HEAR: Homomorphically Encrypted Allreduce

Marcin Chrapek
Department of Computer Science
ETH Zurich
Zurich, Switzerland
marcin.chrapek@inf.ethz.ch

Mikhail Khalilov
Department of Computer Science
ETH Zurich
Zurich, Switzerland
mikhail.khalilov@inf.ethz.ch

Torsten Hoefler
Department of Computer Science
ETH Zurich
Zurich, Switzerland
torsten.hoefler@inf.ethz.ch

ABSTRACT

Allreduce is one of the most commonly used collective operations. Its latency and bandwidth can be improved by offloading the calculations to the network. However, no way exists to conduct such offloading securely; in state-of-the-art solutions, the data is passed unprotected into the network. Security is a significant concern for High-Performance Computing applications, but achieving it while maintaining performance remains challenging. We present HEAR, the first high-performance system for securing in-network compute and Allreduce operations based on homomorphic encryption. HEAR implements carefully designed and modified encryption schemes for the most common Allreduce functions and leverages communication domain knowledge in MPI programs to obtain decryption and encryption routines with high performance. HEAR operates on integers and floats with no code base and no or little hardware changes. We design and evaluate HEAR, showing its minimal overhead, and open-source our implementation. HEAR represents the first step towards achieving confidential HPC.

CCS CONCEPTS

• **Networks** → **In-network processing**; • **Security and privacy** → **Distributed systems security**; • **Computer systems organization** → **Distributed architectures**.

KEYWORDS

Message Passing Interface, Allreduce, In-Network Computing, Homomorphic Encryption, Confidential Computing

ACM Reference Format:
Marcin Chrapek, Mikhail Khalilov, and Torsten Hoefler. 2023. HEAR: Homomorphically Encrypted Allreduce. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3581784.3607099

1 INTRODUCTION

Allreduce is the most used [21, 62, 76, 10] message-passing interface (MPI) collective in which P processes belonging to a communicator $\mathbb{S}$ aggregate a vector of data element-wise and return the aggregated values to each process. Summation, product, and `max`/`min` operators

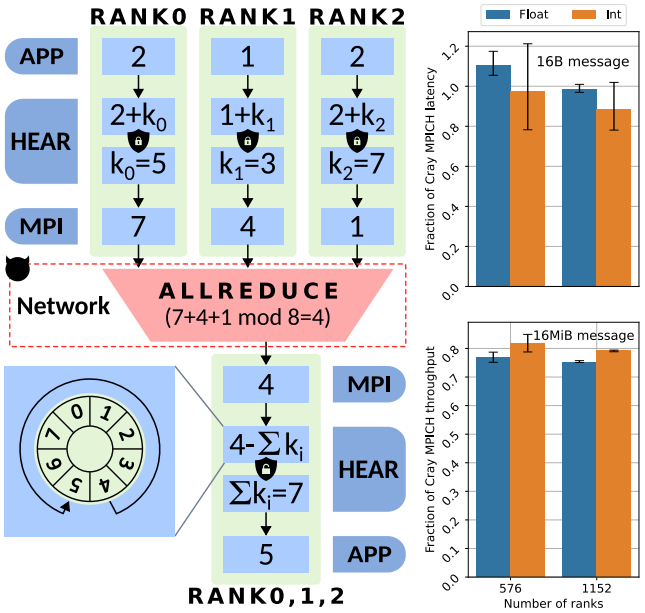

Figure 1: A simplified overview of HEAR running within a secure environment with an untrusted network. HEAR conducts homomorphically encrypted Allreduce by shifting the values on a ring, ensuring losslessness and enabling performant decryption through spatial knowledge of the program.

are typical examples of aggregation functions [64]. Many applications, such as scientific workloads [21, 44, 89], deep learning [88, 20, 8], graph processing [100, 51], and big data analytics [100, 101], rely on Allreduce. Recent studies show that Allreduce, together with the related Reduce collective, are the most commonly invoked collectives taking over 30-40% of the total collective operation time in core hours [21, 76] and making *"MPI_Allreduce [...] the most significant collective in terms of usage and time"*.

To increase the performance and lower the amount of transmitted data of Allreduce, the processes can exploit in-network computing (INC). INC allows hosts to offload the Allreduce function to the networking devices such as switches [45, 60, 84] or network interface cards (NICs) [86, 50]. INC provides two critical advantages: latency and bandwidth usage reductions. The former is lowered by 3-18x [45, 60] which results in performance gains of 1.5-5.5x [60, 84]. The latter can be reduced by 2x [27], resulting in a lower network contention and system power savings as modern datacenter interconnects constitute 15-50% of the overall power usage [2].

However, INC is insecure, and currently, no method exists for securing it. As the information required for computations is used unencrypted, the state-of-the-art sends the INC data unprotected

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11…$15.00
https://doi.org/10.1145/3581784.3607099

in the network [45], making INC vulnerable to attacks. This raises an issue as an increasing number of users, companies, and governments emphasize data anonymity and confidentiality [73, 40, 96]. The growing importance of security led to confidential computing (CC) gaining attention.

CC refers to methods to protect sensitive data while being processed [67, 77]. One of the crucial elements of CC is homomorphic encryption (HE). HE allows conducting arithmetic operations on the encrypted messages without needing to decrypt them first [3, 98, 70]. HE has been identified to have significant potential in applications such as medical records [68, 97], biometric data [29], financial data [68], advertising [68], and AI [7, 91]. While HE provides strong security guarantees, most existing schemes are impractical because of the increased size of the encrypted message (ciphertext) over the original message (plaintext) or the low performance [3].

Performance overheads usually make HE impractical for HPC use cases such as securing INC. While security is an essential feature of HPC, enabling it to achieve its anticipated benefits [47], HPC has unique security requirements unlike those in typical IT systems. One of the most crucial differentiators is the trade-off between performance and security, where the users consider security valuable only if it does not slow down calculations significantly [47]. Such requirements made confidential HPC (cHPC) restrictive.

We present HEAR, a first step towards achieving cHPC. HEAR is a novel system for securing INC and Allreduce operations based on homomorphic encryption. HEAR implements carefully selected and modified encryption schemes for the most common Allreduce functions. It leverages the knowledge of the communication domain in the MPI programs to obtain decryption and encryption routines with high performance and low to no ciphertext overhead, thus fulfilling the HPC security requirements. HEAR operates on integers and floats with no or minimal hardware changes. Figure 1 displays a simplified overview of HEAR while conducting an intuitive version of the integer Allreduce summation. Each MPI rank encrypts the data by adding on a ring a pseudorandom key deterministically generated from initialization keys known by all ranks. The scheme provides homomorphic addition properties, and because of modulo arithmetic, no information is lost. The resulting aggregated value can be efficiently decrypted by subtracting the sum of all the keys. As the performance plots show, HEAR achieves around 80% of the native network throughput and introduces low latency overheads, considering the provided security benefits. In summary, our contributions are:

(1) Defining a suitable INC and cHPC threat model alongside HEAR, a novel framework for confidential Allreduce operations allowing users to choose between performance and security. Homomorphically based HEAR works on integers and floats and supports addition, multiplication, and `XOR`.
(2) Defining and discussing the security of a new floating point HE scheme suitable for HPC usage. Presenting the analysis of the precision loss against the gained security for the floating points within the scheme.
(3) Designing, implementing, and evaluating HEAR, the first high-performance Allreduce framework based on MPI stack and datatype, allowing for confidential INC.
(4) Open-sourcing HEAR[1] as a library for use with any MPI standard implementation without changing the code base and recompiling applications.

2 BACKGROUND

Security attacker models: Attacker models play a crucial role in evaluating the security of cryptographic schemes. One widely used attacker model is the indistinguishability under the chosen-plaintext attack (IND-CPA) model, which assesses a scheme's resilience to attacks where an adversary can choose plaintext messages to encrypt and observe the corresponding ciphertexts [58]. IND-CPA is equivalent to semantic security, where only negligible information can be extracted about the plaintext from the ciphertext [41]. Another important attacker model is the ciphertext-only attack (COA) [11], which evaluates a scheme's resistance to attacks where an adversary can only access ciphertexts. Attacker models are essential in providing a rigorous evaluation of the security of cryptographic systems. Their use in proofs offers a high level of confidence in the effectiveness of a given cryptographic scheme.

Secure environments: The ability to evaluate programs in a secure environment is a crucial element of CC. Secure environments can be achieved by, for example, using Trusted Execution Environments (TEEs) [102]. TEEs are isolated, secure execution environments that protect sensitive data and code from being accessed or tampered with by unauthorized parties outside the TEE [81]. TEEs such as Intel SGX [25], AMD SNP-SEV [57] or ARM TrustZone [74] are based on hardware security mechanisms that create secure and isolated regions within the system's memory. While TEEs provide security of the data on the host, their communication is not well defined [103]. Existing solutions [93] do not support INC and would require increasing the threat boundary. To the best of our knowledge, no prior research allows data from a secure environment to be combined with INC.

Combining MPI, INC, and secure environments: Usually, the communication between MPI processes is either unencrypted [5, 93, 79] or encrypted using end-to-end (E2E) techniques such as TLS [80]. However, turning the encryption off for some applications is unacceptable, and E2E encryption prevents the users from leveraging INC. For TEEs, turning off encryption makes little sense as it would expose the protected data. Relaying the encryption and decryption to the network devices increases the threat boundary and requires either key sharing or attestation. For key sharing to work with INC, keys would need to be shared with all the networking devices as the ones involved in the computation are not known a priori, especially for dynamically routed networks. Additionally, this would introduce decryption and encryption latencies on the involved networking devices. On the other hand, attesting and opening TEEs on all network devices introduces further overhead in TEE context switching. Each context switch requires flushing of translation lookaside buffers or last-level caches [25], making achieving the line rate and low latencies challenging.

The current unencrypted INC schemes have two main issues that should be solved: data confidentiality and result authentication. In the former case, the INC devices can read unprotected data and obtain potentially confidential information. In the latter

[1] https://github.com/spcl/libhear

	PHE							SWHE			FHE				
	[78]	[42]	[33]	[9]	[72]	[23]	[85]	[14]	[83]	[12]	[55]	[19]	[13]	[34]	[17]	**HEAR**
R1	◐	○	◐	◐	◐	○	●	●	○	○	○	○	○	○	○	●
R2	●	●	●	●	●	◐	●	●	◐	◐	◐	●	●	●	●	●
R3	◐	○	◐	◐	◐	●	●	●	○	○	○	○	○	○	○	●
R4	○	○	○	○	○	◐	◐	○	◐	◐	◐	●	●	●	●	●

Table 1: Comparison of the available encryption schemes with different requirements. R1: At most 2x ciphertext inflation, R2: A large number of operations, R3: Low operation complexity, R4: Allow for many operation types. ○ signifies not fulfilling, ◐ partially fulfilling, ● entirely fulfilling.

case, the network devices can manipulate the data freely without the MPI program realizing it. In our work, we primarily focus on confidentiality. We also discuss the authentication in Section 5.5.

Homomorphic encryption (HE): HE allows conducting arithmetic operations on encrypted messages. Encryption scheme with encryption E and message space M is homomorphic for an operation $\star$ if it fulfills $E(x_1) \star E(x_2) = E(x_1 \star x_2) \forall x_1, x_2 \in M$. We categorize HE into three categories: partially, somehow, and fully homomorphic schemes [3]. Partially HE (PHE) allows a single operation type on encrypted data, e.g., addition or multiplication. Somewhat HE (SWHE) can evaluate multiple operation types but only a limited number of times. Fully HE (FHE) allows for arbitrary operations and unlimited evaluations. The choice between different schemes depends on the application requirements.

3 DESIGN REQUIREMENTS

The requirements of confidential HPC are radically different from those of typical HE applications, making most HE schemes impractical due to the increased size of the ciphertext over plaintext (ciphertext inflation), a limited number of operations, and high computation intensities of encryption, decryption, and homomorphic operations. HEAR cannot use an off-the-shelf encryption scheme because such overheads are unacceptable in a distributed HPC environment. HEAR required a design of its own HE methods inspired by some existing frameworks [85, 14]. We present a comparison of HEAR and evaluate common encryption schemes to show their shortcomings. We summarize this analysis in Table 1.

R1: At most 2x ciphertext inflation. INC provides bandwidth reduction of up to 2x [27]. As the sent message is encrypted, the cryptographic schemes should have ciphertext inflation as low as possible and not larger than 2x. Otherwise, one of the main advantages of INC would be eliminated. Most HE schemes do not fulfill this condition. FHE schemes are notorious for large inflation, reaching 10,000x for bit-sized inputs with some keys reaching gigabytes [3, 23]. Newer FHE schemes have considerably reduced the inflation [19, 18], which, as the size of the message grows to infinity, can reach 2x still eliminating the INC advantages. Conceptually simpler and more efficient PHE schemes also rarely achieve this goal. Paillier [72], RSA [78] or ElGamal [33] obtain at least 2x. Unmodified additive ring-based schemes also do not offer a solution here due to increasing key arrays [85]. HEAR bases its operation on additive ring schemes and cleverly uses the evaluated program's structure and communication to eliminate ciphertext inflation for integers. For floats, HEAR provides the designers with a parameter that controls the trade-off between inflation, security, and the achieved precision.

R2: A large number of operations. Many HE schemes suffer from a limited number of operations that they can perform, after which the result is not guaranteed to be decrypted appropriately. For example, SWHE schemes usually increase noise after each iteration [3]. Bootstrapping [39] is the technique commonly used in FHE schemes to deal with this problem. It requires periodic refreshing of the ciphertext to reduce the aggregated noise, which is typically slow and requires large keys [3]. Additionally, some HE schemes that support a limited number of operations have a variable ciphertext length [23]. The length depends on the limit of the number of operations leading to more complex hardware. HEAR offers an unlimited number of operations and a constant ciphertext size for integers and floats.

