SoftVN: Efficient Memory Protection via Software-Provided Version Numbers

Muhammad Umar[†], Weizhe Hua[†], Zhiru Zhang[†], G. Edward Suh[†§*]
[†]Cornell University, Ithaca, NY, USA
[§]Meta AI, Cambridge, MA, USA
{mu94,wh399,zhiruz,gs272}@cornell.edu,edsuh@fb.com

ABSTRACT

Trusted execution environments (TEEs) in processors protect off-chip memory (DRAM), and ensure its confidentiality and integrity using memory encryption and integrity verification. However, such memory protection can incur significant performance overhead as it requires additional memory accesses for protection metadata such as version numbers (VNs) and MACs. This paper proposes SoftVN, an extension to the current memory protection schemes, which significantly reduces the overhead of today's state-of-the-art by allowing software to provide VNs for memory accesses. For memory-intensive applications with simple memory access patterns for large data structures, the VNs only need to be maintained for data structures instead of individual cache blocks and can be tracked in software with low efforts. Off-chip VN accesses for memory reads can be removed if they are tracked and provided by software. We evaluate SoftVN by simulating a diverse set of memory-intensive applications, including deep learning, graph processing, and bioinformatics algorithms. The experimental results show that SoftVN reduces the memory protection overhead by 82% compared to the baseline similar to Intel SGX, and improves the performance by 33% on average. The maximum performance improvement can be as high as 65%.

CCS CONCEPTS

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

KEYWORDS

Trusted execution environment (TEE), Memory protection

ACM Reference Format:
Muhammad Umar, Weizhe Hua, Zhiru Zhang, G. Edward Suh. 2022. SoftVN: Efficient Memory Protection via Software-Provided Version Numbers. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3470496.3527378

*Work was done at Cornell University.

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527378

1 INTRODUCTION

Many emerging data-intensive applications consume private or sensitive data, which demand strong security protection. For example, machine learning (ML) algorithms often need to collect, store, and process a large amount of personal and potentially private data from users to train a model. Moreover, due to their high computational demand, these computations are often performed on a remote server in the cloud rather than a client device such as a smartphone. Unfortunately, in traditional computing systems, private user data may be exposed or misused by the remote server if it is either compromised or malicious.

A promising approach to providing strong confidentiality and integrity guarantees even under untrusted software and potential physical tampering is to rely on trusted hardware to create a hardware-protected execution environment. For example, Intel SGX [30] provides a trusted execution environment (TEE) called an enclave, which protects security-sensitive computation and data even from privileged software such as an operating system. The cryptographic protection of off-chip memory, namely memory encryption and integrity verification, represents an essential technology to enable the TEE. However, the off-chip memory protection also represents the main source of performance overhead in the traditional secure processor designs [12, 30, 44, 47]. Each cache block is encrypted before being written back to memory, and decrypted and verified on a read.

To hide decryption latency, TEEs often use counter-mode encryption, which allows cipher (AES) operations to be overlapped with memory reads. However, the counter-mode encryption requires a version number (VN) for each cache block. In order to handle arbitrary memory access patterns, VNs that count per-cache-block writes are typically stored in the main memory (DRAM). To protect the integrity of off-chip memory, either a message authentication code (MAC) or a cryptographic hash also needs to be attached to each cache block in memory. Moreover, to ensure freshness and prevent replay attacks, the integrity verification requires a tree of MACs. Unfortunately, the additional VN and MAC accesses can lead to significant performance overhead for memory-intensive workloads, even in presence of metadata caches. The depth of the integrity tree also limits the maximum size of protected memory, and makes protection difficult and more expensive for systems with a large DRAM.

In this paper, we propose a new memory protection scheme, named SoftVN, which enables low-cost memory encryption and integrity verification even for memory-intensive applications by allowing software inside a TEE to control VNs for a part of its protected memory space. We study more than 30 kernels across three classes of memory-intensive applications, and make the following

key observations. The memory-intensive kernels often have regular memory access patterns and update multiple data elements in a data structure in a similar fashion, resulting in the same number of writes to many data blocks in a data structure. This implies that a large number of data blocks have the same VN, which can easily be determined in software by counting the number of writes to the data structure. Moreover, the writes to these large data structures are often sequential, as how a deep neural network (DNN) kernel writes to output feature maps.

For data structures with regular write patterns, SoftVN enables software to provide the VN for reads by tracking the VN per data structure in software.[1] This removes the need to fetch and verify the off-chip VNs on reads, which accounts for the majority of today's memory protection overhead. Because write-backs can happen in the background, reads are more critical to performance. The software-provided VNs are still checked and written to off-chip memory in the background so that they can be used for mechanisms transparent to software such as cache write-backs. SoftVN also provides hardware support for providing VNs for reads and updating VNs for sequential writes so that these operations can be performed with small changes to software. Note that SoftVN allows reads to a data structure to be performed with an arbitrary access pattern even though it is designed for data structures with simple write patterns. Also, using SoftVN is optional. With SoftVN, software maps data structures in a way that the associated VNs can be easily tracked to the SoftVN region while keeping the rest of code and data in the memory space guarded by the traditional memory protection.

While conceptually straightforward, allowing software to control VNs for memory encryption and integrity verification introduces nontrivial challenges that need to be addressed through a careful hardware-software co-design. For example, one challenge comes from the granularity mismatch between load/store instructions and off-chip accesses. While software writes one word at a time, off-chip memory protection needs to ensure that all writes to a cache block are collected together so that the block is encrypted at most once with a new VN. The software-provided VNs also represent the number of writes from software instead of the number of writes to off-chip memory. To address these challenges, we introduce a Secure Memory Buffer (SMB) in a processor core and use virtual addresses (VA) instead of physical addresses (PA) for memory protection. Another challenge is efficiency; a new hardware-software interface needs to specify VNs at run-time with low overhead. To address this, we introduce a hardware VN table and a minimal ISA extension. For security, we also need to ensure that software-provided VNs do not introduce a vulnerability where an encryption counter is used multiple times for one memory block using the same key. This paper provides the details of how the software-provided VNs can be realized while providing both security and efficiency.

We study the memory access behaviors of a variety of applications, including deep neural networks (e.g., convolutional neural networks, recommender models, and language models), graph processing (e.g., BFS and PageRank), and bioinformatics (e.g., genome alignment), and show how the VNs for the large data structures can be tracked in software. By exploiting the software-provided VNs, we can greatly reduce the overhead for off-chip memory protection.

[1]Note that we use the term data structure to refer to a group of memory locations that share the same write patterns and the number of writes, such as a tile in a tiled matrix multiplication, not necessarily the entire software data structure.

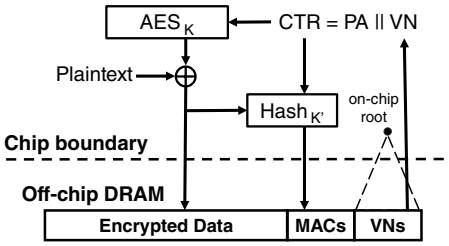


Figure 1: Traditional memory encryption and integrity verification.

To demonstrate the effectiveness of SoftVN, we evaluate the overhead of 12 representative benchmarks from three different categories of memory-intensive applications. The experimental results show that SoftVN can provide memory encryption and integrity verification with only 6% overhead on average. On the other hand, applying the existing memory protection schemes on data-intensive applications leads to 42% overhead on average, 86% in the worst-case (which is reduced to 14% with SoftVN) and even higher overhead with lower memory bandwidth. The proposed SoftVN scheme achieves average and maximum performance improvements of 33% and 65% respectively and an average overhead reduction of 82% over a baseline Intel SGX scheme.

2 BACKGROUND

2.1 Trusted Execution Environments

TEEs [1, 3–6, 11, 12, 24, 26, 30, 44, 45, 47, 50, 53] provide hardware-protected execution environments where confidentiality and integrity are ensured even under an untrusted OS or physical attacks. Intel's Software Guard Extension (SGX) represents one of the state-of-the-art trusted computing designs. SGX establishes a secure environment called an enclave, and a remote user can offload confidential computation and data into the enclave.

SGX reserves a special memory region for each enclave and protects that memory region from all non-enclave memory accesses, including accesses from an OS kernel, a hypervisor, and peripherals. The confidentiality, integrity, and freshness of the special region in DRAM is protected by the Memory Encryption Engine (MEE) [14]. However, SGX's threat model excludes side-channel attacks such as power analysis and memory side-channel attacks. In the next section, we provide details on today's off-chip memory protection scheme employed in SGX and point out the main sources of overhead.

