt data as well as ensure its freshness. The major difference is that TNPU stores the VNs in an on-chip cache and GuardNN relies on a CPU program to generate all the VNs. The program needs to store all the VNs in use in secure memory, and securely communicate them to the accelerator (a difficult problem in itself). Both still perform integrity checking at the level of individual memory blocks.

Our scheme, Securator, improves upon these ideas and proposes a natural extension, where we perform freshness and integrity checks at the layer level. Given that no data is stored and no checking is done at the block-level (like previous schemes), there is an associated performance gain (20.56%). Additionally, there is no need to run a VN-generation module on the host CPU using a TEE. To realize this, we thoroughly characterize the memory access patterns of different kinds of ML accelerators, efficiently encode them, and pass the encoding to a small hardware circuit that automatically generates all VNs at runtime (without external intervention). This eliminates the need for storing and managing VNs in on-chip caches or main memory regions. Furthermore, we leverage the insight that the only consumer for the output(s) of a layer are a few layers that it is connected to – random access of output data is not required because these consumer layers access the data in a structured fashion. Hence, computing and storing MACs at the layer-level is good enough as long as the next layer accesses all the data (in any order), which is often the case.

To summarize, **the main contributions in this paper** are as follows: ① Characterization of traffic in CNNs and popular data pre/post processing algorithms, ② A method to succinctly encode the traffic pattern, ③ A method to generate VNs on the fly using such patterns, ④ A technique to perform integrity checking at the level of layers, and ⑤ A detailed experimental analysis of Securator that shows a 20.56% speedup over the nearest competing work. ⑥ Given that the overheads are low, we additionally assess the benefits of interspersing the execution with the running of a dummy network (for the purpose of adding noise) or widening each layer. This helps reduce the possibility of model extraction attacks (MEA) by utilizing timing or address-based side channels [37] substantially.

The paper is organized as follows. Section II presents the background of TEEs and basic convolution operations in neural networks. Section III presents the threat model and the security guarantees provided by Securator. We characterize the workloads in Section IV, present an analytical model for automatic version number generation in Section V, and present the architecture of Securator in Section VI. Section VII presents an extensive security analysis of the proposed design and finally, VIII reports the experimental results, Section IX presents the related work and we conclude in Section X.

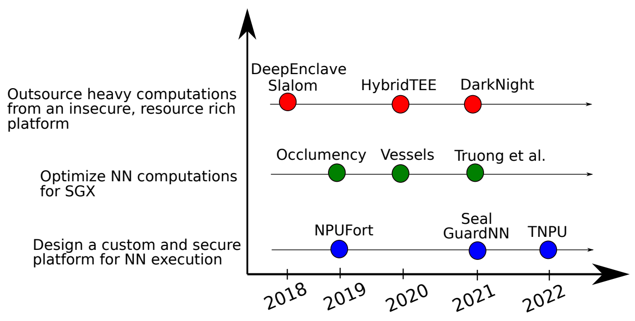

Outsource heavy computations from an insecure, resource rich platform
Optimize NN computations for SGX
Design a custom and secure platform for NN execution
DeepEnclave
Slalom
HybridTEE
DarkNight
Occlumency
Vessels
Truong et al.
NPUFort
Seal
GuardNN
TNPU
2018
2019
2020
2021
2022

Fig. 1: A timeline depicting a pragmatic shift from outsourcing execution from an unsecure compute-rich platform to designing an optimized and secure custom TEE for accelerating ML workloads.

II. Background

A TEE provides an isolated execution environment for security-sensitive applications, ensuring data confidentiality and integrity. The security community has relied on them for performing secure data computations even in an insecure environment.

A. Intel SGX

SGX is a TEE provided by Intel to enable remote code execution in a *secure enclave* by isolating security-sensitive code and data from other applications.

1) SGX-Client (10^{th} Gen Intel CPUs): *SGX-Client* provides a protected memory of limited size, 256 MB, which is accessible from within the secure enclave only. If the data size exceeds this limit, there will be a significant overhead due to pages' eviction and encryption.

Confidentiality With Memory Encryption: SGX uses *AES-CTR* (counter mode) for performing encryption. A data block P is XORed with a one-time pad, which is generated by encrypting the address of the data block PA and the counter value C (major+minor counter) with a secret key. The secret key is a concatenation of the enclave ID E, PUF P, and a random number generated at boot time R. The entire process can be written as: $P \oplus AES_{(E||P||R)}(PA||C)$, where $||$ represents a concatenation operator and $\oplus$ represents a XOR operator. Each block in a page is assigned a 6-bit minor counter, which is concatenated with a 64-bit page-level major counter. The counter value is incremented when a modified cache block is evicted from the last-level cache (LLC) [35]. The major counter is incremented once a minor counter overflows. These steps ensure that the combined counter value C is never reused, thus avoiding a *replay attack*. The counter values are stored inside a secure cache known as a *counter-cache*. If an entry is not present in the cache, it is fetched from main memory. A Merkle tree guarantees the integrity of the counters stored in DRAM.

Integrity and Authenticity Verification using Message Authentication Codes (MACs): MACs are encrypted hashes, that are used to ensure that an adversary does not alter the data stored in an untrusted location. A unique MAC is generated for each data block and the corresponding counter value. When a data block is fetched from main memory, its MAC is also fetched and verified. As the data encryption takes into account the block address, an attacker cannot swap the $\langle data, MAC \rangle$ pair with another pair. Sadly, these security guarantees come with significant overheads, and thus the size of protected memory is limited to 256 MB.

2) SGX-Server (11^{th} and 12^{th} Gen Intel CPUs): Intel recently launched a scalable and efficient TEE [33] for Intel 3^{rd} Gen scalable Xeon servers with an additional feature, *Total Memory Encryption (TME)*, to overcome the size limitations of SGX-Client. This feature provides a secure memory size of 512 GB. Sadly, Intel compromised hardware-based integrity and replay protection in order to securely encrypt the entire memory (as mentioned in their documentation [22], [33]). SGX-Server uses AES-XTS for performing total memory encryption, which does not rely on per-block counters. With the elimination of counters and the Merkle tree, the need for data caching and tree traversal are also eliminated, thus reducing the associated overheads.

B. Convolution

A convolution operation [25] is the heart of a convolutional neural network (CNN). The single-image version has a basic 6-loop structure. We iterate through the input and output feature maps (*ifmap* and *ofmap*, resp.), and compute the convolutions. For the ease of explanation, we assume that they have the same size (both are referred to as an *fmap*) and are 2D matrices (each element is a *pixel*).

```
for(k=0; k<K ; k++){        // K: #output fmaps
 for(c=0; c<C ; c++){       // C: #input fmaps
  for(h=0; h<H ; h++){      // H: #rows in an fmap
   for(w=0; w<W ; w++){ // W: #cols in an fmap
    for(r=0; r<R ; r++){   // R: #rows in a  filter
     for(s=0; s<S ; s++){ // S: #cols in a  filter
        ofmap[k][h][w]+=ifmap[c][h+r][w+s] *
              weights[k][c][r][s];
}}}}}}
```

Listing 1: A basic convolution operation

The loop order in Listing 1 can be written as $k \triangleright c \triangleright h \triangleright w \triangleright r \triangleright s$, where, the operator $\triangleright$ shows the order of the nesting. Due to the restricted capacity of on-chip buffers, we often group pixels into *tiles*. In this case, an example notation for a tiled execution where the *fmaps* (*channels*) are tiled (rows and columns grouped) will be of the form, $k \triangleright c \triangleright h_T \triangleright w_T \triangleright r \triangleright s \triangleright h \triangleright w$; the subscript $_T$ indicates the tile number, and the iterator without a subscript retains its previous meaning (pointing to a single element). Figure 2 shows a generic example, where k, c, h, and w are tiled. We will follow a consistent terminology for all iterators, e.g., C is the number of *ifmaps* (input channels), C_T is the size of a channel tile, c_T is the iterator for a channel tile, and c is the iterator of a channel (see Table I).

C. TNPU and GuardNN

We shall compare Securator with two state-of-the-art proposals: GuardNN [15] and TNPU [22]. A brief description follows (see Section IX for more details).

