IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2021, No. 3, pp. 756–779. DOI:10.46586/tches.v2021.i3.756-779

SEAL-Embedded: A Homomorphic Encryption Library for the Internet of Things

Deepika Natarajan[1] and Wei Dai[2]

[1] University of Michigan, Ann Arbor, MI, dnataraj@umich.edu
[2] Microsoft Research, Redmond, WA, wei.dai@microsoft.com

Abstract. The growth of the Internet of Things (IoT) has led to concerns over the lack of security and privacy guarantees afforded by IoT systems. Homomorphic encryption (HE) is a promising privacy-preserving solution to allow devices to securely share data with a cloud backend; however, its high memory consumption and computational overhead have limited its use on resource-constrained embedded devices. To address this problem, we present SEAL-Embedded, the first HE library targeted for embedded devices, featuring the CKKS approximate homomorphic encryption scheme. SEAL-Embedded employs several computational and algorithmic optimizations along with a detailed memory re-use scheme to achieve memory efficient, high performance CKKS encoding and encryption on embedded devices without any sacrifice of security. We additionally provide an "adapter" server module to convert data encrypted by SEAL-Embedded to be compatible with the Microsoft SEAL library for homomorphic encryption, enabling an end-to-end solution for building privacy-preserving applications. For a polynomial ring degree of 4096, using RNS primes of 30 or fewer bits, our library can be configured to use between 64–137 KB of RAM and 1–264 KB of flash data, depending on developer-selected configurations and tradeoffs. Using these parameters, we evaluate SEAL-Embedded on two different IoT platforms with high performance, memory efficient, and balanced configurations of the library for asymmetric and symmetric encryption. With 136 KB of RAM, SEAL-Embedded can perform asymmetric encryption of 2048 single-precision numbers in 77 ms on the Azure Sphere Cortex-A7 and 737 ms on the Nordic nRF52840 Cortex-M4.

Keywords: Homomorphic Encryption, IoT, embedded systems

1 Introduction

The emergence of the Internet of Things (IoT) has led to a dramatic increase in the number and pervasiveness of devices present around the globe. As IoT devices continue to radically transform domains of daily life, including privacy-sensitive domains such as healthcare, manufacturing, and home automation, users are becoming increasingly aware of the privacy and security risks that accompany the device-to-cloud model adopted by most smart devices. In particular, due to their limited memory capacities and low-power operational states, small embedded devices often rely on external sources of storage and processing to obtain useful insights from device-collected data. Such deployments often utilize cloud computing to perform analytics on this data, which must inherently be carried out while the data is in plaintext form.

Though the above deployment scenario is standard in most domains, it raises several security and privacy concerns: First, the cloud trusted computing base, or the set of hardware, software, and firmware components that a user must trust for security in the cloud, exposes a broad surface for attackers to exploit. Moreover, protection against certain classes of cloud attacks, such as side channel or physical attacks, are often difficult or

Licensed under Creative Commons License CC-BY 4.0.
Received: 2020-10-15 Accepted: 2020-12-15 Published: 2021-07-09

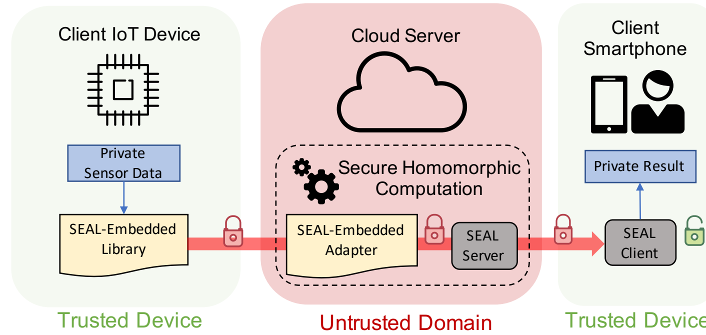

Figure 1: Secure end-to-end computational flow enabled by the SEAL-Embedded solution.

impossible to achieve in a shared cloud setting. Additionally, so-called "honest-but-curious" or even malicious cloud infrastructure or server application providers may constitute an added threat as they eagerly attempt to extract and store private information from user data (e.g. for purposes such as targeted advertising).

Homomorphic encryption (HE), a cryptographic technology that enables secure computation on encrypted data, has often been posed as a solution to the aforementioned threats. Since the construction of the first fully homomorphic encryption scheme in 2009, several improvements have been introduced to make HE efficient enough for practical use. One recent HE construction, known as the CKKS scheme [CKKS17], enables *approximate* homomorphic encryption for efficient processing of floating point data. Prior works have established CKKS as a leading contender for efficient homomorphic logisitic regression and other machine learning tasks [KSW+18, JKLS18, KSK+18, BGP+20, KSLM20]. The CKKS scheme is a natural choice for IoT devices since it can efficiently perform secure computation on the type of real-valued data often sampled by sensors.

However, the computationally heavy nature of homomorphic encryption operations has still largely prevented HE from being ported to the IoT domain. Indeed, currently available homomorphic encryption libraries require much more memory than constrained devices are able to provide. For example, for a polynomial ring with degree 4098 and three 30-bit primes, a commonly used parameter set, a simple CKKS encode-and-encrypt operation with Microsoft SEAL [SEA20] requires a minimum of 4 MB in memory allocation - much larger than the 256 KB of main memory that is typical of many embedded devices. Additionally, the CKKS scheme presents an additional complexity in that its encoding operation requires the implementation of double-precision arithmetic modules (e.g. the Fast Fourier Transform) in addition to modular integer units (e.g. the Number Theoretic Transform) for encryption, while comparable HE schemes such as BFV [FV12] and BGV [BGV12] only require the latter. This significantly increases the memory and computational requirements for an efficient CKKS implementation.

In this work, we show that while there are several memory and performance challenges to overcome in the embedded domain, it is ultimately possible and practical to use HE to encrypt data on IoT devices and to take advantage of secure computation in the cloud. More concretely, we make the following contributions:

- We identify the specific memory and performance challenges related to enabling HE on embedded devices and discuss several techniques to overcome them. In particular, we show how to use RNS partitioning, data type compression, and memory pooling, along with a detailed memory re-use scheme to reduce the memory consumption of HE encoding and encryption while still enabling high performance. Importantly, by employing RNS partitioning, our library's memory usage is constrained only by the degree of ciphertext polynomials rather than the full size of ciphertexts.

- We implement our techniques in SEAL-Embedded, the first homomorphic encryption library targeted for constrained embedded devices, featuring the encoding and encryption procedures of the CKKS scheme. As part of our design, we include a server-side adapter that acts as a translation layer from SEAL-Embedded to the Microsoft SEAL library [SEA20] for homomorphic encryption. Together, these components enable an end-to-end HE deployment solution for embedded devices, as shown in Figure 1.
- We demonstrate the utility, efficiency, and versatility of our solution by deploying our library on two embedded devices with varying degrees of constraints. In particular, we target a standalone Cortex-M4 device for low-power implementations, as well as the high-performance Cortex-A7 on an Azure Sphere IoT platform.
- We provide performance and memory usage results for a variety of configurations enabled by our library, for a parameter set that enables insightful analytics, such as linear inference, on encrypted data. With 136 KB of RAM, we achieve 77 ms and 737 ms for asymmetric encode-and-encrypt for 2048 single-precision numbers on a Cortex-A7 and Cortex-M4, respectively.

Organization The remainder of this paper is organized as follows: Section 2 covers preliminaries, including an introduction to the CKKS scheme and arithmetic algorithms. Section 3 gives a high-level overview of our library's components and usage in a secure remote computation scenario. Section 4 provides further details of our library's implementation, including various optimizations and their impact on computational efficiency and memory consumption. Finally, Section 5 presents the performance and memory consumption results of our library evaluation on Cortex-A7 and Cortex-M4 devices.