R3: Low operation complexity. Many HE schemes are complex. The encryption, decryption, and operation times are typically slow and in the range of milliseconds to hours [3]. FHE encryption schemes can take minutes to conduct simple 16-bit additions, not due to the complexity of the conducted operations but due to their ciphertext inflation [23]. Even if we operate in the limit where ciphertext inflation does not matter, encryption and decryption remain complex operations [17]. PHE schemes suffer from more complex functions involving, for example, large modulus operations [78, 33, 72]. HEAR is defined by basic arithmetic and uses MPI communication structure to allow for performant operation.

R4: Allow for many operation types. SWHE schemes allow a subset of operations while FHE schemes all of them [3]. Some PHE frameworks offer multiple schemes based on principles readily convertible to different operations [85, 23]. HEAR follows a similar strategy and implements six schemes to enable various operations.

4 THREAT MODEL

We assume three parties operating within the same network; a set $\mathbb{S}$ of P processes together with HEAR running within secure environments that established a communication domain, the cluster user who legitimately submits a program and its data to be evaluated, and an adversary who has modify access to the data input methods and can observe the whole network. Due to the number of available options [102], we do not limit ourselves to a particular type of secure environment. We leave it to the user to decide their level of trust, e.g., whether they would like to run a TEE on the whole or parts of the program or simply trust the nodes. We assume that the secure environment prohibits the adversary from accessing the confidential parts of the program and HEAR's keys and methods. The adversary tries to obtain confidential information by eavesdropping on the network transmissions that leave the secure environment. Any elements within the network, such as the NICs and routers, are untrusted. The adversary can observe the network and influence the data provided to the securely running program. An example of an adversary would be a malicious sysadmin. Figure 2 presents an overview of the threat model with the trusted elements in a green outline and untrusted ones in a red outline.

To differentiate between the security levels HEAR provides against such an adversary, we introduce three types of safety: temporal, local, and global. We assume processes submit a vector of values to HEAR, and we consider the plaintext to be any element within a provided Allreduce vector on any of the processes. We

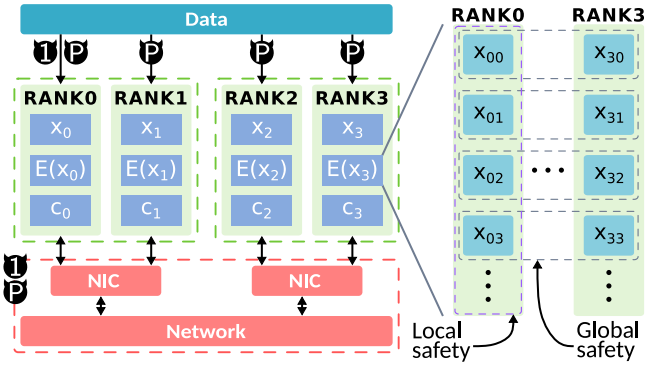

Figure 2: The threat model of HEAR with the processes running within secure environments, untrusted network, and overview of the provided safety.

marked the various plaintexts with x_{ij} in Figure 2. In such an environment, with a high probability, temporal safety ensures a given plaintext value during consecutive Allreduce operations is encrypted differently. Local safety ensures plaintexts in a process-local vector are encrypted differently during the same Allreduce operation. Global safety ensures plaintexts across different processes are encrypted differently. Figure 2 displays what we mean by local and global safety, showing sets of values that should not get encrypted to the same ciphertext.

HEAR always provides local and temporal security. We define two attacker types to distinguish between situations with and without global safety:

(1) **Single-process attacker** 1: can submit message data to one process. In practice, this could be a process to which the operator has obtained access through, for example, expensive side-channel attacks.
(2) **Multi-process attacker** P: can submit message data on all P processes. In practice, this could happen at the end of program evaluation, where the sysadmin pretends that the communication has ended. At the same time, they overtake the communication domain and continue submissions to obtain information from the previously logged messages.

Both are stronger versions of passive attackers just observing the network. We primarily consider confidentiality attacks on the message contents. We discuss integrity protection in Section 5.5.

5 HEAR DESIGN

HEAR optimizes the performance and scalability of supported data and operation types while minimizing the need to change existing INC hardware. HEAR is built on symmetric primitives, allowing for homomorphic operations in the network with consecutive independent ciphertexts, yielding ground for performant parallelization. All the schemes used in HEAR can be intuitively summarized as:

$$E(x) = x \star \text{noise} \quad D(x) = x \star \text{noise}^{-1}$$

For an encryption function E, decryption function D, plaintext x, operation $\star$, and a noise derived from keys providing security. In Sections 5.1-5.3, we define these parameters for distinct data and operation types, reflecting the underlying value encoding differences. Each scheme includes definitions of encryption/decryption, and an overview of performance, security, and lossiness. Table 2 displays the supported operations and their main properties. Table 3 provides intuition behind the presented math.

Key Generation: HEAR is initialized by generating keys within processes belonging to a given communicator. The initialization is per communicator, even if some processes are already initialized in a different communicator. In this phase, each of the P processes with rank i involved in a communicator $\mathbb{S}$ generates a local starting random key $k_i^s \in \mathbb{Z}_n$. k_i^s is kept secret and is securely shared with selected other processes. Each rank i stores its key, and the keys of rank $i-1$ and 0. Those keys are necessary for better performance, as discussed in Section 5.1.4. Rank 0 also generates a single collective random key $k_c \in \mathbb{Z}_n$ together with the encryption $k_e \in \mathbb{Z}_m$ and progression $k_p \in \mathbb{Z}_m$ keys. The three values are kept secret and broadcasted securely to all the other processes. Additionally, we let a pseudorandom function (PRF) $F : \{0,1\}^n \times \{0,1\}^m \to \mathbb{Z}_d$ be a trapdoor function defined as $F_k(x) = F(k,x)$ and mapping values of length n with key of length m to elements in $\mathbb{Z}_d$. F needs to be a cryptographically secure PRF such as AES [31].

After the initialization phase, each process has six keys providing performant scaling as $\Theta(1)$ in space. d, m and n are user selectable parameters. The output length d is dictated by the size of the datatype operated on; integers and floats have different encryption requirements resulting in different output lengths. The required application security and the used PRF dictate the key size m. The input value n is usually defined by the used PRF.

HEAR relies on the communication structure of `Allreduce` to achieve high performance. During any call to conduct an `Allreduce`, the processes within the communicator increase the collective random number as $k_c \leftarrow F_{k_p}(k_c)$. These updates are necessary to provide temporal safety. Then each rank i encrypts each element j of the vector of values x_i to a ciphertext $c_i[j] = E(x_i[j])$ using one of the schemes defined in Sections 5.1-5.3 dependent on the data and operation types. After an aggregation using an operation $\odot$, each of the ranks obtains $c[j] = \bigodot_{i\in\mathbb{S}} c_i[j]$ which can be decrypted using a function $D_{\mathbb{S}}(c[j]) = \bigodot_{i\in\mathbb{S}} x_i[j]$. In HEAR, the local safety is guaranteed by using different j to encrypt different elements of a vector, while the global safety is provided by the different local starting keys k_i^s.

5.1 Integer operations

We define integer-based schemes that do not require hardware changes, are lossless, and have no ciphertext inflation.

5.1.1 Addition. We define addition on an abelian additive group $\mathbb{Z}_d$ with order $d > 1$ dictated by the datatype length. For example, for 64-bit integers, PRF F would return 64-bit pseudorandom numbers. Subtraction can be implemented using two's complement.

Encryption: We encrypt values as:

$$c_i[j] = \begin{cases} x_i[j] + F_{k_e}(k_i^s + k_c + j) & i = P-1 \\ x_i[j] + F_{k_e}(k_i^s + k_c + j) - F_{k_e}(k_{i+1}^s + k_c + j) & \text{otherwise} \end{cases} \quad (1)$$

Decryption: We decrypt as $D_{\mathbb{S}}(c[j]) = c[j] - F_{k_e}(k_0^s + k_c + j)$.

Losslessness: Using a well-known modulo arithmetic property $(a_1 + a_2) \mod 2^b = (a_1 \mod 2^b + a_2 \mod 2^b) \mod 2^b$ we

	Int, Fixed point		Int, Bool	Float, Complex		
	MPI_SUM	MPI_PROD	MPI_LXOR, MPI_BXOR	MPI_SUM v1	MPI_SUM v2	MPI_PROD
Lossiness	Lossless	Lossless	Lossless	Minor	Medium	Minor
Security	IND-CPA❶❷	IND-CPA❶❷	IND-CPA❶❷	COA❶	COA❶❷	COA❶❷
Ciphertext inflation	None	None	None	Precision tradeoff	Precision tradeoff	Precision tradeoff
Hardware changes	None	None	None	Minimal, FPU	Minimal, FPU	Minimal, FPU

Table 2: Summary of supported operation and data types together with their properties.

note that $((m+n) \mod 2^b - n \mod 2^b) \mod 2^b = (m+n-n) \mod 2^b = m \mod 2^b$. Hence, the operations are lossless.

Security: The scheme defined by encrypting x with $x+k$ is IND-CPA secure as long as k is pseudorandom and unique between encryptions as shown in [15, 14, 85] which our scheme fulfills yielding semantic security under both adversary models.

5.1.2 Multiplication. Is defined on an abelian multiplicative group $\mathbb{Z}_d^*$ with the order $d > 1$ equal to the datatype length where we use a pre-selected generator g of the group. Modulo division is equivalent to multiplying by the modular inverse of a number.

Encryption: we encrypt values as:

$$c_i[j] = \begin{cases} x_i[j] \times g^{F_{k_e}(k_i^s + k_c + j)} & i = P-1 \\ x_i[j] \times g^{F_{k_e}(k_i^s + k_c + j) - F_{k_e}(k_{i+1}^s + k_c + j)} & \text{otherwise} \end{cases} \quad (2)$$

Decryption: We decrypt as $D_{\mathbb{S}}(c[j]) = c[j] \times g^{-F_{k_e}(k_0^s + k_c + j)}$.

Losslessness: As g is a generator, each noise used for encryption in Equation 2 has an inverse implying no loss of information due to inverse multiplication in modulo arithmetic.

Security: The scheme is IND-CPA secure under both adversary models as the keys are unique and pseudorandom as shown in [85].

5.1.3 Logical and binary `XOR`*.* Are defined similarly to AES, where d equals the datatype length.

Encryption: we encrypt values as:

$$c_i[j] = \begin{cases} x_i[j] \oplus F_{k_e}(k_i^s + k_c + j) & i = P-1 \\ x_i[j] \oplus F_{k_e}(k_i^s + k_c + j) \oplus F_{k_e}(k_{i+1}^s + k_c + j) & \text{otherwise} \end{cases} \quad (3)$$

Decryption: We decrypt as $D_{\mathbb{S}}(c[j]) = c[j] \oplus F_{k_e}(k_0^s + k_c + j)$.

Loss of information: `XOR` is inverible and follows $a \oplus a = 0$, $a \oplus 0 = a$ yielding losslessness.

Security: The scheme is equivalent to the IND-CPA AES-CTR algorithm [31, 30] under both adversary models due to the unique and pseudorandom inputs provided.

5.1.4 Performance. Apart from simple algebraic operations, our methods require evaluating two PRFs for encryption and one for decryption. Additionally, encryption and decryption in multiplication require $O(\log d)$ modulo exponentiation. As our modulo is 2^n, techniques such as the 2^k-ary method can improve the worst-case performance [43].

To improve performance, we use the canceling technique [14, 85]. For all the methods, the noise from process $i+1$ is eliminated by the noise from process i. This lowers the overall number of operations. The version of the scheme presented in Figure 1 is implemented without the aforementioned cancellation. The encryption then has one PRF evaluation, but decryption involves computing $\bigodot_{i \in \mathbb{S}} F_{k_e}(k_i^s + k_c + j)$ scaling as $\theta(P)$ instead of $\theta(1)$.

5.2 Fixed point operations

Fixed point transmissions can be implemented similarly to integer transmissions. The implicit scaling factor has to be agreed upon before any computations and is shared securely with all ranks. Calculations rely on the same algorithms as in the integer case. The number of involved processes can be used to obtain the correct output scaling factor for multiplication and division. Fixed point arithmetic is helpful in some machine learning applications involving quantization and inference [48, 63, 56, 35, 53, 69] and is often used as a replacement for floating points in HE schemes [17, 24].

5.3 Floating point operations

While floats form the basis for most scientific computing, to the best of our knowledge, there does not exist a floating point HE framework that would be performant. To address this gap, we introduce HFP, a part of HEAR. Before formally discussing HFP in Section 5.3.1, we make the following assumption about the representation of a floating point number a with a sign bit s, a base b, an exponent e, and mantissa m:

$$a = (-1)^s \times m \times b^e = (-1)^s \times 1.m_1 m_2 ... m_{l_m} \times b^{e_1 e_2 ... e_{l_e}} \quad (4)$$

We assume e of binary length l_e and m of binary length l_m. We set the mantissa m to a normalized hidden one format where the leading one is implicit and is not stored in the underlying binary representation. We set the exponent e to a two's complement format. We define the set of values this encoding represents as $\mathbb{F}$. While similar to the IEEE 754 standard [54], we do not limit ourselves to its specifics (subnormals, special numbers, offset stored exponents, etc.). The base b is agreed upon a prior (e.g., 2 in IEEE 754).