In GuardNN, every tile is associated with a version number (VN), which is incremented on every memory write. The

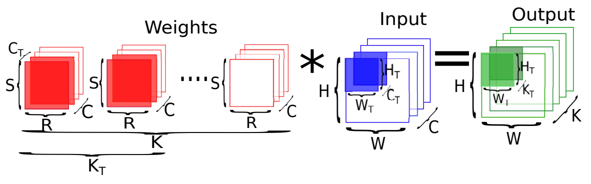


Fig. 2: Graphical representation of a tiled convolution operation. The *ifmaps*, *ofmaps*, input and output channels are tiled.

VNs are managed by the scheduler that runs on the host CPU. MACs are generated and maintained per memory block. GuardNN advocates for a larger block size (512 B); however, that unnecessarily constrains the subsequent layer to read data in that order, which we found to be impractical for modern CNNs where dataflow patterns are different for each layer. TNPU on the other hand, maintains VNs in a Tensor Table that is stored in the host CPU's secure memory. It is protected by an integrity tree. MACs are maintained per block, and are stored in an on-chip MAC cache.

III. System Design and Threat Model

We use the same high-level system design and threat model as previous works [15], [22].

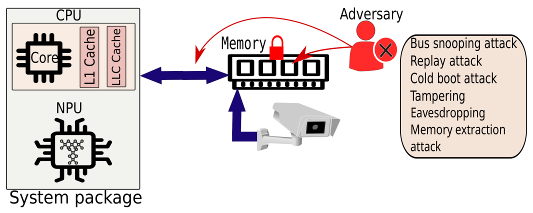


Fig. 3: An overview of the system design and threat model (note that it is not necessary that the NPU and CPU are in the same package)

A high-level overview of the system is shown in Figure 3 along with the possible attacks that can be mounted. The CPU and NPU may or may not be in the same package; all that we need is a way for the CPU to populate DRAM memory (accessible to the NPU) with the input and the NN model. They could share DIMMs or have any other method of communication. The CPU can optionally run a scheduler (possibly on a TEE) to coordinate the actions of the NPU, or the NPU could completely take over after the CPU has sent it a few initialization instructions. Our design is unaffected by this choice. Insofar as the NPU is concerned, it needs to securely run an ML model, layer by layer. We only focus on inferencing in this paper (like [22]).

The CPU, NPU, and caches are within the TCB. A hacker can however target the main memory and the NPU-memory bus. A hacker in this case would include the OS, hypervisor, malicious program, or even a person who has physical access to the main memory or memory bus. She can try to read the data (eavesdropping), tamper with the data (integrity attack), replace the contents of memory addresses with old data (replay attack), masquerade as a different entity (authentication attack), read the memory address sequence from the memory bus and try to figure out the model parameters (MEA). Securator protects against all types of aforementioned attacks.

IV. Characterization of DNN Workloads in a Secure Environment

Simulation configuration (similar to [22])			
Parameter	*Value*	*Parameter*	*Value*
PE array	32 × 32	Counter cache (secure NPU & TNPU)	4 KB
Global buffer	240 KB	MAC cache (secure NPU & TNPU)	8 KB
Frequency	2.75 GHz	Write mode	Write back
Dual-channel DRAM	DDR 4, 100 cyc (lat)	Block Size	64 B

List of benchmarks (similar to [15], [40])		
Workload	*Layers*	*Parameters*
MobileNet [14]	23	4.2 million
ResNet [13]	18	11 million
AlexNet [21]	13	62 million
VGG16 [38]	24	138 million
VGG19 [38]	19	143 million

Terminology used	
Term	Meaning
H	# rows
H_T	# row tiles
h_T	row tile iterator
h	row iterator

TABLE I: NPU configuration, list of benchmarks, and the terminology used in the paper

A. Setup and Benchmarks

We characterize the behavior of popular benchmarks on an in-house cycle accurate CNN simulator that has been rigorously validated with SCALE-Sim [34] and native hardware. It relies on a systolic array architecture to perform convolutions. We relied on the Timeloop [28] tool to provide the most optimal dataflow pattern. We show the configuration of the simulated system in Table I. We simulated a vanilla, *unsecure* accelerator version as the *baseline*. The baseline architecture provides no assurances of security. It is comprised of a systolic array of 32 × 32 PEs arranged in a grid-like fashion. PEs comprise a computational unit and a register file. The neural networks are regular and deterministic, so rather than caches, we use 240 kB of scratchpad memory as the on-chip global buffer (similar to previous work [22]). When PEs are performing computations, the system employs a double buffering technique in which data is loaded into the on-chip buffer from memory. The workloads comprise popular neural network benchmarks as shown in Table I. *Layers* represents the total number of layers and *Parameters* represents the total number of tunable model parameters present in a specific benchmark. Additionally, we simulate a *secure* configuration that is constructed upon the baseline with all the security guarantees similar to SGX-Client (see Section II-A1). The MACs and counter values are saved in the 8 KB MAC cache and 4 KB counter cache, respectively (values taken from [22]). All the models fit within the DRAM. Finally, note that **performance** is defined as the reciprocal of the simulated execution time (appropriately normalized).

1) Performance Insights: Figure 4 shows the performance results. The secure configuration is 45% slower than the baseline, TNPU is 27% slower, and GuardNN is 75 % slower. Clearly, reducing the size of secure memory helps, and having

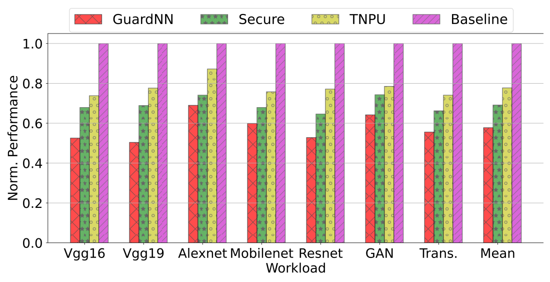


Fig. 4: Performance comparison

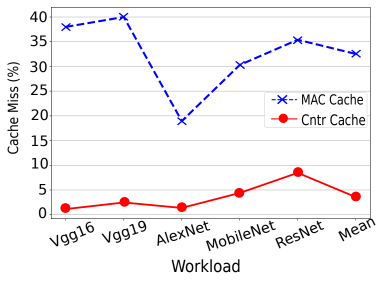


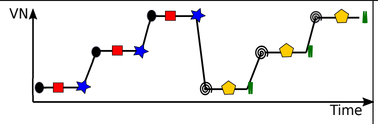
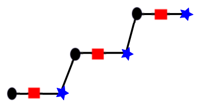
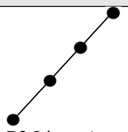
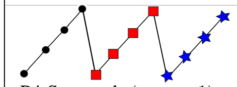
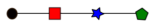
Fig. 5: Cache miss rates for the MAC cache and counter cache, respectively.

a MAC cache also helps (as opposed to not having one like in GuardNN). Figure 5 shows the miss rates for the MAC cache and counter cache in the secure configuration. We observe a high hit rate for the counter cache and a relatively lower hit rate for the MAC cache. Each 64-byte data block can store 16 four-byte pixels (element in a feature map). Each page of 64 blocks can store 64 × 16 pixels implying that a counter cache entry can keep track of $64 \times 16 = 1024$ pixels. For uniform streaming data, after a cache miss, we may observe a cache hit for the next 1024 consecutive pixels. On the other hand, a 64-byte line can store only 8 MACs (each 8 bytes). Since each MAC protects a block of 16 pixels (64 bytes), this means that a MAC cache block can track $16 \times 8 = 128$ pixels, which is 8× less than a counter cache block. This means that the MAC cache miss rate will be much higher; however, on the flip side its miss penalty will be much lower – primarily a DRAM memory read as opposed to traversing a Merkle tree for counters (DRAM+cache). In any case, streaming data such as pixels in DNNs have poor temporal locality and the low absolute values of miss rates attest that.

Conclusion: MAC caches have a very poor hit rate. Moreover, frequently accessing secure memory to read VNs and MACs has a high overhead. Hence, we should try to avoid both.

