SecNDP: Secure Near-Data Processing with Untrusted Memory

Wenjie Xiong*§, Liu Ke*†§, Dimitrije Jankov‡, Michael Kounavis*, Xiaochen Wang*, Eric Northup*, Jie Amy Yang*, Bilge Acun*, Carole-Jean Wu*, Ping Tak Peter Tang*, G. Edward Suh*◇, Xuan Zhang†, Hsien-Hsin S. Lee*

*Meta, †Washington University in St. Louis, ‡Rice University, ◇Cornell University
wenjiex@fb.com, ke.l@wustl.edu, dj16@rice.edu,
{michaelkounavis, xiaochenwang, digitaleric, amyyang, acun, carolejeanwu, ptpt, edsuh}@fb.com,
xuan.zhang@wustl.edu, leehs@fb.com

***Abstract*—Today's data-intensive applications increasingly suffer from significant performance bottlenecks due to the limited memory bandwidth of the classical von Neumann architecture. Near-Data Processing (NDP) has been proposed to perform computation near memory or data storage to reduce data movement for improving performance and energy consumption. However, the untrusted NDP processing units (PUs) bring in new threats to workloads that are private and sensitive, such as private database queries and private machine learning inferences. Meanwhile, most existing secure hardware designs do not consider off-chip components trustworthy. Once data leaving the processor, they must be protected, *e.g.*, via block cipher encryption. Unfortunately, current encryption schemes do not support computation over encrypted data stored in memory or storage, hindering the adoption of NDP techniques for sensitive workloads.**

In this paper, we propose *SecNDP*, a lightweight encryption and verification scheme for untrusted NDP devices to perform computation over ciphertext and verify the correctness of linear operations. Our encryption scheme leverages arithmetic secret sharing in secure Multi-Party Computation (MPC) to support operations over ciphertext, and uses counter-mode encryption to reduce the decryption latency. The security of the encryption and verification algorithm is formally proven. Compared with a non-NDP baseline, secure computation with SecNDP significantly reduces the memory bandwidth usage while providing security guarantees. We evaluate SecNDP for two workloads of distinct memory access patterns. In the setting of eight NDP units, we show a speedup up to 7.46× and energy savings of 18% over an unprotected non-NDP baseline, approaching the performance gain attained by native NDP without protection. Furthermore, SecNDP does not require any security assumption on NDP to hold, thus, using the same threat model as existing secure processors. SecNDP can be implemented without changing the NDP protocols and their inherent hardware design.

***Keywords*-Security and Privacy, Near-Data Processing, Cryptography, Privacy-Preserving Machine Learning**

I. Introduction

The classical von Neumann architecture separates computation from data storage. However, memory and interconnect bandwidth have not been able to keep up with the scaling out and scaling up of the processor cores, hitting the (in)famous *memory wall* [77]. For data-intensive applications with high parallelism and localized memory accesses, memory bandwidth has become a major performance bottleneck. Therefore, techniques such as near-data processing (NDP) offer much promise in both performance and energy consumption by offloading computation to main memory (*i.e.,* DRAM) [8], [10], [24], [36], [44] or even storage [45], [64], [76]. In Samsung's Aquabolt-XL (HBM2-PIM), for example, addition and multiplication operators are supported in DRAM [46].

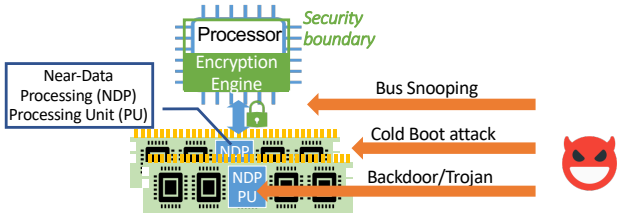

Figure 1. Threat model of SecNDP.

At the same time, many data-intensive applications work on sensitive data, such as queries on private databases (*e.g.*, medical history, user information) and machine learning inference using private models (*e.g.*, models that need IP protection [17] or may reveal the private training dataset [68]). As more computation is outsourced to the cloud, sensitive workloads are becoming more susceptible to cyber and physical attacks. In addition to the pursuit of better performance and lower energy, guaranteeing security and privacy is now considered a first-class citizen in designing future computing systems. To protect the confidentiality and/or integrity of program execution, secure hardware systems designed with a Trusted Execution Environment (TEE) [1], [2], [4], [14], [73] have been proposed and implemented in commercial processors to enforce protection. All of these secure hardware designs consider off-chip components to be untrusted (in Figure 1) – be it a TEE design in CPU [14], [73], CPU and GPU [75], or other accelerators [11], [33]. In this threat model, off-chip memory needs to be encrypted for confidentiality and a message authentication code (MAC) for each data block in memory needs to be computed and checked for integrity.

However, such existing memory protection mechanisms prevent the adoption of NDP in a TEE, because the NDP

§ The first two authors contributed equally to this work.

Processing Units (PUs) only have access to encrypted data, a.k.a., ciphertext. One may apply homomorphic encryption (HE) [23], [59] to enable computations over ciphertext. However, the state-of-the-art HE incurs at least four orders of magnitude performance slowdown [60]. The overhead of HE is too high to let NDP outperform a TEE without NDP. Another approach for secure NDP is to include the NDP PUs inside the TEE [5], [9]. However, this approach requires trusting multiple hardware vendors, an extra key exchange protocol, and hardware components on the NDP PUs, resulting in an undesirably large attack surface for the trusted computing base (TCB). There is no existing practical solution to achieve secure NDP for untrusted memory.

In this paper, we propose SecNDP – a lightweight encryption and verification scheme for a secure processor to use untrusted NDP. SecNDP uses a secure Multi-Party Computation (MPC) protocol [20] between the processor and the untrusted memory, where a block cipher generates the processor's share of secret on-chip without additional off-chip accesses. Like MPC, SecNDP encryption supports addition and scale multiplication efficiently. Like counter-mode encryption [72], SecNDP has low decryption latency with only one adder on the performance critical path. Furthermore, SecNDP introduces a verification tag based on a linear checksum [42] to verify the correctness of the results from the untrusted NDP PU. To efficiently implement SecNDP, we propose architectural support in a secure processor. The SecNDP design only requires relatively small changes in the processor while leaving the NDP protocols and hardware intact. SecNDP's performance and accuracy are evaluated through detailed cycle-level simulations using two real-world data-intensive workloads, *deep learning recommendation inference* and *medical data analytics*. The results show that SecNDP is up to 7.46× faster while using 18% less memory energy compared to the unprotected non-NDP baseline, approaching the performance and energy efficiency of native NDP without protection. The following summarizes the key contributions:

- We propose the first encryption and verification schemes that enable practical and provably secure computation in untrusted NDP.
- We tailor-design the SecNDP architecture, to demonstrate low decryption and verification latency, and low area and power overhead on the processor. The design does not modify NDP PUs and protocols.
- We demonstrate SecNDP's performance benefits and accuracy on two real-world data-intensive use cases.

II. Threat Model

SecNDP is to be used with a TEE design. We follow the threat model of a typical TEE, shown in Figure 1. The processor (CPU, GPU, or an accelerator) is trusted and considered secure, *i.e.*, there is no hardware backdoor. An attacker cannot observe or manipulate the internal state of the processor. We assume an attacker's software co-located in the processor cannot access protected data (directly or through side channels) within the processor. A privileged attacker may access unencrypted data in memory. Data buses are susceptible to passive eavesdropping and unauthorized modifications. For volatile memory such as DRAM, the attacker can conduct a cold-boot attack [29] to dump the memory content. For non-volatile memory, the attacker can unplug the storage to access and modify the content. The NDP processing units (PUs) are untrusted. NDP PUs may have a backdoor or a Trojan to leak data or return a malicious computation result.

III. Background

A. Near-Data Processing (NDP)

For data-intensive applications, such as machine learning [10], [36], [44], [51], [76], similarity search [47], data base [35], graph processing [6], and stencil computing [70], memory bandwidth has become the major performance bottleneck. Many of such workloads deal with vectors and matrices and conduct linear operations, which is our focus.

Depending on the data reuse rate, moving data from memory or storage to CPU for computation is inefficient [69]. Many prior works propose near-memory processing using 3D/2.5D-stacked DRAM technology (such as HMC/HBM) [7], [38], [43], [57]. Accelerators inside the data buffer devices in a commodity DIMM are also proposed to support large-scale workloads [8]. For specific workloads such as sparse embedding operations in recommendation models, tailor-designed light-weight NDP systems are proposed to leverage rank-level parallelism with higher speedup potential [10], [36], [37], [44]. In addition to near-memory processing, near-storage processing is proposed to process data near storage with larger capacity [45], [64], [76]. With the accelerated development of 3D stacking and emerging memory technologies compatible with CMOS process [49], [52], productizing NDP on a computing platform is closer to reality than ever [7], [37], [43], [46].

B. TEEs and Memory Protection

In hardware-based TEE designs, such as AEGIS [73], Mi6 [14], as well as commercial solutions including Intel SGX [4], AMD SEV [1], and ARM TrustZone [2], off-chip memory is untrusted. Off-chip data must be encrypted for confidentiality protection and data integrity needs to be verified.

Confidentiality Protection with Counter-Mode Encryption. In counter-mode encryption [42], [54], [72], a data block (e.g., a cache line) is eXclusive-ORed (XOR) with a One-Time Pad (OTP) (Figure 2(a)). The OTP is an encrypted counter x_e generated using a block cipher such as Advanced Encryption Standard (AES) with the address of the data block and a unique version number v as inputs. The cipher uses the processor's secret key K. The security

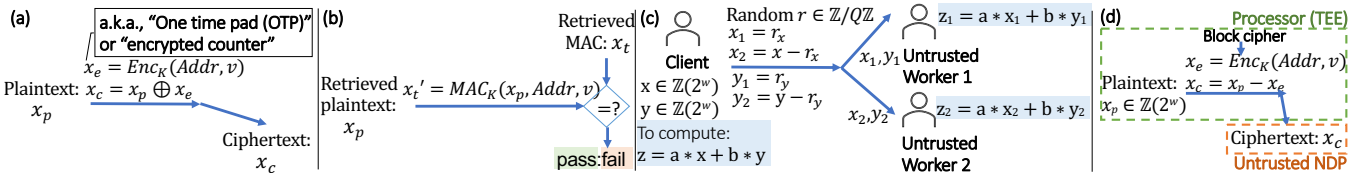

Figure 2. (a) Counter mode encryption. (b) Integrity verification using MAC. (c) Arithmetic secret sharing in secure MPC. (d) Proposed SecNDP.

of the counter-mode encryption relies on the uniqueness of the version number v for each address. If the version number is reused for multiple plaintext blocks at the same address, the attacker can learn the relationship between the plaintext blocks.

When the processor fetches a data block from memory, the OTP can be computed using the address and the version number in parallel with the data access. When the ciphertext is returned, only one XOR with the OTP is needed for decryption. Counter-mode encryption is widely used for memory protection due to its functional parallelism resulting in lower latency [67], [72].

Integrity using Message Authentication Codes (MACs). To detect an unauthorized modification, a keyed message signature, a.k.a., Message Authentication Code (MAC), is stored in memory for each data block. When the processor fetches a data block from memory, its MAC is also fetched. Independently, we use the secret key to compute the MAC to see if it matches with the retrieved one (Figure 2(b)). Without the secret key, the probability that an attacker generates the corresponding MAC for an altered message is negligible. Thus, any change in data from memory will result in a MAC mismatch. To prevent attacks from replaying stale data values or copying valid data from a different address with valid MAC, the address $Addr$ and version number v are also incorporated into the MAC. The integrity verification passes only when both the $Addr$ and v also match. To protect the integrity of the version numbers v, these values are kept on-chip or protected with an integrity tree [62]. To protect both confidentiality and integrity, authenticated encryption with associated data (AEAD) schemes such as the AES GCM mode [54] and the AES CWC mode [42] are designed to use one secret key to both encrypt data and generate a MAC.

Linear Checksum and MACs. A hash function or checksum maps messages into short outputs. A universal hash has the property that the probability of any given pair of messages having the same checksum is small. Such a universal hash function can be used as the building block for a MAC. Linear Modular Hashing [30] is an almost universal hash function suitable for a fast software or hardware implementation. It is applicable to variable-sized data and used in fast message authentication, such as the AES CWC mode [42]. In this paper, we use a linear modular hash not only for its performance but also to leverage its linearity to verify linear operations in NDP.

C. Secure Multi-Party Computation (MPC)

One popular approach to outsource sensitive workloads to untrusted workers is to use secret sharing [20], [55]. MPC secret sharing splits a secret into multiple shares and distributes them to workers. Each worker performs computation over its share locally. Assuming the workers will not collude, it is information-theoretically impossible for each worker to recover the secret from its share. SecNDP let the TEE hold one share and the memory hold another share, and thus, do not require the non-collusion assumption to hold among NDP PUs.

Arithmetic secret sharing [20], [55] is one of the most widely used secret sharing schemes. Let $\mathbb{Z}(2^{w_e})$ denote the integer ring of size 2^{w_e}. The shares are constructed such that the sum of all shares is equal to the original secret value $x \in \mathbb{Z}(2^{w_e})$. Figure 2(c) shows arithmetic sharing between two workers. Each worker holds a share of x, where shares are denoted by $x_1 \in \mathbb{Z}(2^{w_e})$ and $x_2 \in \mathbb{Z}(2^{w_e})$, and $x = x_1 + x_2$. Arithmetic secret sharing enables ciphertext-to-ciphertext additions and additions/multiplications by constants over ciphertext. For example, to compute $z = a * x + b * y$, each worker performs an addition on their respective shares of x and y. In the end, the client collects the two partial results z_1 and z_2, and adds them together to obtain the final result. Compared to HE schemes on high-dimensional rings [60], the operations for each worker are simple and lightweight. This is with arithmetic secret sharing.

All the operations of arithmetic sharing are in the integer ring $\mathbb{Z}(2^{w_e})$. Thus, if the original data are in a floating-point format, they must be quantized into fixed-point numbers or integers. For machine learning models, previous studies have shown that quantization does not have significant impact on the accuracy of certain ML models [31], [41], [74]. More recently, MPC in floating-point arithmetic has also been explored [27], but such evaluation is outside the scope of this work.

IV. SecNDP Encryption and Verification Schemes

In this section, we describe an arithmetic encryption and verification scheme. The encryption supports computation over ciphertexts while providing confidentiality guarantees. The verification scheme further protects the integrity of data using extra data tags. The encryption scheme can be used alone without verification, or with verification for stronger security.

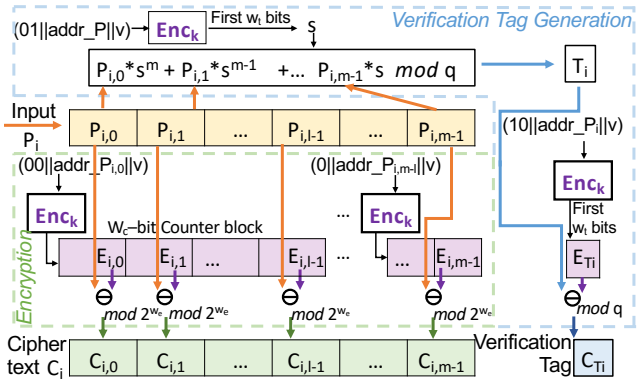

Figure 3. Diagram of the proposed arithmetic encryption and verification tag generation scheme. Here, l is the number of data elements in a data block of the block cipher size (*i.e.*, $l = w_c/w_e$). $\ominus$ represents subtraction in the ring $\mathbb{Z}(2^{w_e})$. $E_{i,j}$ is the j-th OTP substring associated with row i.