5.3.1 Cryptographic scheme. We formally define a floating point encryption scheme HFP=(gen, enc, dec) relying on multiplying the float x by a noise. Beforehand, we provide the intuition behind it. We first move from a representation such as IEEE 754 to one that operates on a ring of values. We then place the floats along such a ring, ensuring that multiplication or addition can be evaluated. The adversary does not know the shifting in the ring, which provides security. The float value ring is achieved by creating a ring on the exponents. In the case of addition, we expand the exponent length by two bits and introduce a new addition arithmetic. In the case of multiplication, we turn the exponent into a ring without expansion. Both might require hardware changes, as explained in Section 5.3.6.

A different perspective to look at this scheme is by considering log number systems [59, 92] where the multiplication of arguments turns into an addition of their logarithms. Moving the floating point into a logarithmic space allows us to change the encoding from Equation 4 to fixed point integers, enabling the same security as in the integer case.

	Int, 4 bits, modulo 2^4=16, subgroup generator 3						Float, half precision ($l_e = 5$, $l_m = 10$)			
	MPI_SUM (sec. 5.1.1)		MPI_PROD (sec. 5.1.2)		MPI_BXOR (sec. 5.1.3)		MPI_SUM (sec. 5.3.3)		MPI_PROD (sec. 5.3.2)	
	Rank 1	Rank 2	Rank 1	Rank 2	Rank 1	Rank 2	Rank 1	Rank 2	Rank 1	Rank 2
Values	[1, 5]	[3, 8]	[2, 4]	[7, 2]	0011	0010	1.75×2^{7}	1.25×2^{9}	1.125×2^{9}	1.375×2^{1}
Expected	[4, 13]		[14, 8]		0001		1.6875×2^{9}		1.547×2^{10}	
Noise	[2, 1]	[1, 7]	$[3^1 = 3, 3^2 = 9]$	$[3^1 = 3, 3^0 = 1]$	0101	1001	1.5×2^{13}		1.75×2^{22}	1.25×2^{-13}
Method	Add (eq. 1)		Multiply (eq. 2)		XOR (eq. 3)		Multiply (eq. 7)		Multiply (eq. 6)	
Encrypted	[2, 15]	[4, 15]	[2, 4]	[5, 2]	1111	1011	1.3125×2^{21}	1.875×2^{22}	1.575×2^{44}	1.719×2^{-12}
Reduced	[6, 14]		[10, 8]		0100		1.266×2^{23}		1.354×2^{33}	
De-noise	[2, 1]		$[3^{-1} = 11, 9^{-1} = 9]$		0101		1.5×2^{13}		1.75×2^{22}	
Method	Subtract		Multiply		XOR		Divide		Divide	
Decrypted	[4, 13]		[14, 8]		0001		1.6875×2^{9}		1.547×2^{10}	

Table 3: Simplified examples of encryption and decryption for different schemes and operations. Note that 3 is only a subgroup generator used for illustrative purposes and does not provide the required security.

Similarly to the key generation procedure at the beginning of Section 5, we assume a PRF $F_k(x)$ mapping inputs of length n and generating values in $\mathbb{F}$. n is a user-selectable security parameter, and the size d of the output of the PRF is the size of the encrypted floating point data type plus γ. γ represents a user-selectable parameter controlling the inflation of the data type and its tradeoff with precision loss and COA security. γ is at least zero and is discussed further in the section.

gen(1^n): generate a symmetric key from uniform $k \in \{0,1\}^n$.

enc(k, x): choose uniform $r \in \{0,1\}^n$. Consider plaintext $x \in \mathbb{F}$ decomposed as $(-1)^{s_x} \times m_x \times 2^{e_x}$ and noise $F_k(r) \in \mathbb{F}$ as $(-1)^{s_f} \times m_f \times 2^{e_f}$ where $l_{m_f} = l_{m_x} - \delta + \gamma$ and $l_{e_f} = l_{e_x} + \delta$. Output the ciphertext c:

$$c = x \otimes F_k(r) = (-1)^{s'} \times m' \times 2^{e'} \tag{5}$$

where $\otimes$ is defined such that $s' = s_x + s_f \mod 2$, $m' = m_x \times m_f$ and $e' = e_x + e_f \mod 2^{l_e+\delta}$. The resulting m' should be of length $l_m - \delta + \gamma$. If m' is not in the hidden one format, it must be normalized by shifting.

δ is necessary for correct functioning and security purposes as described in Section 5.3.5. δ introduces additional bits to the exponent, increasing its length. The value of δ depends on which homomorphic operation the encryption should support. It equals zero for multiplication and two for addition. γ controls precision loss due to encryption/decryption by adding ciphertext inflation bits to the representation. The optimal choice of γ depends on the supported homomorphic operation. The most performant operations would be aligned with current data type lengths, implying γ equal to zero for multiplication and two for additions.

dec(k, r, c): output the plaintext $x = c \otimes F_k(r)^{-1}$ where $F_k(r)^{-1} = (-1)^{s_f} \times 1/m_f \times 2^{-e_f}$ with appropriate mantissa normalization.

Security: we discuss the scheme's security. In HFP, both sign and exponent bits operate on arithmetic rings. As such, they produce uniform outputs, giving the adversary no advantage other than brute-forcing over possible keys and finding the one providing the seen output. However, as we multiply mantissas in our encryption, the resulting ciphertexts will not be uniform but produce a piecewise smooth logarithmic distribution [28]. Such probability distribution gives the adversary a statistical advantage as some plaintexts are more likely to produce specific ciphertexts.

We show that this statistical edge is negligible and evaluate it by considering a probabilistic polynomial time adversary $\mathcal{A}$. $\mathcal{A}$ can use Bayesian statistics [95] to find the optimal strategy to obtain an edge over a simple brute-force attack. $\mathcal{A}$ implements a maximum a posteriori (MAP) [38] estimator for the plaintext X based on ciphertext C. $\mathcal{A}$ observes ciphertext c and makes a guess x_g as:

$$x_{g|c} = \text{argmax}_x \Pr(X = x | C = c)$$

Using Bayesian rules and $\Pr(C = c)$ being independent of X:

$$x_{g|c} \sim \text{argmax}_x \Pr(C = c | X = x) \Pr(X = x)$$

We assume an uninformative uniform prior $\Pr(X = x)$ over all possible mantissa leading to the likelihood:

$$x_{g|c} \sim \text{argmax}_x \Pr(C = c | X = x)$$

$\mathcal{A}$ obtains the likelihood by measuring possible ciphertexts given the random output of the PRF. We examined the average probability of success for MAP for FP32 to measure the statistical edge $\mathcal{A}$ gets through our encryption. For each possible mantissa X we measured all the possible ciphertexts C by iterating over all the possible PRF outputs. Assuming that PRF outputs are uniform, we obtained the likelihood by normalizing the results. Adversary $\mathcal{A}$ achieves the average probability of a guess of 3.57×10^{-7}. The maximum probability of a guess over all of X is 3.58×10^{-7} and the minimal 2.38×10^{-7}. The reference uniform probability of a guess is 1.19×10^{-7}. The adversary $\mathcal{A}$ only obtains a minor advantage.

Importantly, this type of attack can easily become unfeasible for the adversary $\mathcal{A}$. As the number of possible PRF outputs grows exponentially with the data type size (γ controllable), attacking through such a ciphertext attack is impractical, displaying COA properties. Additionally, obtaining keys and inputs of the PRF through this strategy is also not viable as the key space grows exponentially with the number of key bits n, assuming that the PRF is secure and that the input of the PRF is random between encryptions.

5.3.2 Multiplication. We base multiplication on canceling the noise between nodes and the exponent logic described in Section 5.3.5 without any exponent inflation ($\delta = 0$).

Encryption: we encode values as in equation 4. Then $x_i[j]$ can be encrypted to a ciphertext $c_i[j]$ assuming the same definition of

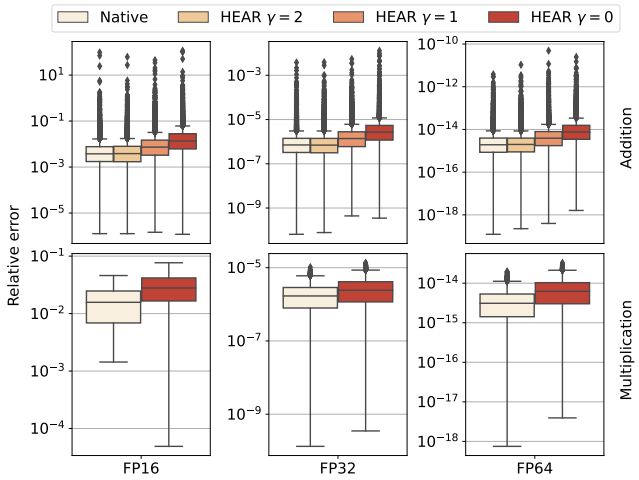

Figure 3: The precision loss against the different floating point types for multiplication and addition operations. γ signifies the number of bits by which the ciphertext has expanded to reduce precision loss.

$\otimes$ as in Section 5.3.1 and equation 5 as:

$$c_i[j] = \begin{cases} x_i[j] \otimes \frac{F_{k_e}(k_i^s + k_c + j)}{F_{k_e}(k_{i+1}^s + k_c + j)} & i = P - 1 \\ x_i[j] \otimes F_{k_e}(k_i^s + k_c + j) & \text{otherwise} \end{cases} \tag{6}$$

Decryption: we decrypt as $\mathrm{D}_{\mathbb{S}}(c[j]) = c[j] \otimes F_{k_e}^{-1}(k_0^s + k_c + j)$.

Loss of information: multiplication and division in the scheme introduce precision loss. To evaluate how much information is lost when passing through encryption and decryption, we used a multiple precision floating point number framework MPFR [36] based on GMP [46]. We ran 10,000 iterations, summing 10,000 randomly selected floats, resulting in an exponential sampling of values. Such sums are representative of typical numbers observed in the working conditions of HEAR. We measured the loss by comparing the absolute difference between the value after passing through HEAR and the native float value relative to the sum obtained using 1024 bits of precision. Figure 3 presents the relative precision loss against the floating point type used for multiplication. The precision differs from the native version by about one-tenth of an order of magnitude. We did not expand the ciphertext for multiplication, implying $\gamma = 0$. In the case of multiplication $\delta = 0$, hence, we operate on the same precision as normal floats.

Security: as shown in section 5.3.1, the scheme is COA secure. The multiplication scheme is robust against both adversary models and provides all three types of safety.

5.3.3 Addition. We base addition on the expanded exponent logic described in Section 5.3.5.

Encryption: we encode values as in equation 4. Then $x_i[j]$ can be encrypted to a ciphertext $c_i[j]$ assuming the same definition of $\otimes$ as in Section 5.3.1 and equation 5 as:

$$c_i[j] = \mathrm{E}(x_i[j]) = x \otimes F_{k_e}(k_c + j) \tag{7}$$

Decryption: we decrypt as: $\mathrm{D}_{\mathbb{S}}(c[j]) = c[j] \otimes F_{k_e}^{-1}(k_c + j)$.

Loss of information: analogously to the multiplication operation, division, and multiplication introduce information loss. Figure 3 also presents the relative precision loss against the floating point type used for addition. The results are obtained similarly to the multiplication case with larger sums of 100,000 elements. We present three cases for ciphertext inflation. The best for precision is $\gamma = 2$. However, it yields two additional bits in the representation, losing some advantages of INC. Similarly to multiplication, the precision loss is only a small part of the order of magnitude. In the worst case ($\gamma = 0$), the precision drops slightly less than an order of magnitude.

Security: as shown in section 5.3.1, the scheme is COA secure. The addition scheme is robust against the single-process adversary model where the adversary can only infer a single message, and global safety is unnecessary. This is crucial as the addition only provides temporal and local safety.

5.3.4 Alternative addition. We provide an alternative addition scheme that sacrifices performance and precision for more robust security for applications that require global safety. We base it on the multiplication of exponents. We first encode the values as exponents of an exponential function and then reduce them multiplicatively, which implies a sum of their exponents. We then decode the values after the decryption.

Encryption: we encode the values as $x_i[j] \rightarrow e^{x_i[j]} = a_i(x_i[j])$. We then use the multiplicative float scheme to send the values a_i.

Decryption: after obtaining the product $A = e^{\sum_{i \in \mathbb{S}} a_i}$ we decode by taking the logarithm as $\log(A) = \sum_{i \in \mathbb{S}} x[j]$.

Loss of information: Exponentiation leads to decreased dynamic range and loss of precision. This is especially true for values far from the origin. The method is useful when the values are known to be in a small range, e.g., in machine learning weights [90] which are normalized [82].

Security: the scheme provides the same security guarantees as multiplication.

5.3.5 Exponent logic. We introduce a new exponent arithmetic, which is conducted modulo $l_e + \delta$ where l_e is the length of the exponent of the original floating point, δ is zero for multiplication and two for addition. The modifications are necessary due to security reasons. We first describe the logic for conceptually simpler multiplication and move to addition afterward. We use the exponent with $l_e = 4$ (from 0 to 15) as an example to display the concepts.

Floating point multiplication is normally conducted by adding two exponents, multiplying the mantissas, and normalizing the result. Multiplying two numbers as described in Section 5.3.1 could lead to an exponent overflow. The typical IEEE standard would cap those values at the maximum infinity exponent. However, we cannot have such a cap if we consider encryption. For example, if the adversary knows that infinity has exponent $2^{l_e} - 1 = 15$, they also know that the maximum possible exponent is encrypted as 14, allowing for arbitrary decryption by mapping all other exponents and breaking any security guarantees. Thus, we allow the exponent to operate like a ring in our multiplicative case. Multiplications and divisions can then be reverted, and message exponents get encoded randomly along the ring, yielding the same security guarantees for the exponents as in the integer case.