2 Preliminaries

2.1 Notation

We define the elements of the integer quotient ring $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ as integers in $(-q/2, q/2]$ and represent them in our software implementation with unsigned integers in $[0, q-1]$. Throughout this paper, we take n to be a power of two. We choose the $2n$-th cyclotomic polynomial $x^n + 1$ to define a quotient ring $R = \mathbb{Z}[x]/(x^n + 1)$ whose elements are integer polynomials with degree less than n. The residue ring of R modulo q is denoted as $R_q = R/qR$. We use a_i to denote the i-th element of a vector a or the i-th coefficient of a polynomial a. Operator $[\cdot]_q$ reduces an element modulo q. For a finite set S, $\mathcal{U}(S)$ denotes the uniform distribution on S. We denote the ternary distribution $\mathcal{U}(\{-1, 0, 1\}^n)$ as $\mathcal{U}(R_3)$ (since they are computationally identical), and the centered binomial distribution parameterized by η as $\mathcal{B}_\eta$, whose samples range in $\mathbb{Z} \cap [-\eta, \eta]$. We use $\mathcal{B}_\eta[x]$ to denote the distribution of polynomials whose coefficients are sampled from $\mathcal{B}_\eta$

2.2 Homomorphic Encryption and Ring-Learning with Errors

Homomorphic encryption is a cryptographic technology that allows for secure computation on encrypted data. Throughout this paper, when we refer to homomorphic encryption, we are specifically referring to a somewhat homomorphic encryption scheme that allows both addition and multiplication on ciphertexts, with a limitation on the multiplicative depth of computation (a so-called "leveled" homomorphic encryption scheme). HE schemes typically include five main computation modules: parameter selection, key generation, encryption, evaluation, and decryption. Parameters are selected based on a desired security level and the multiplicative depth of computation that will be performed on ciphertexts.

Key generation is performed by a trusted party and creates a secret key for decryption and encryption. Key generation can also include the generation of one or more public keys for encryption and/or other public functional keys to be used in evaluation, all generated from the same underlying secret key. Encryption is non-deterministic and can be either symmetric or asymmetric, while evaluation refers to computation on encrypted data (typically performed by an untrusted party) to yield encrypted results. Finally, decryption is performed by a trusted party with access to the secret key.

The security of most efficient homomorphic encryption schemes, such BGV [BGV12], BFV [FV12], and CKKS [CKKS17], relies on the Ring Learning with Errors (RLWE) problem. Given a key s chosen from a key distribution over R, an RLWE sample $(b, a) \in R_q^2$ is constructed by sampling a from $\mathcal{U}(R_q)$ and noise e randomly from an error distribution over R and computing $b = as + e \pmod q$. The RLWE assumption is that the distribution of (b, a) and $\mathcal{U}(R_q^2)$ are computationally indistinguishable. For our implementation, we choose $\mathcal{U}(R_3)$ as our key distribution and $\mathcal{B}_{21}$ as our error distribution, and choose secure encryption parameters in compliance with the HE security standard [ACC+18].

The aforementioned HE schemes all support a technique called *batching*. Here, batching refers to the encoding of multiple messages (where each message may consist of several bits) into a single plaintext. The resulting batched plaintext can then be encrypted into a single corresponding batched ciphertext. Moreover, computation on batched ciphertexts can be performed in a single-instruction-multiple-data (SIMD) fashion on the underlying messages, reducing the cost of homomorphic evaluation by several orders of magnitude.

2.3 CKKS

In 2017, Cheon, Kim, Kim, and Song introduced the first HE scheme for approximate homomorphic encryption [CKKS17]. While other HE schemes can only preserve integer modular arithmetic, producing modulo-reduced results instead of exact computation for values that exceed a chosen *plaintext* modulus, the CKKS scheme allows for truncation of the lower bits of the underlying floating point values. This truncation, which can be controlled by carefully selecting the scale parameter Δ to mark the position of the decimal point, allows the results of CKKS homomorphic evaluation to occupy a dynamic range. These properties have led to the CKKS scheme becoming widely considered as the most desirable scheme for floating point computation, where approximation of values can be tolerated. This makes CKKS an excellent fit for computations such as machine learning, which naturally operate on floating point, imprecise values, and is evidenced by several prior works demonstrating efficient CKKS-based logisitic regression and other machine learning tasks [KSW+18, JKLS18, KSK+18, BGP+20, KSLM20]. By contrast, the CKKS scheme would not be a good fit for computations such as private-set intersection, where exact values are typically required. We choose to focus on the CKKS scheme as the basis for our library (though our library contains all the core components for the BFV scheme as well), since its ability to compute on floating point values makes it a natural fit to encode the analog data typically captured by IoT device sensors.

Below, we give a summary of the CKKS encoding and encryption procedures. For simplicity, we omit the description of CKKS key generation, evaluation, and decoding/decryption procedures, since these do not need to be implemented on the device itself. We refer readers to [CKKS17] and [CHK+19] for detailed descriptions of each module.

`CKKS.Encode`$(z \in \mathbb{C}^{n/2},\ \Delta \in \mathbb{R})$: Let T be a multiplicative subgroup of $\mathbb{Z}_{2n}^*$ with order $n/2$. The subring $\mathbb{H} = \{(z_j)_{j \in \mathbb{Z}_{2n}^*} : z_j = \overline{z_{-j}}\}$ of $\mathbb{C}^n$ can be identified with $\mathbb{C}^{n/2}$ via the natural projection $\pi : (z_j)_{j \in \mathbb{Z}_{2n}^*} \mapsto (z_j)_{j \in T}$. Given a complex $2n$-th root of unity $\zeta = e^{\pi i/n}$, the canonical embedding $\sigma : \mathbb{C}[x]/(x^n + 1) \mapsto \mathbb{C}^{n/2}$ is an injection and is defined as $\sigma(a) = (a(\zeta^j))_{j \in T}$. Denote as $\lfloor \sigma^{-1}(\pi^{-1}(z)) \rceil_R$ the discretization of $\sigma^{-1}(\pi^{-1}(z))$ to R. A scalar $\Delta \geq 1$ is multiplied before the discretization so that the rounding error does not affect significant bits. Output plaintext polynomial $m = \lfloor \Delta \cdot \sigma^{-1}(\pi^{-1}(z)) \rceil_R \in R$.

`CKKS.SymEncrypt`(secret key $\mathtt{sk} \in R_3$, plaintext $m \in R$): Sample $a \leftarrow \mathcal{U}(R_q)$ and coefficients of $e \in R$ from $\mathcal{B}_{21}$. Output $(-a \cdot \mathtt{sk} + e + m, a) \in R_q^2$.

`CKKS.AsymEncrypt`(public key $\mathtt{pk} = (\mathtt{pk}_0, \mathtt{pk}_1) \in R_q^2$, plaintext $m \in R$): Sample $u \leftarrow \mathcal{U}(R_3)$ and coefficients of $e_0, e_1 \in R$ from $\mathcal{B}_{21}$. Output $(u \cdot \mathtt{pk}_0 + e_0 + m, u \cdot \mathtt{pk}_1 + e_1) \in R_q^2$.