2.2 Memory Protection

As shown in Figure 1, existing memory protection techniques [14, 17, 44] leverage symmetric key cryptography to ensure the confidentiality of data stored in off-chip memory. Prior arts typically use the counter-mode encryption (AES-CTR) to hide AES latency. The AES-CTR mode 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

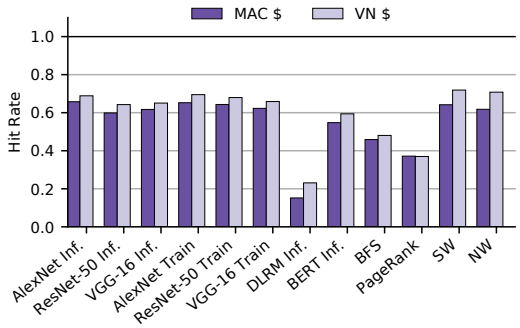


Figure 2: 32 KB metadata cache hit-rates for MACs and VNs.

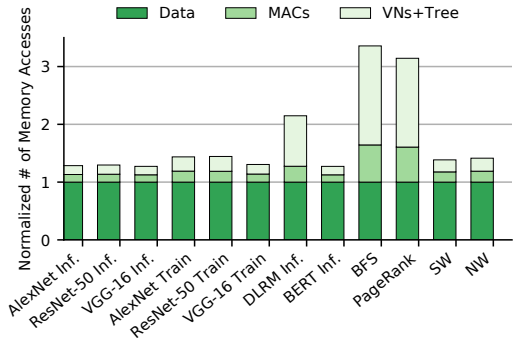


Figure 3: DRAM accesses (misses in LLC and metadata caches) by type, normalized to data accesses.

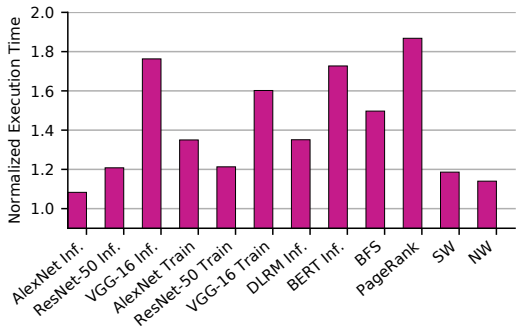


Figure 4: Traditional memory protection overhead.

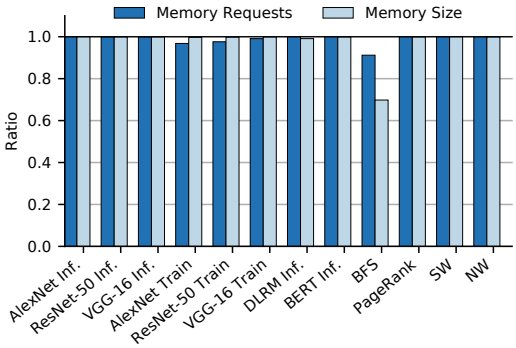


Figure 5: The percentage of off-chip data memory requests and working sets that can use software-provided VNs.

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 decrypts the block. Let K_{Enc}, U, V be the AES encryption key, plaintext, and ciphertext, respectively. The AES encryption can be formulated as $V = U \oplus AES_{K_{Enc}}(PA||VN)$, where $||$ and $\oplus$ represent bit-field concatenation and XOR, respectively.

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 reusing 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.

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 location 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) [13] is used to verify the MACs hierarchically in a way that the root of the tree is stored on-chip. As shown in Figure 1, a traditional memory protection scheme [36] 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. It is worth noting that MEE in SGX uses Carter-Wegman MAC [14] as the hash function. In this paper, we use AES-GCM [9] for both encryption and message authentication. Let us denote the key, plaintext, and ciphertext as K_{IV}, U, V, respectively. The MAC of an encrypted data block can be calculated as $MAC = H_{K_{IV}}(V||PA||VN)$.

2.3 Limitations of Today's Memory Protection

As the VNs are stored in off-chip memory, the integrity and freshness of VNs need to be protected with MACs. The performance overhead of the off-chip memory protection mainly comes from accessing the VNs and MACs, and traversing the integrity tree stored in DRAM. To mitigate this overhead, recently-used VNs and MACs are cached on-chip. However, even with dedicated metadata caches, today's memory encryption and integrity verification can lead to a significant slowdown. For memory-intensive applications, the VN/MAC caches often have low hit-rates (see Figure 2) due to capacity misses and low temporal locality. The memory protection also results in significant bandwidth and performance overhead. For example, metadata accesses increase the off-chip bandwidth usage by 2–3× for a recommender model (DLRM) and graph applications (BFS and PageRank) as shown in Figure 3, both of which involve

significant random reads which trigger traversal of an uncached part of the integrity tree. The performance overhead is also significant for large machine learning models and graph applications with low locality (see Figure 4).

In addition, for memory-intensive applications with a large working set, the size of the protected memory region is large, which leads to a proportionately deeper integrity tree, further increasing the performance overhead. For example, the working set for large recommender models such as variants of DLRM [32] can be over 1 TB. Supporting a protected memory region of this size requires a depth of 9 for the integrity tree. Compared to SGX with a protected memory size of 128 MB, the tree depth is more than doubled.

If memory protection can be provided without accessing VNs and their integrity-tree in off-chip memory during data reads, we can significantly reduce the performance overhead.

3 SOFTVN DESIGN

3.1 Intuition

While memory-intensive applications process a large amount of data, they tend to have well-defined access patterns to various data structures. We studied more than 30 computational kernels across three classes of memory-intensive applications (Section 4), and observed the access patterns.

We found that write patterns in kernels are usually regular (sequential) and same for many elements in a data structure. For example, kernels such as matrix-multiply and activation in a DNN layer update all elements of a tile in an output feature map sequentially the same number of times. This is indicative of a general trend in such applications to perform similar operations on a number of data elements stored together in data structures. As such, all elements within a data structure such as a tile in a tiled matrix multiplication can share the same VN. Note that we use the term 'data structure' to broadly refer to a software-managed group of memory locations that share the same write pattern, not necessarily the entire software data structure. In most cases, the size of a data structure is much larger than a cache line, and the number of data structures is much smaller than the number of cache-line-sized blocks in memory. The overhead of memory protection can be reduced significantly if a VN is tracked per data structure instead of per cache block. Due to sequential writes, software can easily compute the VN per data structure, such as a feature map, by keeping a counter for each data structure.

Moreover, some data structures are read-only within a kernel and use one (constant) VN that was last used to write them, even if their read pattern is arbitrary. For example, the model parameters in a DNN inference are read-only after initialization, and only need a single constant as the VN. They may be read sequentially or even randomly. Similarly, input feature maps are read (possibly multiple times) by kernels, and can use one VN within a kernel.

We also note that whereas both cache reads and write-backs access off-chip VNs in traditional protection, only the reads are on the critical path for a program's progress; write-backs can take place in the background. In that sense, optimizing reads is more important for performance.

Leveraging these observations, we propose to optimize off-chip memory protection by allowing software to provide VNs for a portion of its memory space, during *memory reads*. We call this memory protection scheme SoftVN. In SoftVN, the off-chip VNs are determined by software, reflecting the number of writes by software, and are no longer arbitrary as a result of unpredictable cache eviction in traditional memory protection. Software provides the VN for reads from the SoftVN region; hence, the VN fetch and integrity tree traversal during reads are avoided, and the counter-mode decryption can begin much sooner, improving performance. For writes, a new write VN per cache block (usually the read VN+1), is calculated by hardware and stored in memory. This VN still has to be stored off-chip for mechanisms transparent to software execution, e.g. a cache line eviction, paging and prefetching. However, the updates to the off-chip VNs are performed in the background, via the metadata cache, which should have a high hit-rate as the writes are sequential. Recall from Section 2.3 that the VN metadata cache originally had a low hit-rate, due to capacity misses from large data structures or random reads. For writable regions, SoftVN improves performance by using software-provided VNs to avoid VN accesses on a memory read, deferring the VN accesses to the background for eventually updating/incrementing. For read-only data regions, the off-chip VN accesses are completely eliminated, reducing pressure on the metadata cache.