The addition is more complex. Typical floating point addition works by comparing the exponents of two numbers, shifting the mantissa of the one with the smaller exponent by the difference

between the exponents, and adding the mantissas. If we assume two floats f_1 and f_2, and two random numbers r_1 and r_2, once multiplied together $f_1r_1 + f_2r_2$, the sum of $f_1 + f_2$ cannot be extracted because of different scales. Thus, all the numbers within one summation chain need to be scaled with the same random number to preserve the order of the numbers. Furthermore, we lose the order if we consider multiplication between f_1r_1 that overflows the exponent. For example, a number with exponent $2^{l_e} - 1 = 15$ can be encrypted to an exponent equal to 0, making it the smallest number. This is crucial for addition as comparing two exponents decides which number gets shifted and by how much. We propose a solution to this problem by expanding the exponent by one bit and using this structure in a novel way. The comparison operator now changes to two comparisons. For two exponents, e_1 and e_2, we calculate $e_1 - e_2$ and $e_2 - e_1$. We choose the smaller of the two as the difference and treat the number from which the other was subtracted as the larger one. Notably, only the smaller difference will correspond to a valid exponent. For example, for the aforementioned $l_e = 4$, we would work in arithmetic modulo $2^{l_e+1} = 32$. Then, if we assume $e_1 = 2$ and $e_2 = 21$, $e_1 - e_2 = 13$ and $e_2 - e_1 = 19$. Note that 19 is not a valid exponent, so $e_1 > e_2$ for the addition operation. Finally, when encrypting as described in Section 5.3.1, the maximum result of mantissa multiplication in the case of two mantissas having just binary ones is above 2, which requires shifting the mantissa to the right and adding one to the exponent. However, this additional one can again break the order. For example, for $e_1 = 31$ and $e_2 = 16$, if encryption increases e_1 by one due to mantissa overflow, $e_1 = 0$ gives indistinguishable exponents as both differences are equal to 16. Thus, we must increase the exponent by one more bit. The resulting ciphertext exponent has two more bits than the plaintext.

Increasing the exponent range without allowing overflow is invalid, making the scheme vulnerable to rainbow table type attacks [61]. For example, suppose we do not allow overflows, set $e_1 = 15$, and the exponent of random noise $e_r = 16$. Then $e_1 + e_r = 31$. In that case, the adversary knows this number could be obtained only with one combination of maximum plaintext exponent and maximum random noise exponent.

5.3.6 Limitations. As Section 5.3.5 points, HEAR cannot directly support any caps such as Infinity, not a number (NaN), zero, or any other special number due to security constraints. Zero gets first encoded as the smallest possible number and then treated as such. The support for NaN or Infinity could be added with a special bit indicating such a status. However, the bits of the mantissa, exponent, and sign must still be scrambled. An alternative approach to detect zero or infinity is using additional exponent bits, which would signal an under- or overflow if still set after decryption. This is already provided by addition. Furthermore, as HEAR is incompatible with IEEE 754, floating point operations in HEAR require changes in the FPU. The new FPU can be emulated in software if the INC hardware allows for this. Otherwise, HEAR requires hardware changes enabling features such as working on nonstandard exponents described in Section 5.3.5. Alternative schemes have been developed to replace IEEE 754 [49], for which HEAR could also be suitable. Finally, HEAR operates on a tradeoff of precision and ciphertext inflation (quantified by γ). The lost precision can be compensated for by additional bits, which increases bandwidth usage.

5.4 Other operations

In HEAR, the orders of magnitude faster operation than in other schemes comes from the invertible nature of encryption. All of our schemes rely on the existence of an easily computable inverse operation. However, some logical operations such as `AND` and `OR` have no inverse and cannot be easily ported to our framework. These could be implemented as summations where if the sum is zero, `OR` and `AND` are zero. If it's `P`, `OR`, and `AND` are one. Otherwise, `OR` is one, and `AND` is zero. Yet, this considerably increases bandwidth usage as our ciphertext grows as $O(\log_2 P)$.

Furthermore, some operations such as `min` and `max` are not allowed due to security constraints. If we enable the network to compare two values and determine which is larger, the adversary can encrypt an increasing set of values and determine the plaintext. Thus, all these operations must either use FHE schemes or be performed within the TEEs.

Similarly, arbitrary user-defined functions are not allowed due to the same security concerns. The users can specify functions based on the implemented schemes as long as these use only one operation type or are preprocessed in a secure environment. For example, to implement a variance calculation of a random variable X with mean zero, the nodes can compute X_i^2 on their samples and sum the results using HEAR. Furthermore, one can freely implement INC functions that combine modes of supported operations, e.g., adding data from even ranks and subtracting data from odd ranks.

5.5 Result verification

While we covered the confidentiality of data, we have not discussed verification. If the secure environment within which we run the application is based on TEEs, some of them might require integrity protection [25]. HE is malleable by design [14] and provides no integrity checks. This issue can be resolved by adding a homomorphic message authentication code (HoMAC) [16] derived from the encrypted values. While conceptually similar to normal MACs, HoMACs enable conducting homomorphic operations on the data and produce tags that authenticate the computation [16]. More precisely, each rank generates a tag $\sigma_i[j]$ for each ciphertext in the Allreduce vector $c_i[j]$, and sends to the network a pair of values $(\sigma, c_i[j])$. σ is computed as:

$$\sigma = \text{HoMAC}_{s_i[j]}(c_i[j]) = \frac{s_i[j] - c_i}{Z} \bmod p$$

where $c_i[j]$ is the ciphertext, $s_i[j]$ is per ciphertext homomorphic key generated randomly, Z the overall verification key and p a prime number of length λ bits. λ represents a security parameter. INC conducts a sum over all of the pairs $\sum_{i \in \mathbb{S}(c_i[j], \sigma_i[j])} = (c_t[j], \sigma_t[j])$. After the reduction, the ranks verify that $\sum_{i \in \mathbb{S}} s_i[j]$ is equal to $c_t[j] + \sigma_t Z \bmod p$. The operations can be improved by using a canceling method similar to the one in Sections 5.1-5.3 [14]:

$$\sigma = \text{HoMAC}_{s_i[j]}(c_i[j]) = \frac{s_i[j] - s_{i+1}[j] - c_i}{Z} \bmod p$$

where we ignored the edge conditions for rank 0 analogous to the encryption case. While HoMACs solve the problem of result verification, they are not free. The overhead is linear with the security parameter and might cause more than 200% inflation for reasonable 64-bit p. Additionally, the version of the scheme for

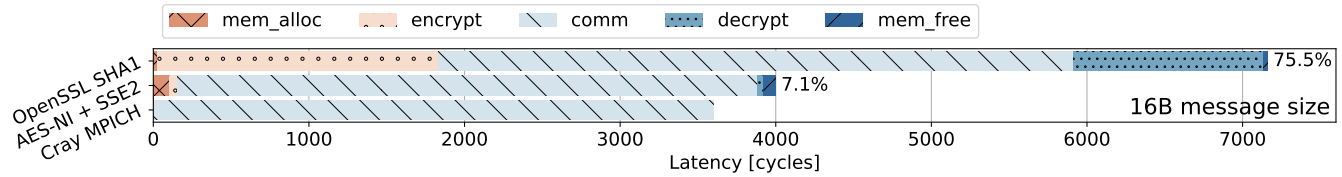

Figure 4: The latency breakdown of the critical path for a 16 Bytes MPI_Allreduce integer summation call on two ranks for two methods and their overheads as a percentage of the communication time measured with 100,000 iterations.

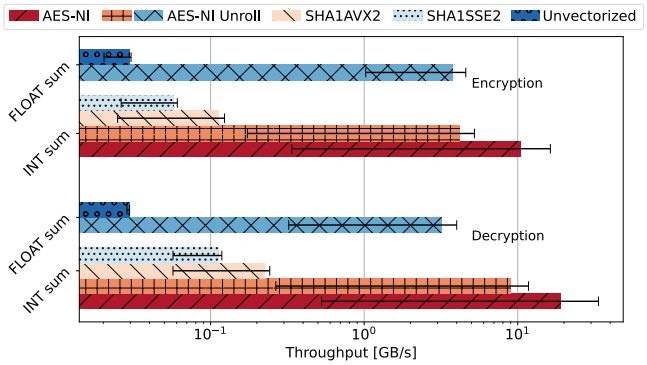

Figure 5: The encryption throughput for different PRNG methods as measured with 100 iterations on a single node. The standard deviation displays the performance across multiple buffer sizes.

multiplication [16] is scaling as 2^N where N is the number of operations. Assuming $\log N$ INC operations, the space overhead grows as $\theta(N)$ and is not scalable. To the best of our knowledge, HoMACs for the XOR operation and floats do not exist.

6 IMPLEMENTATION

We describe `libhear`, the end-to-end implementation of HEAR. `libhear` is a middleware C++ library that adds encryption and decryption functionality to `MPI_Allreduce` using `PMPI` and resides between the application and the MPI runtime. As `libhear` is based on the `PMPI` interface, it is MPI implementation-independent. Additionally, `libhear` does not change the code base or require application recompilation. To enable `MPI_Allreduce` encryption, the user only needs to conduct an `LD_PRELOAD` of `libhear` while running the MPI job. We discuss how `libhear` implements the steps of HEAR and how we optimized the library for performance, including implementing vectorization, AES-NI instructions, a memory pool, and network pipelining. Finally, we describe the experimental setup for all the results in Section 7.

Initialization: `libhear` performs the required key generation, exchange, and per rank state initialization of HEAR during communicator creation (e.g., `MPI_Init`, `MPI_Comm_create`. After initialization, `libhear` intercepts the `MPI_Allreduce` calls and performs encryption and decryption for specific data and operation types, including `MPI_INT` with `MPI_SUM`, and `MPI_FLOAT` with `MPI_SUM`.

Cryptographic operations: `libhear` provides a highly optimized version of encryption and decryption for the x86 ISA. We refined our implementation to enable two scenarios with distinct performance requirements targeted by INC:

(1) High-throughput for large message sizes (e.g., 16 MiB) and 100-800s Gbit/s link bandwidths.
(2) μs-scale latency of inter-process synchronization focusing on small messages such as 16 B.

As critical path profiling of naïve HEAR implementation suggests in Figure 4, the PRF calls are the largest contributor to the time of encryption. We focused on choosing the best PRF for our needs. We investigated multiple possibilities and focused on SHA1 [32] and AES [31] as primary candidates who achieve good performance [93].

High-throughput scenario: Figure 5 shows encryption and decryption throughput of HEAR on a single Intel Xeon E5-2695 v4 @ 2.10GHz core for both integer and float summation. The vectorized integer implementation using OpenSSL [94] SHA1 achieves less than 1GB/s/core for encryption and decryption predominantly because of the lack of hardware acceleration. We optimized performance by utilizing 128-bit AES-NI x86 ISA extensions [4]. We combined AES-NI with loop unrolling to enable automatic SSE2 128-bit vectorization by the compiler. Such a version achieves 5GB/s/core for encryption and 11.7GB/s/core for decryption. To further improve performance, we hand-tuned SSE2 vectorization, which increases encryption and decryption throughput up to 9GB/s/core and 18GB/s/core. In `libhear`, SHA1 is unsuitable for modern (100-200 Gbit/s) and emerging (400-800 Gbit/s) HPC network line rates, while the AES-NI can saturate them with a modest number of one to five processes per node (PPN).

Using the insights from integer experiments, we avoided SHA and focused on AES for floats. We developed two variations for float summation: non-vectorized and loop unrolled versions. Even without hand-tuning, the automatically vectorized variant is an order of magnitude faster than the Aries NIC bandwidth of 0.347GB/s/core.

Low-latency scenario: Figure 4 displays the latency breakdown for the critical path of integer summation collected with x86 RDTSC counters. Similarly to the throughput case, SHA1 is worse and introduces larger latencies than AES-NI. The hardware-enhanced AES-NI also reduces CPU cycle overhead for small messages to 7.1% compared to SHA1-based `libhear` implementation achieving 75.5%. We selected AES-NI-based implementation as the PRF backend of `libhear` and conducted all further experiments on such a version.

Communication: To further optimize `libhear` for data-heavy applications such as gradient summing in distributed ML [8], we exploit non-blocking MPI collectives and implement network pipelining. We use non-blocking `MPI_Iallreduce` to overlap the processing of receive and send buffers with communication. Specifically, we overlap the decryption of $n-1$th send-buffer block and encryption of $n+1$th block with the encrypted nth block reduction done in the network or the MPI implementation.

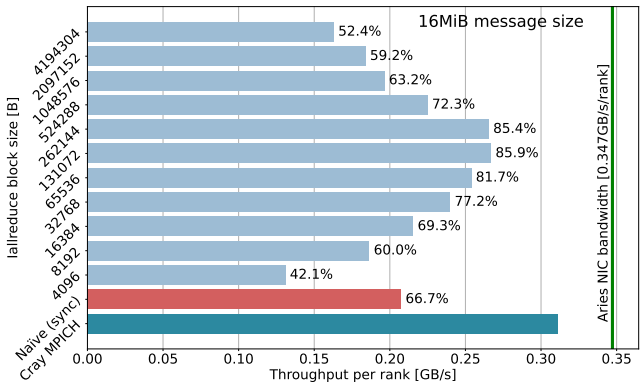

Figure 6: The 16MiB encryption/decryption throughput for different pipelining blocks as measured with 1000 iterations. We show for reference the non-pipelined, synchronous Naïve version and the baseline Cray MPICH. We display the fraction of performance as compared to Cray MPICH.

