MGX: Near-Zero Overhead Memory Protection for Data-Intensive Accelerators

Weizhe Hua†, Muhammad Umar†, Zhiru Zhang†, G. Edward Suh†§∗
†Cornell University, Ithaca, NY, USA
§Meta AI, Cambridge, MA, USA
∗Work was done while at Cornell University.

{wh399,mu94,zhiruz,gs272}@cornell.edu, edsuh@fb.com

ABSTRACT
This paper introduces MGX, a near-zero overhead memory protection scheme for hardware accelerators. MGX minimizes the performance overhead of off-chip memory encryption and integrity verification by exploiting the application-specific properties of the accelerator execution. In particular, accelerators tend to explicitly manage data movement between on-chip and off-chip memories. Therefore, the general memory access pattern of an accelerator can largely be determined for a given application. Exploiting these characteristics, MGX generates version numbers used in memory encryption and integrity verification using on-chip accelerator state rather than storing them in the off-chip memory; it also customizes the granularity of the memory protection to match the granularity used by the accelerator. To demonstrate the efficacy of MGX, we present an in-depth study of MGX for DNN and graph algorithms. Experimental results show that on average, MGX lowers the performance overhead of memory protection from 28% and 33% to 4% and 5% for DNN and graph processing accelerators in a wide range of benchmarks, respectively.

CCS CONCEPTS
• Security and privacy → Security in hardware; • Computer systems organization → Architectures.

KEYWORDS
Off-chip memory protection, version number generation, secure accelerators, neural networks, graph algorithms

ACM Reference Format:
Weizhe Hua, Muhammad Umar, Zhiru Zhang, G. Edward Suh. 2022. MGX: Near-Zero Overhead Memory Protection for Data-Intensive Accelerators. In The 49th Annual International Symposium on Computer Architecture (ISCA ’22), June 18–22, 2022, New York, NY, USA. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3470496.3527418

1 INTRODUCTION
As the technology scaling slows down, computing systems are increasingly relying on hardware accelerators to improve performance and energy efficiency. For example, modern machine learning (ML) models such as deep neural networks (DNNs) are often quite compute-intensive and increasingly run on hardware accelerators [13, 40] for both performance and energy efficiency. Similarly, hardware accelerators are widely used for other compute-intensive workloads such as video decoding, signal processing, cryptographic operations, genome assembly, etc. This paper proposes a novel off-chip memory protection scheme for hardware accelerators, named MGX (Memory Guard for Xelerators), using secure acceleration of DNN and graph algorithms as the primary applications.

In many applications, the hardware accelerators may process private or sensitive data, which need strong security protection. For example, ML algorithms often require collecting, storing, and processing a large amount of personal and potentially private data from users to train a model. Moreover, due to its high computational demand, both training and inference are often performed on a remote server rather than a client device such as a smartphone, implying that the private data and ML models may be exposed if the server is compromised or malicious.

A promising approach to providing strong confidentiality and integrity guarantees under untrusted environments is to create a hardware-protected trusted execution environment (TEE), also called an enclave as in Intel SGX [59]. The cryptographic protection of off-chip memory represents an essential technology to enable the hardware-protected TEE. The off-chip memory protection also represents the main source of performance overhead in the traditional secure processor designs. For a general-purpose processor, the memory protection schemes need to be able to handle any sequence of memory accesses to arbitrary memory locations, and typically protect memory accesses at a cache-block granularity. In secure processors, the counter-mode encryption is used to hide decryption latency, where the counter value is typically a concatenation of the memory address and a version number (VN). The version number is stored in memory and incremented on each write of an encrypted block. To protect integrity of off-chip memory, a message authentication code (MAC) needs to be attached to each cache block in memory. Moreover, the integrity verification requires a tree of MACs to prevent replay attacks. Unfortunately, the VN and MAC accesses can lead to non-trivial performance overhead for memory-intensive workloads. In order to extend the TEE to application-specific accelerators, we need a more efficient memory protection scheme that can protect memory-intensive workloads with low overhead.
In this paper, we show that memory encryption and integrity verification can be performed with almost no performance overhead for an accelerator by customizing protection to the accelerator-specific memory access pattern. We make key observations that the application-specific accelerators explicitly move data between on- and off-chip memories following a relatively simple pattern that is specific to an application, and that the off-chip data movements usually use a granularity that is larger than a cache block. The relatively simple and static memory access patterns imply that version numbers can often be calculated from the on-chip state without storing them in off-chip memory. The coarse-granularity data movement suggests that the version numbers for memory encryption and the MACs for integrity verification can be maintained at a coarse granularity to reduce the overhead.

We study the memory access behaviors of DNN inference and training as well as two representative graph algorithms, and show how to determine the version numbers using the scheduling and the state of the accelerator. By generating version numbers on-chip and performing protection at an application-specific granularity, MGX can eliminate most of overhead for off-chip memory protection; no version number is stored in the off-chip memory, no integrity tree is needed, and each MAC protects a large amount of data instead of one cache block.

To evaluate the effectiveness of MGX for DNN accelerators, we performed extensive experiments using cycle-level simulations based on SCALE-Sim [69], an open-source DNN accelerator simulator from ARM research. The overhead of applying MGX to both DNN inference and training is less than 5% on the state-of-the-art DNN models. For graph accelerator, we performed the experiments using a combination of RTL and cycle-level simulations based on an open-source graph accelerator [33]. The simulation results on two important graph algorithms, PageRank and Breadth-first Search (BFS), show that MGX can provide both memory encryption and integrity verification with very low overhead.

This paper makes the following major contributions:

- We propose MGX, a near-zero overhead memory protection scheme for accelerators. MGX minimizes the performance overhead of memory protection by assigning counter values for data using on-chip state and performing coarse-grained memory protection.
- We demonstrate the applicability of MGX by showing a concrete implementation of MGX for DNNs and graph algorithms and detailed analyses of a genome sequence alignment accelerator and an H.264 video decoder accelerator.
- We evaluate the secure accelerators with MGX and show that the overhead is 3.2% and 4.7% for DNN inference and training, and 5.1% and 4.9% for PageRank and BFS.

2 Secure Accelerator Architecture

The goal of a secure accelerator is to protect the confidentiality and the integrity of the state and data of the accelerator even in an untrusted environment where the host software and the off-chip memory cannot be trusted. For example, the secure DNN accelerator aims to protect inputs, outputs, weights, and all intermediate results.