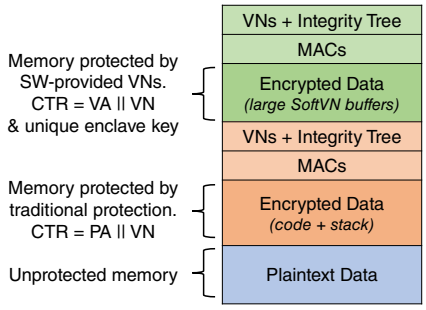


Figure 6: Memory regions in a system with SoftVN.

In a TEE with SoftVN, software inside the TEE determines how to protect different types of data, with traditional or SoftVN protection (Section 3.2). Similar to the way that software inside an enclave is trusted not to output secrets and responsible for protecting secrets from side channels in today's TEEs, software also needs to provide correct VNs for reads for functional correctness. Note that the integrity of an application's code and control data are still protected with the traditional memory protection so that software can be trusted to provide intended VNs.

We studied a diverse range of applications (see Section 4). As shown in Figure 5, we found that these applications can provide VNs to major data structures that account for a large portion of off-chip memory accesses and space.

3.2 Overview

SoftVN is proposed as an extension to the traditional memory protection scheme shown in Figure 1. The enclave's virtual (and physical) address space now contains three distinct regions as in Figure 6. The base protection region uses traditional memory protection, which can be used for code and data whose VNs are difficult to generate in software. A new region, the SoftVN region, is intended to store large data structures whose VNs can be assigned

and tracked in software. An enclave application can choose which memory protection is needed for its data by placing them in an appropriate virtual address (VA) range. The address ranges are configured at the initialization, included in attestation, and fixed for each enclave. The rest of the memory space is outside of an enclave and used for I/O.

Figure 6 shows that the SoftVN region also has MACs and off-chip VNs. The VNs are still stored in off-chip for mechanisms transparent to software, such as cache eviction to DRAM, paging and prefetching. However, the VNs for memory reads in this region come from software, and a data block can be decrypted without accessing VN in off-chip memory. Note that the SoftVN region uses a VA with a VN as a counter (CTR) for encryption so that the virtual-to-physical mapping does not affect the encryption counter. Also, SoftVN uses a different AES key for each enclave to avoid leakage between enclaves.

3.3 Challenges

While the concept of using software-provided VNs is relatively straightforward, exposing memory encryption to software introduces a set of new technical challenges that need to be addressed to realize the idea in modern processor designs.

In SoftVN, software uses a common VN for multiple memory blocks with the same write pattern to efficiently maintain VNs[2]. However, this design makes an implicit assumption that writes to one cache block are collected together, and the entire cache block is encrypted at most once with a given VN. Unfortunately, in modern processors with caches, a cache block may be evicted and written back to memory at any time. For example, consider the case where an application sequentially updates an entire cache block. From the application's perspective, this cache block is updated only once, and the VN only needs to be incremented by one. However, during writes to the cache block, a partially-modified block may be written back to memory multiple times. For security, software needs to be able to ensure that multiple writes to one cache block can be performed without an eviction. SoftVN introduces an in-processor buffer for this purpose.

In SoftVN, VNs only reflect the number of writes from software within each enclave, and the same pair of a memory address and a software-provided VN may be used by multiple enclaves. Also, in enclaves, an untrusted OS is responsible for paging and the virtual-to-physical memory mapping can change over time. Therefore, the traditional counter-mode construction that uses a concatenation of a physical address and a VN may not be unique and cannot be used with software-provided VNs. To address these problems, SoftVN uses a different AES key for each enclave and uses virtual instead of physical addresses in constructing its encryption counter.

The software-provided VNs imply that software needs to specify a VN for each memory access instruction, which can be an expensive proposition if VNs need to change frequently. For efficient memory protection, a new hardware-software interface is needed that allows software to specify VNs with low overhead i.e. a small number of additional instructions, and minimal changes to existing code. In SoftVN, we leverage the observation that many memory locations share one VN, which only needs to be updated infrequently, and propose new instructions and hardware support that allows the usual load/store instructions to be used with no change.

[2]This construction is also allowed in today's memory encryption, and still secure because a VN is concatenated with a memory address to form a counter value for memory encryption.

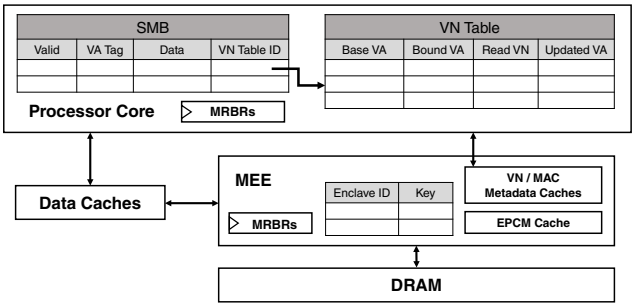


Figure 7: Hardware structures in SoftVN. An SMB, a VN table, and a per-enclave key table are added for SoftVN.

3.4 Hardware Architecture

SoftVN needs a small set of new hardware modules to support encryption using software-provided VNs and virtual addresses (VAs) as shown in Figure 7. The key metadata and operations of SoftVN are discussed below and depicted in Figure 8.

We assume that memory region boundary registers (MRBRs) exist inside both the core and the MEE, and contain the physical address ranges for the three memory regions in Figure 6. These registers are used to identify the region of a memory load/store both inside the core and the MEE.

In our design, software tracks VNs for large memory buffers. To allow software to efficiently specify a VN for loads, we introduce a hardware VN table inside the core, which stores VNs for VA ranges, has a fixed number of entries, e.g. 16, and is explicitly manipulated by software with new instructions. On a read from the SoftVN region, the VN table is used to determine the VN, which is used to decrypt a data block if it needs to be fetched from off-chip memory. The size of the VN table is known to software as an architecture parameter.

To avoid an unpredictable eviction during multiple writes to a cache block, we also place a small buffer called the Secure Memory Buffer (SMB), between each core and its L1 data cache. The SMB contains cache blocks in the SoftVN region while a processor writes to them. Typically, kernels write to only a small number of buffers at a time. The SMB only needs to hold one cache line per data structure being written. For the kernels that we tested, a 256-B SMB that can support up to 4 64-B *slots* for different data structures that are simultaneously updated is sufficient. To software, SMB appears as a fully-associative L0 cache that is accessed with a VA; software simply uses regular load/store instructions to access VAs that are loaded into SMB.

The VN table is first populated with the VA ranges for all the buffers that a software kernel needs to use with software-provided VN; then an output buffer from those ranges is mapped to an SMB slot for writes. The SMB slot points to the corresponding entry of the VN table, which is used to access the read VN for loading data into SMB. At the end of a kernel, software clears the VN table entries and the SMB slots that are in use.