Algorithm 1: Arith. Encryption, *Arith-E*$(K, P, Addr)$

Inputs: P, K, $Addr$; //$Addr$ is the address of P
$v \leftarrow \mathcal{V}()$; //unique version, also considered padded with zeros
$cnt_c \leftarrow size(P)/w_c$; //number of blocks of block cipher size
for $i = 0$ *to* $cnt_c - 1$ //each data block
do
  $Addr_i \leftarrow Addr + i \times (w_c/8)$; //addr. of block i in bytes
  $e_{Addr_i} \leftarrow E(K, 00||Addr_i||v)$; //OTP of block i
  for $j = 0$ *to* $(w_c/w_e) - 1$. //for every w_e-bit element
  do
    $e_j \leftarrow e_{Addr_i}[j \times w_e : (j+1) \times w_e]$; //OTP of element j
    $p_j \leftarrow P[Addr_i + j \times w_e : Addr_i + (j+1) \times w_e]$;
    $c_j \leftarrow p_j - e_j \mod 2^{w_e}$; //ciphertext of element j
  end
  $C_{Addr_i} \leftarrow c_0 || \ldots || c_{w_c/w_e - 1}$; //concatenating ciphertexts
end
Return concatenation of all C_{Addr_i};

A. Preliminaries and Notations

We use vector-matrix multiplication as an example operation as it is the most critical and widely-used kernel in machine learning and other data-intensive applications. Section VI-A introduces two use cases that use variants of vector-matrix multiplication. Our scheme can be generalized for other operations as well. We use P to denote the plaintext of a 2-D matrix of size $n \times m$, $P_{i,j}$ to denote a element in the array, p or P_i to denote a row vector in P, and a to denote a vector of dimension n. Each element in p and a is an integer (or a fixed point number) of width w_e. We assume w_e to be a power of 2 and smaller than a cache line size. We use $E(K, X)$ to denote the output of a w_c-bit block cipher (*e.g.*, AES) that encrypts a w_c-bit message X using a w_K-bit secret key K. The elements of the matrix are also considered as consisting of chunks that are w_c bits long. Each chunk contains $l \leftarrow w_c/w_e$ elements.

B. SecNDP Arithmetic Encryption

SecNDP arithmetic encryption is a combination of arithmetic secret sharing and counter-mode encryption, as shown in Figure 2 (d). The scheme details are depicted in Algorithm 1 and Figure 3. First, the plaintext P will be divided into cnt_c binary strings (chunks) of size w_c. The starting physical address of each chunk and the version number v will be used as the input to the block cipher, which generates an OTP using a standard block cipher (*e.g.*, AES). The concatenation of all cnt_c OTP blocks (e_0, e_1, ...) will have the same number of bits as the plaintext P.

Then, the plaintext P and the concatenation of all the OTP blocks e are divided into binary strings of size w_e. The j-th strings are denoted by p_j and e_j, respectively. The ciphertext c_j is the plaintext p_j subtracted by the corresponding OTP block e_j in the ring $\mathbb{Z}(2^{w_e})$. We name the scheme *"Arithmetic Encryption Scheme"*, because the ciphertext c_j and the OTP block e_j can be seen as the arithmetic shares of the secret p_j.

C. Computation over Ciphertext in SecNDP

After the arithmetic encryption completes, each of the processor and the NDP device holds an arithmetic share of the secret. They can then follow the standard MPC protocol [20], [55] to conduct computation. With arithmetic secret sharing, additive operations such as addition and multiplication with a non-private scale are lightweight. In this paper, we demonstrate how to perform a multiplication with a non-private vector.

Figure 4 (a) shows an example of a non-private vector a multiplied with a private matrix P. In the initialization step (T0), the matrix P is encrypted by Alg. 1 and the resulting ciphertext C is stored in untrusted memory or data storage. When there is a vector a to compute $a \times P$ (T1 in Figure 4), the untrusted NDP will compute $a \times C$. The NDP operation is exactly the same as the one in the unprotected scenario. Meanwhile, the processor will use the address and the version number v to generate the OTP blocks E for the entire plaintext P and compute $a \times E$. In the end, the NDP PU sends its share of result back to the processor, and the processor adds the two shares for the final result.[1] The encryption scheme has the property that $E + C = P$, and thus $res = C_{res} + E_{res} = a \times C + a \times E = a \times P$.

D. Comparing SecNDP with TEE and Unprotected NDP

Vector-matrix multiplication in a TEE and unprotected NDP are also shown in Figure 4. All of them have the initialization step (T0) that initializes data and loads data to memory. In the TEE and SecNDP, the data is encrypted before being loaded to memory. Regarding the computations performed, when compared to a TEE baseline, SecNDP

[1]Note that all the operations are in the ring $\mathbb{Z}(2^{w_e})$, and thus, overflow could happen. However, overflow can be detected by our verification scheme.

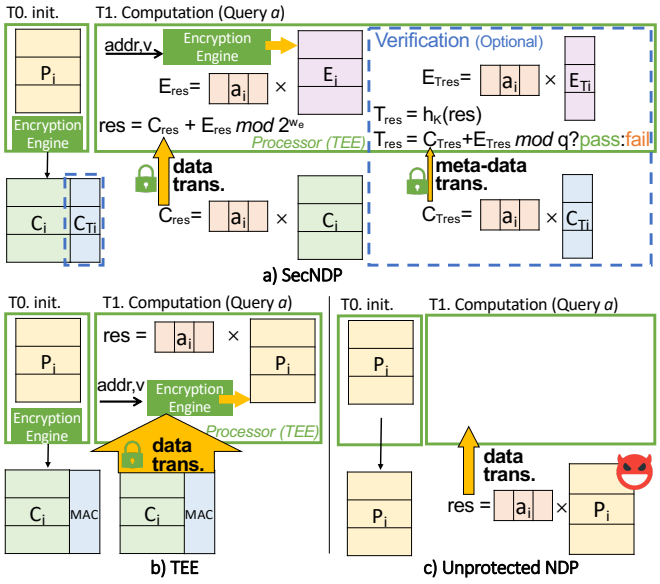

Figure 4. Vector-matrix multiplication in SecNDP, TEE, and unprotected NDP. NDP and SecNDP significantly save memory bandwidth usage, which is the bottleneck for data-intensive workloads. Green boxes indicate encrypted or protected content. Blue boxes show the verification tags.

only needs to transfer the results back to the processor, and thus reduces memory bandwidth usage and saves energy. When there are multiple NDP PUs, they can access memory and perform the computations in parallel to improve performance. However, since both the processor and NDP need to compute on their own share of secret, SecNDP does not save computation on the processor side. Nonetheless, for data-intensive workloads, memory bandwidth, rather than computation, is the main bottleneck. Hence, performance can be significantly improved by SecNDP over the non-NDP baseline.

Compared to an unprotected NDP, the off-chip operations and the required data movement are exactly the same. Hence, there is no modification in the NDP implementation needed to support SecNDP. For decryption and verification, the OTP (E_{res}) and verification tag ($E_{T_{res}}$) computations can be done in parallel with the NDP operations. Only one extra final addition is added to the performance critical path (more details in Section V-E3), which results in negligible performance overhead.

E. Security of the Arithmetic Encryption

The arithmetic encryption applies two-party arithmetic secret sharing between the TEE and the memory, with the TEE's share of secret (*i.e.*, OTP) generated from a block cipher. If an attacker can distinguish c_j (in Alg. 1) from a random number, an oracle can be built to break the block cipher. The security is similar to that of counter-mode encryption [42], [54][2].

[2]We have the proof and formal definitions in the appendix of the full paper at https://eprint.iacr.org/2021/1642.

Let $\mathsf{E}_D(K, Addr, v)$ denote the randomized encryption systems $E(K, D||Addr||v)$. The secret key $K \in \{0,1\}^{w_K}$ is drawn from the uniform distribution. D is a binary string, which can be one of '01', '00' or '10' (in Alg. 1, 2, and 3, respectively). We consider that version values $v \in \{0,1\}^{w_v}$ are drawn from distribution $\mathcal{V}()$ and padded with zeros. Version numbers are not in the control of any adversary but drawn by algorithms 1, 2 and 3. Let $\mathbf{Adv}^{\mathsf{E}_D()}_{|Q|}$ be the distinguishing advantage[3] associated with $\mathsf{E}_D()$ and query budget $|Q|$, where version values are unique per distinct input and not in the control of the distinguisher.

Theorem 1 (Arithmetic Encryption is Secure)**.** *Let Arith-E$(K, P, Addr)$ be the arithmetic encryption system of Alg. 1. Let also $\mathcal{A}^{Arith\text{-}E(),Q_e}_{\mathsf{CPA}}$ be an adaptive chosen plaintext adversary with query budget $|Q_e|$ attacking Arith-E$(K, P, Addr)$, where plaintext is P, $Addr$. Then, the advantage of this adversary is bounded in the following way:*

$$\mathbf{Adv}(\mathcal{A}^{Arith\text{-}E(),Q_e}_{\mathsf{CPA}}) \leq \frac{1}{2^{w_K}} + \mathbf{Adv}^{\mathsf{E}_{00}()}_{|Q|'} \qquad (1)$$

where $|Q|' = \frac{m \cdot n \cdot w_e}{w_c} \cdot |Q_e|$.

The version number v should be generated securely and have its integrity protected. We will discuss how v is managed in Section V-A. The security level depends on the key size w_K and the block cipher. Block ciphers such as AES are considered to be well designed pseudo-random permutations, and their encryption output is practically indistinguishable from random output. Thus, if $\mathsf{E}_{00}()$ is based on AES, $\mathbf{Adv}^{\mathsf{E}_{00}()}_{|Q|'}$ is negligible.

F. SecNDP Verification

Since the NDP is not trusted, the secure processor also needs to verify the correctness of the computation (including overflow). Here, we propose a verification scheme for linear operations. We use a MAC-then-encrypt strategy [61] to construct a verification tag. In the case of linear operations, we choose Linear Modular Hashing [30], [42] as the MAC ($h_K()$ in Alg. 2) and apply SecNDP arithmetic encryption to encrypt the resulting MAC (in Alg. 3 and Figure 3). Here, q is a big prime number, *e.g.*, $2^{127}-1$ which is w_t bits long. The memory side stores a verification tag C_{T_i} for each row vector P_i in P.

To verify the computation results, the secure processor computes a MAC of the result using the secret key and matches that with the retrieved MAC, similar to memory integrity protection in Figure 2(b). Here, the NDP and the processor need to compute together to retrieve the MAC leveraging arithmetic encryption. Specifically, as shown in Figure 4 (a), the NDP computes the result of a vector multiplication over the encrypted checksums to get an encrypted

[3]Distinguishing advantage of block cipher $\mathsf{E}_D()$ is the probability that an attacker can distinguish the output of $\mathsf{E}_D()$ from a truly random output, *i.e.,* breaking the block cipher.

Algorithm 2: Linear Checksum, $h_K(P_i)$

Inputs: K, P_i, **Output:** T_i
$P \leftarrow$ matrix containing P_i; $paddr(P) \leftarrow$ Address of P;
$v \leftarrow \mathcal{V}()$; //version padded with zeros, drawn once for P
$s \leftarrow$ first w_t bits of $E(K, 01||paddr(P)||v)$;
Return $T_i \leftarrow \sum_{j=0}^{m-1} P_{i,j} \times s^{(m-j)} \mod q$;

Algorithm 3: Encrypted MAC, *el-MAC*(K, P_i, $Addr_i$)

Inputs: K, P_i, $Addr_i$ //Address of P_i, **Output:** C_{T_i}
$T_i \leftarrow h_K(P_i)$; //linear checksum, coming from Alg. 2.
$v \leftarrow \mathcal{V}()$; //version padded with zeros, drawn once for P
$E_{T_i} \leftarrow$ first w_t bits of $E(K, 10||paddr(P_i)||v)$; //OTP for tag
Return $C_{T_i} \leftarrow T_i - E_{T_i} \mod q$; // ciphertext of tag;

tag $C_{T_{res}} = a \times C_T$. The processor computes $E_{T_{res}} = a \times E_T$, and $C_{T_{res}} + E_{T_{res}}$ will be used as the retrieved MAC. Due to the linearity of h_K, $h_K(res) = h_K(a \times P) = a \times h_K(P) = a \times (C_T + E_T) = C_{T_{res}} + E_{T_{res}}$ holds.

As shown in Figure 4(a), our verification scheme associates a tag C_{T_i} with each row of the matrix. Like other memory integrity protection schemes, our scheme requires extra memory to store the verification tag. However, because the tag is for each row, the tag is relatively small compared to the data when the matrix has large row sizes. Furthermore, we use arithmetic encryption for tags. Thus, NDP computes and returns the tag of the result $C_{T_{res}} = a \times C_T$, and not all the tags of the rows that participate in the vector matrix multiplication operation, reducing the memory bandwidth requirement.

G. Security of the Verification

The security of the verification scheme is derived from the fact that the verification tags are always encrypted on the memory side and s remains to be a secret to the untrusted memory. In order to generate the valid verification tag for a new value, an attacker needs to correctly guess s. Replay attacks are prevented by including a version number in the MAC (Alg. 2).

Theorem 2 (Integrity of the Weighted Summation)**.** *Let ws-MAC$_K(P, Addr)$ and ws-Verify$_K(C, Addr)$ be the MAC and verification oracles associated with the weighted summation operation (i.e., vector matrix multiplication). Let also $\mathcal{A}_{\mathsf{MAC}}^{ws\text{-}MAC(),Q_s,Q_v}$ be a standard MAC adversary, playing a MAC forgery game on weighted summation oracles. Q_s is the set of sign queries issued by the MAC adversary. Q_v is the set of verification queries issued by the same adversary. Then, the probability that this adversary can successfully create a forgery is bounded in the following way:*

$$\mathbf{Adv}(\mathcal{A}_{\mathsf{MAC}}^{ws\text{-}MAC(),Q_s,Q_v}) \leq \frac{m \cdot |Q_v|}{q} + |Q_v| \cdot (\mathbf{Adv}_{|Q|_{00}}^{\mathsf{E}_{00}()} + \mathbf{Adv}_{|Q|_{01}+1}^{\mathsf{E}_{01}()} + \mathbf{Adv}_{|Q|_{10}+n}^{\mathsf{E}_{10}()}) \quad (2)$$

where $|Q|_{00} = \frac{n \cdot m \cdot w_e}{w_c} \cdot |Q_s|$, $|Q|_{01} = |Q_s| + |Q_v|$, $|Q|_{10} = n \cdot (|Q_s| + |Q_v|)$, and q is the prime found in the definition of Algorithms 2 and 3.

The security level depends on the prime number q, number of queries served $|Q_s|$, $|Q_v|$, the dimensions of the matrix in terms of number of rows n and columns m, and the security of block ciphers $\mathsf{E}_{00}()$, $\mathsf{E}_{01}()$ and $\mathsf{E}_{10}()$. We choose a prime number q that is the largest prime number in 2^{w_t}. For example, we use $w_t = 127$ and $q = 2^{127} - 1$, considering both security and performance. If we consider a 1024-dimension matrix row, we can serve 2^{53} queries without changing key, while maintaining a security level higher than 64 bits.