Figure 1 shows the threat model of the secure accelerator and Table 1 summarizes the potential threats and the corresponding protection mechanisms. As shown in the figure, the hardware TCB mainly includes the CPU TEE and the accelerator. The host processor is assumed to have a trustworthy TEE with a secure communication channel to the accelerator as in recent proposals for GPU TEEs [82]. Following the typical threat model for secure processors, we assume that the internal operations and state of an accelerator cannot be directly observed or modified by an adversary through physical attacks. The off-chip memory is assumed untrusted; the secure accelerator needs memory protection to encrypt confidential data and detect unauthorized changes in values stored in DRAM. We do not consider side-channel attacks such as the memory address, timing, and power side channels.

The accelerator-specific kernels are attested and then executed on the trusted control processor of the accelerator. For external communications, the accelerator needs to support a secure key-exchange protocol to establish trust and securely communicate with a remote user or a TEE. Specifically, the secure accelerator includes a unique private key (SK_acc), embedded by a manufacturer. We assume that a user obtains the corresponding public key using a private key infrastructure as in Intel SGX or Trusted Platform Modules (TPMs). The accelerator also provides remote attestation using its private key so that a user can authenticate hardware, the hash of firmware/configuration of the accelerator, the hash of the application kernel, and the hash of the input and output data.

Our threat model is representative of the typical TEE threat model and common for both the baseline memory protection and MGX. In MGX, both a memory protection unit and an application kernel that issues commands to the accelerator need to be trusted for memory protection. The kernel also needs to be included in remote attestation and protected by running on a control processor inside an accelerator using on-chip memory. Note that software inside a TEE such as the application kernel is already inside the
Figure 2: Memory encryption and integrity verification.

TCB of a typical TEE; software inside the TEE is allowed to output a secret.

To use the secure accelerator, a user sends a command to initiate a new session, which will have the accelerator clear its internal state, set a pair of new symmetric keys for encryption and integrity verification, enable protection mechanisms, and establish a secure (encrypted and authenticated) communication channel with the user using a standard protocol such as an TLS. After initialization, a user sends an application kernel and user data through the encrypted channel. The accelerator loads the data by decrypting it with the communication key, and placing it in protected memory that is encrypted with the memory encryption key. Once the execution is finished, the accelerator returns the encrypted results.

3 MGX: NEAR-ZERO OVERHEAD MEMORY PROTECTION FOR ACCELERATOR

This section first describes state-of-the-art memory protection scheme (i.e., memory encryption and integrity verification). Then, we introduce the MGX scheme, which provides secure and low-overhead memory protection by leveraging the regular and mostly static memory access patterns of specialized accelerators. Finally, we provide an example on tiled matrix multiplication to better explain the proposed MGX scheme.

3.1 Memory Protection Basics

Memory Encryption. As shown in Figure 2(a), existing techniques [25, 30, 72] typically use the counter-mode encryption (AES-CTR) to hide AES latency. AES-CTR requires a non-repeating counter value for each encryption under the same AES key. In a secure processor, the counter value often consists of the physical memory address (PA) of the data block that will be encrypted and a (per-block) version number (VN) that is incremented on each memory write. When a data block is written, the encryption engine increments the VN and then encrypts the data. When a data block is read, the encryption engine retrieves the VN used for encryption and then decrypts the block. Let $K_{	ext{Enc}}, U, V$ be the AES encryption key, plaintext, and ciphertext, respectively. The AES encryption can be formulated as $V = U \oplus K_{\text{Enc}}(P||V)$.

As general-purpose processors can have an arbitrary memory access pattern, the VN for each cache block can be any value at a given time. In order to determine the VN for a later read, a secure processor needs to store the VNs in DRAM. To avoid re-using the same counter value, the AES key needs to change once the VN reaches its maximum, which implies that the size of the VN needs to be large enough to avoid frequent re-encryption. For example, Intel SGX [25] uses a 56-bit VN for each 64-byte data block, which introduces 11% storage and bandwidth overhead. In general, the VNs cannot fit on-chip and are stored in DRAM. As the VNs are stored in the off-chip memory, the integrity and freshness of VNs also need to be protected with MACs to ensure the confidentiality.

Integrity Verification. To prevent off-chip data from being altered by an attacker, integrity verification cryptographically checks if the value from DRAM is the most recent value written to the address by the processor. For this purpose, a MAC of the data value, the memory address, and the VN is computed and stored for each data block on a write, and checked on a read from DRAM. However, only checking the MAC cannot guarantee the freshness of the data; a replay attack can replace the data and the corresponding VN and MAC in memory with stale values without being detected. To defeat the replay attack, a Merkle tree (i.e., hash tree) [24] is used to verify the MACs hierarchically in a way that the root of the tree is stored on-chip. As shown in Figure 2(a), a state-of-the-art method [66] uses a Merkle tree to protect the integrity of the VNs in memory, and includes a VN in a MAC to ensure the freshness of data. Let us denote the key, plaintext, and ciphertext as $K_{\text{Enc}}, U, V$, respectively. The MAC of an encrypted data block can be calculated as $MAC = H_{\text{Enc}}(V||PA||VN)$. The overhead of integrity verification is nontrivial as it requires traversing the tree stored in DRAM. To mitigate this overhead, recently verified MACs are stored in a cache. However, caching is often not effective for data-intensive applications such as large ML models. Moreover, the Merkle tree poses a scalability challenge because its depth needs to increase with the size of the protected memory.

Figure 3 shows the memory traffic overhead of applying the traditional memory protection for DNN inference and training, PageRank, and BFS. For all applications, the number of memory accesses increases by at least 23.1% and over 49.2% in the worst case. The average memory traffic overheads for DNN inference, DNN training, PageRank, and BFS are 36.1%, 40.4%, 26.3%, and 25.6%. Among the applications, DNN training is most heavily affected by off-chip memory protection because it requires access to a large amount of data. In addition, the memory traffic overhead of accessing the VNs can greatly exceed that of accessing the MAC, as it requires a Merkle tree to verify the integrity of the VNs.

3.2 Intuition

The main overhead of memory protection comes from storing and accessing VNs and MACs for protecting the confidentiality and integrity of data and verifying the MACs for VNs hierarchically in the off-chip memory. Because many accelerators such as DNN and graph accelerators are often memory-intensive, these additional meta-data accesses can lead to nontrivial performance overhead. We propose to significantly reduce the memory protection overhead by generating VNs without storing them in memory and customizing the protection granularity based on the application-specific memory access pattern.