Note that the above description for `CKKS.Encode` is different from the description given in [CKKS17], which defines the canonical embedding as $R \mapsto \mathbb{C}^{n/2}$ and therefore applies discretization to $\pi^{-1}(z)$. This difference has no effect on the correctness of encoding, since the introduced rounding error can be scaled down by Δ during the decode operation. Additionally, the above description shows that CKKS is actually able to encode *complex* floating-point values rather than just real values. Since prior works have largely been unable to take advantage of complex encoding in real-world use cases, we assume an encoding of real inputs $z \in \mathbb{R}^{n/2}$ throughout this paper.

2.4 Residue Number System

To be compliant with the Homomorphic Encryption Security Standard [ACC+18], HE schemes must be able to support multi-precision integer arithmetic. To efficiently compute integer arithmetic of lengths larger than a single processor word-size, prior works make use of the Chinese Remainder Theorem (CRT) (equivalently, the residue number system (RNS)) and choose the modulus q to be a product of smaller pair-wise coprime moduli $q_0, q_1, \cdots, q_{L-1}$ (e.g. primes). This establishes an isomorphism $R_{q_0 q_1 \ldots q_{L-1}} \mapsto R_{q_0} \times R_{q_1} \times \cdots \times R_{q_{L-1}}$. To avoid costly CRT conversions, which involve multi-precision integer arithmetic, full-RNS variants of HE schemes were developed to allow ciphertexts to remain in RNS form. The full-RNS variant of CKKS was proposed in [CHK+19] and is implemented by all libraries that support CKKS, including Microsoft SEAL [SEA20], PALISADE [PAL], HEAAN [HEA], and HElib [HEl]. Under this RNS variant, an element in R_q can be stored as an array of elements in $R_{q_0}, R_{q_1}, \ldots, R_{q_{L-1}}$.

2.5 Number Theoretic Transform (NTT)

One major performance bottleneck of HE schemes is polynomial multiplication. The Number Theoretic Transform (NTT) is a commonly utilized technique for fast polynomial multiplication. The NTT is similar to the Fast Fourier Transform (FFT) except that it operates over a prime field instead of the complex field. In both FFT and NTT algorithms, inputs are multiplied with twiddle factors (powers of an n-th primitive root of unity) and combined with each other in a "butterfly"-like manner. To reduce the polynomial product modulo $x^n + 1$ for our ring structure, we perform a negacyclic convolution on the inputs using a special transform whose twiddle factors are powers of a primitive $2n$-th root of unity, called the "negacyclic NTT". In the rest of paper, we use "NTT" to refer to this negacyclic NTT. The method for using the negacyclic convolution for polynomial multiplication between two $(n-1)$-degree polynomials consists of applying the negacyclic NTT to each polynomial, multiplying the transformed polynomials together in a point-wise manner, and performing an inverse negacyclic NTT on the result to get the n-dimensional product. This process is dominated by the NTT/INTT operations, which gives the NTT-based multiplication process an overall complexity of $O(n \log n)$. We use the NTT algorithm for negacyclic convolution as summarized by [LN16], given in Algorithm 1 in Appendix A. Our optimized NTT implementation is described in detail in Section 4.5.

2.6 Fast Fourier Transform for σ^{-1} Projection

The σ^{-1} map in the `CKKS.Encode` procedure (described in Section 2.3) can be performed by evaluating a polynomial at $\zeta^0, \zeta^{-1}, \ldots, \zeta^{-(n-1)}$, where ζ is a primitive $2n$-th root of $x^n + 1$ in $\mathbb{C}$, e.g., $e^{\pi i/n}$. This process can be computed through a negacyclic inverse FFT

(IFFT) over $\mathbb{C}$, which is conceptually similar to the inverse negacyclic NTT defined over $\mathbb{Z}_q$. We adapted the inverse negacyclic NTT algorithm described in [LN16] to an efficient IFFT algorithm for σ^{-1}, given in Algorithm 2 in Appendix A. We describe our implementation and optimizations for the IFFT in more detail in Section 4.4.

2.7 Related Work

Modern HE schemes are lattice-based and post-quantum in nature. As such, they share many algorithmic commonalities with other post-quantum encryption schemes such as CRYSTALS-KYBER [SAB+19], NewHope [PAA+19], SABER [DKRV19], and NTRU [ZCH+19]. However, HE schemes operate on vectors and in rings with dimensions orders of magnitude larger than those in post-quantum cryptosystems, leading to significantly higher memory consumption for HE implementations. This makes HE much more difficult to implement on embedded devices, since these devices often feature limited memory availability and performance constraints. While several post-quantum-based encryption schemes have been demonstrated on embedded devices [ABCG20, MKV20], no homomorphic encryption scheme has been ported to the embedded domain.

Other HE schemes such as CGGI [CGGI20] and DM [DM15], also known as TFHE and FHEW, respectively, feature lightweight encryption algorithms with $O(n)$ complexity. Although these schemes may be suitable for embedded devices, since they feature low computational overhead and low memory consumption during encryption, they have many orders of magnitude higher ciphertext-to-message size expansions. In particular, each ciphertext in these schemes encrypts only a single bit (or very few bits). Additionally, the bootstrapping operation, a technique which enables unlimited computation on ciphertexts, is both necessary and costly for these schemes, since it must be applied after each ciphertext operation during evaluation. This results in a much slower evaluation performance compared to leveled HE schemes such as CKKS, BFV, and BGV. Taking CKKS with a 4096-degree ring and 72-bit modulus as an example, the ciphertext-to-message size expansion ratio for 2048 20-bit messages is 7.2, compared to a ratio of $16,000$ in CGGI; squaring 2048 20-bit encrypted messages takes a fraction of a millisecond in CKKS, while a single *logic gate* in CGGI with 1-bit inputs has an overhead of 13 ms.

Prior works [GHS12, LN14, DSES14] have shown that it is possible to achieve homomorphic encryption indirectly by first evaluating a block cipher to encrypt a message, and then having an untrusted party homomorphically evaluate the block cipher's decryption circuit to obtain an HE ciphertext from the initial block cipher encryption. The computational cost of evaluating the decryption circuit can be minimized using a low-depth block cipher [AGR+16]. However, this technique does not allow for the efficient floating point arithmetic evaluation, since messages are in an extension field.

Both [CDKS20] and [BGGJ20] support conversion of ciphertexts from a lightweight LWE encryption to an RLWE-based HE encryption. The former [CDKS20] has low ciphertext-to-message size expansion and supports packing. However, it does not allow message operations to be performed in a SIMD fashion on RLWE ciphertexts, thereby losing several orders of magnitude speedup. The latter [BGGJ20] uses TLWE ciphertexts, which are inefficient in storage and communication cost compared to packed ciphertexts.

In [LM20], the authors demonstrated a key-recovery attack on the CKKS scheme. However, the attack requires an attacker to have access to a decryption oracle, e.g. by allowing the attacker to query the client device for the decryption of any previously encrypted ciphertexts. Since this attack arises from an incorrect instantiation of the HE protocol within a full application workflow, it does not weaken the security of our work.

3 SEAL-Embedded Design

The SEAL-Embedded solution consists of two main components: the device library, and the cloud adapter. The device library contains optimized functions for encoding and both symmetric and asymmetric encryption of the CKKS scheme. A user can use the SEAL-Embedded solution for secure end-to-end homomorphic computation as follows: After gathering the private sensor data on the embedded device, the user invokes the SEAL-Embedded library to encode and encrypt the data into a CKKS ciphertext. The user then transfers the ciphertext to an instance of the adapter in the cloud over a network of the user's choice. The adapter translates the ciphertexts into a form interpretable by the SEAL library, which can then be invoked by an application to perform homomorphic evaluation on the still-encrypted data. Finally, the computed results can be sent back to the client, who can then use the SEAL library to decrypt and decode the results.