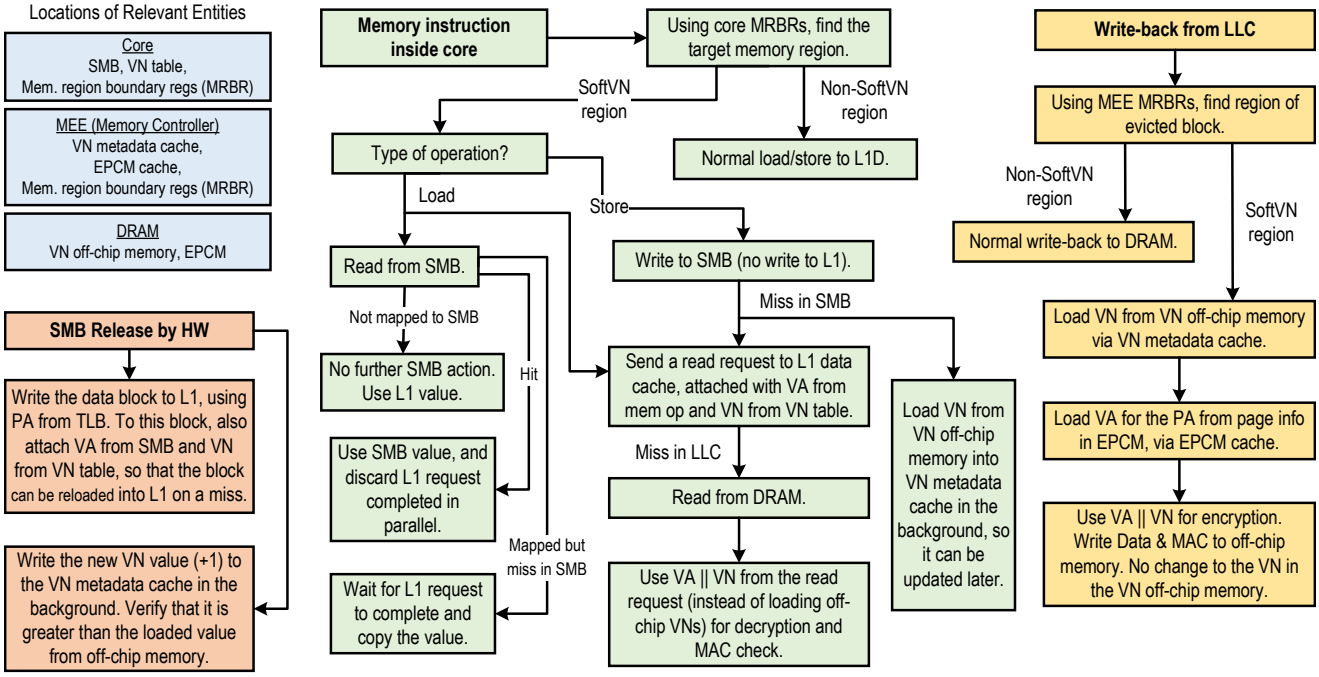


Figure 8: The overview of the SoftVN metadata and operations (load/store operations, SMB releases, LLC write-backs).

SMB exploits the sequential write pattern of typical memory-intensive kernels to support writes with minimal software changes. To modify data blocks in the SoftVN region, software maps a VA range of an entire output buffer to one of the slots in the SMB. Then, hardware automatically loads, via the regular memory hierarchy (using a VN from the VN table and VA from SMB as part of the memory request), one cache block into SMB on a write to that block, and writes (releases) the cache block to the L1 cache upon a write to another block in the same output buffer. Thus, software does not need to explicitly load and unload the SMB for each cache line in the output buffer, except for the very last block that is released explicitly by software. Also, whereas loading the SMB from memory (i.e. write-allocate) uses a VN from the VN table in the memory request, the actual off-chip VN also needs to be eventually loaded, so it can be updated on an SMB release (see below). As an optimization, as soon as SMB loads a new cache line, its VN to be updated is fetched into the metadata cache in the background so it may hit in the metadata cache by the time of an SMB release.

When a data block is released (evicted) from SMB, it is written to the L1 cache. To cater for misses in the L1 cache, the block VA from SMB and the VN from VN table are also provided, so the block can be reloaded if required. At the same time, a new VN is calculated by incrementing the read VN (read VN+1), and this overwrites the VN value loaded in the VN metadata cache in the background. To detect software bugs that reuse one VN multiple times for one location, the metadata cache raises an exception if the new VN is not greater than the VN loaded from off-chip memory for the given memory location. Note that instead of cache eviction incrementing the VN as in traditional protection, releasing a block from SMB increments the VN. Later, write-backs to the SoftVN region in off-chip memory just read the updated VN and use it for encryption. An SMB release also updates the corresponding VN table entry to keep track of the latest VA that was released, in the 'Updated VA' attribute. This is necessary for some kernels which re-read an earlier part of the same sequentially-written output buffer, which needs to use VN + 1 for reads.

For processor writes in the SoftVN region (detected via core MRBRs), only the SMB is accessed; whereas for reads in this region, data can be present in either SMB or the L1 cache. Hence, for reads, L1 is accessed in parallel to SMB; the L1 read request also contains a VN from the VN table and a VA from the memory instruction. The SMB value takes precedence if found; see Figure 8 for details.

The number of SMB slots is an architecture parameter known to software. It is the software's responsibility to keep track of the available slots. If software tries to create more mappings than slots, an exception is raised, which results in a termination of the enclave. Similarly, a write to the SoftVN region that is not mapped into SMB raises an exception.

Both the VN table and SMB are considered a part of the processor's architecture state, and tied to the enclave environment. Their contents need to be saved/restored on a context switch. For example, Intel SGX stores an enclave process state in the State Save Area (SSA). The additional data size is similar to existing data that may optionally be saved, such as AVX registers.

As explained earlier, MEE needs VAs in the CTR for memory transactions in the SoftVN region (detected using MEE MRBRs). For decryption in MEE, reads can use the VA sent from the core with the memory request to the caches. For encryption, cache write-backs from the LLC to off-chip memory need to determine a VA separately. Fortunately, a TEE already needs to store VAs to verify

Instruction	Arguments
SoftVN_SetVN	Table-ID, VA, Length, Read-VN
SoftVN_InvalidateVN	Table-ID
SoftVN_Map	SMB-Slot, Table-ID

Table 1: Extensions to the ISA.

address translation if the translation is controlled by an untrusted OS. For example, SGX stores the VAs in the Enclave Page Cache Map (EPCM), located in the off-chip memory protected by traditional memory protection. The EPCM also contains page ownership, permission and status bits for each protected page. In our design, we use VAs from the EPCM for encryption. For efficiency, the EPCM entries are cached on-chip inside the MEE. We use a 128-bit EPCM entry per page, estimated from SGX documentation [19]. When a page is swapped out from an enclave, paging first flushes the page from caches before updating the EPCM so that write-backs use correct VAs. Also, the software-provided VNs in memory follow the corresponding data on paging.

SoftVN requires each enclave to use a different AES key for encrypting its SoftVN region. To support per-enclave encryption key, the MEE is extended with a table that stores a key for each enclave ID.

3.5 Instruction Set Extension

We introduce new hardware instructions to manage the SMB and the VN table. Table 1 summarizes the new instructions. Note that both the VN table and SMB instructions require ID/slot numbers for explicit management. The arguments to these instructions are passed via general purpose registers.

Software uses `SoftVN_SetVN`, which sets an entry in the VN table, to specify a VN that should be used to read a VA range. To read data in the SoftVN region, an enclave program simply uses regular load instructions, which use a VN from the VN table to read data from off-chip memory.

To modify a data region, software uses `SoftVN_Map` to map one of the address ranges in the VN table into an SMB slot. On a write to this range, the corresponding cache line is loaded into SMB through the memory hierarchy, using the VN from the linked VN table entry. There is no explicit UnMap instruction; the SMB automatically releases a cache line upon a write to a different line in the same address range.

When an address range is no longer in use, software releases the VN table entry with `SoftVN_InvalidateVN`. This also releases the *last* cache line in SMB and clears the SMB slot.

The top function in Figure 9 is the original code of a kernel that adds two arrays element-wise and stores the result in one of the arrays. The bottom function of the figure shows the kernel modified for SoftVN. The read VNs represent the value last used to write data to the arrays. The only code changes come from setting/clearing the VN table and mapping the output buffer to SMB for writes. The read VN is also incremented to reflect a write.

3.6 Implications of SoftVN

3.6.1 Programmability. In SoftVN, software has the additional responsibility of keeping track of the correct VN for its data structures in the SoftVN region. Ideally, the software-provided VN should be

```
// original kernel code
void Add_Array_Elemwise(float *x, float *y, const int N)   {
    for (int i = 0; i < N; ++i)
        x[i] += y[i];
}
// modified kernel code for use with SoftVN
void Add_Array_Elemwise(float *x, float *y, const int N,
      uint64_t *VN_x, uint64_t *VN_y)    {
   // set read VN Table ID 0
    SoftVN_SetVN(0, y, N * sizeof(float), *VN_y);
    // set read/write VN Table ID 1
    SoftVN_SetVN(1, x, N * sizeof(float), *VN_x);
    // map Table ID 1 to SMB slot 0
    SoftVN_Map(0, 1);
    for (int i = 0; i < N; ++i)
        x[i] += y[i];
   // free VN Table ID 0
    SoftVN_InvalidateVN(0);
   // free VN Table ID 1, also frees SMB slot 0
    SoftVN_InvalidateVN(1);
    // update the original VN
    *VN_x += 1;
}
```

Figure 9: Example kernel code and its changes for SoftVN.

designated at a coarse granularity to reduce the number of VNs that software needs to track. For applications with a small number of buffers, programs can easily track VNs by storing them in local variables. For more complex applications, programs can use a separate data structure such as a dictionary to keep track of VNs for different data structures. Note that applications can still use the traditional protection for data structures whose VNs cannot be efficiently maintained in software.