V. Analytical Characterization of Patterns in ML Workloads

Imagine a pair of observers sitting at the global buffer (GB) of the NPU and recording the VN of every tile that is read or written. Let us refer to them as the *read-observer* and *write-observer*, respectively. For a layer, they will see a sequence of VNs. Prior works have used a tabular data structure to store the latest VN for each tile (just before it is written to

Expression	**Possible patterns**	**Expression**	**Possible patterns**	
$[1^{\alpha_K}, 2^{\alpha_K} .. {\alpha_C}^{\alpha_K}]^{\alpha_{HW}}$	 P1:Multi-step	$1^{\alpha_K \alpha_{HW}}, 2^{\alpha_K \alpha_{HW}} .. \alpha_C^{\alpha_K \alpha_{HW}}$	 P2:Step	 P3:Linear($\alpha_K \alpha_{HW} = 1$)
	 P4:Sawtooth ($\alpha_K = 1$)	$1^{\alpha_K}, 2^{\alpha_K} .. {\alpha_C}^{\alpha_K}$		 P5:Line($\alpha_C = 1$)
Pattern Diagrams: The colored dots, squares, and stars, etc., represent accesses for a group of tiles (fetched and processed at one go) The x-axis represents time; the y-axis represents the VN. Axes not shown in other figures to enhance readability.				

Tiling style	**Input reuse**		**Output reuse**	
	Loop order	**Rd/Wr pattern**	**Loop order**	**Rd/Wr pattern**
Tile movement along the channel				
❶ Partial channel [4], [44] ❷ Partial-multi-channel [44]	$h_T \triangleright w_T \triangleright c \triangleright k_T$ $h_T \triangleright w_T \triangleright c_T \triangleright k_T$	WP: $[1^{\alpha_K}, 2^{\alpha_K} \ldots {\alpha_C}^{\alpha_K}]^{\alpha_{HW}}$ RP: $[1^{\alpha_K}, 2^{\alpha_K} \ldots [\alpha_C - 1]^{\alpha_K}]^{\alpha_{HW}}$ $Patterns$: P1,P2,P3,P4,P5	$h_T \triangleright w_T \triangleright k_T \triangleright c$ $h_T \triangleright w_T \triangleright k_T \triangleright c_T$	WP: $1^{\alpha_K \alpha_{HW}}$ RP: – $Patterns$: P5
Tile movement along the width/height				
❸ Partial channel [30] ❹ Partial-multi-channel [44]	$c \triangleright h_T \triangleright w_T \triangleright k_T$ $c_T \triangleright h_T \triangleright w_T \triangleright k_T$	WP: $1^{\alpha_K \alpha_{HW}}, 2^{\alpha_K \alpha_{HW}} \ldots \alpha_C^{\alpha_K \alpha_{HW}}$ RP: $1^{\alpha_K \alpha_{HW}}, 2^{\alpha_K \alpha_{HW}} .. [\alpha_C - 1]^{\alpha_K \alpha_{HW}}$ $Patterns$: P2,P3,P5	– – –	– – –
❺ Channel-wise [4]	$c \triangleright k_T$; $c_T \triangleright k_T$;	WP: $1^{\alpha_K}, 2^{\alpha_K} .. {\alpha_C}^{\alpha_K}$ RP: $1^{\alpha_K}, 2^{\alpha_K} .. [\alpha_C - 1]^{\alpha_K}$ $Patterns$: P2,P3,P5	$k_T \triangleright c$ $k_T \triangleright c_T$	WP: 1^{α_K} RP: – $Patterns$: P5
❻ Full-channel [42]	$h_T \triangleright w_T \triangleright k_T$	WP: $1^{\alpha_K \alpha_{HW}}$ RP: – $Patterns$:P5	$h_T \triangleright w_T \triangleright k_T$	WP: $1^{\alpha_K \alpha_{HW}}$ RP: – $Patterns$:P5

TABLE II: Pattern table for convolution: various possibilities for scheduling the input tiles for input reuse and output reuse. $\alpha_K = \frac{K}{K_T}; \alpha_C = \frac{C}{C_T}; \alpha_{HW} = \frac{H.W}{H_T.W_T}$; RP $\rightarrow$ VN read pattern, WP $\rightarrow$ VN write pattern, – refers to empty or not applicable. Assumption: As per the loop order, we fetch a *group of tiles* that is read, processed, and written in one go. They have the same VN, which is generated by the aforementioned WP/RP patterns.

main memory). The same VN needs to be fetched from the table when the tile is read the next time. We argue in this section that using a conventional table is an overkill. An ML application has a very well-defined behavior, and the sequence of VNs seen by the observers can be very nicely characterized (depending upon the type of data reuse and fmap). We can thus supplant the multi-KB table stored in the host CPU's secure memory with a simple mathematical formula processor that can generate VNs at runtime. This will be a part of the NPU, and the storage requirement will be limited to a few registers.

A. *Convolution Layer*

In convolution, there are three possible types of data reuse: input reuse, weight reuse, and output reuse [30].

1) Input Reuse (IR): The goal of this scheme is to minimize the number of times an *ifmap* is accessed. The stages are: ① The tiled *ifmaps* are loaded into the GB. ② The *ifmaps* are entirely reused in order to compute the corresponding *ofmaps*. ③ The partially computed *ofmaps* are stored back to memory. They are retrieved for the next *ifmap* and updated. Let us look at various patterns generated by scheduling *ifmap* tiles in different ways [30]. The following text heavily refers to Table II that compiles the most popular data flow patterns. The notations are similar to Section II-B.

Rows ❶ and ❷ ► *Partial (multi) channel loading:*
Consider the first entry's loop order with $K_T = 1$: $h_T \triangleright w_T \triangleright c \triangleright k$. We do not show the rest of the iterators because it is assumed that once the data corresponding to a set of input/output tiles is fetched (as per the loop order), the rest of the processing happens within the NPU. We are not concerned with internal aspects of the NPU, we only care about the accesses to main memory. Now, as per this loop order, we consider an input tile, and then we compute the results for a set of *ofmaps*. We cycle through the *ofmaps*, and then move to the next *ifmap*.

Example: Let us understand the pattern generation process using a simple example. Let us assume, $C = 2$, $K = 3$, and the GB can hold only one *ofmap* tile at any point of time. Each *ifmap* tile is reused by three filters to generate 3 distinct partially computed *ofmap* tiles. Each *ofmap* tile is associated with a VN. After the *ifmap* tile has been fully utilized, all three *ofmap* tiles will have the same VN (VN=1), as they are in the same computational stage (only one *ifmap* channel is processed). This leads to a VN write pattern 1^3 as seen by the Write-observer. Thereafter, the next *ifmap* tile from the next channel is read from memory. These partially computed *ofmap* tiles are again sequentially updated – we increment the VN of all the tiles from 1 to 2 (as they get evicted from the GB) leading to the following pattern: $1^3, 2^3$. We illustrate the pattern generation using a pattern diagram (Table II) in which the X-axis represents the time instance and the Y-axis represents the updated VNs.

After processing both the channels of an *ifmap* tile, we schedule the input tile movement along the w-axis and the aforementioned process will repeat until all the $\alpha_{HW} = \frac{H \times W}{H_T \times W_T}$ *ifmap* tiles are processed. The same version number creation process will be repeated α_{HW} times, leading to a final VN pattern of $(1^3, 2^3)^{\alpha_{HW}}$. Additionally, if we group a set of K_T output tiles and consider them together, then this expression will change to 1^{α_K}, where $\alpha_K = K/K_T$. We can generalize the equation and write it as: $(1^{\alpha_K}, 2^{\alpha_K} .... \alpha_C^{\alpha_K})^{\alpha_{HW}}$.

The VN updates will generate a simple deterministic pattern: Multi-step or Sawtooth. Additionally, the *ofmap* tile's *read pattern* will be mostly identical to the write pattern. The only difference is that we will not read the final *ofmap*. The final *ofmap* will be read in the subsequent layer. In Row ❷, we consider a group of C_T channels together. The expressions are similar.