Memory allocation: During `libhear` initialization, we pre-allocate a page-aligned memory pool to handle intermediate send buffer blocks. The memory pool helps to avoid dynamic memory allocation with `malloc` and alleviates the cost of memory registration (pinning) done by the underlying MPI implementation on the Remote Direct Memory Access (RDMA) data path [65]. Figure 6 displays the throughput per rank achieved with different block sizes. With an optimal block size of 131-262KiB, we achieved nearly 0.27GB/s/rank, which is 14% less than the native Cray MPICH at full CPU utilization (i.e., 36 PPN).

Results validation: We validated the correctness of `MPI_FLOAT` scheme in `libhear` by running 10 million iterations of number encryption-decryption. Observed relative error of numbers after decryption varied from the original number by average 1.3×10^{-7}. `MPI_INT` summation correctness was verified by comparing the contents of `MPI_Allreduce` receive buffers of `libhear` and reference MPI implementation using `std::memcmp` function.

7 EVALUATION

We evaluate the overhead of `libhear` for communication on a supercomputing system Piz Daint [75]. We deploy our workloads on compute nodes with two 18-core Intel Xeon E5-2695 v4 @ 2.10GHz CPUs based on the Broadwell microarchitecture coupled with 128 GB of DDR3 memory. Piz Daint is based on the 100Gbit/s Aries interconnect. We compile our code with `clang (v.13.0.1)` and `-O3 -ffast-math -march=native` flags and use Cray MPICH (`PrgEnv-cray/6.0.10`) as a reference MPI implementation. We evaluate the latency and throughput scaling of `libhear` using OSU micro-benchmarks (v7.1) [71]. We then analyze its influence on the performance of a set of deep neural network (DNN) proxy applications relying heavily on Allreduce [52]. Similarly to Section 6, we focus on two crucial aspects of INC, throughput improvements for large messages and latency improvements for small messages.

7.1 Scaling benchmarks

Throughput: For large messages, we evaluate `libhear` throughput scaling with respect to the number of MPI ranks. Figure 7 shows

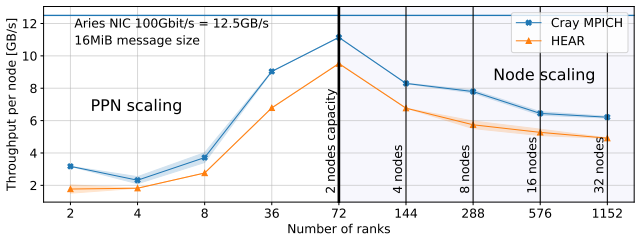

Figure 7: The scaling of the MPI_Allreduce throughput as the number of ranks increases with two sections: PPN scaling on two nodes and node scaling. The boundaries display min/max range while the lines are the means. HEAR scales consistently and achieves about 80% of Cray MPICH's throughput.

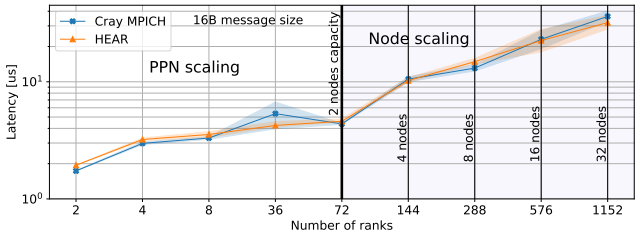

Figure 8: The scaling of the MPI_Allreduce latency as the number of ranks. We show two scaling sections: PPN on two nodes and > 2 nodes. The boundaries display min/max while the lines are the means. HEAR scales well with low overhead.

the throughput of `libhear` alongside the native Cray MPICH for integer summations. We run two trials with 1000 iterations each. We display two sections: PPN scaling and node scaling. For the PPN scaling, we increased the number of ranks on a two-node allocation without increasing the number of nodes. For the node scaling, we doubled the number of nodes at each iteration. At its peak, Cray MPICH reaches 11.1GB/s, steadily reducing the performance afterward due to communication overhead between the nodes. At its peak, `libhear` reaches a throughput of 9.5GB/s, achieving 85% of Cray's MPICH bandwidth. `libhear` scales in the same way as Cray MPICH, consistently achieving around 80% of the performance provided by the native implementation.

Latency: We evaluate `libhear` 16B latency scaling with respect to the number of ranks. Figure 8 shows the latency of `libhear` alongside the native Cray MPICH for integer summations. We run two trials with 100,000 iterations each. Similarly to the throughput experiment, we display two sections: PPN scaling and node scaling. For the PPN scaling, we increased the number of ranks on a two-node system without increasing the number of nodes. For the node scaling, we doubled the number of nodes at each iteration. As expected, the two sections display different scaling. `libhear` is again scaling like the native Cray MPICH and does not introduce significant latency overhead. As the number of ranks increases, the noise within the network grows considerably [26], as visible by the minimum and maximum range. This leads to `libhear` achieving even lower latency than Cray MPICH, indicating minimal overhead `libhear` introduces. The overhead is small enough to hide within the network noise for a larger number of ranks.

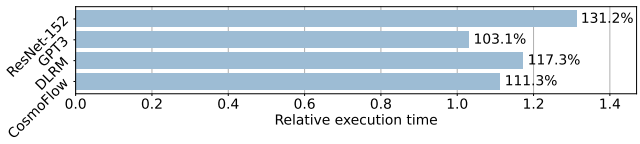

Figure 9: Simulated relative average execution time of one iteration of distributed DNN training while running with HEAR as normalized to without it. GPT3 training parallelized across 384 ranks (48 nodes, 8 PPN); ResNet-152, DLRM, and CosmoFlow scaled to 256 MPI ranks (8 nodes, 32 PPN).

7.2 Performance of DNN training

We also assess overheads from `libhear` using several distributed deep neural network (DNN) training proxy-workloads [52]. We select this workload as a significant part of the distributed Stochastic Gradient Descent (SGD) neural network training algorithm consists of a gradient averaging using Allreduce [8]. As such, we consider DNN training as a worst-case scenario, giving a realistic upper bound of the `libhear` impact on the overall application throughput. In our setup, the gradient averaging part of distributed SGD is modeled using `MPI_Allreduce` with `MPI_FLOAT` (FP32) data type with message size proportional to the number of model parameters.

In Figure 9, we report simulated performance overheads of one training iteration for ResNet-152, GPT3, DLRM, and CosmoFlow models. We observe the highest overheads (1.31×) with ResNet-152 training, whose communication part consists of only Allreduce calls. For other models (i.e., DLRM, CosmoFlow, and GPT3) where the communication also includes the synchronization with other MPI collectives (e.g., `MPI_Alltoall`) and point-to-point operations, `libhear` introduces minor overhead that varies from 3% to 17.5%. We note that these overheads could be eliminated by further overlapping computation (i.e., training on GPUs) with non-blocking HEAR communication that could be implemented in hardware.

8 HEAR EXTENSIONS

HEAR is implemented in software and uses hardware acceleration for the PRF evaluation to enhance its encryption and decryption routines. However, some methods, such as converting a standard float to HEARs representation or addition and multiplication on the new type, create overhead that could be lowered by implementing the operations in hardware.

Our open-source HEAR implementation, `libhear`, allows users to add new data types and operations transparently and with low overhead. It provides the PRF implementation in a vectorized, hardware-accelerated way, allowing for further high-performance expansion. While Allreduce and Reduce are the most commonly used collectives, HEAR can be extended to other collectives such as broadcast, all-to-all, scatter, and gather communication. These would work similarly to Allreduce, however, without any INC. One-to-one communication can also be implemented using a matrix of keys rather than a constant number of keys. However, this scales as $\theta(N)$ in space, worse than $\theta(1)$ of the other HEAR methods. In one-to-one communication, HEAR could be used for atomic operations. Such broad support might be significant if HEAR is implemented in hardware. It can then avoid having multiple encryption standards over the same communication layer.

9 CHALLENGES AND OPPORTUNITIES

HEAR shows that cHPC can be achieved while maintaining crucial performance. However, HEAR only opens the discussion about cHPC in the community. cHPC poses fundamental challenges where the additional security introduces latency, energy, and silicon costs taking resources from other performance features. Even relatively simple algorithms such as AES can have overcomplicated implementations that reach 4.2 cycles/byte/thread [4] of encryption and decryption while using 275pJ/bit of energy [1]. While this can be acceptable for long-distance networks, performance on integrated and short-distance links such as CXL [87] might suffer. Furthermore, the current TEE ecosystem does not support cHPC. Challenges such as lack of Non-Uniform Memory Access (NUMA) [6] and RDMA [93] support, hardening and optimizing HPC libraries like Open MPI [37], creating scalable distributed attestation [22], and enhancing resource orchestrators such as Slurm [99] remain open. Despite all challenges, cHPC can enable performance-hungry personalized medicine (e.g., AI-based gene analysis, private chatbot doctors) and finance (e.g., preventing fraud, running hedge-fund proprietary algorithms) [67, 66]. cHPC offers strong security guarantees, where no confidential data is shared with the computing agent and where computation results can be entrusted. As HEAR demonstrates, by leveraging fundamental and streamlined security technologies such as novel homomorphic computing schemes, we can achieve these security gains while preserving performance.

10 CONCLUSIONS

We presented HEAR, a first-of-a-kind framework supporting homomorphically encrypted Allreduce operation enabling confidential INC. Unlike many HE schemes, HEAR is geared toward the specific security needs of HPC. To quantify those, we presented a threat model corresponding to the common HPC adversaries, which concerned itself with local, global, and temporal safety. We also outlined HEAR and its design. HEAR operates on both integers and floats with variable security. It requires no or little changes in the hardware it is running on. HEAR is either lossless or causes only a small amount of precision loss. HEAR does not require ciphertext inflation, which can be introduced to recover precision. We also discussed how result verification can be supported within HEAR.

We prototyped HEAR in the form of `libhear`, allowing users to run it on top of their applications without recompiling. We have optimized `libhear`'s performance for two common INC cases: small latency and large message throughput. We showed that `libhear`'s overhead is small in both synthetic and application benchmarks for the provided security and performance benefits from enabling INC. Finally, we open-sourced `libhear`. We believe HEAR represents the first milestone towards cHPC.

ACKNOWLEDGMENTS

This project received funding from EuroHPC-JU under grant agreement RED-SEA, No 055776, the UrbanTwin project, and a donation from Intel. We also thank CSCS for providing computational resources for this project. Furthermore, we thank Surendra Anubolu and Mohan Kalkunte (both Broadcom), Michael Steiner (Intel), Madlen Koblinger, and Bogdan Ursu for constructive discussions about this project and assistance in preparing this article.

REFERENCES

[1] Eslam G. AbdAllah, Yu Rang Kuang, and Changcheng Huang. 2020. Advanced Encryption Standard New Instructions (AES-NI) Analysis: Security, Performance, and Power Consumption. In *Proceedings of the 2020 12th International Conference on Computer and Automation Engineering* (ICCAE 2020). Association for Computing Machinery, New York, NY, USA, (May 16, 2020), 167–172. ISBN: 978-1-4503-7678-5. DOI: 10.1145/3384613.3384648.

[2] Dennis Abts, Michael R. Marty, Philip M. Wells, Peter Klausler, and Hong Liu. 2010. Energy proportional datacenter networks. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (ISCA '10). Association for Computing Machinery, New York, NY, USA, (June 19, 2010), 338–347. ISBN: 978-1-4503-0053-7. DOI: 10.1145/1815961.1816004.

[3] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. 2018. A Survey on Homomorphic Encryption Schemes: Theory and Implementation. *ACM Computing Surveys*, 51, 4, (July 25, 2018), 79:1–79:35. DOI: 10.1145/3214303.

[4] Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, and Ronen Zohar. [n. d.] Breakthrough AES Performance with Intel® AES New Instructions.

[5] Ayaz Akram, Anna Giannakou, Venkatesh Akella, Jason Lowe-Power, and Sean Peisert. 2021. Performance Analysis of Scientific Computing Workloads on General Purpose TEEs. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). (May 2021), 1066–1076. DOI: 10.1109/IPDPS49936.2021.00115.

[6] Ayaz Akram, Anna Giannakou, Venkatesh Akella, Jason Lowe-Power, and Sean Peisert. 2021. Performance Analysis of Scientific Computing Workloads on General Purpose TEEs. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, Portland, OR, USA, (May 2021), 1066–1076. ISBN: 978-1-66544-066-0. DOI: 10.1109/IPDPS49936.2021.00115.

[7] Louis J. M. Aslett, Pedro M. Esperança, and Chris C. Holmes. 2015. A review of homomorphic encryption and software tools for encrypted statistical machine learning. *CoRR*, abs/1508.06574. Retrieved Mar. 27, 2023 from http://arxiv.org/abs/1508.06574 arXiv: 1508.06574.

[8] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis. *ACM Computing Surveys*, 52, 4, (Aug. 30, 2019), 65:1–65:43. DOI: 10.1145/3320060.

[9] Josh Benaloh. 1994. Dense probabilistic encryption. In *Proceedings of the Workshop on Selected Areas of Cryptography*, 120–128.

[10] David E. Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E. Grant, Thomas Naughton, Howard P. Pritchard, Martin Schulz, and Geoffroy R. Vallee. 2020. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience*, 32, 3, e4851. DOI: 10.1002/cpe.4851.