V. Architectural Support for SecNDP

The SecNDP scheme can be applied to any TEE (CPU, GPU, ASIC accelerator, etc.) and work with any untrusted near-memory or near-storage processing hardware. In this section, we describe a SecNDP architecture and design for a computing platform supported by a TEE-enabled processor with near-memory processing.

Baseline NDP Architecture: Figure 5 depicts the components supporting NDP in a dual in-line memory module (DIMM). NDP architecture support (blue boxes) comprises of an NDP protocol (*i.e.*, NDP commands), CPU ISA extensions for issuing NDP packets, and NDP PUs on the memory side. NDP PUs can compute and access their memory region in parallel. For example, the Rank-NDP PUs run in parallel to access the memory within their DRAM rank simultaneously and perform computations to obtain the intermediate results. Therefore, the peak compute and memory throughput of the NDP grows with the number of ranks. Each NDP PU contains registers to hold its intermediate results. Multiple register allow multiple NDP operations to overlap without sending intermediate results back to a CPU. For workloads that need to store a number of intermediate results simultaneously, the number of NDP PU registers can become the bottleneck and more registers can improve performance.

ISA extensions for NDP: The processor issues special instructions that offload NDP packets to the memory controller, which then dispatches NDP commands (Figure 5) to the NDP PUs. There are two types of NDP instructions (and the corresponding commands): `NDPInst` that controls the arithmetic computation, and `NDPLd` that loads the value in an NDP register from and to the processor. `NDPInst` has all the operands for issuing an NDP command, including a data address, the operation `Op` to be carried out, vector size `vsize`, data size `dsize`, an immediate operand value `Imm`, and source/destination register IDs `RegID`. Figure 5 also shows the value in an NDP command for vector-matrix multiplication $a \times P$. With this instruction, an NDP command will be issued to let an NDP PU first multiply each element

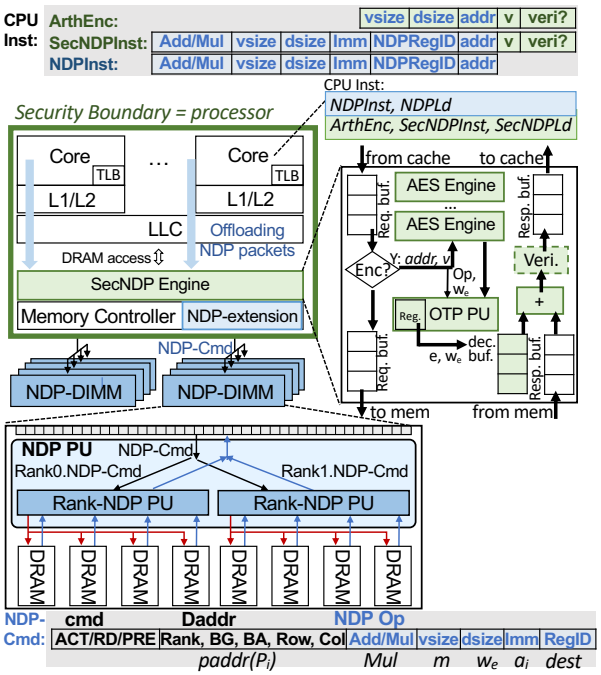

Figure 5. Micro-architecture design for SecNDP. Blue boxes show the components of baseline NDP. Greens boxes show the new components enabling SecNDP.

in the row vector at $paddr(P_i)$ by a_i and add the result to the value at the destination register. After adding all the row vectors, the register will hold the final result. In the end, `NDPLd` is issued to load the results from the NDP PU register back to the processor.

A. SecNDP Architecture Overview

The green boxes in Figure 5 show the components supporting SecNDP. There are two main components: new SecNDP instructions and a SecNDP engine. Note that SecNDP does not change the NDP operations. The NDP commands and NDP PUs remain unchanged.

For efficient version number storage, there are a number of existing solutions [56], [78]. In this paper, we let version values v be managed by trusted software inside the TEE. The integrity of v is then protected by the processor's TEE. The software ensures that v is not reused for the same address. However, a memory region can share the same version number [56]. For NDP workloads such as machine learning, many data structures are read-only or are updated in big chunks. A single data chunk only needs a single version number. Managing the version numbers by software is flexible and efficient, although letting software manage v, comes with the need for new instructions.

B. SecNDP ISA Extensions

The newly proposed instruction `ArithEnc` is for the initial arithmetic encryption and verification tag generation. It has all the operands needed for encryption. `SecNDPInst` is for NDP computation over ciphertext. Its format is similar to `NDPInst` (see Figure 5). Compared to `NDPInst`, `SecNDPInst` has two extra fields: the version number v and one extra bit indicating whether verification is needed. In the micro-architecture, extra fields in data buffers are required to pass v to the encryption engine. The changes are similar to how `NDPInst` operands are offloaded to the memory controller. The new instruction `SecNDPLd` is similar to `NDPLd`, but will also verify the data when loading the data.

C. SecNDP Engine

Since SecNDP is a scheme based on the counter-mode encryption, the encryption engine will be similar to that found in the existing secure hardware. The SecNDP engine is located near the memory controller in the processor.

1) Encryption Engine: The Encryption Engine has a secret key K and takes the address $paddr$ and version v as inputs. For `SecNDPInst`, the encryption engine will generate an OTP, *i.e.*, the processor's share of secret E in Figure 4. Thus, the number of AES engines should be chosen to match the NDP memory throughput (evaluated in Section VII-A).

2) OTP PU: SecNDP needs this processor enhancement to compute on its share of secret, *i.e.*, E in Figure 4. The operations in the OTP PU are the same as the NDP operations. Thus, the OTP PU is designed to have the same number of registers and the same logic as NDP PUs. The OTP PU will simply take the NDP commands ($paddr$, w_e, m, a) and replicate the operations. At the end of the computation, the OTP PU and the NDP PU will each hold their arithmetic share of the secret in their register. Usually, NDP is supported by a lightweight processing unit. Likewise, the complexity of the OTP PU will also be lightweight. For example, for a vector-matrix multiplication, it only needs an integer ALU. The throughput of the OTP PU should match that of the encryption engine.

3) Verification Engine: For verification, a MAC (Alg. 2) is generated from data vectors. Thus, a verification engine is a part of the SecNDP Engine.

Comparing the SecNDP engine to a conventional encryption engine in a TEE, only the OTP PU and the final adder is new. SecNDP would also require more encryption engines depending on the NDP throughput. Because we let software in a TEE manage the version numbers, SecNDP saves the logic and storage for managing version numbers.

D. Verification Tag Storage and Operation

To store verification tags, extra memory space is required. As with conventional memory integrity protection choices, there are three options. (1) *Ver-coloc*: co-locating the tag with data. When software allocates memory for data, it also allocates extra space for the verification tag next to the data. With co-location, the data and tag are likely to be

within the same DRAM row, saving memory access time and energy. (2) *Ver-sep*: allocating tags separately from the data. At the boot-up time or TEE loading time, a system can allocate a designated physical address region to store tags. The advantage of this option is that the software binary layout does not need to be changed for verification. (3) *Ver-ECC*: storing the tags in an ECC chip, and storing the ECC bits in separate memory space [63]. Both data and its tag are fetched in one memory access, while ECC bits are fetched only when corruption is detected. The advantage is that it co-locates the tag with data and fully utilizes the bandwidth of commercial hardware. The disadvantage is that the ECC chip has a fixed capacity and is not flexible for different vector and tag sizes. Also, the SecNDP scheme only verifies the final computation result over a potentially large memory region. If the verification fails, all the ECC of related memory regions need to be checked.

When an NDP operation is to be verified, the NDP needs to compute on the tags. The operation on the data vector ($a_i \times C_i$) will also apply to the tag ($a_i \times C_{T_i}$), as shown in Figure 4(a). There are two possible designs. The SecNDP Engine can issue an extra NDP instruction to compute the tag. Another possible design is to extend each operation. Instead of operation on a vector ($a \times C_i$), an operation on a vector and a tag $a \times [C_i|C_{T_i}]$ is conducted. The second design needs to change the NDP PU logic to have extended registers. Note that the computation for tags is in a prime field. If we choose $q = 2^{127} - 1$, the computation is similar to standard integer arithmetic, but has extra logic when an overflow is incurred [13].

E. Implementation of SecNDP Instructions

1) `ArithEnc`*:* The AES engines use $paddr$ and v to generate the OTPs. Each plaintext data value is then subtracted by its OTP bits (following Alg. 1) and written back to memory like a cache line flush. If the verification bit is set, a MAC (Alg. 2) is computed in the verification engine, and the encryption engine uses $paddr$, m, v to encrypt the tag (Alg. 3).

2) `SecNDPInst`*:* The memory controller passes the operands to the SecNDP Engine and also issues NDP commands. As shown in Figure 4(a), the OTP PU and the NDP PU perform the same computation on the OTP and the ciphertext, respectively. The intermediate results will be in the corresponding registers in the OTP PU and the NDP PU. If verification is needed, extra computation on the tag is performed as described in Section V-D.

3) `SecNDPLd`*:* On the `SecNDPLd` instruction, the value in the desired NDP PU register is loaded to the `resp. buf.` in the processor (see Figure 5). The value of the corresponding register in the OTP PU register is loaded to `dec. buf.` and added to the value from NDP. Only one adder is on the critical path when the encrypted result is returned to the processor core. If verification is needed, the data result is further passed to the Verification Engine to compute the checksum. Meanwhile, the tag is obtained by adding the intermediate tag in the OTP PU and the NDP PU. If the tag matches the checksum of the data result, the verification passes. Otherwise, the verification fails and an interrupt will be triggered. The verification can be in the critical path (1–2 cycle), or can be speculated [48].

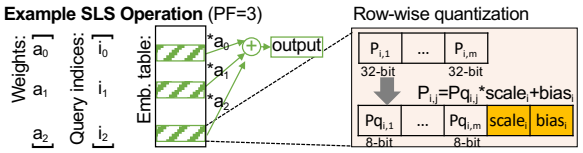

Figure 6. (Left) Embedding table lookup (SLS operation) in deep learning recommendation model. (Right) Row-wise quantization to reduce memory footprint.

VI. Evaluation Methodology

A. Workloads

We use two categories of representative NDP use cases. The first one is deep learning recommendation inference [28], [58], which exhibits sparse and irregular memory access patterns. The second one is data analytics for biomedical applications, which perform accesses over contiguous memory regions. A pooling factor (*PF*) denotes the level of data aggregation, *i.e.*, the ratio of the raw data size to the size of the computed result.

(1) Deep Learning Recommendation Inference: Recommendation models, such as Deep Learning Recommendation Model (DLRM) [58] are structured to take advantage of both continuous and categorical features of individual users. In addition to continuous feature processing using fully-connected (FC) layers commonly found in deep learning, the categorical features are captured by large embedding tables with sparse lookup and pooling operations.

Embedding tables are organized as a set of potentially millions of vectors of dimension m. An embedding table lookup operation (*i.e.,* SparseLengthsSum or SparseLengthsWeightedSum (SLS) operation in Caffe2 [3]) performs a weighted summation for a set of vectors. As illustrated in the left part of Figure 6, an SLS query consists of a list of indices $[i_0, i_1, ..., i_{PF-1}]$ and a weight vector a of dimension *PF*. The result of the pooling is a vector of dimension m and each element of the vector is $res_j = \sum_{k=0}^{PF-1} a_{i_k} \times P_{i_k,j}$. The embedding operations in recommendation models often access irregular indices from large tables, impeding the memory performance that further limits the overall service throughput. Previous studies showed that NDP can significantly speed up embedding lookups in recommendation models [36], [76].

In our evaluation, we offload the embedding table lookup (*i.e.*, SLS operations) to NDP and the rest of the model runs on a CPU. We consider the recommendation models as the

Table I
PARAMETERS OF DLRM MODELS IN EVALUATION

	bottom FC	top FC	# Emb.	total Emb. size
RMC1-small	256-128-32	256-64-1	8	1GB
RMC1-large	256-128-32	256-64-1	12	1.5GB
RMC2-small	256-128-32	256-128-1	24	3GB
RMC2-large	256-128-32	256-128-1	64	8GB

Table II
SIMULATION PARAMETERS AND CONFIGURATIONS

DRAM Parameters
DDR4-2400MHz, rank_size=8GB, tRC=55, tRCD=16, tCL=16, tRP=16, tBL=4, tCCD S=4, tCCD L=6, tRRD S=4, tRRD L=6, tFAW=26
AES Encryption Engine Parameter [22]
Throughput = 111.3Gbps, i.e., 1.15ns per 128-bit block.

service provider's intellectual property (IP), while the values containing users' private information in the embedding tables as the data that need to be protected.

Memory footprint optimization by quantization. For efficiency, the values in the embedding tables are often quantized to a smaller bit width [21]. For example, in row-wise quantization, a vector of 32-bit values can be quantized into a vector of 8-bit integers with a scale and bias per row (Figure 6 right). When the vector is queried, each value in the vector is multiplied by the scale and then the bias is added to recover the original value ($P_{i,j} = Pq_{i,j} \times scale_i + bias_i$). Proper quantization schemes reduce memory footprint while keeping the model accuracy [21].

However, when the row-wise quantization is applied, for each computation there is an extra multiplication with scale ($Pq_{i,j} \times scale_i$), making computation over ciphertext less efficient. Thus, we propose and evaluate table-wise and column-wise quantization schemes, where the scale and the bias are assigned on a per-table or a per-column basis. With table-wise and column-wise quantization, the SLS operation can be first performed without per-row scale, *i.e.*, $resq_j = \sum_{k=0}^{PF-1} a_{i_k} \times Pq_{i_k,j}$. In the last step, the per-table or per-column scale and bias are used to get the final result, *i.e.*, $res_j = resq_j \times scale_j + bias_j$. The total size of the per-column or the per-table scale and the bias is much smaller than the table size. With the table-wise or column-wise quantization, the vector-matrix multiplication can be directly applied to quantized values. For performance evaluation, we assume the scale and bias for quantization can be cached in the processor, and thus, the table-wise and column-wise quantization have the same performance.

Recommendation Model Parameters. For performance evaluation, we use DLRM models with the representative model parameters in Table I. Each embedding table row has $m = 32$ elements. We also consider a different quantization scheme that quantizes 32-bit values into 8-bit values (*i.e.*, 2 cache lines into about 0.5 cache line per vector.) We use a randomly-generated query trace with $PF = 40$ and 80, and a query trace from a production model with a pooling factor *PF* ranging from 50 to 100. For encryption and replay attack prevention, each embedding table uses a version number, and the enclave software manages at most 64 version numbers. For model accuracy evaluation, we use a production-scale recommendation model consisting of hundreds of embedding tables with production data set.

(2) Medical Data Analytics. The second use case we consider here is data analytics over a private data set. To study whether a certain disease is related to certain genes, statistical hypothesis tests need to be performed. Consider a data set containing the expression level for $m = 10000$ genes of $n = 500,000$ patients (e.g., [16]). This is used to compute the test statistics (e.g., p-value of t-test [71]), and the summation (or average) of the gene expression level of patients and non-patients.