Rows ❸ and ❹ ► *Movement along the Width/Height:* Consider the loop order, $c \triangleright h_T \triangleright w_T \triangleright k_T$. Basically, we first cover an *ifmap* and then move to the next. This means $\alpha_{HW} = (H \times W)/(H_T \times W_T)$ *ifmap* tiles will be accessed one after the other. Each *ifmap* tile will thus partially generate all the K *ofmaps* (in groups of K_T). All these groups of *ofmap* tiles will have the same VN. Then we shall move to the next *ifmap* to update all the partially computed output tiles. We increment the VNs accordingly. The process will repeat until all the *ofmap* tiles are fully computed.

Rows ❺ ► *Channel-wise:*

In this scheme, a single input channel (channel-wise) or a group of input channels (multi-channel) of size $H \times W$ constitute an input tile. Due to the stationary nature of the input tile, it will be reused $\alpha_k = K/K_T$ times to generate all of the α_k *ofmap* tiles of size $H \times W$ leading to the pattern (1^{α_K}). These partially computed output tiles are updated when a new *ifmap* tile is fetched from memory. The operation will be repeated until all input channels are processed.

Rows ❻ ► *Full Channel:*

This is a simple scheme in which all the channels required for the generation of an output tile are available. The VN for a tile will be updated only once due to the availability of all the channels, leading to the pattern $1^{\alpha_K \alpha_{HW}}$.

2) Output Reuse (OR): An output tile is completely computed in this scheme before being sent to memory. Initially, ① successive channels of the same *ifmap* are loaded into the GB. ② The partial *ofmap* tile is reused until it is fully computed. The write patterns are very simple because there is no write-read-update cycle. They are of the form 1^x. There is no read pattern because partially computed *ofmap* data is never read.

Rows ❶ and ❷ ► *Partial(multi) channel loading*: Prior to storing an output tile in memory, it must be completely computed. The VN of an output tile of size $H_T \times W_T$ remains the same as it does not leave the GB. Then we proceed to the next output tile. The pattern generated above will repeat for $(H \times W)/(H_T \times W_T)$ times for each of the $(H \times W)/(H_T \times W_T)$ tiles.

3) Weight Reuse: The steps involved in this process are as follows. Consider the first row. ① The tiled weight matrix (4D tensor) of size $C_T \times K_T \times R \times S$ is loaded in memory. ② C_T *ifmaps* of dimension $H \times W$ are loaded in the GB [42]. ③ The weights are reused to compute K_T *ofmaps* of dimension $H \times W$. The pattern generation methodology is similar to the previous schemes. The generated patterns for all the rows are shown in Table III.

Tiling style	**Loop order**	**Pattern**
	Filter-wise movement	
❶ Multi-channel wise [30]	$c_T \triangleright k_T$	WP:$1^{\alpha_K}, 2^{\alpha_K} \dots \alpha_C^{\alpha_K}$; RP:$1^{\alpha_K}, 2^{\alpha_K} \dots (\alpha_C - 1)^{\alpha_K}$ $Patterns$: P2,P3,P5
	Channel-wise movement	
❷ Channel-wise [30]	$k_T \triangleright c$	WP: 1^{α_K}; RP: $-$ $Patterns$: P5
❸ Full -filter [42]	k_T	WP: 1^{α_K}; RP: $-$ $Patterns$: P5

TABLE III: Pattern table for convolution: Different methods of scheduling weight tiles for weight reuse.

B. Other Kinds of Layers

Let us now look at some other kinds of layers in DNNs.

Generative-Adversarial Networks (GANs): GANs [10], [36] are composed of a discriminator and a generator network. To generate fake images, the generator uses deconvolution, whereas the discriminator uses convolution to discern between fake and real images. The pattern generation approaches for general convolution presented in Table II will work for any kind of convolution including deconvolution.

Matrix Multiplication: We analyze the data access pattern in the case of matrix multiplications as it is extensively used in several workloads such as transformers [43]. We classify the different tiling scenarios and present our findings in Table IV.

Tiling Style	**Loop order**	**Pattern**
❶ Fix P [26]	$h_T \triangleright c_T \triangleright w_T$	WP: $(1^{\alpha_W}, 2^{\alpha_W} \dots \alpha_C^{\alpha_W})^{\alpha_H}$ RP: $(1^{\alpha_W}, 2^{\alpha_W} \dots (\alpha_C - 1)^{\alpha_W})^{\alpha_H}$
❷ Fix Q [26]	$c_T \triangleright w_T \triangleright h_T$	WP: $(1^{\alpha_H}, 2^{\alpha_H} \dots \alpha_C^{\alpha_H})^{\alpha_W}$ RP: $(1^{\alpha_H}, 2^{\alpha_H} \dots (\alpha_C - 1)^{\alpha_H})^{\alpha_W}$
❸ Fix R [26]	$w_T \triangleright h_T \triangleright c_T$	WP: $1^{\alpha_{HW}}$ RP: $-$

TABLE IV: Pattern table for matrix multiplication ($R = P \times Q$): Various methods for tiling the input. Dimensions of P: $H \times C$, Q: $C \times W$ $\alpha_C = C/C_T; \alpha_H = H/H_T; \alpha_W = W/W_T$

Image Pre-processing/ Pooling: Numerous ML applications require the input image to be in a specific format. Additionally, it may be necessary to enhance an image's features prior to computation. This necessitates an analysis of the data access patterns for various image pre-processing methods.

We divide image pre-processing applications into three computation styles. The output channel is sometimes completely dependent on a single input channel, the scenario is represented as `Style-1` $S_x = T_x(X)$, where T_X represents the *transformation function*, and X represents the input element. Because there is no requirement to store partially computed outputs in memory, the output access pattern will be linear. However, as illustrated in Table VIII (in the Appendix), the number of output tiles will vary. We observe that pooling and `Style-1` computations follow a similar pattern. Typically, a window is positioned above the image to perform computations relative to the surrounding pixels in order to generate an output pixel. `Style-2` depicts a scenario in which all input channels are merged to form a single output channel $S = T(R, G, B)$, whereas `Style-3` depicts a scenario in which all input channels are merged using various transformations to form multiple output channels. Please refer to Table IX and Table X (in the Appendix).

Additionally, with regards to multiple reuse, we can easily capture such scenarios; we don't discuss many more possible combinations because they are quite rare in practice, we have space constraints and we were also limited by the dataflows that Timeloop can simulate.

Insight: We note that the pattern of VN updates is highly deterministic and is quite similar across a range of operations. All of the patterns can be expressed using a single master equation: $(1^\eta, 2^\eta \dots \kappa^\eta)^\rho$. The triplet $\langle \eta, \kappa, \rho \rangle$ exactly specifies the read/write VN pattern.

VI. Design of Securator

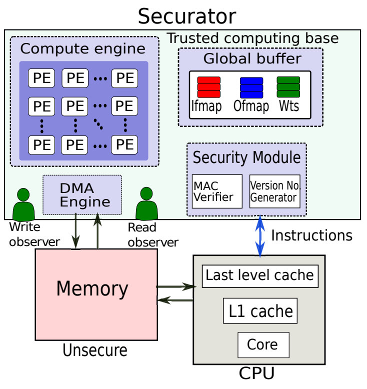


Fig. 6: The high-level design of Securator

A. Overview

Securator provides a hardware-assisted secure environment for the execution of neural networks. A high-level design of the framework is presented in Figure 6. The host CPU securely delivers instructions (using a shared key) to the accelerator via a PCIe link to execute a layer of the CNN. It also points to the location of the encrypted data stored in memory. Thereafter, the accelerator starts the execution. Initially, all the model parameters and the user input are encrypted and stored in memory. During the processing, the data is transferred to the NPU to perform the convolution. The compute engine(CE) is made up of a PE array and some local storage. When a layer is completed successfully, the accelerator writes the encrypted data to memory. In the case of a security breach, a system reboot is performed.

A security breach is detected by the security module, which will be in charge of securing the data and protecting it from various threats. To reduce the overheads and memory traffic, we integrate the VN generator with the security module such that the processed versions can be directly consumed. We explain the modules in the following subsections.