We found that data-intensive kernels usually operate on a small number of data structures at a time. In the kernels we studied, no more than 2 outputs are written simultaneously using the SMB (out of 4 possible), and no more than 6 (out of 16) VN table entries are used at a time. Moreover, we found that scaling up the workload does not necessarily require more SMB or VN table entries; e.g. while the size of data structures grow, larger ML models do not necessarily use more data structures at a time, i.e. the sizes of the SoftVN hardware structures do not need to increase for bigger workloads.

SoftVN introduces some small constraints for programming. For example, buffers in the SoftVN memory region need to be aligned to cache line boundaries, which can be achieved through a modified `malloc`. As the VN table has a limited number of entries, each kernel also needs to limit the number of data structures that use SoftVN at the same time.

We modified the data-intensive kernels in our experiments, and found that the changes required for software to use SoftVN are fairly small and straightforward for typical kernels. As shown in Table 2, for the applications that we studied (Section 4), the number of lines of code that are added is very small, so is the increase in dynamic instructions. Note that these software changes only need to be made to a part of the program, i.e. data-intensive kernels that represent majority of off-chip data accesses. The changes to many different kernels are similar, e.g. specifying the read VNs and mapping the output buffers. We believe that these changes can be automated by a compiler. Also, for most workloads, these modified kernels can be a part of well-vetted libraries or frameworks, and would not have to be re-invented and re-verified by individual programmers.

Application	Kernel Count	Kernel Examples	# LoC Added	SoftVN Memory Working Set Size	Example VN Increments	Max. SMB & VN Table Usage
DNNs Inf./Train (CNNs, DLRM, BERT)	28	GEMM, Im2Col, Pool, Activate, Normalize etc. Backprop versions of these	6 per kernel on average	222 MB in ResNet Inf. to 15 GB in DLRM	+1 for output feature maps per kernel write, fixed VN for weights during inference	1/4 & 6/16
BFS	1	Breadth-First Search	6	231 MB	Fixed VN for read-only input graph	0/4 & 3/16
PageRank	1	PageRank	13	272 MB	+1 for rank vector per iteration	1/4 & 5/16
SW	1	Local Alignment	10	477 MB	+1 for output matrices per sequence alignment	2/4 & 4/16
NW	1	Global Alignment	10	477 MB	+1 for output matrices per sequence alignment	2/4 & 4/16

Table 2: Kernels in example applications.

3.6.2 Memory Allocation. In SGXv1 [30], the maximum workload (heap) memory size has to be specified at enclave creation time. Similarly, in SoftVN, the memory ranges for the software-provided VNs and the traditional memory protection are specified during initialization. Then, a programmer can specify which memory protection is used for each data structure by placing it in the corresponding virtual addresses. In our experiments, we found that many memory-intensive applications allocate their buffers initially and use them during an execution, which simplifies memory allocation. If heap memory in the SoftVN region needs to be freed and reused dynamically, the VNs for the reused memory locations must be carefully tracked by the program (or the enclave memory manager); it needs to keep track of the most recent (maximum) VN that was used for a freed memory chunk and avoid reusing the same VNs.

3.6.3 Multi-core Support. SMB is meant as a temporary space that holds data to combine multiple writes to a cache block into one memory write. In that sense, we treat SMB similar to registers from the cache coherence perspective, even though it keeps memory blocks; hardware does not provide coherence for data in SMB, and software is responsible for ensuring that different threads do not write to one cache block concurrently. We studied multi-threaded implementations of DNN models, and found that multiple threads process different data and write to different parts of output buffers.

3.6.4 SIMD Extensions. While we described our scheme for a scalar processor, SoftVN can be extended for SIMD extensions of cache line granularity, which use wide registers and explicitly move data between registers and memory. The VN table can be traversed to obtain the VN for reads. If SIMD registers match the width of a cache block, as in AVX-512, writes can be made without SMB, using the read VN+1.

3.7 Security Analysis

SoftVN assumes the typical threat model in today's TEE/enclave such as Intel SGX where software inside an enclave is trusted and needs to be written correctly to be secure; enclave software can output its data unencrypted, and is also responsible for protection against side channels [18]. Similarly, we assume that enclave software provides correct VNs.

Security when enclave software is correctly-written: When trusted enclave software provides correct read VNs, SoftVN provides confidentiality and integrity protection that is equivalent to that of the traditional memory protection.

Confidentiality: The counter-mode encryption ($V = U \oplus \mathrm{AES}_K(\mathrm{CTR})$, CTR= VA||VN) is secure if each encryption uses a unique CTR value for a given key (K). In SoftVN, each enclave uses a different key (K), and SMB provides a new VN (read VN+1) on each write (release) of a cache block. The new VN is updated/stored in memory and used for encryption on a write-back to memory. The VNs are protected by an integrity tree in memory and MACs on paging, as in SGX. If the read VN is correctly provided by enclave software, CTR is unique for each write of a cache block.

Integrity: The integrity of a read from the SoftVN region is checked by the MAC ($\mathrm{H}_K(V||\mathrm{VA}||\mathrm{VN})$), similar to the traditional protection except the key is per-enclave and the address is VA instead of PA. Without knowing the MAC key, an adversary cannot change/generate a value (V). Relocating V to another enclave or address is detected through the per-enclave key and VA. VN prevents replays; for replay attacks, a VN from enclave software needs to be replayed as well. SoftVN still uses the traditional memory protection for code and other data including software-provided VNs so that the integrity of enclave software and VNs are protected.

Information leak through VNs: VNs are per-enclave, and one enclave cannot access VNs for other enclaves.

Implication of bugs in enclave software: While not necessary for correctly-written enclave software, SoftVN performs checks to detect software bugs and reduce security risks.

Confidentiality: The metadata cache in SoftVN checks to ensure that the new VN on a write (SMB release) is greater than the VN from the last write to the same address. If buggy software provides a stale VN, this check raises an exception before the VN is re-used for encryption. Thus, a write VN is guaranteed to be unique for each address, and confidentiality is ensured even if VN management in enclave software is buggy.

Integrity: Both substitution and relocation attacks are detected by the MAC, independent of VNs from enclave software. If buggy software provides an incorrect VN for a read, a MAC check still detects the bug in most cases. If the VN was never used before or does not match the current content in memory, the MAC check will fail. The bug is not detected only when enclave software provides an old VN and an attacker rolls back memory to the matching stale value at the same time.

Note the SoftVN also raises an exception on functional bugs such as accesses to the SoftVN region without specifying a VN. In summary, SoftVN ensures confidentiality and substitution/relocation protection even when VN management in software is buggy.

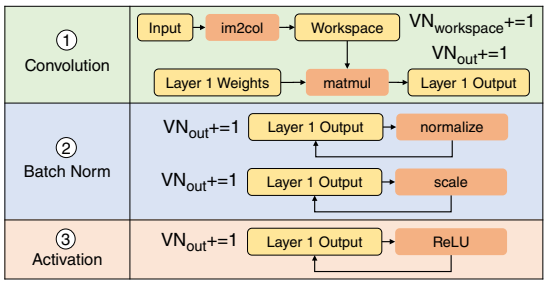
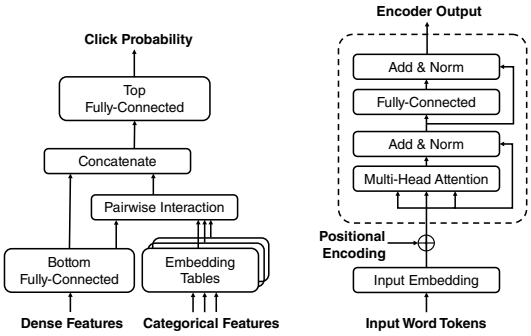


Figure 10: An example sequence of DNN layers with their kernels (orange) and processed data buffers (yellow).



(a) DLRM Recommender Model (b) BERT input & encoder block

Figure 11: Recommender and NLP model structures.

4 EXAMPLE APPLICATIONS

We studied a diverse range of applications to understand the applicability of SoftVN. Below we briefly describe each application and how we track its VNs.

4.1 Deep Neural Networks

DNNs provide state-of-the-art performance on a variety of ML tasks. DNNs consist of a succession of layers of different types, typically including convolution, fully-connected, and pooling. Each layer's computation calls one or more kernels from a DNN library, such as linear operation (e.g., matrix multiplication), matrix rearrangement (e.g., image-to-column), activation, etc. A DNN library also allocates a number of memory buffers to store the intermediate results (features) and network parameters (weights) of the layers. Each subsequent layer (and constituent kernels) processes the output of the earlier layers, and feeds its output to the next.