Adapter. The SEAL-Embedded cloud adapter is designed to be a translation layer between the outputs of the SEAL-Embeddedd client device and the Microsoft SEAL library for homomorphic encryption. This design decision allows updates to the SEAL API to be handled by a simple update to the adapter in the cloud rather than a more cumbersome device library update. The adapter is further responsible for interpreting and post-processing the data sent from the device, constructing the objects required by the SEAL server library, collecting the received data into these objects, and finally passing the objects to the SEAL library. Since the adapter does not rely on any private user data, it can be deployed safely in the untrusted cloud alongside the instance of the SEAL server.

Key Generation. The remaining step for the end-to-end flow described above is the generation of the encryption keys. Although the SEAL-Embedded library contains functions for secret key and public key generation, it is likely that a user will want to use the SEAL-library to generate their keys ahead of device deployment. This would allow a user to, for instance, also generate any additional public keys that may be required by cloud operations. Therefore, the SEAL-Embedded adapter also contains functions for generating the public and secret keys in a form that is interpretable by the SEAL-Embedded library. These keys can then be generated in a trusted environment using the SEAL-Embedded adapter and either installed on the embedded device along with the target application image or sent to device at run-time. For extremely constrained devices, these keys can even be partitioned and sent on a per-prime basis. This scenario is compatible with our implementation, which also computes all operations on a per-prime basis.

3.1 Threat Model

For our threat model, we consider a *client* or *user* to be a trusted party capable of performing key generation, encryption, and decryption in trusted environment, and a *server* to be an untrusted party to which the client would like to offload computation on sensitive data. We use the term *device* to refer to a trusted environment owned and operated by the client (e.g. an embedded device) that is responsible for collecting the private data. The goal of our system is to provide a client device with the capability of homomorphically encoding and encrypting the data collected by the device in order to facilitate its offload to an untrusted server. In particular, we wish to ensure the privacy of data from the point at which it leaves the client device to the point at which any results are returned to the client.

We consider the following classes of attacks, and discuss each in turn below:

- network attacks - A network attacker sees all traffic across all non-private networks. A network attacker may act actively or passively and may exploit side-channel leakage transmitted over the network to learn user data.

- cloud attacks - A cloud attacker may refer either to an external entity who can corrupt a cloud sever (e.g. by exploiting bugs in large cloud operating systems) or a cloud infrastructure provider themselves. A cloud attacker may attempt to circumvent traditional protections in the cloud to obtain read access to private data through conventional software methods (e.g. buffer overflows), side-channel means such as timing or power analysis, or physical attacks such as cold-boot attacks, or if in control of the entire server, may simply read all server state. We consider cloud attackers to be *semi-honest*, meaning that they will follow the prescribed server protocol, but will seek to learn as much private user information as possible.
- device attacks - A device attack refers to an attack on the device itself. These attacks may be remote and injected through direct or indirect (e.g. side-channel) software means, or may refer to physical attacks on devices in deployment settings. The latter may be particularly relevant in certain IoT use-cases, where edge nodes are sometimes deployed in public areas (e.g. city-wide sensors).

We discuss the threat protections that our system is able to provide at a high level and our deployment assumptions. First, we assume that any transfer of public or secret keys for HE encryption can be transferred to the device securely, either before device deployment or over a secure channel. We also assume that the deployed IoT device has access to standard network protection mechanisms for confidentiality, integrity, and freshness of transmitted data. These network security mechanisms are still required on top of homomorphically encrypted ciphertexts since they are necessary for protection against active attacks and secure entire network messages (including any network packet headers) rather than just the ciphertext data itself. In our implementation of SEAL-Embedded on the Azure Sphere, for example, communication channels are protected by the TLS protocol. Further, as discussed in the introduction, homomorphic encryption systems send data to the server in encrypted form. The key property of homomorphic encryption schemes is that they allow the untrusted server to operate on the encrypted data directly without needing to decrypt the data or know the secret key. Since no party can learn anything about the underlying data through the encrypted ciphertexts alone, HE is able to provide protection against all aforementioned cloud attackers. HE itself is not robust against malicious cloud attackers, who can manipulate encrypted ciphertexts in the cloud in addition to viewing server state. However, HE can still offer data privacy against these adversaries as well.

A remaining threat that we consider is the threat of attacks on the device directly. We consider these threats largely out of scope, since we assume that an attack on the device can easily access the underlying private user data (e.g. sensor samples) or the source of data itself. Since an attacker would already have access to the raw data in this scenario, any additional protections on ciphertexts, keys, or error polynomials would be extraneous. However, if a secret key is stored on device, a device attacker could theoretically also obtain and use this key to decrypt all previously encrypted information stored in the cloud, thereby expanding the scope of a device attack to a potentially large amount of previously offloaded user data. To mitigate this situation, users can choose to update their device secret keys frequently, thus limiting the interval of data that a device attacker could decrypt using the user's secret key. However, this would mean that any data encrypted under the old secret key would no longer be able to be analyzed alongside any data encrypted with the new secret key.[1] A better solution to mitigate this problem is to encrypt data on the device using a public key instead of a secret key, thereby removing the need for the secret key to be stored on the device entirely. A CKKS public key is much larger in

[1]Assuming circular security, it would be possible for a user to initiate a secure *key-switching* operation in the cloud to transform previously encrypted data into a ciphertext that could be decrypted by the new secret key, as described in [CKKS17]. However, after performing this key-switching operation, a user would still be left with the problem that an attacker possessing the new secret key would be able to decrypt the previously encrypted data.

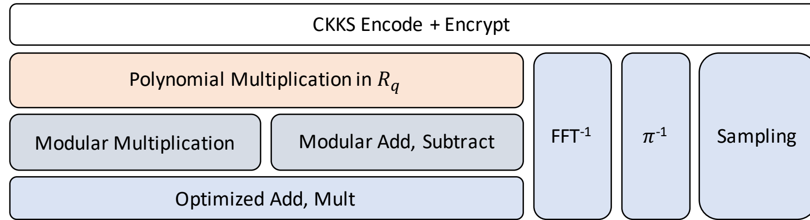

Figure 2: SEAL-Embedded library architecture.

size than a secret key, and therefore imposes a much larger memory requirement on the device. Nevertheless, our system is able to provide memory and computationally efficient implementations for public key CKKS encryption as well, often with the same or similar RAM requirements as the secret key implementation for the same parameter set.

One subtlety that exists even with all the above considerations is the possibility that timing variation on the device could reveal information about the underlying secret data. For example, variations in sample time for sampling an error polynomial on device, or from operations such as modular multiplication or addition including error or secret key coefficients, may manifest as detectable time variations further downstream in the end-to-end flow. These variations may be particularly observable in a streaming case, where a device is computing and sending encryptions continuously. It may therefore be possible for a network or cloud attacker to learn some information about the secret values from this timing variation alone. To defend against this threat, we implement all secret-dependent computations in a constant-time manner.[2] This mitigation can also reduce the ability of a co-located device application from learning secret values as well and therefore ensures a better separation between applications on the device. Since device data leakage from other channels (e.g. power side-channels) does not transfer to a detectable side channel over the network, we do not provide protection from power analysis on the device itself.

Finally, we note that the post-quantum nature of HE additionally provides some robustness against quantum attacks. In particular, attacks mounted by a quantum-time adversary on HE ciphertexts would not be able to access the underlying private data. For our implementation, we leave quantum attacks out of scope. However, if quantum protection is desired, several existing libraries [wol, pqc] offer solutions for quantum-secure encryption of messages across an untrusted channel (such as a public network).