The data set contains sensitive medical information (e.g., the gene expression level) and needs to be protected. We store the data set in the memory after encryption. When researchers want to study a certain disease, they query the data set by giving a list of patient IDs in the data set, and let the NDP unit compute the summation. A query is comprised of a list of patient IDs which are used for aggregating their gene expression level. Usually the queried patient IDs are not sparse.

Database Parameters: In performance simulation, we consider a database with $m = 1024$ genes and conduct summation over $PF = 10,000$ patients (40MB in total).

B. Evaluation Setup

SecNDP Performance Simulation. Based on the simulation framework in [36], we built a cycle-level simulation framework with the following components: (1) a physical addresses mapping module, (2) an NDP packet generator, (3) an encryption engine, and (4) an NDP DIMM consisting of DRAM devices, arithmetic units, and control logic. We use Ramulator [40], a cycle-level DRAM simulator, for the DRAM devices. Table II summarizes the parameters and configurations.

During the simulation, we emulate the packet generation and scheduling steps taken by the software stack and the memory controller. First, we apply a standard page mapping method [26] to generate the physical addresses from a trace of embedding lookups by assuming that the OS randomly selects free physical pages for each logical page frame. This physical address trace is then fed to Ramulator.

We implemented a cycle-level NDP module on top of Ramulator including the logic in the NDP PU. The *NDP_rank* parameter denotes the number of Rank-NDP PUs in the system, and the parameter *NDP_reg* denotes the number of registers per NDP PU. To evaluate the NDP latency, the packet generator divides the physical memory requests into packets of NDP commands that are then sent to the cycle-level simulator. NDP activates all ranks under the memory channel. For every NDP packet, the NDP commands are dispatched to the parallel ranks and the latency is bounded by the slowest rank. The total latency of NDP also includes

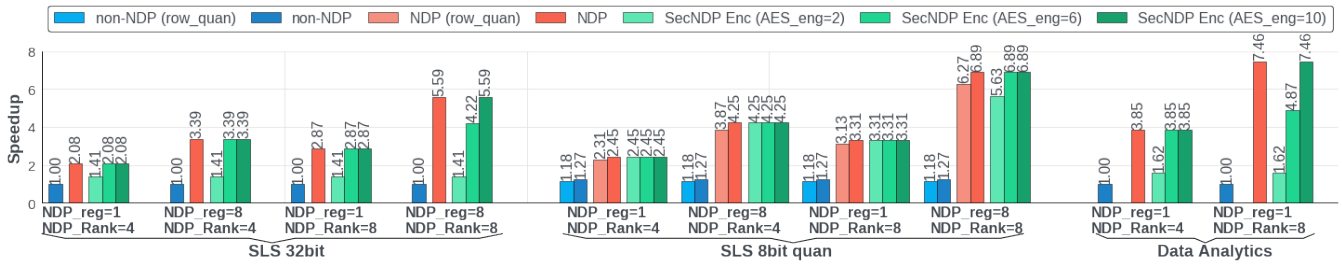

Figure 7. Performance of unprotected non-NDP baseline (in blue), unprotected NDP (in red), and SecNDP-Enc with different numbers of AES engines (in green). For SLS 8-bit quantization, (row_quan) denotes the result of the row-wise quantization scheme, and the other bars are for both the column-wise and table-wise quantization schemes[5]. With a sufficient number of AES engines, encryption in SecNDP shows the same speedup as unprotected NDP.

the DRAM cycles during initialization to configure memory-mapped control registers and a cycle in the final stage to transfer the sum/partial-sum to the processor using `NDPLd`.

To evaluate the throughput of the AES engine and the OTP PU, we use the performance number of a fully pipelined AES design [22]. We assume the addition and multiplication on the counter block are pipelined cycle-by-cycle after AES encryption. Combining the off-chip NDP latency and the encryption engine throughput, we estimate the throughput of SecNDP. The final throughput is the smaller one of the NDP throughput and the OTP throughput.

Whole System (TEE + SecNDP) Evaluation. In some cases, only part of the computation will be offloaded to SecNDP (NDP portion). The remaining part will be executed in the CPU TEE (CPU portion). To estimate the performance slowdown of the CPU portion and to establish a CPU TEE performance reference, we measure the execution time on two Intel machines with SGX enclaves [4]. One is an Intel Xeon E-2288G CoffeeLake (CFL) CPU, with 168MB SGX EPC, 16MB L3, and 32GB DRAM. The other one is an Intel Xeon Platinum 8370C IceLake (ICL) CPU, with 96GB SGX EPC, 48MB L3, and 192GB DRAM. When the workload fits in caches (*e.g.*, the CPU portion of DLRM), SGX ICL has about 5% slowdown. To evaluate the end-to-end performence, we ran the whole model on the ICL machine to obtain the execution time breakdown between the CPU portion and the NDP portion, and use the speedup (or slowdown) of each portion to evaluate the execution time.

Arithmetic Precision and Application Accuracy. We apply a quantization scheme to a production-scale model and use a production dataset with 40K samples to evaluate the model accuracy. The accuracy is presented using *Logloss* [21], a widely used loss function for prediction models.

Power and Area. We use the memory trace from our performance simulation setup and DRAMPower tool [18] to estimate the DRAM chip energy. We use CACTI-IO [34] to evaluate the energy between the DRAM chip and the NDP PU (located inside DIMM's buffer chip) and the energy of DIMM IO. To estimate the energy and the area of the SecNDP engine, we refer to an AES design [22] and use model in [66] for the OTP PU and the verification engine at 45nm process node.

[5]As discussed in Section VI-A-(1), because we cache the scale and bias for quantization in the processor, the column-wise and the table-wise quantization schemes have the same speedups. Hence, we display one bar for both the column-wise and table-wise quantization in the middle group of SLS 8bit quan. The (row_quan) bars represent the original row-wise quantization scheme in the baseline and unprotected NDP scenarios.

Table III
SECNDP SPEEDUP AGAINST UNSECURED BASELINE AND SGX.

	RMC1-small	RMC1-large	RMC2-small	RMC2-large	Data Analytics
unprotected non-NDP	1x	1x	1x	1x	1x
unprotected NDP	2.46x	3.11x	4.05x	4.44x	7.46x
SGX-CFL	0.0038x	0.0037x	N/A	N/A	0.1738x
SGX-ICL (no int. tree)	0.59x	0.60x	N/A	N/A	0.57x
SecNDP	**2.36x**	**3.02x**	**3.95x**	**4.33x**	**7.46x**

VII. EVALUATION RESULTS

In this section, we evaluate the performance (in execution time), precision, energy, and area of the SecNDP architecture. The results show the performance and energy consumption of the SecNDP approach and that of unprotected NDP. The precision loss in SecNDP shows negligible impact on the accuracy of the recommendation model. The area overhead of the SecNDP engine is also small.

A. Performance Evaluation

Overall End-to-End Performance. Table III illustrates the performance of SecNDP compared with the non-NDP baseline and Intel SGX. By leveraging eight NDP PUs, SecNDP demonstrates 2.3x to 4.3x speedup for end-to-end DLRM models with *batch size=256* and 7.46x for the medical data analytics. The performance of SecNDP is close to that of an unprotected NDP as also shown in the table. Meanwhile, using SGX shows considerable slowdown (Table III). This is because the working set sizes of the workloads do not fit in on-chip caches. Due to the malloc size limit by the current SGX library, we could only run RMC1 in SGX. We observe 6x–300x slowdown for the CFL SGX enclave[6], and 1.8x – 2.6x slowdown for the ICL SGX enclave[7]. For SecNDP,

[6]CFL processors rely on an integrity tree to protect against replay attacks, and thus only support a limited number of protected memory pages, causing frequent page swapping in this case.

[7]ICL processors do not have integrity tree for replay attack prevention.

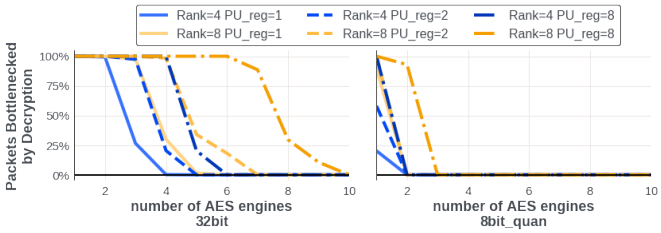

Figure 8. Percentage of NDP packets for SLS operations bottlenecked by decryption bandwidth for confidentiality protection of SecNDP.

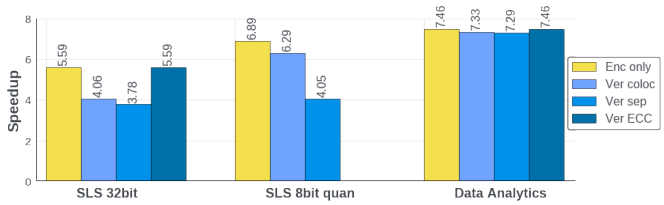

Figure 9. Speedup of various SecNDP encryption and verification schemes with *NDP_rank=8*, *NDP_reg=8*.

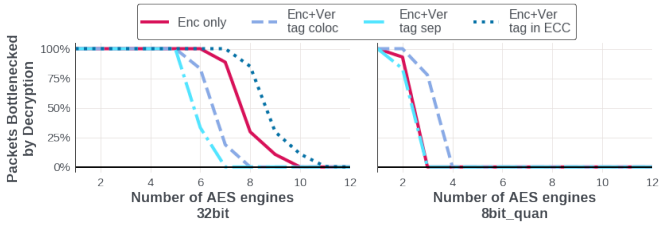

Figure 10. Percentage of NDP packets for SLS operations bottlenecked by decryption bandwidth with *NDP_rank=8*, *NDP_reg=8*.

Table III shows the performance with the verification scheme (*Ver-ECC*) that stores tags in the ECC chip. We will present the performance of unprotected NDP and SecNDP with various verification schemes (introduced in Section V-D) in more detail next. Overall, SecNDP demonstrates significant performance improvement over a CPU TEE without NDP.

NDP Performance - Unprotected. The red bars in Figure 7 summarize the performance of the SLS operations (NDP portion of DLRM) using different quantization schemes and that of the data analytics workload across different NDP settings (*NDP_rank*, *NDP_reg*). For SLS operations, quantization provides 17–27% speedup for all row-wise (row_quan), column-wise, and table-wise quantization in both the NDP and non-NDP settings. This is because the embedding tables are smaller after quantization. Further, with more *NDP_rank* and *NDP_reg*, NDP tends to have higher speedup. More *NDP_rank* allows more parallel memory accesses. More *NDP_reg* allows more intermediate SLS results, making the workload among the NDP PU more evenly distributed. For *NDP_rank=8* and *NDP_reg=8*, the speedup reaches 5.59x without quantization, and 6.89x with quantization. The data analytics workload exhibits more regular memory access patterns that are distributed evenly across all the *NDP_rank* PUs. Thus, it results in higher performance speedup (up to 7.46x) than irregular SLS operations. Also, since there is only one resulting sum, more *NDP_reg* does not help further for the data analytics workload.

NDP Performance - Encryption-Only. We evaluate the performance of SecNDP over different numbers of AES engines (Figure 7). When there is only a small number of AES engines, the decryption becomes the performance bottleneck. As the number of AES engines increases, the speedup reaches that of the unprotected NDP in all settings, indicating that the performance bottleneck eventually shifts to the memory bandwidth.

Figure 8 shows the percentage of NDP packets that is bottlenecked by the AES bandwidth for SLS operations. If the workload has enough parallelism and no NDP PU is idle, more *NDP_rank* require more AES engines to match the NDP throughput. For example, when *NDP_rank=8*, we need ten AES engines to match the memory throughput in the burst mode for the system setting in Table II. However, in practice, applications may not be able to fully utilize the peak memory bandwidth and less AES engines may be sufficient. For example, Figure 8 shows that eight AES engines are enough to match the NDP throughput for roughly 70% of the NDP packets for SLS operations without quantization. Quantization can also significantly reduce the number of AES engines needed; with quantization, only about one third of the AES engines are needed. This is because less OTP is required for the decryption.

NDP Performance - Encryption+Verification. We evaluate three design options for SecNDP's verification tags (*Ver-coloc*, *Ver-sep*, and *Ver-ECC* in Section V-D), and assume the NDP PUs have enough bandwidth to process tags. We use a 128-bit tag for each vector. Figure 9 shows the performance results of *NDP_rank=8* and *NDP_reg=8* with twelve AES engines. For the data analytics workload, because the row vector size m is greater than that of SLS, and the 128-bit tag is relatively small compared to data, the verification overhead is small. Figure 10 presents the percentage of the NDP packets bottlenecked by the decryption operation for SLS operations.

As shown in Figure 9, without quantization, *Ver-coloc* and *Ver-sep* show lower speedup because of the extra memory accesses for the tags. *Ver-sep* shows more performance degradation because tags and data are not co-located, in other words, an additional DRAM row buffer activation is required. *Ver-ECC* has the same speedup as the encryption-only (*Enc-only*) case as no extra DRAM access is required. As shown in the left part of Figure 10, *Ver-ECC* needs more AES engines to match the bandwidth requirement.

With quantization, the embedding vector size becomes shorter than a cache line and the corresponding tags cannot fit in the ECC chip; *Ver-ECC* does not work. Hence, we only show the results for *Ver-coloc* and *Ver-sep* in Figure 9. With *Ver-coloc*, one cache line fetch can retrieve both data and tag, and thus the performance is close to *Enc-only*. But *Ver-coloc*

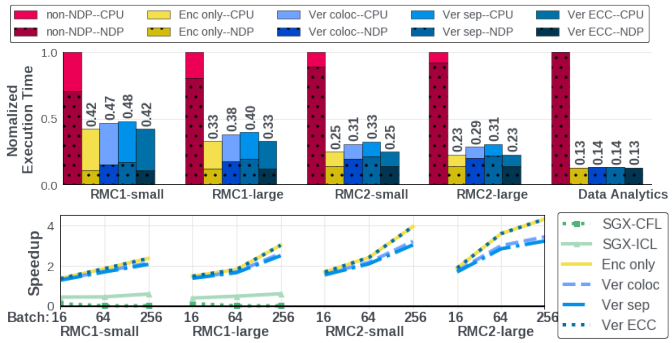

Figure 11. (Top) Normalized execution time in SecNDP with *NDP_rank=8*. The breakdown execution time of NDP portion (–NDP) and the CPU TEE portion (–CPU) of the workloads is shown in the stacked bar. (Bottom) Recommendation inference speedup for different batch sizes in SecNDP.

Table IV
ACCURACY OF DIFFERENT QUANTIZATION SCHEMES

	LogLoss	LogLoss degradation
32-bit floating point	0.64013	0
32-bit fixed point	0.64013	$-3.6 * 10^{-10}$
table-wise quantization (8-bit)	0.64059	0.07%
column-wise quantization (8-bit)	0.64027	0.02%

still cannot reach the performance of *Enc-only*, because with the tag present, the data is not aligned with the cache line boundary. In some cases, two contiguous cache lines are still needed. Compared to *Enc-only*, *Ver-coloc* requires more AES engines to decrypt the tag in parallel with the data. *Ver-sep* leads to about 40% performance degradation over *Enc-only*, because two separate cache lines are accessed instead of one. Even though *Ver-sep* does not change an application's memory layout and can support a variety of tag sizes, it has the worst performance. Co-locating tags and data either in ECC or in the memory has better performance but requires a more complicated system implementation.