The memory access pattern of a DNN is typically deterministic, unless dynamic pruning is involved, or embedding table lookups are present (described shortly). Software can track the VNs for different buffers or tiles. The VN for sequentially written buffers such as feature maps is incremented once per write. Read-only buffers, such as weights during inference, can use a constant value as the VN. During training, weights are also updated sequentially. Figure 10 shows a back-to-back sequence of layers in a neural network, in which the buffer for output of Layer 1 is written sequentially four times over three layers, increasing its VN by 4.

SoftVN can also support multiple writes per kernel, such as in a tiled matrix multiplication, in which output tiles are written once for each input tile multiplication, and all tiles eventually share a single VN which is set for the entire matrix.

For our experiments, we use DarkNet [35], a DNN framework that supports many popular DNN models, and extended it as required for new models. Below, we describe variants of DNN models which we implemented.

4.1.1 Convolutional Neural Networks (CNNs). Mainstream ML models used to classify images are mostly based on CNNs, which typically consist of convolution, pooling, and fully-connected layers. Popular models include AlexNet [23], VGG [41], and ResNet [16]. We consider both inference over multiple runs and training of these networks over a few minibatches with a small batch size, on the ImageNet [7] dataset.

4.1.2 Recommender Models. Recommender models are widely used to provide personalized recommendations to online shoppers and social media users. Such models take both dense (numerical) and sparse (categorical) features, and predict the probability of a click on a suggested item. While the dense features are readily processed using fully-connected layers, the categorical features first have to be transformed into numerical features by indexing into large embedding tables, which may be hundreds of GBs in size. Since such embedding table lookups access random locations, they have low spatial locality and can benefit significantly from software-provided VNs. For inference, these embedding tables are read-only – they are only read after written once during loading. Therefore, we assign a single VN to all the embedding tables. For our experiments, we use DLRM [32] (Figure 11a) over a synthetic dataset.

4.1.3 Language Models. In recent years, Transformers [48] have become the state-of-the-art in natural language processing (NLP). We implemented BERT [8] inference as an example NLP application. A BERT model (Figure 11b) consists of only the encoder blocks from the Transformer architecture. BERT processes a long sequence of input word tokens at once, and uses attention and residual fully-connected layers. Our BERT model uses 4 encoder blocks, and 12 attention-heads per block.

4.2 Graph Algorithms

4.2.1 Breadth-First Search (BFS). BFS is a standard graph algorithm with a variety of uses. We consider a BFS task that traverses through an entire graph, and assigns a parent to each node and marks its distance relative to the root node. In our implementation, the read-only input graph adjacency list uses SoftVN. The BFS node queue is updated at a fine granularity and uses the traditional protection, as do the graph nodes which are visited and updated in a random order. For experiments, we performed BFS over an existing graph dataset [22] with 1.8 million nodes and 29 million edges.

4.2.2 PageRank. PageRank is a well-known algorithm used in ranking search results [34]. It takes a graph as input, and calculates the relative importance of each node using incoming node edges. The algorithm runs iteratively until the importance scores (or PageRank vector) converge. In our implementation, the read-only hyperlink and dangling node matrices derived from the input graph are assigned a single VN during a load. Two PageRank vectors are alternately used as previous and current outputs, and thus can be

assigned a VN equal to the iteration count. The elements of the previous PageRank vector are read randomly as the hyperlink matrix is sparse. For the experiments, we used the same existing dataset [22] as in BFS.

4.3 Bioinformatics

Genome alignment is an important memory-intensive task in bioinformatics. For local and global alignments, we study the Needleman-Wunsch (NW) [33] and Smith-Waterman (SW) [42] dynamic programming (DP) algorithms, respectively. In these algorithms, DP matrices are written sequentially after processing elements towards the above and left of the current element, and thus can be written to memory with a single output VN. For the experiments, we used two genome sequences with a length of 10K each.

5 EXPERIMENTAL RESULTS

5.1 Methodology

To evaluate the performance of SoftVN, we performed detailed simulations of the applications described in Section 4. We used ZSim [39], a cycle-level CPU simulator based on Pin [29], integrated with Ramulator [21]. The simulation parameters are shown in Table 3. We implement a scheme similar to Intel SGX MEE [14] as the baseline, which includes an 8-ary Merkle tree with 56-bit VNs and MACs per 64-byte cache line. Both the baseline and SoftVN memory protection engines are inserted between the LLC and the DRAM. AES-GCM mode is used to perform authenticated encryption in both schemes. Each memory request triggers additional metadata accesses. The memory protection engine is also augmented with VN, MAC and EPCM caches to store the most recently accessed metadata. We model 16 GB each for the base and SoftVN protected memory regions. The memory regions are large enough for our applications, and no paging is needed. We report results for only the main computation portion for each application e.g. forward-pass in a neural network, not the initialization.

Processor	3 GHz, Out-of-Order
ROB Depth	224
Issue/Retire Width	8 max. per cycle
Caches	L1-I/D: 32 KB, 4 ways, 4 cycle lat. L2: 1 MB, 8 ways, 12 cycle lat. L3: 8 MB, 16 ways, 27 cycle lat. 64 B cache lines
DRAM	DDR4-2400 32 GB 1 Channel × 4 Ranks Read/Write Queue Size: 32
MAC/VN $	32 KB each, 4 ways
EPCM $	4 KB, 4 ways
AES	128-bit, 40 cycle latency
GF Multiply	8 cycle latency

Table 3: Simulation parameters.

For each application, we simulate three modes: a system with no memory protection (NP), an Intel SGX-like baseline protection scheme (BP), and one with SoftVN (BP+SoftVN).

5.2 Performance Overhead

Figure 12 shows the execution time of both BP and SoftVN, normalized to no protection (NP). The performance overhead for BP is substantial compared to NP. The usage of software-provided VNs leads to average and maximum performance improvements of **33%** and **65%** respectively, and an average overhead reduction of **82%** over the baseline. On average, SoftVN incurs only **6%** performance overhead, compared to 42% of the baseline. The worst case BP overhead of 86% is reduced to 14% in SoftVN. Figure 13 depicts each application's MPKI (miss-per-kilo-instructions) for the LLC. The BP overhead is higher for applications with a higher LLC miss-rate per instruction, as more LLC misses lead to off-chip accesses. SoftVN can achieve low protection overhead even for memory-intensive applications with a large number of off-chip accesses, which depend on the working set size, computations and the memory access pattern of the underlying kernels, as long as most application memory buffers can use software-provided VNs. Moreover, the VN metadata accesses are reduced by **66%**. The accesses to the EPCM region for reading the VA introduce only a negligible amount of metadata, as the EPCM cache has a very high hit rate, since it stores coarse-grained per-page base VAs.

The benefit of using software-provided VNs is more pronounced for workloads with considerable memory traffic. This is especially true for applications with random (low locality) read accesses such as BFS and PageRank. For random accesses, metadata caches experience a large number of misses as shown in Figure 2, and the MAC tree needs to be traversed from a leaf to a higher-level node on each memory access. The performance improvement for BFS is lower compared to PageRank because BFS uses traditional memory protection for a sizeable portion of its working set (see Figure 5). On the other hand, workloads with high spatial locality such as DNNs show *relatively* high hit-rates for metadata caches. However, if the working set is large, then poor temporal locality in the metadata caches can lead to significant overhead as in VGG-16.

Nevertheless, even in applications with a vast majority of VN accesses being eliminated, a small overhead still remains. This can be attributed to MAC accesses, which have to be fetched per cache line even for the SoftVN region.

5.2.1 Detailed Analysis of DNNs. As a case study, we look at the inference for the VGG-16 CNN using DarkNet. The normalized per-layer execution time for both BP and SoftVN is shown in Figure 14, along with the total size of the layer's working set (weights, input and output feature maps, temporary workspace). All these data buffers and all layers can use software-provided VNs. Figure 14 shows that different layers in the network contribute differently to the overall overhead. The convolution (`conv`) and fully-connected (`fc`) layers involve matrix multiplication of features and weights. The layers with a large working set show high BP overhead, as such sets do not fit in the cache, and also lead to more metadata cache capacity misses. Other layers such as pooling have a smaller working set and little overhead. The figure shows that SoftVN greatly reduces the overhead in the memory-intensive layers.