[11] Alex Biryukov and Eyal Kushilevitz. 1998. From differential cryptanalysis to ciphertext-only attacks. In *Advances in Cryptology — CRYPTO '98* (Lecture Notes in Computer Science). Hugo Krawczyk, (Ed.) Springer, Berlin, Heidelberg, 72–88. ISBN: 978-3-540-68462-6. DOI: 10.1007/BFb0055721.

[12] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. 2005. Evaluating 2-DNF Formulas on Ciphertexts. In *Theory of Cryptography* (Lecture Notes in Computer Science). Joe Kilian, (Ed.) Springer, Berlin, Heidelberg, 325–341. ISBN: 978-3-540-30576-7. DOI: 10.1007/978-3-540-30576-7_18.

[13] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference* (ITCS '12). Association for Computing Machinery, New York, NY, USA, (Jan. 8, 2012), 309–325. ISBN: 978-1-4503-1115-1. DOI: 10.1145/2090236.2090262.

[14] Lukas Burkhalter, Anwar Hithnawi, Alexander Viand, Hossein Shafagh, and Sylvia Ratnasamy. 2020. {TimeCrypt}: Encrypted Data Stream Processing at Scale with Cryptographic Access Control. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), 835–850. ISBN: 978-1-939133-13-7. Retrieved Mar. 21, 2023 from https://www.usenix.org/conference/nsdi20/presentation/burkhalter.

[15] Claude Castelluccia, Aldar C-F. Chan, Einar Mykletun, and Gene Tsudik. 2009. Efficient and provably secure aggregation of encrypted data in wireless sensor networks. *ACM Transactions on Sensor Networks*, 5, 3, (June 4, 2009), 20:1–20:36. DOI: 10.1145/1525856.1525858.

[16] Dario Catalano and Dario Fiore. 2013. Practical Homomorphic MACs for Arithmetic Circuits. In *Advances in Cryptology – EUROCRYPT 2013* (Lecture Notes in Computer Science). Thomas Johansson and Phong Q. Nguyen, (Eds.) Springer, Berlin, Heidelberg, 336–352. ISBN: 978-3-642-38348-9. DOI: 10.1007/978-3-642-38348-9_21.

[17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017* (Lecture Notes in Computer Science). Tsuyoshi Takagi and Thomas Peyrin, (Eds.) Springer International Publishing, Cham, 409–437. ISBN: 978-3-319-70694-8. DOI: 10.1007/978-3-319-70694-8_15.

[18] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In *Advances in Cryptology – ASIACRYPT 2016* (Lecture Notes in Computer Science). Jung Hee Cheon and Tsuyoshi Takagi, (Eds.) Springer, Berlin, Heidelberg, 3–33. ISBN: 978-3-662-53887-6. DOI: 10.1007/978-3-662-53887-6_1.

[19] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *Journal of Cryptology*, 33, 1, (Jan. 1, 2020), 34–91. DOI: 10.1007/s00145-019-09319-x.

[20] M. Cho, U. Finkler, M. Serrano, D. Kung, and H. Hunter. 2019. BlueConnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. *IBM Journal of Research and Development*, 63, 6, (Nov. 2019), 1:1–1:11. DOI: 10.1147/JRD.2019.2947013.

[21] Sudheer Chunduri, Scott Parker, Pavan Balaji, Kevin Harms, and Kalyan Kumaran. 2018. Characterization of MPI Usage on a Production Supercomputer. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. (Nov. 2018), 386–400. DOI: 10.1109/SC.2018.00033.

[22] George Coker et al. 2011. Principles of remote attestation. *International Journal of Information Security*, 10, 2, (June 1, 2011), 63–81. DOI: 10.1007/s10207-011-0124-7.

[23] Eduardo Lopes Cominetti and Marcos A. Simplicio. 2020. Fast Additive Partially Homomorphic Encryption From the Approximate Common Divisor Problem. *IEEE Transactions on Information Forensics and Security*, 15, 2988–2998. DOI: 10.1109/TIFS.2020.2981239.

[24] Anamaria Costache, Nigel P. Smart, Srinivas Vivek, and Adrian Waller. 2017. Fixed-Point Arithmetic in SHE Schemes. In *Selected Areas in Cryptography – SAC 2016* (Lecture Notes in Computer Science). Roberto Avanzi and Howard Heys, (Eds.) Springer International Publishing, Cham, 401–422. ISBN: 978-3-319-69453-5. DOI: 10.1007/978-3-319-69453-5_22.

[25] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 86. http://eprint.iacr.org/2016/086.

[26] Daniele De Sensi, Tiziano De Matteis, Konstantin Taranov, Salvatore Di Girolamo, Tobias Rahn, and Torsten Hoefler. 2022. Noise in the clouds: Influence of network performance variability on application scalability. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6, 3, 1–27.

[27] Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefler. 2021. Flare: flexible in-network allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC '21). Association for Computing Machinery, New York, NY, USA, (Nov. 13, 2021), 1–16. ISBN: 978-1-4503-8442-1. DOI: 10.1145/3458817.3476178.

[28] Carl P. Dettmann and Orestis Georgiou. 2009. Product of n independent uniform random variables. *Statistics & Probability Letters*, 79, 24, (Dec. 15, 2009), 2501–2503. DOI: 10.1016/j.spl.2009.09.004.

[29] P. Drozdowski, N. Buchmann, C. Rathgeb, M. Margraf, and C. Busch. 2019. On the Application of Homomorphic Encryption to Face Identification. In *2019 International Conference of the Biometrics Special Interest Group (BIOSIG)*. 2019 International Conference of the Biometrics Special Interest Group (BIOSIG). (Sept. 2019), 1–5.

[30] Morris Dworkin. 2001. Recommendation for Block Cipher Modes of Operation: Methods and Techniques. NIST Special Publication (SP) 800-38A. National Institute of Standards and Technology, (Dec. 1, 2001). DOI: 10.6028/NIST.SP.800-38A.

[31] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. 2001. Advanced encryption standard (AES). Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, (2001). DOI: 10.6028/NIST.FIPS.197.

[32] Donald E. Eastlake 3rd and Paul Jones. 2001. US Secure Hash Algorithm 1 (SHA1). Request for Comments RFC 3174. Internet Engineering Task Force, (Sept. 2001). 22 pp. DOI: 10.17487/RFC3174.

[33] T. Elgamal. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31, 4, (July 1985), 469–472. DOI: 10.1109/TIT.1985.1057074.

[34] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 144. http://eprint.iacr.org/2012/144.

[35] Seyed Hamed Fatemi Langroudi, Tej Pandit, and Dhireesha Kudithipudi. 2018. Deep Learning Inference on Embedded Devices: Fixed-Point vs Posit. In *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*. 2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2). (Mar. 2018), 19–23. DOI: 10.1109/EMC2.2018.00012.

[36] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A multiple-precision binary floating-point

library with correct rounding. *ACM Transactions on Mathematical Software*, 33, 2, (June 1, 2007), 13–es. DOI: 10.1145/1236463.1236468.

[37] Edgar Gabriel et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, (Sept. 2004), 97–104.

[38] J.-L. Gauvain and Chin-Hui Lee. 1994. Maximum a posteriori estimation for multivariate Gaussian mixture observations of Markov chains. *IEEE Transactions on Speech and Audio Processing*, 2, 2, (Apr. 1994), 291–298. DOI: 10.1109/89.279278.

[39] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Better Bootstrapping in Fully Homomorphic Encryption. In *Public Key Cryptography – PKC 2012* (Lecture Notes in Computer Science). Marc Fischlin, Johannes Buchmann, and Mark Manulis, (Eds.) Springer, Berlin, Heidelberg, 1–16. ISBN: 978-3-642-30057-8. DOI: 10.1007/978-3-642-30057-8_1.

[40] Avi Goldfarb and Catherine Tucker. 2012. Shifts in Privacy Concerns. *American Economic Review*, 102, 3, (May 2012), 349–353. DOI: 10.1257/aer.102.3.349.

[41] Shafi Goldwasser and Silvio Micali. 1984. Probabilistic encryption. *Journal of Computer and System Sciences*, 28, 2, (Apr. 1, 1984), 270–299. DOI: 10.1016/0022-0000(84)90070-9.

[42] Shafi Goldwasser and Silvio Micali. 1982. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing* (STOC '82). Association for Computing Machinery, New York, NY, USA, (May 5, 1982), 365–377. ISBN: 978-0-89791-070-5. DOI: 10.1145/800070.802212.

[43] Daniel M. Gordon. 1998. A Survey of Fast Exponentiation Methods. *Journal of Algorithms*, 27, 1, (Apr. 1, 1998), 129–146. DOI: 10.1006/jagm.1997.0913.

[44] S. Gottlieb, W. Liu, D. Toussaint, R. L. Renken, and R. L. Sugar. 1987. Hybrid-molecular-dynamics algorithms for the numerical simulation of quantum chromodynamics. *Physical Review. D, Particles and Fields*, 35, 8, (Apr. 15, 1987), 2531–2542. pmid: 9957958. DOI: 10.1103/physrevd.35.2531.

[45] Richard L. Graham et al. 2016. Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for Efficient Data Reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*. 2016 First International Workshop on Communication Optimizations in HPC (COMHPC). (Nov. 2016), 1–10. DOI: 10.1109/COMHPC.2016.006.

[46] Torbjrn Granlund and Gmp Development Team. 2015. *GNU MP 6.0 Multiple Precision Arithmetic Library*. Samurai Media Limited, London, GBR, (Oct. 2015). 148 pp. ISBN: 978-988-8381-96-8.

[47] Yang Guo et al. 2023. High-Performance Computing (HPC) Security: Architecture, Threat Analysis, and Security Posture. NIST Special Publication (SP) 800-223 (Draft). National Institute of Standards and Technology, (Feb. 6, 2023). DOI: 10.6028/NIST.SP.800-223.ipd.

[48] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32nd International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, (June 1, 2015), 1737–1746. Retrieved Mar. 23, 2023 from https://proceedings.mlr.press/v37/gupta15.html.

[49] John L. Gustafson and Isaac T. Yonemoto. 2017. Beating Floating Point at its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations*, 4, 2, (Apr. 25, 2017), 71–86, 2, (Apr. 25, 2017). DOI: 10.14529/jsfi170206.

[50] Torsten Hoefler, Salvatore Di Girolamo, Konstantin Taranov, Ryan E. Grant, and Ron Brightwell. 2017. sPIN: High-performance streaming Processing In the Network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC '17). Association for Computing Machinery, New York, NY, USA, (Nov. 12, 2017), 1–16. ISBN: 978-1-4503-5114-0. DOI: 10.1145/3126908.3126970.

[51] Torsten Hoefler, Andrew Lumsdaine, and Jack Dongarra. 2009. Towards Efficient MapReduce Using MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (Lecture Notes in Computer Science). Matti Ropo, Jan Westerholm, and Jack Dongarra, (Eds.) Springer, Berlin, Heidelberg, 240–249. ISBN: 978-3-642-03770-2. DOI: 10.1007/978-3-642-03770-2_30.

[52] Torsten Hoefler et al. 2022. HammingMesh: a network topology for large-scale deep learning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (SC '22). IEEE Press, Dallas, Texas, (Nov. 18, 2022), 1–18.

[53] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized neural networks: training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18, 1, (Jan. 1, 2017), 6869–6898.

[54] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, (July 2019), 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[55] Yuval Ishai and Anat Paskin. 2007. Evaluating Branching Programs on Encrypted Data. In *Theory of Cryptography* (Lecture Notes in Computer Science). Salil P. Vadhan, (Ed.) Springer, Berlin, Heidelberg, 575–594. ISBN: 978-3-540-70936-7. DOI: 10.1007/978-3-540-70936-7_31.

[56] Sambhav R. Jain, Albert Gural, Michael Wu, and Chris Dick. 2020. Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, (Eds.) mlsys.org. https://proceedings.mlsys.org/book/295.pdf.

[57] David Kaplan. [n. d.] AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More.

[58] Jonathan Katz and Yehuda Lindell. 2014. *Introduction to Modern Cryptography, Second Edition*. (2nd ed.). Chapman & Hall/CRC, (Oct. 2014). 603 pp. ISBN: 978-1-4665-7026-9.

[59] N. G. Kingsbury and P. J. W. Rayner. 1971. Digital filtering using logarithmic arithmetic. *Electronics Letters*, 7, 2, (Jan. 28, 1971), 56–58. DOI: 10.1049/el:19710039.

[60] Benjamin Klenk, Nan Jiang, Greg Thorson, and Larry Dennison. 2020. An in-network architecture for accelerating shared-memory multiprocessor collectives. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (ISCA '20). IEEE Press, Virtual Event, (Sept. 23, 2020), 996–1009. ISBN: 978-1-72814-661-4. DOI: 10.1109/ISCA45697.2020.00085.

[61] Himanshu Kumar, Sudhanshu Kumar, Remya Joseph, Dhananjay Kumar, Sunil Kumar Shrinarayan Singh, Ajay Kumar, and Praveen Kumar. 2013. Rainbow table to crack password using MD5 hashing algorithm. In *2013 IEEE Conference on Information & Communication Technologies*. 2013 IEEE Conference on Information & Communication Technologies. (Apr. 2013), 433–439. DOI: 10.1109/CICT.2013.6558135.

[62] Ignacio Laguna, Ryan Marshall, Kathryn Mohror, Martin Ruefenacht, Anthony Skjellum, and Nawrin Sultana. 2019. A large-scale study of MPI usage in open-source HPC applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC '19). Association for Computing Machinery, New York, NY, USA, (Nov. 17, 2019), 1–14. ISBN: 978-1-4503-6229-0. DOI: 10.1145/3295500.3356176.