B. A Programmable Version Generation Scheme

Securator encrypts the output data when the data is evicted from the GB to memory. To automatically generate the VNs for the output data, we thoroughly analyzed the movement of *ofmap tile* data in Section V. An automatic VN generation scheme helps reduce the overheads associated with the storage and management of VNs. Even though the storage requirements per se are not very high (max: 8 KB in prior work [22]), the additional complexity in the design and runtime for reading this table, which is stored in the host CPU's secure memory, is quite onerous. Securator overcomes this limitation by generating VNs automatically (once for every group of tiles) as per the master equation in Section V: $(1^\eta, 2^\eta \dots \kappa^\eta)^\rho$, which is parameterized with a set of triplets, which are provided as an input to the programmable unit. The job of the unit is to generate the VNs at runtime based on the triplets $\langle \eta, \kappa, \rho \rangle$. The triplet is securely shared with the accelerator by the user along with a session-specific encryption key or public key (to decrypt data for the first time). Subsequently, we generate VNs based on memory accesses and the value of the triplet.

C. Details of the Encryption Process

We rely on AES counter-mode encryption (CTR) for encrypting each data block. To encrypt a 64-byte data block, we employ four parallel AES-128 engines. The 128-bit key is created by concatenating the accelerator's secret id (embedded within it) with a random number generated prior to execution. This technique ensures that the key is hardware-specific and changes with each execution. The major counter value is created by concatenating the *fmap* ID and layer ID, whereas the minor counter value is created by concatenating the VN and index of the block within the *fmap*. This approach ensures that the counter value changes in accordance with the index of the block in an *fmap* (the same value is encrypted differently). The counters are encrypted using the AES-CTR mode to generate a one-time pad (OTP). This OTP is XORed with the block data to create the ciphertext (standard algorithm).

D. MAC Generation

Unlike competing work, we do not maintain per-block MACs, we operate at the layer-level. The reason that prior work maintained per-block MACs is because they wanted to give the freedom to subsequent layers to read data randomly (no pre-specified order). We also give the same freedom, with one caveat, which is that a consumer layer needs to verify the output generated by the producer layer, regardless of how they are connected (holds for multi-output, skip and back connections as well). This is a very reasonable restriction because there is no reason to otherwise perform a layer computation if the computation is never used.

When we read/write a block, we compute its MAC. The 32-byte MAC is computed as $MAC = SHA_{256}(P||L||F||VN||I||B)$, where, P is the secret id of the accelerator, L is the layer ID, F is the id of the *fmap*, I is the block index within the *fmap*, and B is the contents of the data block (64-bytes). $||$ is the concatenation operator.

Let the MAC of a block that is being read/written be denoted by MAC_B. We maintain two 256-bit registers: MAC_R (for reads) and MAC_W (for writes). For a block that is read we compute $MAC_R = MAC_R \oplus MAC_B$, where $\oplus$ is the XOR operator. We do the same in the case of writes, albeit with the register MAC_W. As per Bellare et al. [3], this scheme is quite secure (theoretically similar to chaining).

We need to verify that whatever has been written is also read back without tampering. MAC_W embodies everything that has been written. If we analyze all the access patterns in Section V, we observe that in the same layer we read everything back other than data written in the last iteration. This is read back in the next layer. In the next layer, we use one more register MAC_{FR} (first read) that computes a MAC of all the *ifmap* data (*ofmaps* of the previous layer) that is read for the first time. We use the layer id of the previous layer. Note that it is very easy to design a circuit using our master equation to figure out when an input tile is read for the first time (not shown due to a lack of space).

The crucial condition that needs to hold is as follows. This is provable from the equations shown in Section V.

$$MAC_W = MAC_{FR} \oplus MAC_R \tag{1}$$

We can use two pairs of these registers (that alternate across layers) because of the overlaps in terms of usage (need MAC_R for the previous layer when the current layer is being processed).

Let us now consider read-only data such as inputs and filter weights. Their VN remains the same (it is equal to the last-generated VN in the previous layer for *ifmaps* and 1 for filter weights). Without loss of generality, consider *ifmap* data. We maintain a separate MAC_{IR} register for it. If the same *ifmap* tile is read an even number of times, the result of all the XOR operations to update MAC_{IR} should be zero, otherwise it will be equal to the XOR of the MACs – same as the first-read data (MAC_{FR}). This is because the outputs of the previous layer should be the inputs of the current layer. In either case, the inputs are verified because we are verifying that the "first-read" data for the inputs is correct in Equation 1.

Furthermore, it is important to note that Securator is a standalone unit, as we only care about the read-write patterns of the physical memory.

VII. Security Analysis

On the lines of prior work, the trusted computing base (TCB) comprises the accelerator, the code running on the host CPU (assumed to run within a secure execution environment), and their communication interface. The DRAM is not encrypted and securing its contents is the main aim here.

Confidentiality All the data written to DRAM is fully encrypted. The key changes with each run, guaranteeing that each execution of the same application generates unique encrypted data. Moreover, data blocks with the same content and address produce different ciphertexts over time due to the VN, fmap ID, layer ID, and block index (described in Section VI-C).

Integrity The output of a producer layer is verified when it is read in a consumer layer. Equation 1 in Section VI-D states the condition for block-level correctness (whatever is written == whatever is read). We can extend this result to a full layer on the lines of reference [3]. Regardless of the order of accesses, we read and write full layers several times. We just verify the same constraint at the layer level. This process is further strengthened by including the block address as a part of the MAC computation (avoids swapping and replication attacks).

Authenticity Similar to GuardNN [15] we can either use a session key and encrypt all the weights using it, or we can use a public key based mechanism. This is a solved problem in related work.

Freshness VNs ensure that *Securator* does not receive outdated data. We can correctly read data only when we use the latest VN, and a VN changes on every write. This is a standard approach that is extensively used in SGX [5] as well.

Side Channel Attacks We do not consider power and EM side-channel attacks. Cache side-channel attacks are not possible as such accelerators do not use the caches. The main memory access sequence is potentially visible to attackers even though the data in main memory is encrypted. To stop this, we need to use a scheme to stop model extraction attacks.

Model Extraction Attacks Li et al. [24] list various obfuscation methods for preventing MEAs. Our scheme is an orthogonal idea and can be used with any of these methods. For 3 of the 8 proposed schemes, we needed to change the triplets used in our master equation, which holds for all of their schemes. For two schemes, the compiler needs to add an extra layer (we are oblivious to it). The only scheme that needs rigorous evaluation is layer widening as it involves the maximum slowdown because the layers are increased in size to confuse the attacker. It essentially translates to a scalability study, because the more scalable the solution, the larger a layer can be (we can add more redundancy and make it hard for the attacker to extract useful information).

VIII. Evaluation

A. Setup

We showed the detailed simulation setup and list of benchmarks in Section IV. We evaluate 5 designs as shown in Table V. They are the baseline design (no security), *secure* design (*ClientSGX*), TNPU, GuardNN, and Securator. We used CNN benchmarks from Table I, along with GAN [31], and basic transformer [43]. We implemented the hardware of the pattern generator circuit, the SHA-256 and AES circuits in Verilog. We used the Cadence Genus Tool to synthesize, place, and route the design in a 28 nm technology (scaled to 8 nm using the results in [16]). TNPU and *secure* use an 8 KB MAC cache. We conducted simulations for an augmented design, Securator+, that protects against MEA and bus snooping attacks (details in Section VIII-E).