As customized for a particular application domain, specialized accelerators tend to have more predictable and regular memory access patterns compared to general-purpose CPUs. In particular, both DNN inference and training can be scheduled statically based
on the network structure. For example, most popular deep learning (DL) frameworks such as TensorFlow [3] adopt declarative programming and lazy execution, where the DNN network is represented as a data-flow graph (DFG). A DL framework (i.e., compiler) first optimizes the DFG and then generates the scheduling of the computations and the corresponding memory accesses based on the optimized DFG before execution. Therefore, the DL compiler can assign a VN for each memory access based on the schedule without storing the VNs in memory.

Moreover, accelerators may have the same access pattern to a large chunk of memory because they typically operate on blocks of memory larger than cache lines. For example, DNNs write the output feature maps of a layer to DRAM the same number of times because each output feature map is generated following the same schedule. As VNs reflect the maximum number of writes to the corresponding memory block, this regular memory access pattern means that we only need one VN for all the output features of a layer or a tile. If a DNN accelerator only writes the output features to DRAM once per layer (i.e., no tiling), MGX can simply use the layer number as part of the VN. In addition to DNN accelerators, most graph accelerators update the attributes associated with each vertex the same number of times (e.g., mostly once) in each iteration. In that case, the attribute values of vertices can also share the same VN value.

3.3 MGX Scheme

A specialized accelerator typically has an on-chip control processor that receives the statically compiled kernel (i.e., the accelerator-specific executable/binary) and is responsible for executing the kernel, orchestrating the functional units of the accelerator, and keeping track of the state of the accelerator. For example, the BFS algorithm can be considered as a kernel for graph accelerators. In our MGX design, as shown in Figure 2(b), the VNs for reading and writing memory blocks are generated by the kernel running on the trusted on-chip control processor, rather than being stored in memory. The application kernel can assign VNs for memory writes based on the scheduling of compute/memory operations and accelerator state. We found that the kernel only needs to maintain an on-chip state to generate VNs, given an application-specific and coarse-grained nature of accelerator memory accesses. For memory writes, the kernel ensures that the VN is greater than the last-used VN value for the memory location so that the same VN value is never reused for encrypting a memory block. For memory reads, the kernel on the control processor regenerates the VN value used for the most recent write to the same address by using the on-chip state to ensure proper decryption.

In the DNN accelerator, a kernel on the control processor implements a full DNN model, and VNs are computed inside the accelerator without any off-chip communications. However, depending on the complexity of the application, an accelerator may rely more on the host CPU to determine which operations to run. In such cases, the scheduling software in the host CPU TEE can provide additional state to determine VNs when issuing commands to an accelerator control processor. Note that the VN values can be public because the security of the AES-CTR encryption and the MAC only depends on the integrity, not the confidentiality, of VNs.

It is worth noting that MGX does not require static or sequential memory access patterns to generate VNs. Reads do not affect the VNs no matter how irregular they are. Writes can also happen in an arbitrary order using one VN value as long as they occur once per each address. If needed, the control processor can keep additional state for VNs.

Once the VN is determined, the encryption (Enc) engine can decrypt/encrypt each 128-bit data block using the standard AES-CTR method for memory encryption. As VNs no longer need to be stored in DRAM, the integrity protection tree for VNs also becomes unnecessary, greatly reducing off-chip accesses. For integrity protection, MACs still need to be stored in memory. We propose to further reduce the overhead by customizing the size of a memory block that each MAC protects to match the data movement granularity of the accelerator. For example, the CHAI accelerator [89] from Xilinx reads a 512-byte chunk from memory at a time. Using a 64-bit MAC for each 512-byte data block significantly reduces the bandwidth overhead for integrity protection.

Figure 4(a) shows an example of tiled matrix multiplication (MatMul), where two matrices $A$ and $B$ are blocked into two and four tiles, respectively. According to the scheduling shown in Algorithm 4(b), the partial results of $C_1$ and $C_2$ are first computed, then the final results are obtained by summing the partial results. We assume that $A$ and $B$ are previously written to off-chip memory with VN value $n$. Since $A$ and $B$ are read-only during the computation, the MatMul kernel uses $n$ as the VN for reading tiles of $A$ and $B$, as shown in Figure 4(c). In the first two iterations (i.e., iteration 1 and 2 in Figure 4(a)), the accelerator writes the partial results of $C_1$ and $C_2$ to DRAM with an incremented VN value $n + 1$. The VN value for $C_1$ and $C_2$ needs to be incremented as the memory locations can be reused. Since $C_1$ and $C_2$ occupy different memory locations in the off-chip memory under this specific scheduling, the MatMul kernel can assign the same VN value $n + 1$ for both tiles. For the last two rounds, the MatMul kernel first reads the partial results of $C_1$ and $C_2$ with VN value $n + 1$, which is the VN value used to write them. Then, the kernel increments the VN and write the final results of $C_1$ and $C_2$ with VN value $n + 2$. The kernel keeps track of the VN for each matrix (or each tile) in its program state (i.e., VN[A], VN[B], VN[C_1], and VN[C_2]).
3.4 Security Analysis

Encryption – MGX uses the same AES counter-mode encryption that is used by the traditional memory encryption scheme for processors. The only difference between MGX and the traditional scheme is that the VN in MGX is determined/generated by the scheduler based on the accelerator state, rather than being stored in off-chip memory. As long as the version number generator guarantees that the VN is unique for each write to a given memory location, the counter value, which is a concatenation of the memory address and the version number, is different for each encryption. Therefore, the security of the memory encryption in MGX can be reduced to the AES counter-mode encryption, which is one of the approved modes of operation [19, 55]. Note that using one VN for multiple memory locations does not sacrifice security because the counter value to a block cipher in the counter mode includes a memory address in addition to a VN.

Integrity Verification – MGX uses a MAC to protect the integrity of data in memory. The MAC includes the address and the VN in addition to data. This MAC construction is identical to the one that is used in the traditional integrity verification scheme (shown in Figure 2(a)), and protects against replay, relocation, and substitution attacks [60], as long as the version numbers are unique for each write to a location. In the traditional scheme, the version numbers need to be protected separately using a Merkle tree because they are stored in off-chip memory. In MGX, version numbers cannot be tampered with an attacker because they are generated on-chip. Also, MGX requires the scheduler to generate VNs in a way that a VN value is only used at most once for a write to each memory location. Thus, the integrity protection in MGX can be reduced to that of the chosen keyed-hash function.