5.2.2 Sensitivity Analysis. We studied the effect of varying different parameters on performance for both BP and SoftVN.

LLC Size Figure 15 shows the effect on the execution time as the LLC size varies from 2 MB to 8 MB. The figure shows that both the BP and SoftVN overhead numbers increase as the LLC size decreases. It is interesting to note that for VGG-16 inference, the cache miss-rates are high even for 8 MB, so smaller LLC sizes

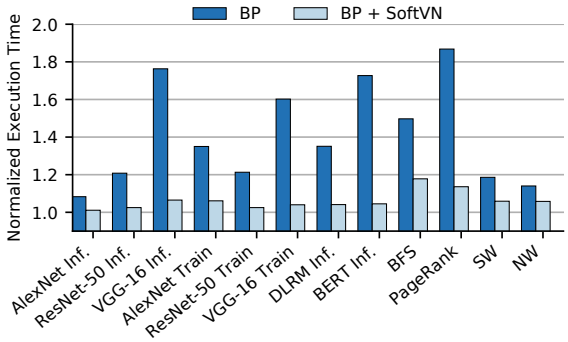


Figure 12: Normalized execution time for BP and SoftVN.

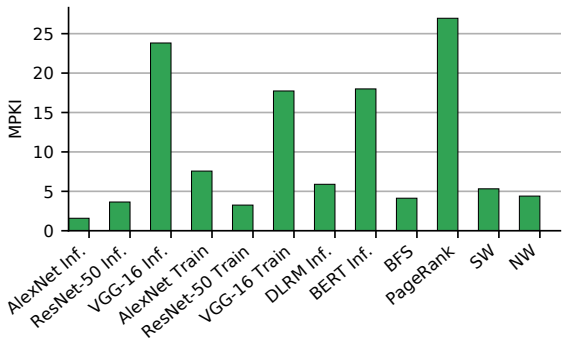


Figure 13: The MPKI for the LLC in each of the applications.

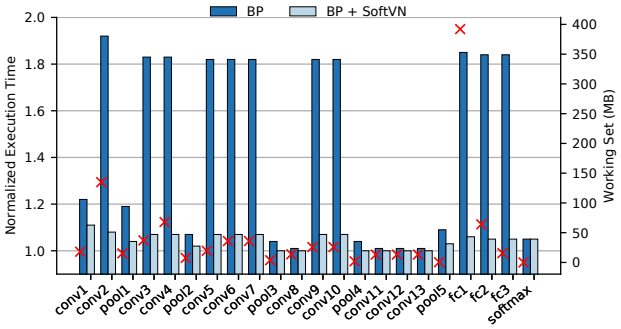


Figure 14: VGG-16 inference per-layer normalized execution times, with working set size marked as ×.

do not increase the overhead significantly, whereas overhead drastically increases for AlexNet at smaller LLC sizes. However, the SoftVN overhead is quite low even for smaller LLC sizes, which suggests a potential to provide even more significant benefits for cases when multiple programs concurrently run and share the LLC and memory channels. Note that the SoftVN overhead increases noticeably for BFS when the cache size is reduced. This is because these applications still rely on the traditional memory protection for a sizeable portion of their data.

Metadata Cache Size Figure 16 shows the execution time when the metadata cache size is varied from 32 KB to 256 KB each for MACs and VNs. The overhead for most applications does not change noticeably due to the large working sets. We separately tested and found that a very large metadata cache size, ranging in megabytes, is needed to achieve noticeable overhead reduction in most applications. Given that there is limited area available for the memory protection engine, significantly increasing metadata caches will be difficult in practice. Using software-provided VNs for reads is an effective way to reduce the performance overhead without requiring large metadata caches.

5.3 Area and Energy Overhead

The hardware overhead of SoftVN is minimal as the new hardware structures are small. SoftVN adds an SMB with 4 64-byte entries and a VN table with 16 32-byte entries per core, which add only a 2.3% area overhead relative to L1 using CACTI [2] estimates. In addition, the MEE has a new 4 KB EPCM cache per chip. Moreover, a significant reduction (66%) in the number of off-chip accesses for metadata (VNs) leads to considerable off-chip energy savings.

6 RELATED WORK

Our scheme builds upon existing processor memory protection mechanisms and optionally enables applications to remove VN accesses by providing VNs in software.

The previous designs for memory encryption [43, 52] use the counter-mode and smaller VNs to optimize memory encryption and tree representation. For integrity verification, several recent efforts [10, 15, 36] propose a counter-based integrity tree to improve efficiency. Prior work [13, 25] also proposes caching metadata to exploit metadata locality. Alternative designs [28, 40, 51] propose to reduce the latency of integrity verification by predicting VN or using a unverified VN speculatively. Various integrity-tree optimizations have been proposed. VAULT [46] proposes variable arity in different levels to cater for different update frequency, and MACs over multiple blocks to lower the overhead. Morphable Counters [37] further reduce the overhead by compressing the counter representation dynamically and using an even denser 128-ary hash tree. Synergy [38] proposes to store MACs inline in ECC DRAM to avoid the extra MAC fetch access. While the previous studies focus on optimizing VNs stored in memory, this paper proposes to expose VNs to software and remove the need to wait for VNs from off-chip memory on reads.

Recent studies [20, 49, 54] propose to extend TEEs to GPUs. GPUs read and write from the device memory to the local memory during and after computation. Recently, researchers observed that many memory locations share the same counter value in GPU kernels, and number of distinct counters is small, and so proposed to scan the memory to discover regions with uniform counters and lower the overhead by caching common counters on-chip [31]. In TNPU [27], a similar observation regarding memory location inside a tensor sharing a single VN was used to propose a tree-less secure neural accelerator with software-managed VNs. The high-level observation that many memory locations share a VN is similar to ours. However, these approaches target different hardware i.e. GPUs and accelerators, whereas we study CPUs with virtual memory and transparent data caches. In our work, we show how exposing the memory protection to software and a careful SW-HW co-design

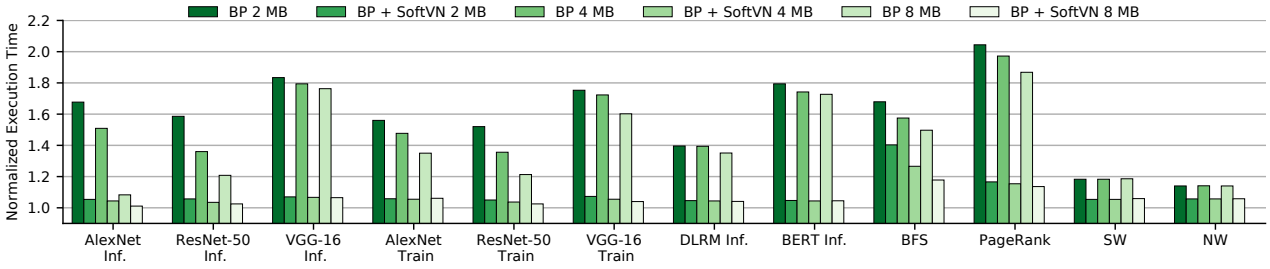


Figure 15: Normalized execution times as the LLC size varies from 2 MB to 8 MB.

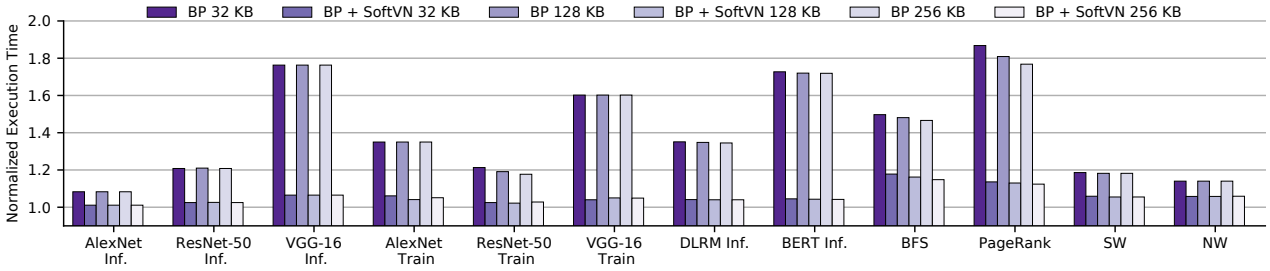


Figure 16: Normalized execution times as the metadata cache size varies from 32 KB to 256 KB each for MACs and VNs.

can enable memory protection with low overhead by efficiently leveraging common VNs across a large number of cache blocks even for CPUs.