3.2 Library Overview

In this section, we describe the structure of our library, including some of our optimizations for performance and memory management. Figure 2 gives a visualization of our library as a layered architecture. We describe each component below, leaving the bulk of optimization details to Section 4.

The base layer of the SEAL-Embedded library consists of low-level arithmetic operations. These include optimized addition and multiplication operations for both 32-bit and 64-bit unsigned integer data types. For ease of portability, SEAL-Embedded is written entirely in C, but includes inline assembly versions of critical operations for further performance gains for ARM architecture targets. Additionally, SEAL-Embedded contains optimized, constant-time variants of Barrett reduction [Bar87] and modular integer arithmetic. We further include a fast variant of modular multiplication that can be used when one operand is known ahead of time. These algorithms are given in more detail in Section 4.1.

[2] Functions in our library may still experience variations in time (e.g. due to differences in caching between runs). Our constant-time code simply ensures that this variation will not be related to a difference in the *value* of secret data.

At the layer above the base layer, SEAL-Embedded implements the negacyclic NTT to accelerate polynomial multiplications. For encryption, SEAL-Embedded contains functions for sampling from uniform, ternary, and centered binomial distributions, using random bytes generated by an extendable-output function (e.g. SHAKE-256) from a random seed generated by an RNG accessible through system APIs or hardware peripherals. We give further details about our choice of sampling algorithms in Section 4.2. For encoding, we implement the inverse of the canonical embedding map using a variant of inverse FFT. SEAL-Embedded offers trade-offs between computational efficiency and storage cost for users to select based on their needs. We discuss these trade-offs and optimizations in Sections 4.3–4.6.

At the top-most layer, SEAL-Embedded contains functions for CKKS encoding and encryption operations. We provide users with preset parameters for encryption. This allows us to omit several unnecessary functions from the library (e.g. generating prime moduli based on the input size, finding the inverse of n in a particular ring), thus enabling a smaller code size and faster run-time execution. However, it is possible for a user to configure the library to use custom parameters as well, so long as they provide the necessary precomputed values. The encoding and encryption functions string the functions in the underlying layers together to convert an input value vector into a CKKS ciphertext. The following section describes the encode-and-encrypt procedure as used in SEAL-Embedded.

3.3 Logical Flow

The process that our library uses to generate a CKKS ciphertext can be logically broken up into an encoding and encryption procedure. First, a user application passes the data it wants to encrypt to our library in the form of an input vector of (either real or complex) floating-point values, along with a chosen scale Δ. During encoding, SEAL-Embedded first applies the π^{-1} projection on these values, essentially performing a reordering of the inputs. Next, SEAL-Embedded applies the σ^{-1} map to reordered values via a variant of inverse FFT over the complex field (described in Section 2.6). The output, once multiplied by the chosen scale Δ and rounded to the nearest integer, is the CKKS plaintext.

Following the encoding procedure, the resulting CKKS plaintext is fed into an encryption sub-module, which is comprised of an RLWE encryption of zero followed by an addition of the CKKS encoded plaintext. In the symmetric case, a uniformly random polynomial in R_{q_i} is sampled and stored as the second polynomial of the ciphertext, and an error polynomial is sampled from the error distribution (set as the binomial distribution in SEAL-Embedded) and reduced modulo q_i. The random polynomial is then multiplied with the secret key and added to the reduced error polynomial. The output of this step is the output of the RLWE encryption of zero: $c_0 = -a \cdot s + e$, $c_1 = a$. Finally, to turn the encryption of zero into a ciphertext encrypting the requested values vector, the plaintext m is added to the c_0 term of the RLWE encryption.

The asymmetric case proceeds similarly to the above, with a few key differences. A public key with two polynomial components is used in place of the uniform random polynomial, while a sampled uniform ternary polynomial is used in place of the secret key. Additionally, two error polynomials are sampled from the chosen error distribution. The resulting RLWE ciphertext is constructed as: $c_0 = u \cdot pk_0 + e_0$, $c_1 = u \cdot pk_1 + e_1$. As in the symmetric case, the plaintext m is added to the c_0 term of the RLWE encryption.

If the full-RNS version of the encode-and-encrypt procedure is used, the above procedures occur with respect to each small modulus q_i. This process is illustrated in Figure 3 for both symmetric and asymmetric encryption. The logical flow described above is useful for understanding the underlying process of the encode-and-encrypt procedure. In SEAL-Embedded, however, we choose to order the operations in a slightly different manner in order to enable optimal performance and memory usage. We describe this re-ordering in more detail in Section 4.3.

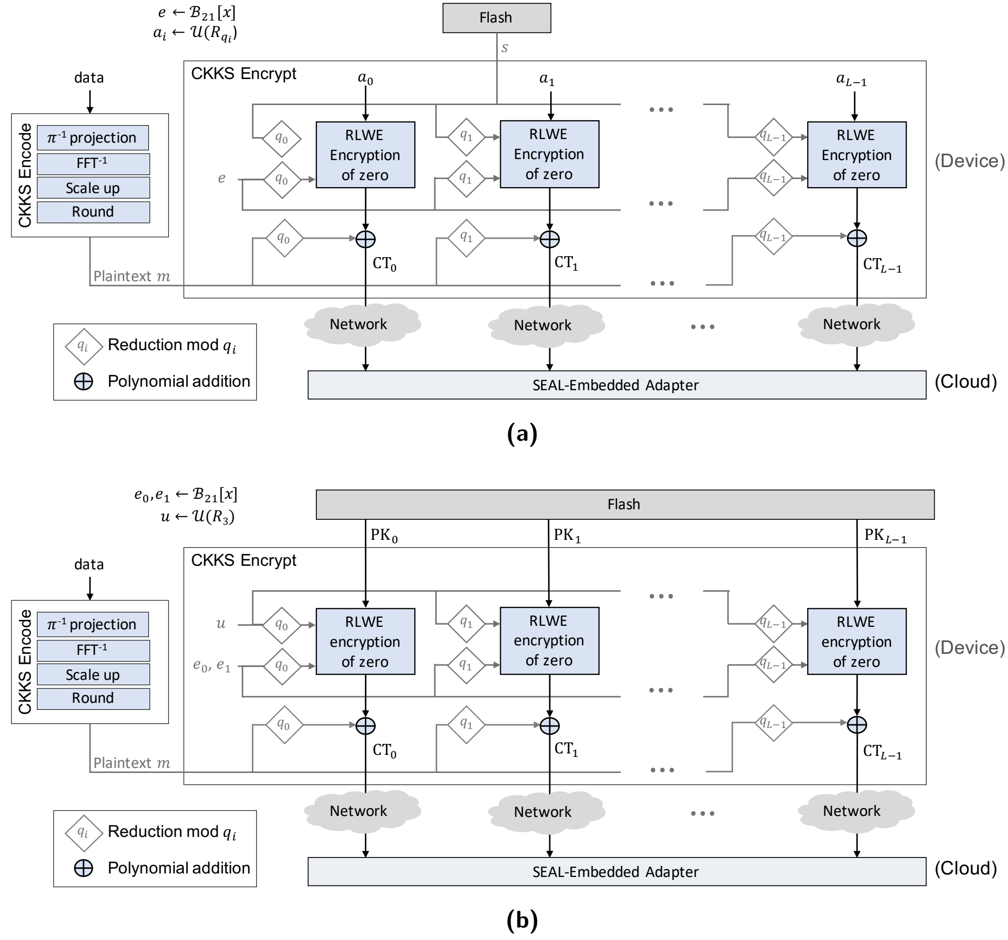

Figure 3: Logical flow of the CKKS encode-and-encrypt procedure used by SEAL-Embedded for symmetric (a) and asymmetric (b) encryption. Each CT_i denotes a two-component ciphertext, while each PK_i denotes a two-component public key. A CKKS ciphertext is generated by adding a CKKS plaintext to an RLWE encryption of zero. Note that in our work, the plaintext m is first added to error e (e_0 for asymmetric encryption) to reduce memory consumption and increase performance. (see Section 4.3).