4 MGX FOR DNN ACCELERATION

This section introduces the background on DNNs, the workflow of DNN acceleration, and discusses how MGX can be applied to enable efficient memory encryption and integrity verification for secure DNN computation.
augmentation for image data) and then forward the processed data and the kernel to the DNN accelerator.

The on-chip control processor executes the kernel. For accelerators that support high-level functions (e.g., convolution and pooling) such as TPUs [40], TVM-VTA [62], and ChaDNN [89], the high-level functions in the kernel can issue multiple low-level instructions to functional units. For example, the MatMul instruction is executed by the matrix multiplication array on the accelerator. The control processor is also responsible for providing the VN values for memory reads and writes required by each instruction. For example, the convolution function in some accelerators is implemented as a nested for loop, where the inner loop computes the partial results of different tiles of the output features and the outer loop accumulates the partial results to obtain the final results of the output features. Similar to the MatMul example discussed in Section 3.3, the kernel code provides the VN values for each tile in the inner loop of the function. Finally, the functional unit performs the DNN computation specified by the low-level instructions.

4.3 Version Number Generation for DNNs

This subsection describes the MGX scheme for DNN inference and training, focusing on how VNs can be determined by the DNN kernel on the control processor. Figure 6 shows how the counter values are constructed for DNN memory protection in MGX. Each counter value includes the address of the memory block being encrypted/decrypted and a 64-bit VN, and is used as the input to AES-CTR encryption. Note that using one VN for multiple memory locations does not sacrifice security because the counter value includes a memory address. The VNs are constructed differently for accessing three different types of data: features, weights, and gradients, ensuring that the counter values are unique for each encryption and are never reused.

DNN Inference – During DNN inference, the accelerator reads the feature maps of the previous layer as the input, performs the computation, and then produces the output feature maps of the current layer. The feature maps of each layer are written to off-chip memory the same number of times, regardless of the scheduling of the accelerator. Therefore, we can keep one unique VN value for the feature maps of each layer (VN\(_F\)). When writing new feature maps generated from a DNN layer, the DNN kernel increments the maximum VN\(_F\) used before and assigns this new value as the VN\(_F\) for these feature maps. In the case that the feature maps are written exactly once at the end of each layer, the DNN kernel can assign VN\(_F\) for the feature maps based on the layer number. For example, the feature maps of the \(i^{th}\) layer have a VN\(_F\) value of \(i\) concatenated with the input count.

If optimizations such as loop reordering and tiling are employed in DNN kernels, the output feature maps can be written to DRAM multiple times within a layer, requiring the VN\(_F\) increment multiple times within a layer (e.g., the VN for the output matrix is incremented twice in the tiled MatMul example). In this case, the DNN kernel can maintain VN\(_F\) in its program state and keeps track of the VN values associated with the feature maps of each layer in the network. As the VN\(_F\) used for writing each feature map is generated by the DNN kernel on the control processor, the DNN kernel can also provide the VN\(_F\) for reading feature maps based on its program state. Once the VN\(_F\) is generated for the feature map, the memory protection unit receives the VN\(_F\) value from the control processor and encrypts/decrypts the feature map using the specified value.

As illustrated in Figure 7(a), the DNN kernel can generate VN\(_F\) for the output feature maps \(y\), even if \(y\) is written to DRAM as many times as the number of tiles. In this specific case, Algorithm 7(b) provides the pseudocode of the augmented convolution function running on the control processor for setting the VNs for memory reads and writes. The VN for the input feature maps \(x\) (VN\(_F\) \([x]\)) remains constant (e.g., \(n\)) as \(x\) is read-only within the layer. Here, \(n\) simply represents the VN for the input feature maps at the beginning of a layer. In the first iteration, the kernel assigns \(n + 1\) as the VN value for the output feature maps \(y\) (VN\(_F\) \([y]\)) and writes \(y\) with that VN value. Then, for the rest of the \(t - 1\) iterations, the kernel reads \(y\) with the current VN\(_F\) \([y]\) value, increments VN\(_F\) \([y]\), and writes the updated \(y\) with the new VN\(_F\) \([y]\) value. Assuming that \(y\) is written to off-chip memory \(t\) times, the final VN value associated with \(y\) should be \(n + t\). Similarly, the VNs for feature maps across different DNN layers can also be determined easily by the kernel. Figure 8(a) shows the computational graph (i.e., DFG) of the forward propagation of a residual block, which is a widely used structure in many modern DNNs [28, 31]. Suppose each conv layer \((L_1, L_2, L_3)\) and the element-wise addition layer \((L_4)\) write their output to DRAM \(t_1, t_2, t_3\), and \(t_4\) times, respectively. The DNN kernel can compute the VN for each feature map in the residual block based on the scheduling as VN\(_F\) \([x_i]\) = \(n + \sum_{k=1}^{t_k} f_k\), where \(n\) is the VN for the input features to the residual block. The VN value of the output features is incremented for different layers to guarantee...
that the same counter value is never reused, even though output features from different layers can exist in memory at the same time.

The weights are read-only during inference. Therefore, we can use a constant as the VN for the weights until they are updated. To allow updating weights, the DNN kernel tracks $V_{W}$ in its program state and keeps track of the number of updates (writes) to the weights.

Note that $V_{F}$ and $V_{W}$ are all kept as part of the program state in the trusted control processor, and there is no VN stored in external memory. For simplicity, the kernel can maintain one $V_{F}$ for the output features of each layer and one $V_{W}$ for the entire network. In this implementation, a 127-layer DNN uses 1 KB on-chip state for VNs. The size of the on-chip state can be reduced by only tracking the non-consumed features (i.e., the features will be used as the input to later layers) or leveraging the network structure statically known to the kernel. For example, the VN for output features can be determined from one counter that keeps the number of inputs processed and the layer number. For the 64-bit VN in our design, an accelerator with a throughput of 1,000 inputs per second for a 1,000-layer DNN can run for 0.28 million years before an overflow. If an overflow happens, MGX requires the memory to be re-encrypted with a new key.

**DNN Training** – One iteration of training consists of a forward propagation and a backpropagation. The forward propagation is the same as inference except that all intermediate features are saved, and can use the VN generation strategy for inference. Here, we describe the VN generation for the backpropagation. Each layer first computes the gradients flowing to the previous layer using the gradients flowing into current layer and the associated weights. Then, the layer’s weights are updated using the incoming gradients and the saved features.