End-to-End Execution Time Breakdown. Figure 11 presents the breakdown of the end-to-end execution time and how the performance speedup scales with different batch sizes. Here, we present the speedup with *NDP_rank=8*, *NDP_reg=8*, and *PF=80* for SLS operations without quantization. For *batch size=256*, compared with the insecure non-NDP baseline, SecNDP achieves an end-to-end model inference speedup, ranging from 2.3x to 4.3x. The results include the slowdown of CPU portion in TEE. SecNDP provides higher speedup for larger batch sizes, while SGX does not scale with batch sizes.

B. Impact of Arithmetic Precision on Application Accuracy

Like in MPC, we only support integer and fixed point operations in SecNDP. To assess the impact of data type precision, we evaluate the accuracy impact from the precision change [32]. We perform the model accuracy evaluation using production-level recommendation models [36]. As shown in Table IV, using 32-bit fixed-point values for the embedding has a negligible impact on the Logloss. Furthermore, using 8-bit table-wise or column-wise quantization leads to similar, negligible Logloss degradation (<0.07%). The accuracy results suggest that SecNDP can provide significant speedup with security guarantee, while maintaining the desired model accuracy requirement.

Table V
MEMORY ENERGY CONSUMPTION OF SECNDP (PJ/BIT)

	DIMM	DIMM IO	SecNDP Engine	Normd. Mem. Energy (PF=80)
unprotected non-NDP	27.42×PF	7.3×PF	0	100%
unprotected NDP	27.42×PF	7.3	0	79.2%
non-NDP Enc	27.42×PF	7.3×PF	0.5×PF	101.5%
SecNDP Enc	27.42×PF	7.3	0.9×PF	81.83%
SecNDP Enc+ver	30.85×PF	8.2	1.01×PF+1.72	92.09%

C. Energy and Area Overhead

In addition to performance improvement, NDP also saves energy by reducing the amount of data transfer between the processor and DIMM. Table V summarizes the energy consumption of memory (including IO) and the SecNDP engine. The *non-NDP Enc* row shows the energy consumption of a TEE without NDP. Although SecNDP requires extra computation in OTP PUs, the energy consumption of OTP PUs is insignificant compared to the overall memory power savings. When *PF=80*, SecNDP saves memory system energy by 18% with encryption only and by 8% with verification. In addition, there will be additional energy savings on the processor cache hierarchy, because less data is moved in and out of the caches. The area overhead of SecNDP is estimated to be 1.625 mm^2 at 45nm node if we use 10 AES engines which match the throughput of OTP PUs and the verification engine. The energy/area overheads can be further reduced with more advanced process nodes.

VIII. RELATED WORK

Many cryptographic schemes have been proposed to protect the confidentiality of sensitive workloads on untrusted platforms, such as HE [15], [19], [23], [53], [59], [60] and MPC [20], [55]. However, HE incurs about several orders of magnitude performance overheads [60]. Thus, deploying HE in untrusted NDP does not outperform the TEE baseline without NDP. MPC has better performance than HE, however, the security of MPC relies on the assumption that the untrusted nodes do not collude, which is not realistic in many system settings. To further verify the correctness of computations results, non-interactive verifiable computation [25], [50] and probabilistically checkable proofs [12], [65] are proposed, but their computation complexity is still too high to be practical for secure outsourcing.

Hardware-based TEE is an alternative solution to protect both confidentiality and integrity of workloads. Analyzing the memory protection of Intel SGX, Vessels [39] showed that deep learning workloads suffer from significant performance degradation, and proposed data movement optimizations. To improve the performance of TEE for data-intensive workloads, untrusted accelerators can be leveraged [31],

[74]. Slalom [74] proposed using arithmetic secret sharing to offload secure computations from a TEE to untrusted GPUs. However, the TEE still needs to store its share of secret in memory and pre-compute the results in an offline phase. Thus, Slalom moves computation from online to offline, but does not reduce computation or memory usage. DarKnight [31] proposed a blinding scheme to offload batched deep learning workloads to untrusted GPU. However, DarKnight cannot protect weights in the accelerator. Another solution is to also trust the accelerators and include the accelerators in the TEE. For memory, previous studies [5], [9] proposed protocols to encrypt and obfuscate the traffic on the memory bus. However, InvisiMem [5] still incurs significant performance, energy and memory space overhead. In addition, such solution not only requires trusting multiple hardware vendors, but also requires coordination and standardization among the vendors. SecNDP is the first work to demonstrate how untrusted off-the-shelf NDP units can be used for secure computation.

IX. Conclusion

The recent progress in near-data processing generates a lot of interest in using NDP to alleviate the memory bandwidth bottleneck for data-intensive applications. However, there is a lack of feasible techniques that protect the confidentiality of off-chip data while taking advantage of NDP. In this paper, we present SecNDP, a lightweight encryption and verification scheme that supports NDP over ciphertext and verifies the correctness of NDP results. With a sufficient number of AES engines, our evaluation on two workloads shows that SecNDP can match the speedup delivered by unprotected NDP. Our energy evaluation shows SecNDP saves memory energy by reducing the data transfer on the memory bus. The SecNDP scheme enables a TEE in the presence of untrusted memory to leverage the performance and energy benefits of NDP securely.

Acknowledgments

The authors would like to thank Muhammad Umar, Henry Wang, Shankaran Gnanashanmugam, Jihang Li, Yuchen Hao, and Haixin Liu for their help in evaluating recommendation system in Intel SGX, and thank Brian Knott, Hao Chen, Chuan Guo for their suggestions. The authors would also like to thank the anonymous reviewers for their insightful comments and suggestions. Liu Ke and Xuan Zhang were partially supported by NSF CCF-1942900.

References

[1] "AMD Secure Encrypted Virtualization (SEV)," https://developer.amd.com/sev/.

[2] "Arm TrustZone Technology," https://developer.arm.com/ip-products/security-ip/trustzone.

[3] "Caffe2," https://caffe2.ai/.

[4] "Intel Software Guard Extensions(SGX)," https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html.

[5] S. Aga and S. Narayanasamy, "InvisiMem: Smart memory defenses for memory bus side channel," in *International Symposium on Computer Architecture (ISCA)*, 2017, p. 94–106.

[6] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *International Symposium on Computer Architecture (ISCA)*, 2015, p. 105–117.

[7] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *International Symposium on Computer Architecture (ISCA)*, 2015, pp. 336–348.

[8] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, "Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems," in *International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[9] A. Awad, Y. Wang, D. Shands, and Y. Solihin, "ObfusMem: A low-overhead access obfuscation for trusted memories," in *International Symposium on Computer Architecture (ISCA)*, 2017, p. 107–119.

[10] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Da Eun Shim, Sung-Kyu Lim, Hyesoon Kim, "FAFNIR: Accelerating Sparse Gathering by Using Efficient Near-Memory Intelligent Reduction," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 908–920.

[11] S. Banerjee, P. Ramrakhyani, S. Wei, and M. Tiwari, "SESAME: Software defined enclaves to secure inference accelerators with multi-tenant execution," *arXiv preprint arXiv:2007.06751*, 2020.

[12] M. Bellare, S. Goldwasser, C. Lund, and A. Russell, "Efficient probabilistically checkable proofs and applications to approximations," in *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, 1993, pp. 294–304.

[13] D. J. Bernstein, "Floating-point arithmetic and message authentication," 2000. [Online]. Available: http://cr.yp.to/papers.html#hash127.

[14] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "MI6: Secure Enclaves in a Speculative Out-of-Order Processor," in *International Symposium on Microarchitecture (MICRO)*, 2019, p. 42–56.

[15] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "Fully homomorphic encryption without bootstrapping," in *Innovations in Theoretical Computer Science (ITCS)*, 2012, pp. 309–325.

[16] C. Bycroft, C. Freeman, D. Petkova, G. Band, L. T. Elliott, K. Sharp, A. Motyer, D. Vukcevic, O. Delaneau, J. O'Connell *et al.*, "The UK biobank resource with deep phenotyping and genomic data," *Nature*, vol. 562, no. 7726, pp. 203–209, 2018.

[17] R. Cammarota, I. Banerjee, and O. Rosenberg, "Machine Learning IP Protection," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2018.

[18] K. Chandrasekar, B. Akesson, and K. Goossens, "Improved power modeling of ddr sdrams," in *2011 14th Euromicro Conference on Digital System Design*, 2011, pp. 99–108.

[19] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Annual International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2017.

[20] I. Damgård, V. Pastro, N. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *Annual Cryptology Conference (CRYPTO)*, 2012, pp. 643–662.

[21] Z. S. Deng, J. Park, P. T. P. Tang, H. Liu, J. Yang, H. Yuen, J. Huang, D. S. Khudia, X. Wei, E. Wen, D. Choudhary, R. Krishnamoorthi, C.-J. Wu, N. Satish, C. Kim, M. Naumov, S. Naghshineh, and M. Smelyanskiy, "Low-precision hardware architectures meet recommendation model inference at scale," *IEEE Micro*, 2021.

[22] P.-K. Dong, H. K. Nguyen, and X.-T. Tran, "A 45nm high-throughput and low latency aes encryption for real-time applications," in *2019 19th International Symposium on Communications and Information Technologies (ISCIT)*, 2019, pp. 196–200.

[23] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, 2012.

[24] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2015, pp. 283–295.

[25] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *Annual Cryptology Conference (CRYPTO)*. Springer, 2010, pp. 465–482.

[26] M. Gorman, "Understanding the Linux virtual memory manager," 2004.

[27] C. Guo, A. Hannun, B. Knott, L. van der Maaten, M. Tygert, and R. Zhu, "Secure multiparty computations in floating-point arithmetic," *arXiv preprint arXiv:2001.03192*, 2020.

[28] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H.-H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The architectural implications of facebook's dnn-based personalized recommendation," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2020, pp. 488–501.

[29] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest We Remember: Cold-Boot Attacks on Encryption Keys," *Commun. ACM*, vol. 52, no. 5, p. 91–98, May 2009.

[30] S. Halevi and H. Krawczyk, "MMH: Software message authentication in the gbit/second rates," in *Fast Software Encryption*, 1997, pp. 172–189.

[31] H. Hashemi, Y. Wang, and M. Annavaram, "DarKnight: An accelerated framework for privacy and integrity preserving deep learning using trusted hardware," in *International Symposium on Microarchitecture (MICRO)*, 2021, pp. 212–224.

[32] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Society for Industrial and Applied Mathematics, 2002.

[33] W. Hua, M. Umar, Z. Zhang, and G. E. Suh, "GuardNN: Secure DNN Accelerator for Privacy-Preserving Deep Learning," *arXiv preprint arXiv:2008.11632*, 2020.

[34] N. P. Jouppi, A. B. Kahng, N. Muralimanohar, and V. Srinivas, "Cacti-io: Cacti with off-chip power-area-timing models," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1254–1267, 2014.

[35] Y. Kang, Y.-s. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart SSD," in *Symposium on Mass Storage Systems and Technologies (MSST)*, 2013.

[36] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C.-J. Wu, M. Hempstead, and X. Zhang, "RecNMP: Accelerating personalized recommendation with near-memory processing," in *International Symposium on Computer Architecture (ISCA)*, 2020, pp. 790–803.

[37] L. Ke, X. Zhang, J. So, J.-G. Lee, S.-H. Kang, S. Lee, S. Han, Y. Cho, J. H. Kim, Y. Kwon, K. Kim, J. Jung, I. Yun, S. J. Park, H. Park, J. Song, J. Cho, K. Sohn, N. S. Kim, and H.-H. S. Lee, "Near-Memory Processing in Action: Accelerating Personalized Recommendation with AxDIMM," *IEEE Micro*, 2022.

[38] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *International Symposium on Computer Architecture (ISCA)*, 2016, pp. 380–392.

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

[40] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible DRAM simulator," in *IEEE Computer architecture letters*, vol. 15, no. 1, 2015, pp. 45–49.

[41] B. Knott, S. Venkataraman, A. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten, "CrypTen: Secure multiparty computation meets machine learning," in *Proceedings of the NeurIPS Workshop on Privacy-Preserving Machine Learning*, 2020.

[42] T. Kohno, J. Viega, and D. Whiting, "CWC: A high-performance conventional authenticated encryption mode," in *International Workshop on Fast Software Encryption*. Springer, 2004, pp. 408–426.

[43] Y.-C. Kwon, S. H. Lee, J. Lee, S.-H. Kwon, J. M. Ryu, J.-P. Son, O. Seongil, H.-S. Yu, H. Lee, S. Y. Kim, Y. Cho, J. G. Kim, J. Choi, H.-S. Shin, J. Kim, B. Phuah, H. Kim, M. J. Song, A. Choi, D. Kim, S. Kim, E.-B. Kim, D. Wang, S. Kang, Y. Ro, S. Seo, J. Song, J. Youn, K. Sohn, and N. S. Kim, "25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications," in *International Solid-State Circuits Conference (ISSCC)*, 2021, pp. 350–352.

[44] Y. Kwon, Y. Lee, and M. Rhu, "TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning," in *International Symposium on Microarchitecture (MICRO)*, 2019, pp. 740–753.

[45] J. H. Lee, H. Zhang, V. Lagrange, P. Krishnamoorthy, X. Zhao, and Y. S. Ki, "SmartSSD: FPGA Accelerated Near-Storage Data Analytics on SSD," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 110–113, 2020.

[46] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, O. Seongil, A. Iyer, D. Wang, K. Sohn, and N. S. Kim, "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product," in *International Symposium on Computer Architecture (ISCA)*, 2021, pp. 43–56.

[47] V. T. Lee, A. Mazumdar, C. C. del Mundo, A. Alaghi, L. Ceze, and M. Oskin, "Application codesign of near-data processing for similarity search," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 896–907.

[48] T. S. Lehman, A. D. Hilton, and B. C. Lee, "PoisonIvy: Safe speculation for secure memory," in *International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[49] H. Li, T. F. Wu, S. Mitra, and H.-S. P. Wong, "Resistive RAM-centric computing: Design and modeling methodology," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 9, pp. 2263–2273, 2017.

[50] B. Libert, T. Peters, M. Joye, and M. Yung, "Linearly homomorphic structure-preserving signatures and their applications," *Designs, Codes and Cryptography*, vol. 77, no. 2, pp. 441–477, 2015.

[51] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *International Symposium on Microarchitecture (MICRO)*, 2018, pp. 655–668.

[52] Q. Liu, B. Gao, P. Yao, D. Wu, J. Chen, Y. Pang, W. Zhang, Y. Liao, C.-X. Xue, W.-H. Chen, J. Tang, Y. Wang, M.-F. Chang, H. Qian, and H. Wu, "33.2 a fully integrated analog reram based 78.4tops/w compute-in-memory chip with fully parallel mac computing," in *International Solid- State Circuits Conference*, 2020, pp. 500–502.

[53] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *Symposium on Theory of computing (STOC)*, 2012, pp. 1219–1234.

[54] D. A. McGrew and J. Viega, "The security and performance of the galois/counter mode (GCM) of operation," in *Progress in Cryptology (INDOCRYPT)*, 2005, pp. 343–355.