4 Implementation Details and Optimizations

4.1 Low-Level Arithmetic

SEAL-Embedded relies on the efficiency of low-level modular multiplication and addition of integers. For our implementation, we choose CKKS parameters that use primes at most 30-bits in size to be compatible with the small RAM size and 32-bit architecture type of most embedded devices. As discussed later in Section 5.1, these parameters are capable of building real-life applications such as linear inference. We implement all modular arithmetic functions in C. For ARM architectures, we also provide inline assembly versions of the modular arithmetic functions using the `UMULL` instruction to produce a 64-bit product of two 32-bit unsigned integers.[3]

Additionally, we implement two versions of integer modular multiplication. The first

[3] `UMULL` is supported on ARM v6-T2 and above architectures.

version contains standard Barrett reduction with both 32-bit and 64-bit wide multiplications: Given a modulus $t < 2^{30}$, a precomputed value $u = \lfloor 2^{64}/t \rfloor$, and integers $x, y < t$, the reduction computes $x \cdot y - \lfloor \frac{x \cdot y \cdot u}{2^{64}} \rfloor \cdot t$ as either $[x \cdot y]_t$ or $[x \cdot y]_t + t$, then possibly subtracts a t (in a constant-time manner) to get the final reduced result. The second version is a much faster variant of the former and can be used for instances where one of the operands x or y is known ahead of time. In this version, we precompute an auxiliary integer $y' = \lfloor y \cdot 2^{32}/t \rfloor$. Then, we calculate $x \cdot y - \lfloor \frac{x \cdot y'}{2^{32}} \rfloor \cdot t$ as either $[x \cdot y]_t$ or $[x \cdot y]_t + t$ (again reducing a final t if necessary).[4] This second version only needs the 32 most-significant bits of the product when computing $x \cdot y'$, and therefore only requires 32-bit multiplication. The second version is especially applicable to the modular multiplication of input values with twiddle factors during the NTT, since the twiddle factors are known before runtime. We rely on both of the above versions of modular multiplication during calculation of the NTT, and refer to the implementation of the second version as the "fast" NTT configuration.

4.2 Random Sampling

SEAL-Embedded samples error polynomials from a centered binomial distribution, the same distribution as widely adopted by Module-LWE and RLWE-based cryptosystems [SAB+19, PAA+19, DKRV19]. The original worst-case to average-case reductions for LWE [Reg05] and RLWE [LPR10] apply to (rounded) continuous Gaussian distributions and extend to discrete Gaussian distributions. However, the implementation of efficient and constant-time Gaussian sampling is a challenging problem [MW17][KRR+18]. By contrast, sampling from a centered binomial distribution $\mathcal{B}_k$ can be performed efficiently and in constant-time by calculating the difference of Hamming weights of two random bit streams of length k. There is no known attack that takes advantage of the shape of this error distribution, assuming its standard deviation is sufficiently large. To align with the HE security standard [ACC+18], which lists secure parameters for distributions with a standard deviation of 3.2, we use a centered binomial distribution with a standard deviation of $\sqrt{21/2} \approx 3.24$. This choice does not reduce security levels and only increases encryption noise by a negligible amount.

Additionally, SEAL-Embedded implements two uniform sampling modules for symmetric and asymmetric encryption. Symmetric encryption requires the second polynomial of a ciphertext to be sampled uniformly from R_q. Since both the CRT and NTT are bijective, we sample this polynomial uniformly from R_{q_i} directly in the NTT form for each q_i in the RNS base. Our implementation uses rejection sampling to obtain a random value less than the maximum multiple of q_i representable by the integer data type, and applies constant-time Barrett reduction modulo q_i to the output. Asymmetric encryption requires a polynomial to be sampled uniformly from R_3 (i.e., n coefficients sampled uniformly from $\{-1, 0, 1\}$). Similar to the symmetric case, we implement this functionality using rejection sampling and a constant-time implementation of modulo 3 reduction.

All three of the aforementioned distributions require a source of random bytes. We obtain these random bytes by first generating a 64-byte random seed (the same size of seed as used in Microsoft SEAL) from the device's RNG. This seed is then extended by an extendable-output function (XOF) to fill a variable length buffer with cryptographically secure pseudo-random bytes. Using an XOF allows SEAL-Embedded to replace half of a symmetrically encrypted ciphertext by the seed to reduce communication cost. Further, since the overhead of random sampling is dominated by the XOF, using an XOF rather than a platform-specific RNG allows for a fairer comparison between different platforms. For our XOF, we use the assembly implementation of SHAKE-256 given in PQM4 [KRSS][5].

[4] According to [Har14], this algorithm should be attributed to the unpublished work [Fär05].

[5] SEAL-Embedded also provides the C version of SHAKE-256 as implemented in [SEA20] as a fallback.

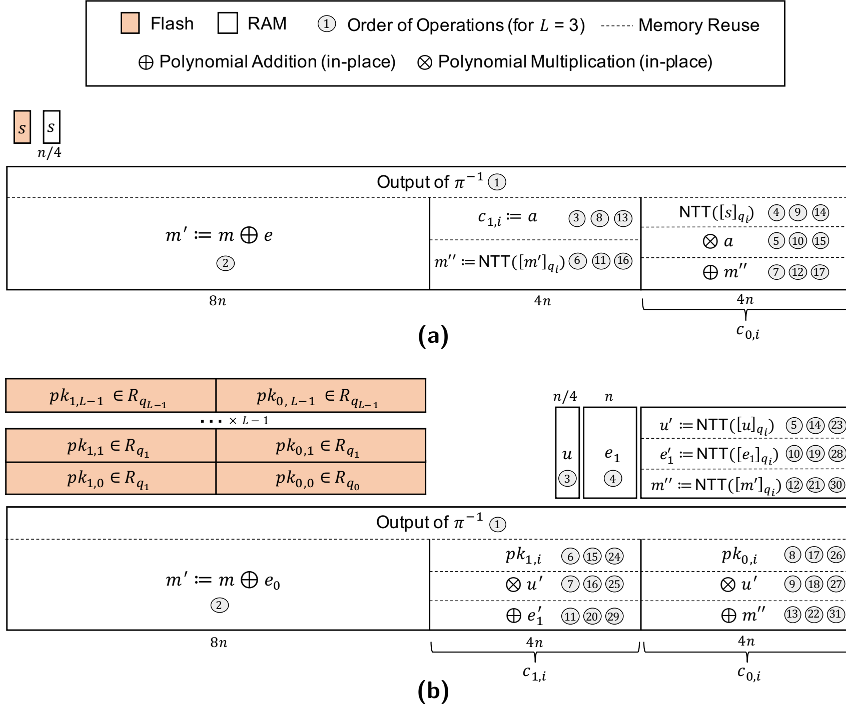

Figure 4: Memory requirements, order of operations, and memory re-use strategy for SEAL-Embedded symmetric (a) and asymmetric (b) encode-and-encrypt procedure configured for high memory efficiency. Block sizes are given in terms of bytes (e.g. m' occupies $8n =$ 32 KB for $n = 4096$), and dashed lines denote memory reuse. Polynomial operators with no left-hand component operate on the previous object in the particular memory block.