VNs are constructed in the same way as in the inference. The backpropagation only adds additional feature reads and does not affect the VN generation for features. The VNs for weights still use $V_{W}$ as all weights are updated the same number of times. However, $V_{W}$ are incremented more frequently, where $V_{W}$ tracks the number of updates to the weights, where the weights are updated exactly once during backpropagation. Gradients in backward pass correspond to the feature maps in forward pass. As illustrated in Figure 8(b), the VNs for gradients can be computed in a way similar to computing the VNs for feature maps. In addition to $V_{F}$ and $V_{W}$, the control processor also needs to keep track of the VN of the gradients ($V_{NC}$) associated with each layer. It is worth noting that broadcast ($L_{0}$) and element-wise addition ($L_{4}$) operations in forward pass become the element-wise addition and broadcast operations in backpropagation after differentiation, respectively.

Figure 8: The VN generation scheme for the forward and backward passes of a residual block — Blue values represent the VN associated with the feature maps (x) and the gradients (y). Broadcast and element-wise addition operations in forward pass become the element-wise addition and broadcast operations in backpropagation after differentiation, respectively.

**5 MGX FOR GRAPH PROCESSING**

This section provides background on graph processing and discusses how MGX can be applied to graph accelerators for different graph algorithms, such as PageRank and Breadth-First Search (BFS).

**5.1 Background on Graph Algorithms**

Graphs represent a popular way to encode connections in many important applications including social networks, electrical grid, circuits, etc. At the same time, processing large graphs requires high performance. To achieve high performance and low power, many ASIC/FPGA accelerators are proposed for graph processing.

In this work, we focus on the GraphBLAS formulation [42], where key graph processing operations (e.g., traversals, shortest path) are formulated as sparse linear algebra operations such as sparse-matrix dense-vector multiplications (SpMV). GraphBLAS extends the expressiveness of linear algebra in representing graph operations by leveraging the concept of semiring.
defined as a 5-tuple \((\mathcal{D}, \oplus, \otimes, I_\mathcal{D}, I_\otimes)\), where \(\mathcal{D}\) is a set, \(\oplus\) is a scalar multiplication operation, \(\otimes\) is a scalar addition operation, \(I_\mathcal{D}\) is the identity of \(\oplus\), and \(I_\otimes\) is the identity of \(\otimes\). The matrix operations on a semiring can correspond to a step in many different graph algorithms, depending on specific operators used in that semiring. For example, PageRank, BFS, and single-source shortest path (SSSP) can be expressed using the following semirings:

- PageRank: \((\mathbb{R}, \times, +, 1, 0)\)
- BFS: \((\text{Boolean}, \&\), |, 1, 0)\)
- SSSP: \((\mathbb{R} \cup \infty, +, \min, 0, \infty)\)

Given the expressiveness and the flexibility of GraphBLAS, an ASIC/FPGA accelerator designed with the GraphBLAS programming interface can be used to execute a rich set of graph algorithms, whereas many existing graph processing accelerators are designed to accelerate one specific graph algorithm such as PageRank and BFS [93, 95]. Therefore, we investigate the applicability of MGX with a focus on GraphBLAS-based graph processing accelerators.

GraphBLAS represents the topology of a graph (i.e., the connectivity between vertices) as a sparse adjacency matrix and the attributes associated with vertices as a sparse or dense vector. Figure 9 provides an example for the presentation of the graph topology and attribute values (i.e., the ranks of vertices) in PageRank using GraphBLAS. The graph topology is encoded with the adjacency matrix, where each non-empty cell represents the weight between the two connected vertices. The rank vector contains the current attribute values associated with the vertices. Then, PageRank algorithm can be expressed as an SpMV operation between the sparse adjacency matrix and the dense rank vector. The resulting updated rank vector remains in a dense format and will be used as the rank vector for the next iteration.

5.2 Version Number Generation for GraphBLAS

MGX is also applicable to graph processing accelerators. Here we discuss how to apply MGX to a GraphBLAS-based accelerator using the PageRank algorithm as an example. While our discussion mainly focuses on PageRank, MGX can also be applied to other graph algorithms supported by GraphBLAS.

PageRank is an iterative algorithm, which computes the rank of each vertex by calculating the likelihood of that vertex being reached. There are three main data structures to store a graph in PageRank — a sparse adjacency matrix, a dense rank vector, and a dense updated rank vector. The sparse adjacency matrix stores all edges in a graph, which is represented as a tuple of the IDs of the source and destination vertices. Due to the sparsity of the graph, the adjacency matrix is usually stored in a compressed sparse format to eliminate redundant memory accesses. Both rank and updated rank vectors are stored in dense format. Each entry in the three data structures typically occupies 4 bytes in memory and each data structure can have several to thousands of millions of entries in real-world graphs [16, 32, 46].

In each iteration of PageRank, the graph processing accelerator needs to update the attribute values of all vertices (i.e., calculate the updated rank vector). GraphBLAS-based accelerators typically use the scheduling as depicted in Figure 10. The graph processing accelerator computes the updated rank vector for a subset of vertices (e.g., \(\{A, B, C, D\}\)) at a time and generates the updated rank vector for all vertices in a sequential manner. When calculating the updated rank vector for a subset of vertices (e.g., \(\{A, B, C, D\}\)), the accelerator accesses a tile of the sparse adjacency matrix between the destination vertices and a subset of source vertices (e.g., the \(2^{nd}\) tile) and the corresponding rank vector (e.g., the rank \(\{E, F, G, H\}\)) to get the partial result of the updated rank vector. After processing all tiles, the final results of the updated ranks for the subset of destination vertices are obtained.

In PageRank, the sparse adjacency matrix is read-only and read sequentially as the adjacency matrix is stored in a sparse format. In addition, the adjacency matrix remains unchanged across different iterations of PageRank. Thus, MGX can assign a constant VN value for the adjacency matrix. However, the size of the sparse adjacency matrix in each tile differs because each destination vertex may be connected to an arbitrary set of source vertices. For example, in Figure 10, the first, second, and third tiles contain six, five, and four edges, respectively. Because the sparse adjacency matrix needs to be read with an irregular block size, integrity verification for the adjacency matrix must be in a fine granularity to avoid unnecessary memory reads. For example, an accelerator may read/write 64-byte chunks of the adjacency matrix at a time, and have a MAC for each chunk.