[55] P. Mohassel and P. Rindal, "ABY3: A mixed protocol framework for machine learning," in *Conference on Computer and Communications Security (CCS)*, 2018, p. 35–52.

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

[57] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2017, pp. 457–468.

[58] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *arXiv preprint arXiv:1906.00091*, 2019.

[59] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *International conference on the theory and applications of cryptographic techniques*. Springer, 1999, pp. 223–238.

[60] B. Reagen, W.-S. Choi, Y. Ko, V. T. Lee, H.-H. S. Lee, G.-Y. Wei, and D. Brooks, "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 26–39.

[61] P. Rogaway, "Authenticated-encryption with associated-data," in *Conference on Computer and Communications Security (CCS)*, 2002, pp. 98–107.

[62] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly," in *International Symposium on Microarchitecture (MICRO)*, 2007, pp. 183–196.

[63] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "Synergy: Rethinking secure-memory design for error-correcting memories," in *International Symposium on High Performance Computer Architecture*, 2018, pp. 454–465.

[64] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A User- programmable SSD," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 67–80.

[65] S. T. Setty, R. McPherson, A. J. Blumberg, and M. Walfish, "Making argument systems for outsourced computation practical (sometimes)." in *NDSS*, vol. 1, no. 9, 2012, p. 17.

[66] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *International Symposium on Computer Architecture (ISCA)*, 2014, pp. 97–108.

[67] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva, "High efficiency counter mode security architecture via prediction and precomputation," in *International Symposium on Computer Architecture (ISCA)*, 2005, pp. 14–24.

[68] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership Inference Attacks Against Machine Learning Models," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 3–18.

[69] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra, "Near-memory computing: Past, present, and future," *Microprocessors and Microsystems*, vol. 71, p. 102868, 2019.

[70] G. Singh, D. Diamantopoulos, C. Hagleitner, S. Stuijk, and H. Corporaal, "NARMADA: Near-memory horizontal diffusion accelerator for scalable stencil computations," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 263–269.

[71] Student, "The probable error of a mean," *Biometrika*, pp. 1–25, 1908.

[72] G. E. Suh, D. Clarke, B. Gasend, M. Van Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *International Symposium on Microarchitecture (MICRO)*, 2003, pp. 339–350.

[73] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for tamper-evident and tamper-resistant processing," in *ACM International Conference on Supercomputing 25th Anniversary Volume*, 2003, p. 357–368.

[74] F. Tramèr and D. Boneh, "Slalom: Fast, verifiable and private execution of neural networks in trusted hardware," in *International Conference on Learning Representations (ICLR)*, 2019.

[75] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on gpus," in *Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2018, pp. 681–696.

[76] M. Wilkening, U. Gupta, S. Hsia, C. Trippel, C.-J. Wu, D. Brooks, and G.-Y. Wei, "RecSSD: Near Data Processing for Solid State Drive Based Recommendation Inference," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 717–729.

[77] Wm. A. Wulf and Sally A. McKee, "Hitting the memory wall: implications of the obvious," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1. ACM, 1995, pp. 20–24.

[78] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," in *International Symposium on Computer Architecture (ISCA)*, 2006, p. 179–190.

APPENDIX

A. Preliminaries and more Definitions

Definitions of the notations used in the paper are shown in Table VI.

Table VI
NOTATIONS

Symbol	Definition
P	plaintext, is a 2-D array of $n \times m$, i.e., n vectors of dimension m.
$P[add_l : add_r]$	plaintext block between physical address add_l and add_r, include add_l and exclude add_r
P_i	the ith vector in the plaintext.
$P_{i,j}$	the jth element in vector P_i.
w_e	Bit width of element in P
w_A	Bit width of addresses
A	address of corresponding plaintext
w_K	Bit width of the secret key
K	processor secret key, w_K bit
w_c	block cipher size, 128 for AES
$E(K, X)$	A block cipher $\{0,1\}^{w_K} \times \{0,1\}^{w_c} \rightarrow \{0,1\}^{w_c}$
$paddr()$	the starting physical address of data in bytes
$size()$	the size of data in bits
v	version number of the plaintext
w_v	Bit width of version, it should be less than $w_c - 37$, considering 38-bit address and 2-bit for encrypting tag
C	Ciphertext. Size is the same as plaintext
PF	pooling factor for summation
w_t	Bit width of verification tag
q	a prime number, $\approx 2^{w_t}$
T_i	checksum of vector P_i
C_{T_i}	encrypted checksum of vector P_i
$x\|\|y$	concatenation of x and y
$x \xleftarrow{\$} X$	uniformly sampling, i.e., selecting at random an element from X and assigning it to x.

In Figure 3, we show the diagram of a vector and matrix multiplication. Algorithms 4 and 5 are the algorithms that compute weighted summation of rows in matrix P (*i.e.,* vector a times a subset of rows in P) and verify the result. In the algorithm description that follows, it is assumed that only a subset of elements from the input matrix P participate in the multiplication, as indicated by index sets $[i_0, i_1, ..., i_{PF-1}]$ and $[j_0, j_1, ..., j_{PF-1}]$. The steps performed on the processor vs NDP are shown separately in different columns. The communication between the two entities is denoted by long arrows.

B. Proof of Correctness

Theorem A.1 *On the correctness of the weighted summation*: Value res computed in step 15, Alg. 4, satisfies the equality $res = (\sum_{k=0}^{PF-1} a_k \times P_{i_k,j_k}) \mod 2^{w_e}$.

Proof: From the Encryption in Alg. 1, for any i and j, we have $C_{i,j} = P_{i,j} - e_{i,j} \mod 2^{w_e}$, and thus, $P_{i,j} = C_{i,j} + e_{i,j} \mod 2^{w_e}$. Thus,

$$res = c_{res} + e_{res} \mod 2^{w_e} \tag{3}$$

$$= [\sum_{k=0}^{PF-1} a_k \times C_{i_k,j_k} + \sum_{k=0}^{PF-1} a_k \times e_{i_k,j_k}] \mod 2^{w_e} \tag{4}$$

$$= [\sum_{k=0}^{PF-1} a_k \times (C_{i_k,j_k} + e_{i_k,j_k})] \mod 2^{w_e} \tag{5}$$

$$= (\sum_{k=0}^{PF-1} a_k \times P_{i_k,j_k}) \mod 2^{w_e} \tag{6}$$

■

Theorem A.2 *On the correctness of the weighted summation verification*: If all the parties in Alg. 5 follow the protocol, and for all $j \in [0, m-1]$, $\sum_{k=0}^{PF-1} a_k \times P_{i_k,j}$ does not exceed 2^{w_e}, then the equality $T_{res} = C_{T_{res}} + E_{T_{res}} \mod q$ holds.

Proof: From the encryption flow of Alg. 1, for any i and j, we have $C_{i,j} = P_{i,j} - e_{i,j} \mod 2^{w_e}$. From the OTP generation flow of Alg. 3, $C_{T_i} = T_i - E_{T_i} \mod q$. All elements in P are integrity protected using the same entity s, which is equal to the first w_t bits of $E(K, 01||paddr(P)/w_c||version)$. Hence, as shown in Alg. 5:

$$LHS = T_{res} = \sum_{j=0}^{m-1} res_j^{m-j} \mod q \tag{7}$$

$$= \sum_{j=0}^{m-1} (\sum_{k=0}^{PF-1} a_k \times P_{i_k,j} \mod 2^{w_e}) \times s^{m-j} \mod q \tag{8}$$

Meanwhile,

$$RHS = C_{T_{res}} + E_{T_{res}} \mod q \tag{9}$$

$$= [(\sum_{k=0}^{PF-1} a_k \times C_{T_k}) + (\sum_{k=0}^{PF-1} a_k \times E_{T_k})] \mod q \tag{10}$$

$$= [\sum_{k=0}^{PF-1} a_k \times (C_{T_k} + E_{T_k})] \mod q$$

$$= [\sum_{k=0}^{PF-1} a_k \times T_k] \mod q \tag{11}$$

$$= [\sum_{k=0}^{PF-1} a_k \times (\sum_{j=0}^{m-1} P_{i_k,j} \times s^{m-j})] \mod q \tag{12}$$

$$= [\sum_{k=0}^{PF-1} (\sum_{j=0}^{m-1} a_k \times P_{i_k,j} \times s^{m-j})] \mod q \tag{13}$$

$$= [\sum_{j=0}^{m-1} (\sum_{k=0}^{PF-1} a_k \times P_{i_k,j}) \times s^{m-j}] \mod q \tag{14}$$

$$\text{Let } overflow_j \triangleq \lfloor (\sum_{k=0}^{PF-1} a_k \times P_{i_k,j})/2^{w_e} \rfloor. \quad (15)$$

$$RHS = [\sum_{j=0}^{m-1} (\sum_{k=0}^{PF-1} a_k \times P_{i_k,j} \mod 2^{w_e} + overflow_j \times 2^{w_e}) \times s^{m-j}] \mod q \quad (16)$$

$$= [\sum_{j=0}^{m-1} (\sum_{k=0}^{PF-1} a_k \times P_{i_k,j} \mod 2^{w_e}) \times s^{m-j}] \mod q + [\sum_{j=0}^{m-1} (overflow_j \times 2^{w_e}) \times s^{m-j}] \mod q \quad (17)$$

$$= T_{res} + [\sum_{j=0}^{m-1} (overflow_j) \times s^{m-j}] \times 2^{w_e} \mod q \quad (18)$$

$$= LHS + [\sum_{j=0}^{m-1} (overflow_j) \times s^{m-j}] \times 2^{w_e} \mod q \quad (19)$$

If there is no overflow, then for all j, $overflow_j = 0$. Hence, $LHS = RHS$.

■

C. Security Proof

We begin our security analysis with the definition of the distinguishing advantage of a cryptographic system. We modify the standard definition to include the number of issued queries as a property of the advantage.

Definition A.1 *Distinguishing advantage for a randomized system*: Let $\mathsf{S}() : \{0,1\}^{w_{in}} \rightarrow \{0,1\}^{w_{out}}$ denote a randomized system associated with inputs of length w_{in} and outputs of length w_{out}, and $\mathcal{R}()$ a truncated output random oracle, associated with the same input and output lengths. Let also $\mathcal{A}_Q^{\mathsf{F}()}$ denote any randomized polynomial time algorithm which issues queries from a set Q of cardinality $|Q|$ to any system $\mathsf{F}() : \{0,1\}^{w_{in}} \rightarrow \{0,1\}^{w_{out}}$, and outputs one of 0 or 1. The distinguishing advantage for system $\mathsf{S}()$ and truncated output random oracle $\mathcal{R}()$ associated with number of queries $|Q|$ is defined as:

$$\mathbf{Adv}_{|Q|}^{\mathsf{S}()} = \max_{\mathcal{A},Q} |Pr[\mathcal{A}^{\mathsf{S}()} \Rightarrow 1] - Pr[\mathcal{A}^{\mathcal{R}()} \Rightarrow 1]| \quad (20)$$

Definition A.2 *Distinguishing advantage for randomized encryption systems tweaked by a version field*: Let $E(K,X) : \{0,1\}^{w_c} \times \{0,1\}^{w_K} \rightarrow \{0,1\}^{w_c}$ be a block cipher associated with input $X \in \{0,1\}^{w_c}$, key $K \in \{0,1\}^{w_K}$, and ciphertext $Y \in \{0,1\}^{w_c}$. Let w_A and w_v be address and version lengths. Let $\mathsf{E}_{00}(K,A,v)$, $\mathsf{E}_{01}(K,A,v)$ and $\mathsf{E}_{10}(K,A,v)$, denote the randomized encryption systems $E(K, 00||A||v||0^{w_c-w_A-w_v-1})$, $E(K, 01||A||v||0^{w_c-w_A-w_v-1})$ and $E(K, 10||A||v||0^{w_c-w_A-w_v-2})$, where $\mathsf{E}_{00}(), \mathsf{E}_{01}(), \mathsf{E}_{10}() : \{0,1\}^{w_A} \times \{0,1\}^{w_v} \times \{0,1\}^{w_K} \rightarrow \{0,1\}^{w_c}$. $A \in \{0,1\}^{w_A}$ is the address input in these systems. $K \in \{0,1\}^{w_k}$ is the encryption key, which is drawn from the uniform distribution, $K \xleftarrow{\$} K_0$, and secret to any randomized polynomial time algorithm issuing queries to $\mathsf{E}_{00}(), \mathsf{E}_{01}(), \mathsf{E}_{10}()$. Version $v \in \{0,1\}^{w_v}$ is a cipher tweak, which is also randomized, drawn from distribution $\mathcal{V}()$, where $\mathcal{V}()$ is not necessarily uniform. The version field is in the control of systems $\mathsf{E}_{00}(), \mathsf{E}_{01}(), \mathsf{E}_{10}()$ and not in the control of the algorithms that issue queries to $\mathsf{E}_{00}(), \mathsf{E}_{01}(), \mathsf{E}_{10}()$. On every distinct query received by $\mathsf{E}_{00}(), \mathsf{E}_{01}(), \mathsf{E}_{10}()$, systems $\mathsf{E}_{00}(), \mathsf{E}_{01}(), \mathsf{E}_{10}()$ produce a different tweak v drawn from distribution $\mathcal{V}()$, and use this tweak to compute an output for the received query. The only restriction imposed on $\mathcal{V}()$ is that no two encryption operations performed by any of $\mathsf{E}_{00}(), \mathsf{E}_{01}(), \mathsf{E}_{10}()$, with the same input A, is tweaked by the same $v \leftarrow \mathcal{V}()$. We use a Boolean parameter $\mathsf{dis} \in \{\mathsf{true}, \mathsf{false}\}$ to denote whether v is disclosed to the algorithms querying $\mathsf{E}_{00}(), \mathsf{E}_{01}(), \mathsf{E}_{10}()$ or not. When parameter is dis is omitted, it is implied that $\mathsf{dis} = \mathsf{true}$. The distinguishing advantage associated with each of the randomized encryption systems $\mathsf{E}_{00}()$, $\mathsf{E}_{01}()$ and $\mathsf{E}_{10}()$ and number of queries $|Q|$ is defined as:

$$\mathbf{Adv}_{|Q|}^{\mathsf{E}_D(),\,\mathsf{dis}} = \max_{\substack{A,Q,\mathcal{V}(),\\ \text{same input queries are}\\ \text{on a different } v \leftarrow \mathcal{V}()}} |Pr[\mathcal{A}^{\mathsf{E}_D()} \Rightarrow 1, \mathsf{dis}] - Pr[\mathcal{A}^{\mathcal{R}()} \Rightarrow 1, \mathsf{dis}]| \quad (21)$$

where D is one of '00', '01', or '10', and $\mathcal{A}$ is any randomized polynomial time algorithm as in Definition A.1.

The security of our proposed systems is established in the standard adaptive chosen plaintext and MAC adversaries. The games which these adversaries play are given in the definitions below.