B. FPGA Prototype

We implemented a proof-of-concept (PoC) on the CHaiDNN [46] framework for FPGA-based accelerators (sim-

Configuration	Integrity (MAC)	Encryption (AES)	Anti-Replay	MEA
Baseline	×	×	×	×
Secure	per-block	CTR	Counters	×
TNPU	per-block	XTS	VN	×
GuardNN	per-block	CTR	VN	×
Securator	per-layer	CTR	VN	×
Securator+	per-layer	CTR	VN	✓

TABLE V: Simulated designs

TABLE VI: Overhead associated with the h/w structures

Module	Area (μm^2)	Power (μW)
AES-128	3900	640
SHA-256	270	40
VN generator	40	4.4

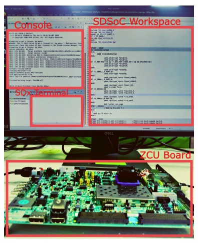


Fig. 7: Experimental setup

ilar to GuardNN [15]) to assess the changes need to the DNN compiler and the work that needs to be done by the host CPU. CHaiDNN v2 was synthesised with Vivado HLS 2018.2 in an SDSoC [19] development environment, with the Xilinx Zynq Ultrascale+ ZCU102 board as the target (shown in Figure 7). The user runs a program to generate the triplets for each layer based on a dataflow computed by Timeloop (or decided a priori). In the frontend, this data is kept as a dummy layer alongside the existing model parameters (first layer of the DNN). The backend design requires bypassing this extra layer, tracking main memory requests, and passing them to the Securator components. We added a *pluggable triplet generating module* (experimentally validated) and other Securator modules to the code of the synthesized accelerator. Integrating the Securator components – AES encryption unit, SHA unit, and VN generator – proved to be easy. The design didn't require any more changes and the code of the host CPU was not modified. Of course, in a more realistic setting we need to create a system for raising an alarm when a security exception is detected. This was not implemented in the PoC.

C. Verilog Synthesis Results

The overhead associated with the hardware structures is shown in Table VI. We incur a marginal area overhead of 4210 μm^2 (area) and a sub-mW power overhead.

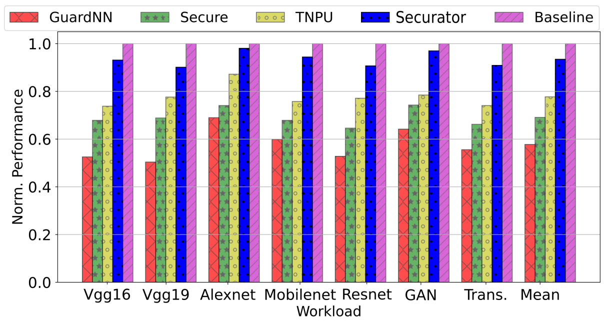


Fig. 8: Normalized performance for different workloads

D. Performance Analysis

Figure 8 presents the performance results using the simulation configuration mentioned in Table I. The baseline configuration is used to normalize the results. We reduced the performance overhead by nearly 20.56% compared to the state-of-the-art scheme TNPU. The fact that TNPU verifies integrity block-by-block results in a large number of MAC accesses. With TNPU, we observed nearly 35% misses in the MAC cache since streaming data results in a high cache miss rate as described in Section IV. This results in an increase in DRAM accesses as shown in Figure 9 (nearly 21%). Additionally, the access to the Tensor Table/tile stored in the secure memory location adds to the overhead. Securator simplifies the design by computing on-chip versions, thus eliminating the requirement for additional DRAM accesses. Because of direct DRAM accesses, there is a good correlation between the DRAM traffic and the performance. A similar correlation between the reduction of accesses and the performance improvement has been seen in prior work such as GuardNN, TNPU (albeit in very different settings).

In GuardNN, the MACs are generated according to the granularity of the accelerator data block (64 bytes). The scheme does not use any MAC cache. All the MACs are stored in an off-chip memory. Each data read or write request is accompanied by a MAC read/write request. This leads to a very high memory traffic with a performance degradation of nearly 64%.

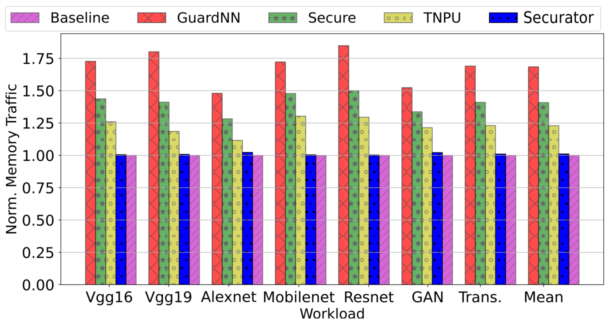


Fig. 9: Normalized memory traffic for different workloads

E. Scalability Analysis: Securator+

Li et al. [24] suggest many methods to prevent MEA and memory-based side-channel attacks. A key technique is layer widening: increase the size of all layers by padding junk data such that it is not possible to find their real size. To investigate the effect of layer widening, we increase the size of the base layer of MobileNet ($111 \times 111 \times 3$) to $3\times$, $4\times$, $5\times$, $6\times$ and $7\times$ the original size (see Figure 10). We observe that Securator is the most scalable and its relative performance increases with the layer size. Its speedup over TNPU increases from 16.6% ($1\times$) to 21% ($7\times$). Hence, Securator+ is the best choice for implementing layer widening to avoid MEA attacks.

F. Limitations

Securator is capable of simulating any DNN, other than those with data-dependent accesses, as they result in unpre-

Work	Security	Freshness(via VNs)		Integrity(via MACs)		MEA	Tile size
	(via encryption)	Granularity	Space	Granularity	Space	prot.	
Outsourcing computations [9], [12], [40]	Partial	Block	$BTm + TM$	Block	BTH	×	64 blocks
SGX+ Optimizations [20], [23], [41]	Full	Block	$BTm + TM$	Block	BTH	×	
Custom accelerators							
NPUFort [45]	Partial	Block	BTV	–	–	Obscure time*	Depends on resources, scheduling scheme and workload
Seal [49]	Partial	Block	BTV	–	–	×	
TNPU [22]	Full	Tile	TV	Block	BTH (on-chip)	×	
	VNs for each tile are stored in a secure memory (similar to SGX)						
GuardNN [15]	Full	Tile	TV	Block	BTH (off-chip)	×	
	VNs are stored and managed by a scheduler running on the host CPU (uses a TEE)						
Securator	**Full**	**Layer**	V	**Layer**	***O(H) (on-chip)***	**Widen layers**	
	VNs are not stored, but generated using on-chip FSM						
H: MAC size, V: VN size, M: Major counter size, m: Minor counter size, T: Total # of tiles, B: # of blocks per tile *Secure and unsecure fmaps have different latencies resulting in a fuzzy separation (in terms of temporal behavior)							

TABLE VII: A comparison of related work

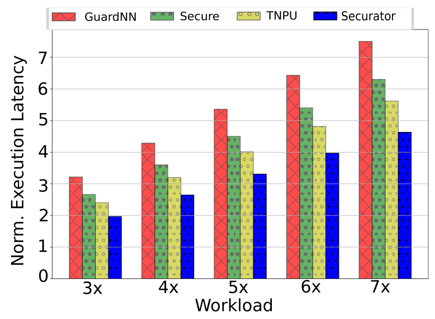


Fig. 10: Normalized execution latencies for scalability analysis (normalized to $111 \times 111 \times 3$)

dictable access patterns. Thus, we cannot design a circuit for its automatic VN generation. However, note that having data dependent accesses is a bad idea for secure processors as it exposes a side channel (data information). Designing a scheme that can do data-dependent access pruning and yet not sacrifice security is a subject of future work.

IX. Related Work

TEEs provide a secure platform for the execution of CNN workloads even in an untrusted environment; additionally, they outperform computationally heavy cryptographic methods such as fully homomorphic encryption [6]. Recent works focus on ❶ outsourcing the computation to an untrusted GPU [12], [40]; ❷ optimizing a pre-existing TEE such as SGX or Tensorcone [9]; and ❸ designing a secure, custom accelerator [15]. We present a brief comparison of related works in Table VII.

A. Outsourcing Computations

Slalom [40] and DarkNight [12] focus on providing a hybrid and secure execution platform. They split the computations between a TEE and an untrusted GPU. Layers are obfuscated and delegated from the TEE to the GPU. The output from the unsecure GPU is verified using the Freivald's algorithm [40]. The central insight behind the data obfuscation is that convolution is a bilinear operation. Instead of directly exposing the inputs to an untrusted third party, the CPU adds controlled noise. The bilinear property will ensure that the noise can later on be removed from the computed result. The overheads associated with input blinding, output verification along with the additional communication overhead due to data transfer from the TEE to the GPU cannot be avoided. Securator relies on a TEE alone, hence, these overheads are not involved.