7 CONCLUSION

In this paper, we propose SoftVN, a memory protection scheme that allows data-intensive applications to use software-provided VNs for accessing data structures whose VNs can easily be tracked in software, thereby reducing VN fetch overhead. We analyzed a number of different applications, and demonstrated a significant reduction in overhead for these when using SoftVN, relative to traditional memory encryption and integrity verification.

8 ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive feedback. 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.

REFERENCES

[1] Thaynara Alves and D. Felton. 2004. *Trustzone: Integrated Hardware and Software Security.* White Paper. ARM.

[2] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM TACO* 14, 2, Article 14 (Jun 2017), 25 pages. https://doi.org/10.1145/3085572

[3] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. 2019. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 42–56. https://doi.org/10.1145/3352460.3358310

[4] David Champagne and Ruby B. Lee. 2010. Scalable architectural support for trusted software. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 1–12. https://doi.org/10.1109/HPCA.2010.5416657

[5] Siddhartha Chhabra, Brian Rogers, Yan Solihin, and Milos Prvulovic. 2011. SecureME: A Hardware-Software Approach to Full System Security. In *Proceedings of the International Conference on Supercomputing* (Tucson, Arizona, USA) *(ICS '11)*. Association for Computing Machinery, New York, NY, USA, 108–119. https://doi.org/10.1145/1995896.1995914

[6] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 857–874. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. *Conf. on Computer Vision and Pattern Recognition (CVPR)* (2009), 248–255.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[9] Morris J. Dworkin. 2007. *SP 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC.* Technical Report. Gaithersburg, MD, USA.

[10] Reouven Elbaz, David Champagne, Ruby B. Lee, Lionel Torres, Gilles Sassatelli, and Pierre Guillemin. 2007. TEC-Tree: A Low-Cost, Parallelizable Tree for Efficient Defense Against Memory Replay Attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2007*, Pascal Paillier and Ingrid Verbauwhede (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 289–302.

[11] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley. 2014. Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 190–202.

[12] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. 2012. A Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing* (Raleigh, North Carolina, USA) *(STC '12)*. ACM, New York, NY, USA, 3–8. https://doi.org/10.1145/2382536.2382540

[13] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. 2003. Caches and hash trees for efficient memory integrity verification. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* 295–306. https://doi.org/10.1109/HPCA.2003.1183547

[14] S. Gueron. 2016. Memory Encryption for General-Purpose Processors. *IEEE Security Privacy* 14, 6 (Nov 2016), 54–62. https://doi.org/10.1109/MSP.2016.124

[15] W. Eric Hall and Charanjit S. Jutla. 2006. Parallelizable Authentication Trees. In *Proceedings of the 12th International Conference on Selected Areas in Cryptography*

(Kingston, ON, Canada) *(SAC'05)*. Springer-Verlag, Berlin, Heidelberg, 95–109. https://doi.org/10.1007/11693383_7

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[17] Michael Henson and Stephen Taylor. 2014. Memory Encryption: A Survey of Existing Techniques. *ACM Comput. Surv.* 46, 4, Article 53 (Mar 2014), 26 pages. https://doi.org/10.1145/2566673

[18] Intel Corporation. 2017. Intel SGX and Side-Channels. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sgx-and-side-channels.html

[19] Intel Corporation. 2021. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3D: System Programming Guide, Part 4. (Jun 2021).

[20] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. 2019. Heterogeneous Isolated Execution for Commodity GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 455–468.

[21] Y. Kim, W. Yang, and O. Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (2016), 45–49.

[22] Christine Klymko, David F. Gleich, and Tamara G. Kolda. 2014. Using Triangles to Improve Community Detection in Directed Networks. In *The Second ASE International Conference on Big Data Science and Computing, BigDataScience*.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (Lake Tahoe, Nevada) *(NIPS'12)*. Curran Associates Inc., USA, 1097–1105. http://dl.acm.org/citation.cfm?id=2999134.2999257

[24] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*.

[25] J. Lee, T. Kim, and J. Huh. 2016. Reducing the Memory Bandwidth Overheads of Hardware Security Support for Multi-Core Processors. *IEEE Trans. Comput.* 65, 11 (Nov 2016), 3384–3397. https://doi.org/10.1109/TC.2016.2538218

[26] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Zhenghong Wang. 2005. Architecture for protecting critical secrets in microprocessors. In *32nd International Symposium on Computer Architecture (ISCA'05)*. 2–13.

[27] Sunho Lee, Jungwoo Kim, Seonjin Na, Jongse Park, and Jaehyuk Huh. 2022. TNPU: Supporting Trusted Execution with Tree-less Integrity Protection for Neural Processing Unit. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA) (HPCA)*.

[28] T. S. Lehman, A. D. Hilton, and B. C. Lee. 2016. PoisonIvy: Safe speculation for secure memory. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783741

[29] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.* 40, 6 (June 2005), 190–200. https://doi.org/10.1145/1064978.1065034

[30] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (Tel-Aviv, Israel) *(HASP '13)*. ACM, NY, USA, Article 10.

[31] Seonjin Na, Sunho Lee, Yeonjae Kim, Jongse Park, and Jaehyuk Huh. 2021. Common Counters: Compressed Encryption Counters for Secure GPU Memory. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 1–13. https://doi.org/10.1109/HPCA51647.2021.00011

[32] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). https://arxiv.org/abs/1906.00091

[33] Saul B. Needleman and Christian D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 48, 3 (1970), 443–453.

[34] L. Page, S. Brin, R. Motwani, and T. Winograd. 1998. The PageRank citation ranking: Bringing order to the Web. In *Proceedings of the 7th International World Wide Web Conference*. Brisbane, Australia, 161–172. citeseer.nj.nec.com/page98pagerank.html

[35] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.

[36] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. 2007. Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*. IEEE Computer Society, Washington, DC, USA, 183–196.

[37] Gururaj Saileshwar, Prashant J. Nair, Prakash Ramrakhyani, Wendy Elsasser, Jose A. Joao, and Moinuddin K. Qureshi. 2018. Morphable Counters: Enabling Compact Integrity Trees for Low-Overhead Secure Memories. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* (Fukuoka, Japan) *(MICRO-51)*. IEEE Press, 416–427.

[38] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi. 2018. SYNERGY: Rethinking Secure-Memory Design for Error-Correcting Memories. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 454–465. https://doi.org/10.1109/HPCA.2018.00046

[39] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 475–486.

[40] Weidong Shi and Hsien-Hsin S. Lee. 2006. ASE. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, Washington, DC, USA, 103–112.

[41] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1409.1556

[42] Temple F Smith, Michael S Waterman, et al. 1981. Identification of common molecular subsequences. *Journal of molecular biology* 147, 1 (1981), 195–197.

[43] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, Washington, DC, USA, 339–. http://dl.acm.org/citation.cfm?id=956417.956575

[44] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003. AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing. In *Proceedings of the 17th Annual International Conference on Supercomputing* (San Francisco, CA, USA) *(ICS '03)*. ACM, New York, NY, USA, 160–171. https://doi.org/10.1145/782814.782838

[45] Jakub Szefer and Ruby B. Lee. 2012. Architectural Support for Hypervisor-Secure Virtualization. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 437–450. https://doi.org/10.1145/2150976.2151022

[46] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 665–678.

[47] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, USA) *(ASPLOS IX)*. ACM, New York, NY, USA, 168–177.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.

[49] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. 2018. Graviton: Trusted Execution Environments on GPUs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 681–696. https://www.usenix.org/conference/osdi18/presentation/volos

[50] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. 20–37.

[51] Weidong Shi, H. S. Lee, M. Ghosh, Chenghuai Lu, and A. Boldyreva. 2005. High efficiency counter mode security architecture via prediction and precomputation. In *32nd International Symposium on Computer Architecture (ISCA'05)*. 14–24.

[52] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. 2006. Improving Cost, Performance, and Security of Memory Encryption and Authentication. *SIGARCH Comput. Archit. News* 34, 2 (May 2006), 179–190.

[53] Jun Yang, Youtao Zhang, and Lan Gao. 2003. Fast Secure Processor for Inhibiting Software Piracy and Tampering. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, USA, 351.

[54] J. Zhu, R. Hou, X. Wang, W. Wang, J. Cao, B. Zhao, Z. Wang, Y. Zhang, J. Ying, L. Zhang, and D. Meng. 2020. Enabling Rack-scale Confidential Computing using Heterogeneous Trusted Execution Environment. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1450–1465.