Definition A.3 *Adaptive chosen plaintext adversary attacking the encryption system Arith-E()*: Let *Arith-E*$(K,P,A) : \{0,1\}^{m \times n \times w_e} \times \{0,1\}^{w_A} \times \{0,1\}^{w_K} \rightarrow \{0,1\}^{m \times n \times w_e}$ be the encryption system defined by Algorithm 1, where P is the matrix input, m, n and w_e are length parameters defined in Section IV-A, A is an address value of length w_A and $K \in \{0,1\}^{w_K}$ is the encryption key. The version input is omitted as this is not in the control of an adversary, but instead drawn from distribution $\mathcal{V}()$ by *Arith-E*() on every distinct encryption operation. An adaptive chosen plaintext adversary $\mathcal{A}_{\mathsf{CPA}}^{\textit{Arith-E}(),Q}$ is defined as a randomized polynomial time algorithm which issues $|Q|$ adaptive chosen queries to encryption system *Arith-E*() from a set $Q = \{Q^{[i]} \leftarrow (P^{[i]}, A^{[i]}),\ i \in [0, |Q|-1]\}$, and observes the responses $C^{[i]} \leftarrow$ *Arith-E*$(K, P^{[i]}, A^{[i]})$. Furthermore, depending on the value of a Boolean parameter dis, the algorithm may

Algorithm 4: Weighted Summation $\sum_{k=0}^{PF-1} a_k \times P_{i_k,j_k} \mod 2^{w_e}$

	Processor	Bus	NDP
2	**Inputs:** K, P, $[i_0, i_1, ..., i_{PF-1}]$, $[j_0, j_1, ..., j_{PF-1}]$, $[a_0, a_1, ..., a_{PF-1}]$		
3	**Output:** res		
4	$C \leftarrow$ *Arith-E*$(K, P, paddr(P))$ // Initial Encryption using Algorithm 1		
5	Processor sends encrypted matrix C to NDP	$\xrightarrow{C}$	NDP receives C
6	Processor requests for a weighted summation of elements of C	$\xrightarrow{NDPSum}$	NDP receives the request
7			$C_{res} \leftarrow \sum_{k=0}^{PF-1} a_k \times C_{i_k,j_k} \mod 2^{w_e}$
8	// Processor generates OTP for P_{i_k,j_k}		
9	**for** $k = 0$ *to* $PF-1$ **do**		
10	$idx_k \leftarrow (paddr(P_{i_k,j_k}) \times 8 \mod w_c)/w_e$;		
11	$v \leftarrow$ version associated with the encryption of matrix P drawn by Algorithm 1;		
12	$E_{i_k,j_k} \leftarrow E(K, 00 \| paddr(P_{i_k,j_k}) \| v)[idx_k \times w_e : (idx_k + 1) \times w_e]$		
13	Processor receives C_{res} from NDP	$\xleftarrow{C_{res}}$	
14	$E_{res} \leftarrow (\sum_{k=0}^{PF-1} a_k \times E_{i_k,j_k}) \mod 2^{w_e}$ // Computes OTP for result		
15	$res \leftarrow C_{res} + E_{res} \mod 2^{w_e}$		
16	**Return** *res*		

Algorithm 5: Verification of Vector Weighted Summations $res_j = \sum_{k=0}^{PF-1} a_k \times P_{i_k,j} \mod 2^{w_e}$ for all $j \in [0, m-1]$

	Processor	Bus	NDP
2	**Inputs:** K, $paddr(P)$, $[i_0, i_1, ..., i_{PF-1}]$, $[a_0, a_1, ..., a_{PF-1}]$		
3	**Output:** pass or fail		
4	Processor requests $C_{res_j}, \forall j \in [0, m-1]$		
5	Processor receives C_{res_j}	$\xleftarrow{C_{res_j}}$	$C_{res_j} \leftarrow (\sum_{k=0}^{PF-1} a_k \times C_{i_k,j}) \mod 2^{w_e}$
6	$res_j \leftarrow \sum_{k=0}^{PF-1} a_k \times P_{i_k,j} \mod 2^{w_e}$ **for all** $j \in [0, m-1]$		
7	// res_j values are computed from steps 8-12 and 14-16 of Alg.4		
8	$v \leftarrow$ version associated with the linear checksum of P, drawn by Algorithm 2;		
9	$s \leftarrow$first w_t bits of $E(K, 01 \| paddr(P) \| v)$;		
10	$T_{res} \leftarrow \sum_{j=0}^{m-1} res_j \times s^j \mod q$;		
11	$v \leftarrow$ version associated with the enc. linear checksum of P, drawn by Algorithm 3;		
12	**for** $k = 0$ *to* $PF-1$ **do**		
13	$E_{T_k} \leftarrow$ first w_t bits of $E(K, 10 \| paddr(P_k) \| v)$;		
14	$E_{T_{res}} \leftarrow (\sum_{k=0}^{PF-1} a_k \times E_{T_k}) \mod q$;		
15	Processor receives $C_{T_{res}}$ // C_{T_k} values have been computed using Alg. 2, 3	$\xleftarrow{C_{T_{res}}}$	$C_{T_{res}} \leftarrow (\sum_{k=0}^{PF-1} a_k \times C_{T_k}) \mod q$
16	**Return** pass **if** $T_{res} = C_{T_{res}} - E_{T_{res}} \mod q$ **else** fail		

or may not have access to the version values v returned from $\mathcal{V}()$ as part of the game. The algorithm succeeds if, in the end of the game, the algorithm can produce a value $K_0 \in \{0,1\}^{w_K}$ which is equal to the encryption key K. The advantage of this adversary is defined as:

$$\begin{aligned}\mathbf{Adv}(\mathcal{A}_{\mathsf{CPA}}^{\textit{Arith-E}(),Q,\mathsf{dis}}) = Pr[K_0 \leftarrow \mathcal{A}_{\mathsf{CPA}}^{\textit{Arith-E}(),Q}, K_0 = K \mid \\ \mathsf{dis},\ C^{[0]} = \textit{Arith-E}(K, P^{[0]}, A^{[0]}), \ldots, \\ C^{[|Q|-1]} = \textit{Arith-E}(K, P^{[|Q|-1]}, A^{[|Q|-1]})\ ]\end{aligned} \tag{22}$$

In the definition and security analysis that follows we will omit the sequences $[i_0, ..., i_{PF-1}]$ and $[a_0, ..., a_{PF-1}]$ passed as input to Algorithms 6 and 7 for the sake of simplicity. These sequences are considered constant and our proof holds for any such sequences.

Definition A.4 *MAC Adversary attacking the weighted summation algorithm*: Let *ws-MAC*$_K(P, A) : \{0,1\}^{n \times m \times w_e} \times \{0,1\}^{w_A} \times \{0,1\}^{w_K} \rightarrow \{0,1\}^{m \times w_e + w_t}$ be the sign oracle defined by Algorithm 6 and *ws-Verify*$_K(\mathcal{C}, A)$: $\{0,1\}^{m \times w_e + w_t} \times \{0,1\}^{w_A} \times \{0,1\}^{w_K} \rightarrow \{\mathsf{pass}, \mathsf{fail}\}$ the verification oracle of Algorithm 7 returning one of pass or fail. P is the matrix row input, n, m, and w_e are length parameters defined in Section IV-A, A is an address value of length w_A and $K \in \{0,1\}^{w_K}$ is the key value used by the sign and verification oracles. The response coming from oracle *ws-MAC*$_K()$ is a bit string of length $m \times w_e + w_t$ and consists of the responses $C_{res_0}, \ldots C_{res_{m-1}}$ and $C_{T_{res}}$ returned from Algorithm 6. $\mathcal{C}$ is a bit string of the same length $m \times w_e + w_t$ consisting of the values $C_{res_0}, \ldots C_{res_{m-1}}$ and $C_{T_{res}}$ which are passed as input to algorithm 7. A MAC adversary $\mathcal{A}_{\mathsf{MAC}}^{\textit{ws-MAC}(),Q_s,Q_v,\mathsf{dis}}$ is defined as a randomized polynomial time algorithm which has access to oracles *ws-MAC*() and *ws-Verify*(), but not to the key value K, submits $|Q_s|$ adaptive chosen sign

Algorithm 6: Weighted Summation Sign Oracle, *ws-MAC*$_K(P,\ Addr,\ [i_0, i_1, ..., i_{PF-1}],\ [a_0, a_1, ..., a_{PF-1}])$

Inputs: K, P, $Addr$, $[i_0, i_1, ..., i_{PF-1}]$, $[a_0, a_1, ..., a_{PF-1}]$ //$Addr$ is the address of P
Output: $C_{res_0}, \ldots, C_{res_{m-1}}, C_{T_{res}}$
Execute steps 4, 5 of Algorithm 4 m times on inputs K, P, $[i_0, i_1, ..., i_{PF-1}]$, $[j, \ldots, j]$, and $[a_0, a_1, ..., a_{PF-1}]$, $\forall j \in [0, m-1]$;
Execute Algorithms 2 and 3 on inputs K and P to produce values C_{T_k} ;
Execute steps 15 of Algorithm 5 on inputs $[i_0, i_1, ..., i_{PF-1}]$, and $[a_0, a_1, ..., a_{PF-1}]$;
$C_{res_0}, \ldots, C_{res_{m-1}} \leftarrow$ values returned from step 5 of Algorithm 5;
$C_{T_{res}} \leftarrow$ value returned from step 15 of Algorithm 5;
Return $C_{res_0}, \ldots, C_{res_{m-1}}, C_{T_{res}}$;

Algorithm 7: Weighted Summation Verification Oracle, *ws-Verify*$_K(C_{res_0}, ..., C_{T_{res}},\ Addr,\ [i_0, ..., i_{PF-1}],\ [a_0, ..., a_{PF-1}])$

Inputs: K, $C_{res_0}, \ldots, C_{res_{m-1}}$, $C_{T_{res}}$, $Addr$, $[i_0, \ldots, i_{PF-1}]$, $[a_0, \ldots, a_{PF-1}]$;
Output: pass_fail;
Execute Algorithm 5 on inputs K, $Addr$, $[i_0, \ldots, i_{PF-1}]$, and $[a_0, \ldots, a_{PF-1}]$;
Use inputs $C_{res_0}, \ldots, C_{res_{m-1}}$ and $C_{T_{res}}$ instead of the values returned from NDP in steps 4-5 and 15 of Algorithm 5;
pass_fail $\leftarrow$ value returned from Algorithm 5 in step 16;
Return pass_fail;

queries and $|Q_v|$ adaptive chosen verification queries to oracles *ws-MAC*() and *ws-Verify*() respectively in any order, and observes the oracle responses. Queries come from sets $Q_s = \{Q_s^{[i]} \leftarrow (P^{[i]}, A^{[i]}),\ i \in [0, |Q_s| - 1]\}$ and $Q_v = \{Q_v^{[i]} \leftarrow (\mathcal{C}^{[i]}, a^{[i]}),\ i \in [0, |Q_v| - 1]\}$. Sign oracle responses have the form $\mathcal{C}^{[i]} \leftarrow$ *ws-MAC*$_K(P^{[i]}, A^{[i]})$ for all $i \in [0, |Q_s| - 1]$. Verification oracle responses have the form $b^{[i]} \leftarrow$ *ws-Verify*$_K(\mathcal{C}^{[i]}, a^{[i]})$ for all $i \in [0, |Q_v| - 1]$. The algorithm succeeds if, in the end of the game, the algorithm can produce verification inputs $(\mathcal{C}^{[r]}, a^{[r]})$ such that *ws-Verify*$_K(\mathcal{C}^{[r]}, a^{[r]}) =$ pass and the verification input $\mathcal{C}^{[r]}$ has not been returned from a sign query before. Depending on the value of the Boolean parameter dis, the adversary may or may not have access to version values. The advantage of this adversary is defined as:

$$\begin{aligned}
&\mathbf{Adv}(\mathcal{A}_{\mathsf{MAC}}^{\textit{ws-MAC}(),Q_s,Q_v,\mathsf{dis}}) = \\
&\quad Pr[\ (\mathcal{C}^{[r]}, a^{[r]})\ \leftarrow \mathcal{A}_{\mathsf{MAC}}^{\textit{ws-MAC}(),Q_s,Q_v,\mathsf{dis}}, \\
&\quad \nexists\, j \in [0, |Q_s| - 1] : (\textit{ws-MAC}_K(P^{[j]}, A^{[j]}) = \mathcal{C}^{[r]} \wedge \\
&\quad a^{[r]} = A^{[j]}),\ \textit{ws-Verify}_K(\mathcal{C}^{[r]}, a^{[r]}) = \mathsf{pass}\ | \\
&\quad \mathcal{C}^{[0]} = \textit{ws-MAC}_K(P^{[0]}, A^{[0]}), \ldots, \\
&\quad b^{[0]} = \textit{ws-Verify}_K(\mathcal{C}^{[0]}, a^{[0]}), \ldots]
\end{aligned} \tag{23}$$

The security of the arithmetic encryption and weighted summation verification algorithms is established by the next two theorems. These theorems are the same as Theorems 1 and 2 of Sections IV-B and IV-F, respectively. In these sections, parameter dis is omitted for the sake of simplicity. In the presentation that follows dis is present:

Theorem A.3 *On the security of arithmetic encryption*: Let *Arith-E*(K, P, A) be the arithmetic encryption system of Definition A.3. Let also $\mathcal{A}_{\mathsf{CPA}}^{\textit{Arith-E}(),Q,\mathsf{dis}}$ be the adaptive chosen plaintext adversary of the same definition. Then, the advantage of this adversary is bounded in the following way:

$$\mathbf{Adv}(\mathcal{A}_{\mathsf{CPA}}^{\textit{Arith-E}(),Q,\mathsf{dis}}) \leq \frac{1}{2^{w_K}} + \mathbf{Adv}_{|Q|'}^{\mathsf{E}_{00}(),\mathsf{dis}} \tag{24}$$

where $|Q|' \leftarrow \frac{m \cdot n \cdot w_e}{w_c} \cdot |Q|$.

Proof: To establish that the bound holds we first convert the conditions $C^{[i]} =$ *Arith-E*$(K, P^{[i]}, A^{[i]})$, $i \in [0, |Q| - 1]$ of relation (22) into an equivalent form that introduces the output of encryption system $\mathsf{E}_{00}()$. We refer to the version values used in the queries issued by the adversary as $v_0, \ldots, v_{|Q|-1}$:

$$\begin{aligned}
&\mathbf{Adv}(\mathcal{A}_{\mathsf{CPA}}^{\textit{Arith-E}(),Q,\mathsf{dis}}) = Pr[K_0 \leftarrow \mathcal{A}_{\mathsf{CPA}}^{\textit{Arith-E}(),Q,\mathsf{dis}}, K_0 = K\ | \\
&\quad \mathsf{dis},\ C^{[0]}[0 : w_e] = P^{[0]}[0 : w_e] \\
&\qquad -\ \mathsf{E}_{00}(K, A^{[0]}/w_c, v_0)[0 : w_e]\ \bmod 2^{w_e}, \ldots, \\
&\quad C^{[|Q|-1]}[0 : w_e] = P^{[|Q|-1]}[0 : w_e] \\
&\qquad -\ \mathsf{E}_{00}(K, A^{[|Q|-1]}/w_c, v_{|Q|-1})[0 : w_e]\ \bmod 2^{w_e}, \\
&\quad \ldots\ ]
\end{aligned} \tag{25}$$

By definition A.2, all invocations to oracle $\mathsf{E}_{00}()$ in the bound of relation (24), can be replaced by invocations to a truncated output random oracle $\mathcal{R}()$ provided that the correcting additive term $\mathbf{Adv}_{|Q|'}^{\mathsf{E}_{00}(),\mathsf{dis}}$ is introduced in the bound. We note that there are $|Q|'$ invocations to $\mathsf{E}_{00}()$. Invocations that occur as part of the same adversary query use the same version value, whereas invocations made as part of different queries use different version values. This is consistent with definition A.2 where version values are considered drawn from an arbitrary distribution $\mathcal{V}()$ not necessarily uniform, which satisfies the stated constraint.