4.3 Memory Management

In order to maintain a small form factor, low price point, and high energy efficiency, embedded devices are often uniquely memory constrained when compared to other computational platforms. This constraint makes running memory-intensive workloads like homomorphic encryption a challenge on embedded devices. Moreover, any embedded encryption library must leave enough RAM free for user applications to operate and collect the data that the user desires to encrypt. Thus, in addition to the module-specific optimizations described in previous sections, we employ multiple techniques to further optimize memory usage in SEAL-Embedded, including RNS partitioning, data type compression, and a novel memory pool allocation strategy. We consider these optimizations together as part of a comprehensive memory re-use strategy, described in more detail below.

RNS Partitioning The first major way in which SEAL-Embedded overcomes memory limitations is by operating with respect to a single RNS prime at any given point. As mentioned in Section 2.4, the isomorphism established by the CRT preserves arithmetic in $R_{q_0 q_1 \cdots q_{L-1}}$ with arithmetic in subrings R_{q_i} that can be performed independently. SEAL-Embedded exploits this property by performing encryption in one subring R_{q_i} at a time, allocating just enough memory to carry out operations for a single prime component. After sending the ciphertext elements to an external server for a particular prime, the device can begin the encryption process with respect to the next RNS prime. This method is in direct contrast to currently available HE libraries such as Microsoft SEAL, which often allocate enough memory for all prime components of a polynomial at once.

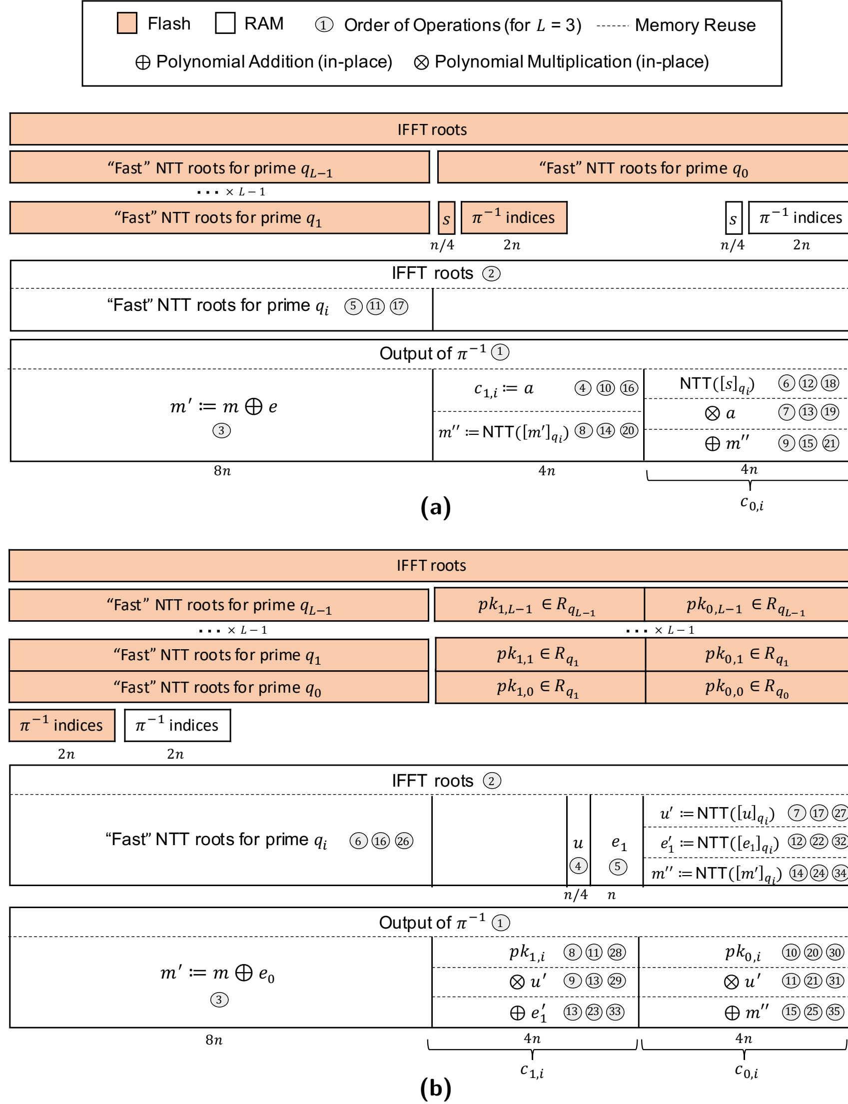

Figure 5: Memory requirements, order of operations, and memory re-use strategy for SEAL-Embedded symmetric (a) and asymmetric (b) encode-and-encrypt procedure configured for high performance. Block sizes are given in terms of bytes (e.g. m' occupies $8n = 32$ KB for $n = 4096$), and dashed lines denote memory reuse. Polynomial operators with no left-hand component operate on the previous object in the particular memory block.

Data Type Compression The second technique SEAL-Embedded employs to reduce memory consumption is memory compression of certain polynomials. We leverage the ternary nature of the secret key polynomial s (or polynomial u, in the asymmetric case) and store its values using just 2 bits per coefficient. SEAL-Embedded also applies compression to the output of error sampling. Since each error sample is an integer in the range $[-21, 21]$, each sample can be represented in just 6 bits. As a compromise between memory efficiency and fast performance from a regular access pattern, we compress the error polynomials to occupy 8-bits of storage per coefficient. When either the ternary or error polynomials are required for arithmetic operation with another polynomial, we read its coefficients and reduce modular q_i (a larger representation) as needed, one coefficient at a time.

Memory Pool The third significant way SEAL-Embedded overcomes memory limitations is via the use of a memory pool. For both symmetric and asymmetric cases, we carefully determined the minimum amount of memory required for efficient operation of the library. SEAL-Embedded uses these sizes to allocate all the memory it needs (for a particular configuration of the library) up front. This one-time allocation has the advantages of providing a higher level of memory access safety and availability throughout the lifetime of the library, ensuring low memory usage by preventing fragmentation, and reducing the number of calls to the costly memory allocator. However, the use of a manually sized memory pool comes with its own challenges; in particular, we need to determine the precise re-use pattern for the memory in the pool, described in more detail below.

Efficient Memory Re-use Strategy Combining the above three strategies led us to carefully optimize the particular order of SEAL-Embedded library operations to ensure the lowest possible memory usage while still enabling high performance. Our strategy is best explained through the following example of how an encoded message $m \in \mathbb{Z}^n$ is passed to encryption operations in R_{q_i}. The naive approach taken by almost all HE libraries is as follows: given the output of encoding $m \in \mathbb{Z}^n$ and a freshly-sampled error vector $e \in \mathcal{B}_{21}^n$, reduce e and m modulo q_i to get e_i and m_i, respectively; perform NTT in R_{q_i} on e_i to generate an encryption of zero; perform NTT in R_{q_i} on m_i and add the result to the encryption of zero. Note that this requires storage for m, e, m_i, and e_i in memory ($17n$ or $33n$ bytes with or without RNS partitioning, respectively, for 3 primes). In our design, by contrast, we sample e coefficient-by-coefficient and add each coefficient to m in-place without modulo reduction, such that m (or $m + e$) occupies the same buffer throughout the full encryption. Using this method, we do not need to allocate additional storage for sampling the error e, and instead proceed with the modular reduction and NTT in R_{q_i} of $m + e$ with respect to RNS partitioning. As a result, our method has the advantage of needing fewer NTTs than the naive case and requires only $12n$ bytes to store $m + e$ and its NTT form. This example demonstrates how we consider all three of the aforementioned memory management strategies to ensure a memory-optimal implementation. Complete outlines of our re-use strategy for low memory and high performance configurations of our library are given by the numbered operations in Figures 4 and 5.