Instead, MGX can calculate the coarse-grained MAC exploiting the fixed scheduling of the graph processing accelerator. As the sparse adjacency matrix remains unchanged, the accelerator partitions the sparse adjacency matrix in the same way across different iterations. Therefore, MGX can protect the confidentiality and integrity of the sparse adjacency matrix at the tile level, where all
elements in the same tile can be protected using a single MAC. The rank vector is also read-only and read sequentially during each iteration, only requiring one VN per graph. Each tile of the updated rank vector is written sequentially and will be written to the off-chip memory for the same number of times, which also only requires one VN per graph. Therefore, the kernel on the control processor only needs to track the number of executed iterations (Iter) of PageRank to calculate VNs, thus requiring only 64-bit additional on-chip state. Specifically, (Iter – 1) is used as the VN when reading a tile of the rank vector and Iter is used as the VN when writing a tile of the updated rank vector. With MGX, the VN can be computed without off-chip memory accesses, eliminating the overhead of memory encryption. MACs can be calculated at coarse granularity to reduce the overhead of integrity verification. Since BFS uses the same SpMV operation as PageRank, the VN generation scheme remains the same and only one Iter counter is added to the accelerator state.

In addition to the SpMV operation adopted in PageRank, sparse-matrix sparse-vector multiplication (SpMSpV) is another important linear algebra operation in GraphBLAS. Compared with SpMV, the only difference is that the SpMSpV operation reads the attribute values associated with the vertices randomly instead of sequentially. MGX can still use the same VN generation scheme for all data structures and the same MAC granularity for the adjacency matrix and the vector with updated attribute values, as in SpMV. However, the vector holding the current attribute values requires a fine-grained MAC. In this case, MGX can still greatly reduce the overhead of off-chip memory protection.

6 EVALUATION

6.1 Accelerator and Simulation Setup

For DNN accelerator, we use cycle-level simulations to (1) compare the performance overhead of multiple memory protection schemes, (2) study the overhead for a larger class of DNN models, and (3) evaluate the overhead for DNN inference and training. Specifically, we use SCALE-Sim [69], an open-source cycle-level DNN accelerator simulator from ARM research. For graph processing, we use a combination of RTL and cycle-level simulation to compare the performance overhead of different memory protection schemes. In particular, we use GraphLily [33], an open-source GraphBLAS accelerator written in HLS.

As shown in Figure 11, both simulation setups have three main components — an accelerator, a memory protection simulator, and off-chip memory. The DNN accelerators are simulated in a cycle-level DNN simulator (i.e., SCALE-sim) to generate a trace of computation and memory events. The HLS graph processing accelerator (i.e., GraphLily) is first synthesized into RTL design. We then use RTL simulation to obtain the trace of computation and memory events. After the memory traces are obtained, a memory protection simulator uses the event trace to calculate the total execution time and the bandwidth usage by simulating protection mechanisms and DRAM accesses. The memory accesses are simulated using Ramulator [44] DDR4 at 2400MHz. The performance of the accelerator is maximized when the memory bandwidth matches the computation throughput, which means the accelerator is neither compute nor memory bounded. Finally, a performance evaluator generates the final performance based on the timing of computation and new memory events (including additional memory events from memory protection).

Accelerator Configurations – To evaluate the MGX for a DNN accelerator under different use cases, we model a large and a small configurations, namely Cloud and Edge, for cloud and edge computing, respectively. Cloud is modeled based on Google TPU-v1 [40] and Edge uses a similar configuration as the Samsung Neural Processing Unit [18]. Specifically, Cloud and Edge contain 64k and 1k processing elements (i.e., MAC units) and 24 MB and 4.5 MB on-chip memory, running at 700 MHz and 900 MHz, respectively. To balance computation and memory bandwidth, we simulate one 64-bit DDR channel for Edge and four 64-bit DDR channels for Cloud. The size of the protected memory is 16 GB. For the graph accelerator, we simply adopt the original design of the GraphLily accelerator. The clock frequency of the graph accelerator is assumed to be 800 MHz. We simulate four 64-bit DDR channels at 2400 MHz to provide enough bandwidth.

Benchmarks – For the DNN accelerator, we evaluate MGX on a variety of DNN architectures — AlexNet, VGG, GoogleNet, and ResNet for image classification and BERT (i.e., Transformer encoder) for language pretraining, and DLRM for personalized recommendation. For each DNN model, we simulate both inference (forward propagation) and training (forward propagation and backpropagation) of the network. The weight update during training is not emulated as no similar operation is available in SCALE-Sim.

For the graph accelerator, we validate the effectiveness of MGX on two widely-used graph algorithms: PageRank and BFS. Both algorithms are executed using the SpMV engine in GraphLily. We perform PageRank and BFS on six existing graph benchmarks, including Google-plus, pokec, livejournal, and reddit from the Stanford Network Analysis Project [53] and ogbl-ppa and ogbn-products from the Open Graph Benchmark [32]. Google-plus, pokec and livejournal are social network graphs. reddit is composed of posts from the Reddit forum. ogbl-ppa and ogbn-products are two large graph datasets containing 576K and 2449K vertices and 42M and 124M edges, respectively.

Memory Protection – We implement the recent memory encryption engine (MEE) design from Intel [25] as the baseline memory encryption. This baseline uses a multi-level 8-ary Merkle tree with 56-bit VNs and MACs, and works at a 64-byte granularity. Similarly, for integrity verification, we implemented the baseline that uses one MAC for each 64-byte block. Because the DNN accelerator has a largely streaming memory access pattern, increasing the
VN/MAC cache does not help unless it is big enough to capture temporal locality across layers. In our experiments, we include a reasonably large (32-KB) on-chip cache for VNs and MACs in the baseline scheme. The VN/MAC cache uses the LRU replacement policy with write-back and write-allocate policies. MGX has no cache for VNs and MACs, and protects the integrity using a MAC per 512-byte block for most applications, amortizing the overhead of memory protection over a large chunk of data. It is worth noting that the MAC granularity of the embedded tables in DLRM is still 64-byte, as fine-grained access to the embedded tables is required.

6.2 Experimental Results

Performance – We compare the accelerator performance for three different protection schemes: no protection (NP), today’s baseline memory protection (BP), and MGX. The results for BP and MGX are normalized to the one with no protection.

Figure 12 compares the memory traffic increase of two DNN accelerator configurations with MGX and BP — Cloud-MGX, Cloud-BP, Edge-MGX, Edge-BP. Cloud-BP and Edge-BP introduce 36.0% and 36.3% more memory accesses on average for inference, respectively. In particular, the inference of the recommendation model (i.e., DLRM) increases the memory traffic by 55%. For training, the average increases in memory accesses are 37.8% and 42.9% for Cloud-BP and Edge-BP. The memory traffic increase is larger for training because the training process accesses more data and has more frequent cache evictions in the VN/MAC cache. Cloud-MGX and Edge-MGX increase the memory traffic by an average of 2.4% and 2.4% for inference and 2.7% and 3.5% for training, respectively. The results demonstrate the advantage of the MGX, which removes VNs stored in DRAM and uses a MAC per 512-byte data block to match the accelerator’s data movement granularity.