Therefore, we establish that:

$$\begin{aligned}\mathbf{Adv}(\mathcal{A}^{\textit{Arith-E}(),Q,\mathsf{dis}}_{\mathsf{CPA}}) \leq Pr[K_0 \leftarrow \mathcal{A}^{\textit{Arith-E}(),Q,\mathsf{dis}}_{\mathsf{CPA}}, K_0 = K \mid \\ \mathcal{R}(A^{[0]})[0:w_e] = P^{[0]}[0:w_e] \\ - \ C^{[0]}[0:w_e] \bmod 2^{w_e}, \ldots, \\ \mathcal{R}(A^{[|Q|-1]})[0:w_e] = P^{[|Q|-1]}[0:w_e] \\ - \ C^{[|Q|-1]}[0:w_e] \bmod 2^{w_e}, \ldots \ ] + \mathbf{Adv}^{\mathsf{E}_{00}(),\mathsf{dis}}_{|Q|'}\end{aligned} \tag{26}$$

Next, we observe that all random oracle responses are statistically independent from each other and from any other event including the selection of key K. Therefore, all conditions that involve random oracle responses can be removed from the bound of (26):

$$\begin{aligned}\mathbf{Adv}(\mathcal{A}^{\textit{Arith-E}(),Q,\mathsf{dis}}_{\mathsf{CPA}}) \leq Pr[K_0 \leftarrow \mathcal{A}^{\textit{Arith-E}(),Q,\mathsf{dis}}_{\mathsf{CPA}}, \ K_0 = K \ ] \\ + \mathbf{Adv}^{\mathsf{E}_{00}(),\mathsf{dis}}_{|Q|'}\end{aligned} \tag{27}$$

We conclude the proof using the fact that the key K is drawn from the uniform distribution:

$$Pr[K_0 \leftarrow \mathcal{A}^{\textit{Arith-E}(),Q,\mathsf{dis}}_{\mathsf{CPA}}, \ K_0 = K \ ] = \frac{1}{2^{w_K}} \tag{28}$$

Theorem 1 directly follows from combining (27) and (28). ■

Theorem A.4 *On the security of weighted summation verification*: Let *ws-MAC*$_K(P, A)$ and *ws-Verify*$_K(\mathcal{C}, A)$ be the sign and verification oracles of Definition A.4. Let also $\mathcal{A}^{\textit{ws-MAC}(),Q_s,Q_v,\mathsf{dis}}_{\mathsf{MAC}}$ be the MAC adversary of the same definition. Then, the advantage of this adversary is bounded in the following way:

$$\begin{aligned}\mathbf{Adv}(\mathcal{A}^{\textit{ws-MAC}(),Q_s,Q_v,\mathsf{dis}}_{\mathsf{MAC}}) \leq \ \frac{m \cdot |Q_v|}{q} + \\ |Q_v| \cdot (\mathbf{Adv}^{\mathsf{E}_{00}(),\mathsf{dis}}_{|Q|_{00}} + \mathbf{Adv}^{\mathsf{E}_{01}(),\mathsf{dis}}_{|Q|_{01}+1} + \mathbf{Adv}^{\mathsf{E}_{10}(),\mathsf{dis}}_{|Q|_{10}+n})\end{aligned} \tag{29}$$

where $|Q|_{00} \leftarrow \frac{n \cdot m \cdot w_e}{w_c} \cdot |Q_s|$, $|Q|_{01} \leftarrow |Q_s| + |Q_v|$, $|Q|_{10} \leftarrow n \cdot |Q_s| + |Q_v|)$ and q is the prime found in the definition of Algorithms 5 and 7.

Proof: A first step in the proof is to establish a bound for the advantage of the MAC adversary $\mathsf{F}()$ if the responses from all verification queries, but the final one $(\mathcal{C}^{[r]}, a^r)$, are equal to fail. This is essentially the probability that the adversary succeeds the first time at query $(\mathcal{C}^{[r]}, a^{[r]})$:

$$\begin{aligned}\mathsf{F}(\mathcal{A}^{\textit{ws-MAC}(),Q_s,Q_v,\mathsf{dis}}_{\mathsf{MAC}}) = \\ Pr[\ (\mathcal{C}^{[r]}, a^{[r]}) \ \leftarrow \mathcal{A}^{\textit{ws-MAC}(),Q_s,Q_v,\mathsf{dis}}_{\mathsf{MAC}}, \\ \nexists \, j \in [0, |Q_s| - 1] : (\textit{ws-MAC}_K(P^{[j]}, A^{[j]}) = \mathcal{C}^{[r]} \wedge \\ a^{[r]} = A^{[j]}), \ \textit{ws-Verify}_K(\mathcal{C}^{[r]}, a^{[r]}) = \mathsf{pass} \mid \\ \mathcal{C}^{[0]} = \textit{ws-MAC}_K(P^{[0]}, A^{[0]}), \ldots, \\ \textit{ws-Verify}_K(\mathcal{C}^{[0]}, a^{[0]}) = \mathsf{fail}, \ldots]\end{aligned} \tag{30}$$

In the conditions of relation (30), oracles *ws-MAC*() and *ws-Verify*() invoke oracles $\mathsf{E}_{00}()$, $\mathsf{E}_{01}()$ and $\mathsf{E}_{10}()$. In fact, there are $|Q|_{00} \leftarrow (\frac{m \cdot n \cdot w_e}{w_c}) \cdot |Q_s|$ invocations to oracle $\mathsf{E}_{00}()$, $|Q|_{01} \leftarrow |Q_s| + |Q_v|$ invocations to oracle $\mathsf{E}_{01}()$, and $|Q|_{10} \leftarrow n \cdot (|Q_s| + |Q_v|)$ invocations to oracle $\mathsf{E}_{10}()$ occurring. Invocations to $\mathsf{E}_{00}()$ serve the purpose of performing arithmetic encryption. This is done by Algorithm 1, invoked by Algorithm 4 in step 4. Invocations to $\mathsf{E}_{01}()$ serve the purpose of computing the entity s. There is only one invocation performed by each sign and verification query. The invocation happens in line 4 of the invoked Algorithm 2 for sign queries, and line 9 of Algorithm 5 for verification queries. Invocations to $\mathsf{E}_{10}()$ serve the purpose of computing OTPs for linear checksums. There are n invocations performed by each sign and verification query. Invocations happen in line 4 of the invoked Algorithm 3 for sign queries, and line 13 of Algorithm 5 for verification queries. As in the proof of Theorem A.3, the invocations to these oracles can be replaced by invocations to truncated output random oracles $\mathcal{R}_{00}()$, $\mathcal{R}_{01}()$ and $\mathcal{R}_{10}()$, provided that a corrective additive term $(\mathbf{Adv}^{\mathsf{E}_{00}(),\mathsf{dis}}_{|Q|_{00}} + \mathbf{Adv}^{\mathsf{E}_{01}(),\mathsf{dis}}_{|Q|_{01}} + \mathbf{Adv}^{\mathsf{E}_{10}(),\mathsf{dis}}_{|Q|_{10}})$ is introduced in the bound of (30). Furthermore, as in the proof of Theorem A.3., random oracle responses are statistically independent from each other and from the verification check. Therefore, all conditions in the bound or relation (30) can be removed in the presence of the corrective additive term:

$$\begin{aligned}\mathsf{F}(\mathcal{A}^{\textit{ws-MAC}(),Q_s,Q_v,\mathsf{dis}}_{\mathsf{MAC}}) \leq \\ Pr[\ (\mathcal{C}^{[r]}, a^{[r]}) \ \leftarrow \mathcal{A}^{\textit{ws-MAC}(),Q_s,Q_v,\mathsf{dis}}_{\mathsf{MAC}}, \\ \nexists \, j \in [0, |Q_s| - 1] : (\textit{ws-MAC}_K(P^{[j]}, A^{[j]}) = \mathcal{C}^{[r]} \wedge \\ a^{[r]} = A^{[j]}), \ \textit{ws-Verify}_K(\mathcal{C}^{[r]}, a^{[r]}) = \mathsf{pass} \ ] + \\ \mathbf{Adv}^{\mathsf{E}_{00}(),\mathsf{dis}}_{|Q|_{00}} + \mathbf{Adv}^{\mathsf{E}_{01}(),\mathsf{dis}}_{|Q|_{01}} + \mathbf{Adv}^{\mathsf{E}_{10}(),\mathsf{dis}}_{|Q|_{10}}\end{aligned} \tag{31}$$

We proceed with the proof estimating a bound for the probability of the event *ws-Verify*$_K(\mathcal{C}^{[r]}, a^{[r]}) = \mathsf{pass}$, when $\mathcal{C}^{[r]}$ has not been returned by a sign query and no conditions are present. Once again, we replace the singe invocation to $\mathsf{E}_{01}()$ and the n invocations to $\mathsf{E}_{10}()$, which happen as

part of the verification check *ws-Verify*$_K(\mathcal{C}^{[r]}, a^{[r]}) = \mathsf{pass}$, with the same number of invocations to truncated output random oracles $\mathcal{R}_{01}()$ and $\mathcal{R}_{10}()$ as above. It is not difficult to see that, if in the check *ws-Verify*$_K(\mathcal{C}^{[r]}, a^{[r]}) = \mathsf{pass}$, Algorithm 7, and the invoked Algorithm 5 query truncated output random oracles, then the probability of the event *ws-Verify*$_K(\mathcal{C}^{[r]}, a^{[r]}) = \mathsf{pass}$ is the probability that a random uniformly distributed entity s is congruent mod q to any of the roots of a polynomial of degree m, also defined mod q. As there can by at most m roots and s mod q is uniformly distributed in the set $[0, q-1]$, it holds that:

$$\mathsf{F}(\mathcal{A}^{\textit{ws-MAC}(),Q_s,Q_v,\mathsf{dis}}_{\mathsf{MAC}}) \leq \frac{m}{q} + \mathbf{Adv}^{\mathsf{E}_{00}(),\mathsf{dis}}_{|Q|_{00}} + \mathbf{Adv}^{\mathsf{E}_{01}(),\mathsf{dis}}_{|Q|_{01}+n} + \mathbf{Adv}^{\mathsf{E}_{10}(),\mathsf{dis}}_{|Q|_{10}+1} \tag{32}$$

where $|Q|_{00} \leftarrow \frac{n \cdot m \cdot w_e}{w_c} \cdot |Q_s|$, $|Q|_{01} \leftarrow |Q_s| + |Q_v|$ and $|Q|_{10} \leftarrow n \cdot |Q_s| + |Q_v|)$.

The bound of relation (32) depends only on the number of queries in sets Q_s and Q_v and not on the queries themselves. Furthermore, the bound is non-decreasing with the number of queries in Q_s, Q_v. We introduce the notation $\mathsf{F}(\mathcal{A}^{\textit{ws-MAC}(),Q_s,Q_v,\mathsf{dis}}_{\mathsf{MAC}})^{[i]}$ to refer to the advantage of relation (30), when the game of the adversary includes only queries up to verification query i, for some $i \leq |Q_v| - 1$ and verification query i is the output of the adversary. Since the bound of relation (32) is non-decreasing with the number of queries in Q_s, Q_v, it holds that:

$$\mathsf{F}(\mathcal{A}^{\textit{ws-MAC}(),Q_s,Q_v,\mathsf{dis}}_{\mathsf{MAC}})^{[i]} \leq \mathsf{F}(\mathcal{A}^{\textit{ws-MAC}(),Q_s,Q_v,\mathsf{dis}}_{\mathsf{MAC}}) \tag{33}$$

for every $i \in [0, |Q_v| - 1]$. Furthermore:

$$\mathbf{Adv}(\mathcal{A}^{\textit{ws-MAC}(),Q_s,Q_v,\mathsf{dis}}_{\mathsf{MAC}}) \leq \sum_{i=0}^{|Q_v|-1} \mathsf{F}(\mathcal{A}^{\textit{ws-MAC}(),Q_s,Q_v,\mathsf{dis}}_{\mathsf{MAC}})^{[i]} \leq |Q_v| \cdot \mathsf{F}(\mathcal{A}^{\textit{ws-MAC}(),Q_s,Q_v,\mathsf{dis}}_{\mathsf{MAC}}) \tag{34}$$

Theorem A.4 follows directly from relations (32) and (34). This concludes the proof and the analysis that establishes the security of the arithmetic encryption and encrypted linear checksum mechanisms.

■

D. Another Construction of Linear Checksum

Here we present another construction for Linear checksum. Instead of only using w_t bits from the AES cipher output as s in Alg. 2, we use all w_c bits in the checksum s.

Algorithm 8: Linear Checksum with More Randomness

Inputs: K, P_i, $paddr(P)$
Output: T_i
$v \leftarrow \mathcal{V}()$ //drawn once for the matrix P
$e_s = E(K, 01||paddr(P)||v)$ //v padded with zeros;
$cnt_s = w_c/w_t$
//we define every w_t-bit substring of e_s
for $k = 0$ *to* cnt_s. **do**
  $s_k = e_s[k \times w_t : (k+1) \times w_t]$;
end
$T_i \leftarrow \sum_{j=0}^{m-1} P_{i,j} \times (s_{[(m-j) \mod cnt_s]})^{\lfloor (m-j)/cnt_s \rfloor}$
$T_i \leftarrow T_i \mod q$
Return T_i;

Proposition: This construction will change Theorem A.4, where the quantity $\frac{m}{q}$ in the bound must be replaced replaced by the quantity $\frac{m}{cnt_s \cdot q}$, which is lower.

Proof: Following similar steps, as in the proof of Theorem A.4, we see that the verification check *ws-Verify*$_K(\mathcal{C}^{[r]}, a^{[r]}) = \mathsf{pass}$ of relation (31) is now reduced to an equation of the form:

$$\sum_{j=0}^{m-1} \mathcal{C}_j \times (s_{[(m-j) \mod cnt_s]})^{\lfloor (m-j)/cnt_s \rfloor} \mod q = 0 \tag{35}$$

for some entities $\mathcal{C}_i$, $j \in [0, m-1]$, which are not all zero, and substrings $s_k, k \in [0, cnt_s - 1]$. Substrings $s_k, k \in [0, cnt_s - 1]$ are random uniformly distributed, and statistically independent of each other. Relation (35) is a non-zero polynomial equation of degree at most $\frac{m}{cnt_s}$ for a single substring s_k. So, with the values of all other substrings given, and by the fundamental theorem of algebra, equation (35) has at most $\frac{m}{cnt_s}$ solutions for this substring s_k. Thus, out of the q possible s_k values, at most $\frac{m}{cnt_s}$ satisfy (35) . Consequently, the bound of Theorem A.4 must change and the the quantity $\frac{m}{q}$ in the bound must be replaced replaced by the lower quantity $\frac{m}{cnt_s \cdot q}$. ■