HybridTEE [9] divides the computation between a constrained-resource local TEE (ARM Trustzone) and a resource-rich remote TEE (Intel SGX) based on the presence of security-critical features. A separate algorithm is used to detect which parts of the image are security-critical (the SIFT and YOLO algorithms are used). However, executing an auxiliary DNN to find security-critical features might lead to high overheads and may compromise system security.

B. Optimizing Intel SGX

Several works such as [20], [23], [41] focus on optimizing the execution of the whole DNN within a TEE. Vessels [20] solves the problem of low memory re-usability and a high memory overhead by creating an optimized *memory pool* for the TEE. This is done by characterizing the memory usage patterns in a CNN layer. Similarly, Truong et al. [41] propose to partition the layers to minimize the memory usage. In Occlumency [23], the authors proposed a memory-efficient feature map allocation technique and partitioning convolution operation to optimize the CNN execution in SGX.

On the contrary, DeepEnclave [11] aims to secure only the user data. It optimizes the memory utilization by executing the initial few layers inside the secure enclave and the latter outside. It is important to note that these works use the vanilla version of client-side SGX where Merkle trees and eviction trees are maintained for the entire data. However, we should note that such memory optimizations lose their utility in modern server-side SGX-based designs where the enclave

Tiling Style	Loop order	Pattern `Style-1`
❶ Channel-wise	k	WP:1^{α_K}
❷ Multi-channel	k_T	RP:–
❸ Partial channel	$h \triangleright w \triangleright k_T$	WP:$1^{\alpha_K \alpha_{HW}}$
❹ Partial-multi-channel	$h_T \triangleright w_T \triangleright k_T$	RP:–
❺ Full-channel	$h_T \triangleright w_T$	WP: $1^{\alpha_{HW}}$, RP: –

TABLE VIII: Pattern table for image pre-processing (`Style-1`)/Pooling $(S_x = T_x(X))$: Possibilities for scheduling the output tiles. (H_T, W_T, C_T) represents the tiling factor. $(C = K)$

Tiling Style	Loop order	Pattern (`Style-2`)
❶ Channel-wise	c	WP: 1, RP: 1
❷ Multi-channel	c_T	WP: 1, RP: 1
Tile movement along the channel		
❸ Partial channel	$h_T \triangleright w_T \triangleright c$	WP: $1^{\alpha_{HW}}$
❹ Multi channel	$h_T \triangleright w_T \triangleright c_T$	RP:–
Tile movement along the width/height		
❺ Partial channel	$c \triangleright h_T \triangleright w_T$	WP: $1^{\alpha_{HW}}, 2^{\alpha_{HW}} \dots \alpha_C^{\alpha_{HW}}$
❻ Multi channel	$c_T \triangleright h_T \triangleright w_T$	RP:$1^{\alpha_{HW}}, 2^{\alpha_{HW}} .. (\alpha_C - 1)^{\alpha_{HW}}$
❼ Full-channel	$h_T \triangleright w_T$	WP: $1^{\alpha_{HW}}$, RP: –

TABLE IX: Pattern table for image pre-processing(`Style-2`: $S = T(R, G, B)$): Methods of scheduling input tiles (K=1)

Output Reuse			Input Reuse	
Tiling Style	Loop order (OR)	Rd/Wr Pattern	Loop order (IR)	Rd/Wr Pattern
❶ Channel-wise	$c \triangleright k_T$	WP:1^{α_K}	$k_T \triangleright c$	WP:$1^{\alpha_K}, 2^{\alpha_K} \dots \alpha_C^{\alpha_K}$
❷ Multi-channel	$c_T \triangleright k_T$	RP:–	$k_T \triangleright c_T$	RP: $1^{\alpha_K}, 2^{\alpha_K} \dots (\alpha_C - 1)^{\alpha_K}$
Tile movement along the channel				
❸ Partial channel	$h_T \triangleright w_T \triangleright k_T \triangleright c$	WP:$1^{\alpha_K \alpha_{HW}}$	$k_T \triangleright h_T \triangleright w_T \triangleright c$	WP:$(1^{\alpha_K}, 2^{\alpha_K} .. \alpha_C^{\alpha_K})^{\alpha_{HW}}$
❹ Multi channel	$h_T \triangleright w_T \triangleright k_T \triangleright c_T$	RP: –	$k_T \triangleright h_T \triangleright w_T \triangleright c_T$	RP:$(1^{\alpha_K}, 2^{\alpha_K} \dots (\alpha_C - 1)^{\alpha_K})^{\alpha_{HW}}$
Tile movement along the width/height				
❺ Partial/Multi channel	–	–	$k_T \triangleright h_T \triangleright w_T \triangleright c$	WP: $1^{\alpha_K \alpha_{HW}}, 2^{\alpha_K \alpha_{HW}} \dots \alpha_C^{\alpha_K \alpha_{HW}}$
❻ Multi channel	–	–	$k_T \triangleright h_T \triangleright w_T \triangleright c_T$	RP:$1^{\alpha_K \alpha_{HW}}, 2^{\alpha_K \alpha_{HW}} \dots (\alpha_C - 1)^{\alpha_K \alpha_{HW}}$
❼ Full-channel	$h_T \triangleright w_T \triangleright k_T$	WP: $1^{\alpha_k \alpha_{HW}}$; RP: –	$k_T \triangleright h_T \triangleright w_T$	WP: $1^{\alpha_{HW} \alpha_K}$; RP: –

TABLE X: Pattern table for image pre-processing (`Style-3`: $S_i = T_i(R, G, B)$)

size can be as large as 512 GB [33]. The key advantage of our scheme is that we provide all security guarantees without the additional overheads of counters and Merkle trees.

C. Designing a Custom TEE

GuardNN [15] and TNPU [22] focus on bringing the security guarantees provided by traditional secure processors such as Intel SGX to custom accelerators. The majority of the overheads in a conventional secure processor comes from cache misses and Merkle tree traversals. GuardNN and TNPU both aim to eliminate the Merkle tree and counters with a more sophisticated and efficient VN management mechanism that makes use of a DNN's extremely deterministic and statically defined memory access patterns.

TNPU needs a "Tensor Table" to keep track of the output tile updates (every update is associated with a new VN). Since input tiles are never updated, the VNs linked with them are never updated. TNPU divides the secure memory into two regions: the first is protected by the information stored in the Tensor Table (Region 1), while the second 128 KB secure memory region is protected by a system similar to *SGX-Client* (Region 2). The Tensor Table protects Region 1, and the table itself is stored in Region 2. MACs are generated at the granularity of individual blocks and they are stored in an 8 KB on-chip MAC cache (overflows go to main memory).

GuardNN relies on counter-mode encryption, which requires VNs. For memory writes, the accelerator generates VNs using on-chip counters, which are managed by the scheduler (running on the host CPU with a secure TEE). For memory reads, VNs are received from the scheduler on the host CPU. GuardNN also generates block-specific MACs which are not stored in an on-chip cache (read directly from main memory).

Due to the high memory requirements of DNNs, generating per-block MACs like GuardNN and TNPU results in high memory overheads. Even if an on-chip cache is used, the situation does not improve significantly because caches are not optimized for streaming data. Securator addresses these two concerns by significantly reducing the number of MACs required to verify the integrity of a DNN. For GPUs, [48] and [27] provide a secure execution environment that is quite similar to GuardNN. They are not specific to neural networks and thus don't leverage their characteristics.

X. Conclusion

We showed that competing works such as GuardNN and TNPU naturally segue into the Securator proposal. It is a logical successor in this line of work, where the move to layer-level security checks is complete. For realizing this, it was necessary to mathematically characterize a large number of memory access patterns (with different levels of stationarity) and encode them efficiently. The VN generator coupled with the MAC verifier helped realize layer-level operations. Because of the reduced need for data storage and consequently reduced DRAM traffic, it was possible to reduce wasted cycles, and achieve an additional 20.56% throughput as compared to TNPU, which is the nearest competing work.

Appendix

We show the results for image pre-processing (`Style-1`, `Style-2`, and `Style-3`) in Tables VIII, IX, and X.

Acknowledgement

This work has been supported in part by the Semiconductor Research Consortium (SRC).