Figure 13 shows the performance of the baseline protection and two MGX variants, MGX — PageRank-MGX, PageRank-BP, BFS-MGX, BFS-BP. As shown optimization to the overhead reduction, we include the results of two MGX variants, MGX_{VN} and MGX_{MAC}, which use only one optimization: on-chip VN generation or coarse-grained MAC. MGX_{VN} is 1.08× and 1.12× slower than no protection on average for inference and training. MGX_{MAC} has a higher overhead than MGX_{VN}, on average 1.16× and 1.20× slower than no protection for inference and training. This result shows that both on-chip VN generation and coarse-grained MAC are important in reducing the overhead of off-chip memory protection.

For the graph accelerator, we compare the memory traffic increase and execution time of PageRank and BFS.

Figure 14: The memory traffic increase of DNN inference and training.

Figure 13: The normalized execution time of the DNN inference and training on different networks models.
in Figure 14(a), PageRank-BP and BFS-BP introduce 26.3% and 25.6% more memory accesses on average, respectively. MGX only adds 1.5% and 1.4% additional memory accesses for PageRank and BFS, respectively. Compared to BP, MGX is able to significantly reduce the meta-data memory accesses, demonstrating the effectiveness of VN generation and coarse-grained MAC.

Figure 14(b) compares the performance of the baseline protection and MGX for PageRank and BFS. BP leads to a significant slowdown for both PageRank and BFS. For PageRank and BFS, the maximum slowdown due to BP is 1.42x and 1.39x, respectively. In contrast, MGX introduces only the maximum overhead of 5.2% for both graph algorithms. Across all benchmarks, BP and MGX have average performance overhead of 32.7% and 5.0%. In addition, the average performance overheads of MGX_VN and MGX_MAC are 9.4% and 18.0% across all benchmarks.

6.3 Case Study on Existing Accelerators

To show how MGX can be applied to existing DNN accelerators, we study CHaiDNN [89], which is an open-source DNN accelerator from Xilinx. CHaiDNN has a relatively simple accelerator interface, which only supports three high-level operations including Convolution, Deconvolution, and Pooling. Activation functions are merged with high-level operations to avoid unnecessary DRAM access and to maximize performance. Because of the high abstraction level of CHaiDNN, a deep neural network like AlexNet can be expressed in less than 20 instructions.

In order to equip CHaiDNN accelerator with MGX, we can implement the MGX scheme using a microcontroller for generating and managing version numbers. Each CHaiDNN instruction can be treated as a DNN layer. For each layer, the microcontroller assigns a VN value to all output features belonging to that layer and keeps track of the VN values in the on-chip VN table (i.e., the microcontroller’s SRAM memory). The VN table also needs to have two counters for weights and inputs as described in Section 4.3. In addition to the microcontroller, we also need to add AES Galois/Counter Mode (AES-GCM) cores [52] for both memory encryption and integrity verification. As the DNN accelerators typically have large processing element arrays and on-chip buffer, the overhead of adding microcontroller and AES-GCM cores is expected to be modest.

7 DISCUSSION

In this section, we discuss two additional cases to show MGX is also applicable to other accelerators such as genome alignment and video decoding. In addition, we also show that MGX can be applied to DNNs with static and even dynamic pruning techniques.

7.1 Applicability of MGX

Genome Alignment – We consider the off-chip memory protection for Darwin [81], which is an accelerator for genome assembly.

While Darwin also relies on a CPU to perform certain initialization operations and control the hardware acceleration, we assume that the CPU and its communications with Darwin are protected separately with a secure computing technology (e.g., Intel SGX) and focus on protecting memory accesses for the accelerators in this discussion.

Figure 15 shows the components and data accesses in Darwin. Darwin consists of two hardware-accelerated parts, D-SOFT and GACT, which use five types of data in off-chip memory: reference sequences, a seed-pointer table, a position table, query sequences and traceback pointers. During initialization for a reference-assisted assembly, the reference sequence, the seed-pointer table, and the position table are loaded (written) into memory once by a CPU; these are later only read by the accelerator. Therefore, the version number for these three data structures can be obtained simply from a counter in the on-chip state of the accelerator, which increments on each new genome assembly (CTRgenome).

After initialization, the CPU loads a batch of query sequences into memory and runs D-SOFT and GACT on the accelerator for each query in the batch. D-SOFT generates a filtered list of candidate positions from seeds in the query that hit in the reference sequence. These are passed on to GACT arrays as tiles for extension i.e. alignment. During these processes, the seed pointer & position tables, and the query & reference sequences are all only read by the accelerator. The output consists of GACT arrays writing traceback pointers for each tile sequentially into the memory. Hence, for the query sequences and traceback pointers, we can keep a counter in the accelerator state that increments for each new query batch (CTRquery), and use the concatenation of CTRgenome and CTRquery as the version number. The traceback pointers are later processed by the software to construct aligned reads.

The GACT part of Darwin is available as open-source RTL, whereas D-SOFT is available as software. We evaluate the performance of GACT for reference-guided assembly, using the latest human genome assembly GRCh38 [14] as reference. For chromosomes 1, X & Y, we generate three sets of reads simulating different sequencers (PacBio, ONT2D, and ONT1D) with varying error profiles, as described in [81].
We use the D-SOFT software to generate a list of candidate positions (or tiles) that are sent to the GACT hardware for alignment. For each tile, a GACT array loads a chunk of reference and query sequences from a specified DRAM offset, performs alignment of that tile, and finally writes the traceback pointers to DRAM. For the memory accesses of each tile, we obtain the data transfer times through a memory protection simulator, with four DDR4-2400 channels. For obtaining the computation time of the tiles, we perform an RTL simulation of GACT, with the same settings as specified in [81]. We assume an ASIC configuration with 64 GACT arrays that can process tiles independently, each containing 64 PEs, running at 800 MHz. As D-SOFT generates calls to GACT for millions of tiles, we simulated only a subset of the tiles. The memory and computation times are used to calculate the overall execution time. Because GACT loads input chunks from effectively random locations in the reference and non-contiguous locations in the query, and since the tile size can be variable, we do not use coarse-grained MACs for GACT, and only simulate the MGX\textsubscript{VN} mode with on-chip VN generation and fine-grained MACs.