[63] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. 2016. Fixed Point Quantization of Deep Convolutional Networks. In *Proceedings of The 33rd International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, (June 11, 2016), 2849–2858. Retrieved Mar. 23, 2023 from https://proceedings.mlr.press/v48/linb16.html.

[64] Message Passing Interface Forum. 2021. *MPI: A Message-Passing Interface Standard Version 4.0*. Manual: (June 2021). https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[65] Frank Mietke, Robert Rex, Robert Baumgartl, Torsten Mehlan, Torsten Hoefler, and Wolfgang Rehm. 2006. Analysis of the memory registration process in the Mellanox InfiniBand software stack. In *Euro-Par 2006 Parallel Processing: 12th International Euro-Par Conference, Dresden, Germany, August 28–September 1, 2006. Proceedings 12*. Springer, 124–133.

[66] Fan Mo, Zahra Tarkhani, and Hamed Haddadi. 2022. Machine Learning with Confidential Computing: A Systematization of Knowledge. arXiv.org. (Aug. 22, 2022). Retrieved Aug. 28, 2023 from https://arxiv.org/abs/2208.10134v2.

[67] Dominic P. Mulligan, Gustavo Petri, Nick Spinale, Gareth Stockwell, and Hugo J. M. Vincent. 2021. Confidential Computing—a brave new world. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. 2021 International Symposium on Secure and Private Execution Environment Design (SEED). (Sept. 2021), 132–138. DOI: 10.1109/SEED51797.2021.00025.

[68] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. 2011. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop* (CCSW '11). Association for Computing Machinery, New York, NY, USA, (Oct. 21, 2011), 113–124. ISBN: 978-1-4503-1004-8. DOI: 10.1145/2046660.2046682.

[69] Markus Nagel, Rana Ali Amjad, Mart Van Baalen, Christos Louizos, and Tijmen Blankevoort. 2020. Up or Down? Adaptive Rounding for Post-Training Quantization. In *Proceedings of the 37th International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, (Nov. 21, 2020), 7197–7206. Retrieved Apr. 6, 2023 from https://proceedings.mlr.press/v119/nagel20a.html.

[70] Monique Ogburn, Claude Turner, and Pushkar Dahal. 2013. Homomorphic Encryption. *Procedia Computer Science*. Complex Adaptive Systems 20, (Jan. 1, 2013), 502–509. DOI: 10.1016/j.procs.2013.09.310.

[71] [n. d.] OSU micro-benchmarks 7.1. https://mvapich.cse.ohio-state.edu/benchmarks/.

[72] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Advances in Cryptology — EUROCRYPT '99* (Lecture Notes in Computer Science). Jacques Stern, (Ed.) Springer, Berlin, Heidelberg, 223–238. ISBN: 978-3-540-48910-8. DOI: 10.1007/3-540-48910-X_16.

[73] Maria Petrescu and Anjala S. Krishen. 2018. Analyzing the analytics: data privacy concerns. *Journal of Marketing Analytics*, 6, 2, (June 1, 2018), 41–43. DOI: 10.1057/s41270-018-0034-x.

[74] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Computing Surveys*, 51, 6, (Jan. 28, 2019), 130:1–130:36. DOI: 10.1145/3291047.

[75] [n. d.] Piz Daint. CSCS. Retrieved Apr. 4, 2023 from https://www.cscs.ch/computers/piz-daint/.

[76] Rolf Rabenseifner. 2004. Optimization of Collective Reduction Operations. In *Computational Science - ICCS 2004* (Lecture Notes in Computer Science). Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, (Eds.) Springer, Berlin, Heidelberg, 1–9. ISBN: 978-3-540-24685-5. DOI: 10.1007/978-3-540-24685-5_1.

[77] Fahmida Y. Rashid. 2020. The rise of confidential computing: Big tech companies are adopting a new security model to protect data while it's in use - [News]. *IEEE Spectrum*, 57, 6, (June 2020), 8–9. DOI: 10.1109/MSPEC.2020.9099920.

[78] R. L. Rivest, A. Shamir, and L. Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21, 2, (Feb. 1, 1978), 120–126. DOI: 10.1145/359340.359342.

[79] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. 2021. {ReDMArk}: Bypassing {RDMA} Security Mechanisms. In 30th USENIX Security Symposium (USENIX Security 21), 4277–4292. ISBN: 978-1-939133-24-3. Retrieved Aug. 27, 2023 from https://www.usenix.org/conference/usenixsecurity21/presentation/rothenberger.

[80] Xiaojun Ruan, Qing Yang, Mohammed I. Alghamdi, Shu Yin, and Xiao Qin. 2012. ES-MPICH2: A Message Passing Interface with Enhanced Security. *IEEE Transactions on Dependable and Secure Computing*, 9, 3, (May 2012), 361–374. DOI: 10.1109/TDSC.2012.9.

[81] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted Execution Environment: What It is, and What It is Not. In *2015 IEEE Trustcom/BigDataSE/ISPA*. 2015 IEEE Trustcom/BigDataSE/ISPA. Vol. 1. (Aug. 2015), 57–64. DOI: 10.1109/Trustcom.2015.357.

[82] Tim Salimans and Durk P Kingma. 2016. Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks. In *Advances in Neural Information Processing Systems*. Vol. 29. Curran Associates, Inc. Retrieved Apr. 6, 2023 from https://proceedings.neurips.cc/paper/2016/hash/ed265bc903a5a097f61d3ec064d96d2e-Abstract.html.

[83] T. Sander, A. Young, and Moti Yung. 1999. Non-interactive cryptocomputing for NC/sup 1/. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. 40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039). (Oct. 1999), 554–566. DOI: 10.1109/SFFCS.1999.814630.

[84] Amedeo Sapio et al. 2021. Scaling Distributed Machine Learning with {In-Network} Aggregation. In 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), 785–808. ISBN: 978-1-939133-21-2. Retrieved Mar. 27, 2023 from https://www.usenix.org/conference/nsdi21/presentation/sapio.

[85] Savvas Savvides, Darshika Khandelwal, and Patrick Eugster. 2020. Efficient confidentiality-preserving data analytics over symmetrically encrypted datasets. *Proceedings of the VLDB Endowment*, 13, 8, (Apr. 1, 2020), 1290–1303. DOI: 10.14778/3389133.3389144.

[86] Whit Schonbein, Ryan E. Grant, Matthew G. F. Dosanjh, and Dorian Arnold. 2019. INCA: in-network compute assistance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC '19). Association for Computing Machinery, New York, NY, USA, (Nov. 17, 2019), 1–13. ISBN: 978-1-4503-6229-0. DOI: 10.1145/3295500.3356153.

[87] Debendra Das Sharma. 2022. Compute Express Link®: An open industry-standard interconnect enabling heterogeneous data-centric computing. In *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*. 2022 IEEE Symposium on High-Performance Interconnects (HOTI). (Aug. 2022), 5–12. DOI: 10.1109/HOTI55740.2022.00017.

[88] Noam Shazeer et al. 2018. Mesh-TensorFlow: deep learning for supercomputers. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (NIPS'18). Curran Associates Inc., Red Hook, NY, USA, (Dec. 3, 2018), 10435–10444.

[89] James S. Sims and Nicos Martys. 2004. Simulation of Sheared Suspensions With a Parallel Implementation of QDPD. *Journal of Research of the National Institute of Standards and Technology*, 109, 2, 267–277. pmid: 27366609. DOI: 10.6028/jres.109.017.

[90] Evangelos Stromatias, Daniel Neil, Michael Pfeiffer, Francesco Galluppi, Steve B. Furber, and Shih-Chii Liu. 2015. Robustness of spiking Deep Belief Networks to noise and reduced bit precision of neuro-inspired hardware platforms. *Frontiers in Neuroscience*, 9, 222. pmid: 26217169. DOI: 10.3389/fnins.2015.00222.

[91] Xiaoqiang Sun, Peng Zhang, Joseph K. Liu, Jianping Yu, and Weixin Xie. 2020. Private Machine Learning Classification Based on Fully Homomorphic Encryption. *IEEE Transactions on Emerging Topics in Computing*, 8, 2, (Apr. 2020), 352–364. DOI: 10.1109/TETC.2018.2794611.

[92] E.E. Swartzlander and A.G. Alexopoulos. 1975. The Sign/Logarithm Number System. *IEEE Transactions on Computers*, C-24, 12, (Dec. 1975), 1238–1242. DOI: 10.1109/T-C.1975.224172.

[93] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. 2020. {sRDMA} – Efficient {NIC-based} Authentication and Encryption for Remote Direct Memory Access. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), 691–704. ISBN: 978-1-939133-14-4. Retrieved Aug. 27, 2023 from https://www.usenix.org/conference/atc20/presentation/taranov.

[94] The OpenSSL Project. OpenSSL: The Open Source toolkit for SSL/TLS. (Apr. 2003).

[95] Rens van de Schoot et al. 2021. Bayesian statistics and modelling. *Nature Reviews Methods Primers*, 1, 1, (Jan. 14, 2021), 1–26, 1, (Jan. 14, 2021). DOI: 10.1038/s43586-020-00001-2.

[96] W. Gregory Voss. 2017. European Union Data Privacy Law Reform: General Data Protection Regulation, Privacy Shield, and the Right to Delisting. (Jan. 5, 2017). Retrieved Mar. 27, 2023 from https://papers.ssrn.com/abstract=2894571. preprint.

[97] Alexander Wood, Kayvan Najarian, and Delaram Kahrobaei. 2020. Homomorphic Encryption for Machine Learning in Medicine and Bioinformatics. *ACM Computing Surveys*, 53, 4, (Aug. 25, 2020), 70:1–70:35. DOI: 10.1145/3394658.

[98] Xun Yi, Russell Paulet, and Elisa Bertino. 2014. Homomorphic Encryption. In *Homomorphic Encryption and Applications*. SpringerBriefs in Computer Science. Xun Yi, Russell Paulet, and Elisa Bertino, (Eds.) Springer International Publishing, Cham, 27–46. ISBN: 978-3-319-12229-8. DOI: 10.1007/978-3-319-12229-8_2.

[99] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing* (Lecture Notes in Computer Science). Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, (Eds.) Springer, Berlin, Heidelberg, 44–60. ISBN: 978-3-540-39727-4. DOI: 10.1007/10968987_3.

[100] Huasha Zhao and John Canny. 2014. Kylix: A Sparse Allreduce for Commodity Clusters. In *2014 43rd International Conference on Parallel Processing*. 2014 43rd International Conference on Parallel Processing. (Sept. 2014), 273–282. DOI: 10.1109/ICPP.2014.36.

[101] Huasha Zhao and John F. Canny. 2013. Sparse Allreduce: Efficient Scalable Communication for Power-Law Data. *CoRR*, abs/1312.3020. Retrieved Mar. 27, 2023 from http://arxiv.org/abs/1312.3020 arXiv: 1312.3020.

[102] Xuyang Zhao, Mingyu Li, Erhu Feng, and Yubin Xia. 2022. Towards A Secure Joint Cloud With Confidential Computing. In *2022 IEEE International Conference on Joint Cloud Computing (JCC)*. 2022 IEEE International Conference on Joint Cloud Computing (JCC). (Aug. 2022), 79–88. DOI: 10.1109/JCC56315.2022.00019.

[103] Jianping Zhu et al. 2020. Enabling Rack-scale Confidential Computing using Heterogeneous Trusted Execution Environment. In *2020 IEEE Symposium on Security and Privacy (SP)*. 2020 IEEE Symposium on Security and Privacy (SP). (May 2020), 1450–1465. DOI: 10.1109/SP40000.2020.00054.

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

10.5281/zenodo.8086366

ARTIFACT IDENTIFICATION

1.1 Abstract

Allreduce is one of the most commonly used collective operations. Its latency and bandwidth can be improved by offloading the calculations to the network. However, no way exists to conduct such offloading securely; in state-of-the-art solutions, the data is passed unprotected into the network. Security is a significant concern for High-Performance Computing (HPC) applications, but achieving it while maintaining performance remains challenging. We present HEAR, the first high-performance system for securing in-network compute and Allreduce operations based on homomorphic encryption. HEAR implements carefully designed and modified encryption schemes for the most common Allreduce functions and leverages communication domain knowledge in MPI programs to obtain decryption and encryption routines with high performance. Representing the first step towards achieving confidential HPC, HEAR operates on integers and floats with little or no hardware changes. We design, implement, and evaluate HEAR, showing its minimal overhead in a library called `libhear`. We evaluate parts of HEAR for correctness, overhead, scaling, precision loss, and example applications. This artifact description outlines the structure of the artifact, allowing for reproducing the results presented in the article, how it connects with the main contributions, and what flows to follow to evaluate it.

1.2 Structure

We split our artifact into two main parts: precision, correctness, and security evaluation of HEAR **A1** and latency, throughput, block size, scaling, and application experiments of its implementation, `libhear` **A2**. Some experiments were run in a distributed fashion using `Slurm` running on the Piz Daint supercomputer. The other experiments can be evaluated locally. All of the experiments were written in `C/C++`. This artifact allows for reproducing all of the obtained results in the article.

1.3 Contributions

We use our experiments to verify the claims connected to the main contributions of the article. Together with the named related experiments, these are:

(1) Defining a suitable INC and cHPC threat model alongside HEAR, a novel framework for confidential Allreduce operations allowing users to choose between performance and security. Homomorphically based HEAR works on integers and floats and supports addition, multiplication, and `XOR`.
   A1 We evaluate the correctness of encryption for summation for floating point and integer schemes.