References

[1] , "Artificial intelligence (ai) hardware market size and forecast," Online, 2022. [Online]. Available: https://www.verifiedmarketresearch.com/product/global-artificial-intelligence-ai-hardware-market/

[2] A. Anderson, *Virtual reality, augmented reality and artificial intelligence in special education: a practical guide to supporting students with learning differences*. Routledge, 2019.

[3] M. Bellare, R. Guérin, and P. Rogaway, "Xor macs: New methods for message authentication using finite pseudorandom functions," in *Annual International Cryptology Conference*. Springer, 1995, pp. 15–28.

[4] Y. Chen, K. Wang, X. Liao, Y. Qian, Q. Wang, Z. Yuan, and P.-A. Heng, "Channel-unet: a spatial channel-wise convolutional neural network for liver and tumors segmentation," *Frontiers in genetics*, p. 1110, 2019.

[5] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptol. ePrint Arch.*, vol. 2016, no. 86, pp. 1–118, 2016.

[6] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "Chet: an optimizing compiler for fully-homomorphic neural-network inferencing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 142–156.

[7] S. Deng, H. Zhao, W. Fang, J. Yin, S. Dustdar, and A. Y. Zomaya, "Edge intelligence: The confluence of edge computing and artificial intelligence," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 7457–7469, 2020.

[8] A. Facon, S. Guilley, X.-T. Ngo, and T. Perianin, "Hardware-enabled ai for embedded security: A new paradigm," in *2019 3rd International Conference on Recent Advances in Signal Processing, Telecommunications & Computing (SigTelCom)*. IEEE, 2019, pp. 80–84.

[9] A. Gangal, M. Ye, and S. Wei, "Hybridtee: Secure mobile dnn execution using hybrid trusted execution environment," in *AsianHOST*. IEEE, 2020, pp. 1–6.

[10] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[11] Z. Gu, H. Huang, J. Zhang, D. Su, A. Lamba, D. Pendarakis, and I. Molloy, "Securing input data of deep learning inference systems via partitioned enclave execution," *arXiv preprint arXiv:1807.00969*, 2018.

[12] H. Hashemi, Y. Wang, and M. Annavaram, "Darknight: An accelerated framework for privacy and integrity preserving deep learning using trusted hardware," in *MICRO-54*, 2021, pp. 212–224.

[13] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[14] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[15] W. Hua, M. Umar, Z. Zhang, and G. E. Suh, "Guardnn: Secure dnn accelerator for privacy-preserving deep learning," *arXiv preprint arXiv:2008.11632*, 2020.

[16] W. Huang, K. Rajamani, M. R. Stan, and K. Skadron, "Scaling with design constraints: Predicting the future of big chips," *IEEE Micro*, vol. 31, no. 4, pp. 16–29, 2011.

[17] O. Iliashenko, Z. Bikkulova, and A. Dubgorn, "Opportunities and challenges of artificial intelligence in healthcare," in *E3S Web of Conferences*, vol. 110. EDP Sciences, 2019, p. 02028.

[18] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," *White paper*, 2016.

[19] V. Kathail, J. Hwang, W. Sun, Y. Chobe, T. Shui, and J. Carrillo, "Sdsoc: A higher-level programming environment for zynq soc and ultrascale+ mpsoc," in *Proceedings of the 2016 ACM/SIGDA international symposium on field-programmable gate arrays*, 2016, pp. 4–4.

[20] K. Kim, C. H. Kim, J. J. Rhee, X. Yu, H. Chen, D. Tian, and B. Lee, "Vessels: efficient and scalable deep learning prediction on trusted processors," in *SoCC*, 2020, pp. 462–476.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[22] S. Lee, J. Kim, S. Na, J. Park, and J. Huh, "Tnpu: Supporting trusted execution with tree-less integrity protection for neural processing unit," in *HPCA*, 2022.

[23] T. Lee, Z. Lin, S. Pushp, C. Li, Y. Liu, Y. Lee, F. Xu, C. Xu, L. Zhang, and J. Song, "Occlumency: Privacy-preserving remote deep-learning inference using sgx," in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019, pp. 1–17.

[24] J. Li, Z. He, A. S. Rakin, D. Fan, and C. Chakrabarti, "Neurobfuscator: A full-stack obfuscation tool to mitigate neural architecture stealing," *arXiv preprint arXiv:2107.09789*, 2021.

[25] D. Moolchandani, A. Kumar, and S. R. Sarangi, "Accelerating cnn inference on asics: A survey," *Journal of Systems Architecture*, vol. 113, p. 101887, 2021.

[26] G. E. Moon, H. Kwon, G. Jeong, P. Chatarasi, S. Rajamanickam, and T. Krishna, "Evaluating spatial accelerator architectures with tiled matrix-matrix multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 1002–1014, 2021.

[27] S. Na, S. Lee, Y. Kim, J. Park, and J. Huh, "Common counters: Compressed encryption counters for secure gpu memory," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 1–13.

[28] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *ISPASS*. IEEE, 2019, pp. 304–315.

[29] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[30] R. V. W. Putra, M. A. Hanif, and M. Shafique, "Romanet: Fine-grained reuse-driven off-chip memory access management and data organization for deep neural network accelerators," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 4, pp. 702–715, 2021.

[31] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[32] Raul Inze., "The cost of machine learning projects," Online, 2019. [Online]. Available: https://medium.com/cognifeed/the-cost-of-machine-learning-projects-7ca3aea03a5c

[33] S. Johnson et al., "Supporting intel sgx on multi-socket platform," Intel Corp., 2021. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/supporting-intel-sgx-on-mulit-socket-platforms.pdf

[34] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic cnn accelerator simulator," *arXiv preprint arXiv:1811.02883*, 2018.

[35] S. R. Sarangi, *Advanced Computer Architecture*. McGraw-Hill Education, 2021.

[36] N. Shrivastava, M. A. Hanif, S. Mittal, S. R. Sarangi, and M. Shafique, "A survey of hardware architectures for generative adversarial networks," *Journal of Systems Architecture*, vol. 118, p. 102227, 2021.

[37] N. Shrivastava and S. R. Sarangi, "Towards an optimal countermeasure for cache side-channel attacks," *IEEE Embedded Systems Letters*, 2022.

[38] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[39] J. Suo, W. Zhang, J. Gong, X. Yuan, D. J. Brady, and Q. Dai, "Computational imaging and artificial intelligence: The next revolution of mobile vision," *arXiv preprint arXiv:2109.08880*, 2021.

[40] F. Tramer and D. Boneh, "Slalom: Fast, verifiable and private execution of neural networks in trusted hardware," in *ICLR*, 2018.

[41] J.-B. Truong, W. Gallagher, T. Guo, and R. J. Walls, "Memory-efficient deep learning inference in trusted execution environments," *arXiv preprint arXiv:2104.15109*, 2021.

[42] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei, "Deep convolutional neural network architecture with reconfigurable computation patterns," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 8, pp. 2220–2233, 2017.

[43] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[44] S. Wang, D. Zhou, X. Han, and T. Yoshimura, "Chain-nn: An energy-efficient 1d chain architecture for accelerating deep convolutional neural networks," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, 2017, pp. 1032–1037.

[45] X. Wang, R. Hou, Y. Zhu, J. Zhang, and D. Meng, "Npufort: A secure architecture of dnn accelerator against model inversion attack," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, 2019, pp. 190–196.

[46] Xilinx, “Haidnn-v2: Hls based deep neural network accelerator library for xilinx ultrascale+ mpsocs.” Xilinx, 2018. [Online]. Available: ttps://github.com/Xilinx/CHaiDNN

[47] X. Yang, T. A. Yang, and L. Wu, “An edge detection ip of low-cost system on chip for autonomous vehicles,” in *Advances in Artificial Intelligence and Applied Cognitive Computing*. Springer, 2021, pp. 775–786.

[48] S. Yuan, A. Awad, A. W. B. Yudha, Y. Solihin, and H. Zhou, “Adaptive security support for heterogeneous memory on gpus.”

[49] P. Zuo, Y. Hua, L. Liang, X. Xie, X. Hu, and Y. Xie, “Sealing neural network models in encrypted deep learning accelerators,” in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1255–1260.