We first compare the memory traffic increase of the baseline protection scheme (BP) with MGX\textsubscript{VN}. The elimination of off-chip VNs leads to a reduction in memory traffic overhead from 34% in BP to 12.5% in MGX\textsubscript{VN}, the remaining overhead coming from the fine-grained MACs. Figure 16 shows the performance overhead of GACT. The average performance overhead for BP is 14%. The performance overhead of genome assembly alignment is lower than DNN and graph algorithms because the Darwin accelerator design is more compute-bound. MGX\textsubscript{VN} further reduces the average performance overhead to only 4%.

**H.264 Video Decoding** – We studied H.264/AVC video decoding [1] as another candidate for MGX memory protection. Figure 17 shows a typical H.264 decoder architecture, which transforms an input bitstream into video frames. The input bitstream is typically encrypted with the standard counter mode [2]. The decoding process outputs different kinds of frames. Whereas I (intra-coded) frames are independent, the P (inter-predicted) frames are calculated using previous frames as a reference. B (bi-directional) frames use later frames as a reference, leading to out-of-order decoding. Therefore, multiple decoded frames are kept in off-chip memory buffers and if needed, are re-read by the inter-prediction stage.

To study how MGX can be applied to a H.264 decoder, we analyzed an open-source implementation [57]. This decoder stores the decoded and reference frames in external memory, and supports the

![Figure 17: Block diagram of a typical H.264 decoder — the inter-prediction module reads reference frames from the off-chip memory for constructing predicted frames.](image)
redundant computations at run time, and memory access patterns may vary for different inputs. However, the variations are still limited; dynamic pruning may skip some of the accesses that exist in the model, but does not introduce extra accesses. It may appear that the MGX does not work with dynamic pruning. However, skipping VNs does not affect the security as long as the VNs are not reused. The memory protection will be functional as long as a write and the corresponding reads use the same VN.

To demonstrate the MGX under static and dynamic pruning, we implemented a variety of pruning techniques in PyTorch and emulated the MGX in software. For pixel-level dynamic pruning, we implemented different compression techniques such as Compressed Sparse Row [6], Compressed Sparse Column [27, 74], and Run-Length Compression [63]. We also tested a dynamic channel pruning scheme similar to [23]. With dynamic pruning, the number of memory accesses to the features is input-dependent and determined at run time. However, we found that the version numbers of the features with dynamic pruning can still be obtained from the same VN generation scheme. Figure 20 illustrates the case where features are dynamically pruned. MGX uses the shared VN to write output features, but only unpruned features (e.g., $x_1, x_3, ..., x_t$) are written to memory. Then, MGX can read the sparse input features (e.g., $x_1, x_3, ..., x_t$) using the same VN. Again, only unpruned features are read from memory. For pruned features, the VN is simply not used and skipped.

8 RELATED WORK

Privacy-Preserving Deep Learning – Homomorphic encryption (HE) and secure multi-party computation (MPC) can provide stronger protection than TEEs by performing all computations in an encrypted format. However, DNN tasks in the HE/MPC solutions [17, 41, 45, 56, 61, 64, 78, 83, 84] are still multiple orders of magnitude slower than the baseline with no protection. A recent work [65] proposes to reduce the latency of HE-based DNN inference to hundreds of milliseconds using specialized hardware. Yet, the overhead in throughput is still quite significant even with an HE accelerator. A secure accelerator with MGX provides a design point to offer hardware-based security with much higher performance.

There are many TEEs [7, 9, 10, 12, 15, 21, 22, 47, 49, 59, 72, 75, 77, 86, 92] proposed for CPUs. Recent studies showed that DNNs can be protected using Intel SGX [43, 50, 79], but with non-trivial overhead of memory protection in SGX. Today’s processor-based TEEs are also limited by the performance of a general-purpose processor. Recent studies [38, 39, 82, 96] proposed to extend today’s TEE by including a GPU. The GPU TEE designs enable high performance, but require both a CPU and a GPU to be protected inside a TEE. Also, ASIC accelerators are often far more energy-efficient compared to GPUs and widely used for high-throughput tasks such as inference. Outsourcing to untrusted GPUs/accelerators [80] is another promising approach. However, secure outsourcing introduces significant computation and storage overhead for the offline phase; for high-throughput applications, performance will still be limited by CPUs.

Recent work [85, 94, 96] proposes to build FPGA/ASIC TEEs as accelerators. TNP [73] is most similar to our work, which also proposes a tree-free off-chip memory protection by exploiting the DNN-specific memory access patterns. In this work, we demonstrate that MGX can further reduce the overhead of integrity verification using coarse-grained MACs and is generally applicable to other data-intensive accelerators beyond DNN accelerators.

Memory Encryption and Integrity Verification – There is a large body of work on memory encryption and integrity verification for general-purpose CPUs, including the counter-mode encryption [71], optimizations to reduce the size of VNs [68, 90], counter-based integrity trees [20, 26, 66, 76], meta-data caching [24, 48], and predicting VNs or using unverified VNs speculatively [51, 70, 88]. The general-purpose protection schemes all require version numbers in off-chip memory, which will pose a challenge for applications with large data sets and random access patterns as in DLRM. MGX introduces a new approach to customize memory protection for a specific application and remove off-chip VNs, which significantly reduces the overhead of the state-of-the-art.

Side-channel Attacks and Protection – A variety of side-channel attacks have been shown against DNN accelerators. Memory and timing side-channels have been used to infer the network structure and weights of DNN models [34, 35, 91]. Power and electromagnetic side-channel attacks have been shown against DNN accelerators. Memory encryption and integrity verification that MGX aims to provide. A secure accelerator needs to be extended with additional countermeasures to prevent the side-channel attacks.

9 CONCLUSION

In this paper, we propose a novel off-chip memory protection scheme for hardware accelerators, named MGX. On average, MGX reduces the performance overhead of memory protection from 28% and 33% to 4% and 5% for DNN and graph processing accelerators, respectively. We also show that MGX is generally applicable to other applications, such as genome assembly alignment and H.264 video decoding.

10 ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive feedback on the earlier version of the manuscript. At Cornell, Weizhe Hua and Muhammad Umar are supported in part by NSF Award CCF-2007832, ECCS-1932501, and CCF-2118709. Weizhe Hua is also supported in part by the Facebook fellowship.