(2) Defining and discussing the security of a new floating point HE scheme suitable for HPC usage. Presenting the analysis of the precision loss against the gained security for the floating points within the scheme.
   A1 We evaluate the relative security loss based on the probability of guessing the corresponding encrypted number in a uniform distribution and the distribution introduced by HEAR. We also show the loss of precision for the floats for encrypted summation.

(3) Designing, implementing, and evaluating HEAR, the first high-performance Allreduce operation framework based on MPI stack and datatypes, allowing for confidential in-network computing.
   A2 We evaluate the latency breakdown of HEAR implementation, `libhear`, the encryption/decryption throughput using different methods, and the optimal size for the pipelining block.

(4) Open-sourcing HEAR as a library for use with any MPI standard implementation without recompiling applications.
   A2 We evaluate the scaling of the scheme in terms of the used processes per node (PPN) and the number of nodes. We also show overheads, for example, applications using their proxy C++ implementation.

REPRODUCIBILITY OF EXPERIMENTS

We present experiments in categories, each showing the expected evaluation times in brackets. We used `Python` 3.7.4 for parsing the experiment results. `seaborn`, `matplotlib`, `pandas`, and `numpy` `Python` libraries are required. The precision experiments rely on the GNU MPFR library version 4.1.0. It can be obtained using a package manager (e.g., `sudo apt-get install libmpfr-dev=4.1`) or by downloading the corresponding release (https://www.mpfr.org/mpfr-4.1.0/). Additionally, the implementation of `libhear` depends on `OpenSSL` version 1.1.1. Some experiments depend on `MPI`. We used `cray-mpich` version 7.7.18. For compilation, we relied on `Clang` 13.0.1. All experiments requiring `MPI` and scaling were run on the Piz Daint supercomputer using the `Slurm` workload manager version 20.11.8. Unless otherwise stated, all the resulting intermediate logs, such as `.csv` files, are expected to be placed in `tests/results`. By default, the output figures are output to `plotting/figures`.

2.1 HEAR related experiments A1

To set the environment for HEAR experiments, run the following command in the root directory of the project:

- `make security accuracy correctness` (10 min)

Compilation using the provided `Makefile` will result in the following binaries:

- `security`;
- `accuracy_addition`;
- `accuracy_multiplication`;
- `hfloat_correctness`;

- `integer_correctness`;

2.1.1 Security evaluation. The security experiment evaluates the likelihood of an adversary guessing the correct plaintext based on ciphertext using the maximum a posteriori (MAP) estimator. The experiment iterates through all possible plaintexts and mantissa keys to evaluate the corresponding likelihoods in parallel on multiple machines relaying on `MPI`. Run the base experiment by executing:

- `mpirun -n 96 ./security` (180 min)

The experiment should be run on 96 cores. This number can be modified by changing the variable `NUM_CORES` in `/tests/security/secruity.cpp` and recompiling. Each rank outputs one log file called `mantissas_overall_start_end.csv` for overall unnormalized maximum likelihood in the processed ciphertexts. To parse the results, execute:

- `python3 tests/security/parse.py` (5 min)

Parsing entails finding the maximum, minimum, and average likelihoods. The printed likelihoods verify the results claimed in Section 5.3.1.

2.1.2 Loss of information. The loss of information experiments evaluate how much precision is lost when encrypting, adding/multiplying, and decrypting a series of floating point numbers. We evaluate this using a series of summations/products for encrypted and nonencrypted values and tracking the corresponding error using an `MPFR` variable with large precision. Run the experiment by executing:

- `./accuracy_addition -n combined` (60 min)
- `./accuracy_mulitplication -n new -t 10000` (30 min)

This will output six `.csv` files in the `tests/accuracy/results` folder which can be parsed by running:

- `cd plotting`
- `python3 accuracy.py` (5 min)

Parsing will output a plot named `accuracy.pdf`, which verifies the accuracy claimed in Sections 5.3.2 and 5.3.3 of the article together with Figure 3.

2.1.3 Float correctness. Float correctness shows that by encrypting and decrypting a number, its value is not lost and differs from the original number only by a certain precision loss. We evaluated a large number of encryptions/decryptions using random numbers. Run the experiment by executing:

- `./hfloat_correctness` (10 min)

The resulting printed average error and runtime will verify the claims from Section 6 about the floating point error.

2.1.4 Integer correctness. Similarly to floats, we verify that encryption and decryption, together with a summation, do not introduce any errors for integers. For that, we encrypt a series of numbers, sum them up, decrypt, and verify that the corresponding result is correct. Run the experiment by executing:

- `./integer_correctness` (10 min)

The program not failing due to assertion errors shows the correctness of encryption and decryption as claimed in Section 6.

2.2 `libhear` related experiments A2

To set the environment for `libhear` performance experiments, run the following command in the root directory of the project:

- `bash ./sourceme.sh` (10 min)

Execution of this script will result in compiling all the experimental toolchain:

- `./build/lib/libhear_critical_path_baseline.so` - RDTSC performance counters profiling of baseline MPI library;
- `./build/lib/libhear_critical_path_naive.so` - RDTSC performance counters profiling of naive `libhear` implementation;
- `./build/lib/libhear_critical_path_mpool.so` - RDTSC performance counters profiling `libhear` implementation optimized for small MPI messages;
- `./build/lib/libhear_mpool_release.so` - release build of `libhear` optimized for small MPI messages;
- `./build/lib/libhear_release.so` - release build of `libhear` optimized for large MPI messages;
- `./build/bin/gpt3` - GPT3 DNN model proxy-benchmark;
- `./build/bin/cosmoflow` - CosmoFlow DNN model proxy-benchmark;
- `./build/bin/resnet` - ResNet-152 DNN model proxy-benchmark;
- `./build/bin/dlrm` - DLRM DNN model proxy-benchmark;
- `./build/bin/osu_allreduce_int` - OSU Allreduce collective benchmark with `MPI_INT` datatype;
- `./build/bin/osu_allreduce_float` - OSU Allreduce collective benchmark with `MPI_FLOAT` datatype;

By default, `<logs_dir>` should be set to `tests/implementation/results` where all of the plotting scripts assume the resulting `.csv` files to be located. This behavior can be changed by modifying the plotting scripts.

2.2.1 Encryption throughput. We evaluate the encryption throughput for integers and floats depending on the used backend. Run the base experiment by executing in the root folder:

- `python3 ./scripts/single_core_encr_tput.py $(pwd)/build/ <logs_dir>` (30 min)

2.2.2 Latency breakdown. We evaluate the latency of Allreduce for integer summation and how it is influenced by `libhear`. Run the base experiment by executing:

- `bash ./scripts/batch_critical_path.sh $(pwd) $(pwd)/build/lib/ <logs_dir>` (30 min)

By default, the experiment runs with 2 MPI ranks and 1 rank per node. The default value can be changed in the script.

2.2.3 Optimal blocksize. We evaluate the optimal pipelining block size for `libhear` by evaluating throughput for various block sizes. Run the base experiment by executing:

- `bash ./scripts/batch_block_size.sh $(pwd) $(pwd)/build/lib/ <logs_dir>` (120 min)

By default, the experiment runs with 96 MPI ranks and 48 ranks per node. The default value can be changed in the script.

2.2.4 Throughput and latency scaling. We evaluate the throughput and latency scaling for `libhear` for integers and floats. Run the base experiment by executing:

- `bash ./scripts/batch_allreduce_<dtype>_scaling.sh $(pwd) $(pwd)/build/lib/ <logs_dir>` (120 min), `<dtype>` denotes Allreduce datatype

By default experiment runs the following configurations of `<number of MPI ranks>/<number of nodes>`: 2/2, 4/2, 8/2, 24/2, 48/2, 96/2. The default value can be changed in the script. For a larger number of nodes, one has to modify the `hostfile` accordingly to contain only the required number of nodes, as the ranks are equally spread out between the nodes.

2.2.5 Example applications. We evaluate the influence of `libhear` on evaluation times of example applications. Run the base experiment by executing:

- `bash ./scripts/batch_dnns.sh ./build/bin/ ./build/lib/ <logs_dir>` (120 min)
- `bash ./scripts/batch_gpt3.sh ./build/bin/ ./build/lib/ <logs_dir>` (120 min)

By default `batch_dnns.sh` runs experiments for DLRM, ResNet, and CosmoFlow proxy-apps on 16 nodes and 16 MPI ranks per node. `batch_gpt3.sh` runs experiments for GPT3 models on 48 nodes and 8 MPI ranks per node. The lower number of nodes is not possible due to the memory usage of these experiments. The output provides the results directly. For plotting, convert these to a `.csv` format.

2.2.6 Results post-processing. To post-process the resulting logs, run:

- `python3 postprocess_synthetic_perf.py <logs_dir>` (10 min)

Postprocessing script will output the following files in `<logs_dir>` `directory`:

- `./critical_path.csv` - CSV table with critical path latency measurements;
- `./single_core_encr_tput.csv` - CSV table with single core encrytion/decryption throughput measurements;
- `./block_size.csv` - CSV table with block size latency/throughput measurements;
- `./allreduce_int_scaling.csv` - CSV table Allreduce with integer datatype latency/throughput measurements;
- `./allreduce_float_scaling.csv` - CSV table Allreduce with float datatype latency/throughput measurements;

2.2.7 Results plotting. To obtain the plots as in the article, run `cd plotting` and:

- `python3 latency_breakdown.py` (5 min) - outputting a single plot `latency_breakdown.pdf` verifying the claims from Section 6 and Figure 4;
- `python3 throughput_enc_dec_combined.py` (5 min) - producing a single plot `throughput_one_core_enc_dec.pdf` verifying the claims from Section 6 and Figure 5;
- `python3 optimal_blocksize.py` (5 min) - resulting in a single file `block_size.pdf` verifying the results claimed in Section 6 Figure 6;
- `python3 throughput_scaling.py` (5 min) - producing a single plot `throughput_scaling.pdf` verifying claims in Section 7.1 and Figure 7;
- `python3 latency_scaling.py` (5 min) - producing a single plot `latency_scaling.pdf` verifying claims in Section 7.1 and Figure 8;
- `python3 dnn_overhead.py` (5 min) - producing a single plot `dnn_overhead.pdf` verifying claims in Section 7.2 and Figure 9;

All resulting plots will be placed in `plotting/figures`.

ARTIFACT DEPENDENCIES REQUIREMENTS

3.1 Artifact Dependencies and Requirements

We provide three ways to evaluate our work:

(1) A QEMU predefined image that allows running a cluster of bare-metal machines, e.g., on Chameleon Cloud.
(2) A Docker image that can be used for local testing and verifying some experiments.
(3) Scripts for a Slurm cluster.

The hardware should have AES-NI support, and no input datasets are needed. The exact package versions used in our experiments in the paper are described in the section “REPRODUCIBILITY OF EXPERIMENTS" while the evaluation versions can be found below.

3.1.1 QEMU image. We provide a predefined image configured as an MPI machine that can be scaled freely to form a cluster. The image should not be used in production due to `SSH` security issues. The image is part of the artifact. We also provide it directly in the Chameleon Cloud interface for ease of evaluation. The image contains all the necessary installed dependencies and the appropriate compiled code. Because of such a setup, the cluster can be scaled without major manual effort. We used the TACC cluster for testing the evaluation and either the Icelake or Haswell nodes. The package versions used in the evaluation are in the Dockerfile provided as the second evaluation method.

3.1.2 Docker image. We also provide a Dockerfile that can be built into an image. The Dockerfile is self-contained and provides all of the requirements. We used Ubuntu 22.04 with Docker version 20.10.23 to run the Docker image. No special hardware is necessary, yet some experiments might not work in this setup (experiments requiring a larger scale).

3.1.3 Slurm cluster. The two previous methods rely on the manual setting of MPI. We also provide scripts for running the work on a Slurm cluster, allowing for easier scalability. The dependencies outlined in the paper and described in the “REPRODUCIBILITY OF EXPERIMENTS" section must first be preinstalled.

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

4.1 Artifact Installation and Deployment Process

4.1.1 QEMU image on Chameleon Cloud (15 minutes).

(1) Spawn the cluster. First, create a reservation with the necessary number of nodes to do that. For the non-scaling experiments, we recommend 2. Then, go to the instances and spawn the required number of instances. In the images, please find the image called `hear` in the available sources. The image is also attached to the artifact, which allows for creating it in the CLI interface. Assign the floating IP address and connect to one of the instances.
(2) Execute `cd homomorphic-mpi/`
(3) Create the file `hostfile` and insert all of the local IP addresses of the instances separated by newlines.

4.1.2 *Docker (15 minutes).*

(1) Compile the Docker image by entering the `homomorphic-mpi` directory and running `docker build -t hear .`. We also provide a ready image in the Docker hub for your convenience. To use it, run `docker pull spcleth/hear`.
(2) If you compiled your container manually, run it using `docker run -it –mount type=bind,src="`(pwd)`", target = / /project/plotting/figureshearbash.If youusedthereadyimage, youcanrun`

4.1.3 *Slurm cluster (3/4 hours).*

(1) Follow the usual installation of the packages outlined within the paper and the “REPRODUCIBILITY OF EXPERIMENTS" section. The ready instructions for some environments can be found in the Dockerfile.
(2) In all of the shell scripts, uncomment commented lines with `srun`, and comment `mpirun` lines out.
(3) In the file `postprocess_synthetic_perf.py` comment existing lines with `ranks=[2,4,8,24,48,96]` and uncomment `ranks=[2,4,8,18,36,72,144,288,576,1152]`.
(4) Use `sbatch` instead of `bash` to deploy the workloads.