Notably, our largest memory requirement comes from the encoding procedure, which at a minimum requires space for $16n$ bytes. Our implementation ensures that most of the memory used by the encoding procedure is re-used by encryption, so that encryption requires zero or only a small amount of extra memory from this initial allocation. Additionally, we allow for multiple allocation strategies for a user to choose from, depending on the requirements of their device and deployment. An example configuration of memory using our library, targeted for low-RAM utilization while still achieving high performance, is given in Figure 4 for symmetric and asymmetric encryption. As another example, for the case where an IoT device is able to support slightly larger memory usage, our library can be also configured to use additional RAM and/or flash data storage to achieve better performance (shown in Figure 5 for symmetric and asymmetric encryption). These options, which are described in more detail in the following sections, allow our library to be used across a variety of devices and scenarios.

4.4 Optimizations and Trade-offs for the IFFT

As discussed in Section 2.6, the σ^{-1} projection in the CKKS encoding procedure can be performed by evaluating an inverse FFT over the complex field using double-precision floating arithmetic. SEAL-Embedded implements an efficient version of this algorithm (see: Appendix A) using the techniques given in [LN16]. While the input to the IFFT in SEAL-Embedded is a vector of n *real single*-precision floating point values, intermediate results are a vector of n *complex double*-precision floating point values. Therefore, the

Table 1: Configurations and data memory requirements in bytes for IFFT roots, NTT roots, and π^{-1} projection in SEAL-Embedded.

Module	Configuration	RAM (B)	Flash (B)	Description
IFFT	compute on-the-fly	0	0	memory efficient
	load	$16n$	$16n$	high performance
NTT	compute on-the-fly	0	0	memory efficient
	compute one-shot	$4n$	0	balanced for memory
	load	$4n$	$4n$	balanced for performance
	load fast	$8n$	$8n$	high performance
π^{-1}	compute on-the-fly	0	0	memory efficient
	compute persistent	$2n$	0	balanced for memory
	load	$2n$	$2n$	balanced for performance
	load persistent	$2n$	$2n$	high performance

minimum memory requirement of the IFFT module is large (e.g. 64K bytes for $n = 4096$). The output of the IFFT, however, is a vector of n *real double*-precision floating point values, because the input values are conjugate pairs generated by the π^{-1} map. We can therefore compress the IFFT output to occupy only half the storage of intermediate results and reuse the other half as "free" memory in following procedures (e.g. encryption).

To aid memory conservation for devices where memory is particularly constrained, SEAL-Embedded can calculate the powers of a $2n$-th primitive root of unity in $\mathbb{C}$ (in short, roots) that are required for the IFFT "on-the-fly" case so that roots require zero RAM or flash data storage. For less constrained devices, we offer another option to store precomputed roots in flash and load them into RAM during the encode-and-encrypt procedure. A naive implementation of the "load" would require an additional n *complex double-precision* elements of RAM consumption. However, we can again mitigate the effect of this requirement through use of our memory pool, as shown in Figure 5. These two options are summarized in Table 1. We note that the high performance option can save significant processing time, as it requires much less double-precision arithmetic ($O(n)$ trigonometric sin/cos functions) than the "on-the-fly" case.

4.5 Optimizations and Trade-offs for the NTT

As discussed in Section 2.5, we used the NTT algorithm described in [LN16] (given in Algorithm 1 in Appendix A) for our NTT implementation, using the modular multiplication algorithms mentioned in Section 4.1. Similar to the IFFT, we implement multiple versions of the NTT (see: Table 1) that either compute roots or access precomputed roots to offer a trade-off between storage requirements and computational efficiency. Each of the NTT algorithms request or generate each power of the primitive root once (specifically, the `bit_reverse`(k)-th power in the k-th request, where $k \in \{1, 2, \ldots, n-1\}$). We additionally implement a faster NTT algorithm (see: "load fast" below) that requires twice the amount of memory to store powers of a $2n$-th primitive root of unity in $\mathbb{Z}_{q_i}$.

In the "compute on-the-fly" version, when a certain root power is requested, we compute its value by exponentiating the primitive root modulo q_i. For pre-selected RNS primes and degree n, we precompute $2n$-th primitive root of unity for each prime. For the k-th request, we compute the `bit_reverse`(k)-th power using the square-and-multiply algorithm in the reverse direction (i.e., from least to most significant bit), resulting in an overal complexity of $O(n \log \log n)$ integer modular multiplications for obtaining the full set of NTT roots. This configuration of the NTT is most useful for a low-memory implementation, since it requires no additional RAM or flash data storage.

In the "compute one-shot" configuration, we compute all root powers together and store them in RAM for the lifetime of encryption with each prime. This requires $4n$

bytes of RAM for requesting all the roots. However, it requires less computation than the "compute on-the-fly" case, since the roots only need to be calculated once per prime. We compute powers of the roots in increasing order, with n integer modular multiplications and n bit-reverse operations, so that they are stored in bit-reversed order for coalesced memory accesses during the NTT computation. In the "load" flavor, all of the roots are precomputed and stored in flash ahead of deployment. The NTTs for each prime are performed together, before which the roots for the target prime are loaded to RAM, and after which the roots for next prime are loaded into the same RAM buffer (overwriting the previous roots). For accessing the NTT roots, this version requires $4n$ bytes RAM, and n bit-reverse operations. However, it requires no integer modular multiplications, and can therefore be much faster than the above two configurations.

Finally, we also include a high performance "load fast" version of the NTT. In this version, each root is coupled with a precomputed auxiliary integer (see: Section 4.1) that enables fast integer modular multiplication with the NTT inputs. First, the roots (and their corresponding auxiliary integers) are pre-loaded into flash. Then, similar to the above cases, the NTTs for each prime are performed together, before which the roots for this prime are loaded to RAM, and after which the roots for the next prime are loaded and overwrite the same buffer. For obtaining the NTT roots, this requires $8n$ bytes RAM and n bit-reverse operations. Multiplications with the roots are faster in this NTT version than in others. Additionally, the NTT values can be reduced modulo q_i in a faster "lazy" way since its multiplication algorithm can tolerate inputs larger than q_i. As a result, we can trade $O(n \log n)$ reductions from $[0, 2q_i - 1]$ in the inner loop for just $O(n)$ reductions from $[0, 4q_i - 1]$ at the end of the computation.

4.6 Optimizations and Trade-offs for the π^{-1} Projection

As discussed in Section 2.3, the input to the encoding procedure (an element of $\mathbb{C}^{n/2}$) is mapped to the subring $\mathbb{H}$ of $\mathbb{C}^n$. Each coefficient in the input vector of length $n/2$ and its conjugation are sent to two calculated destinations in the output vector of length n. Since the IFFT assumes a bit-reversed ordering of the input, our index calculation operation outputs indices in this bit-reversed order. More precisely, the i-th input coefficient is sent to indices $j = \texttt{bit_reverse}(([5^i]_{2n} - 1)/2)$ and $n - 1 - j$ in the output. Our library implements four versions of this projection: "compute on-the-fly", "compute persistent", "load", and "load persistent" (summarized in Table 1).

The "compute on-the-fly" and "compute persistent" versions of the projection compute indices using the same strategies used by the "compute on-the-fly" and "compute one-shot" versions of the NTT (described in Section 4.5), respectively. The main difference between these projection versions is that the "compute persistent" case stores calculated indices in RAM for the lifetime of a program. Indices can instead be precomputed and stored in flash to save computation; the "load" version loads precomputed indices to RAM every time an encoding is performed (and overwrites the index buffer with other data after encoding), while the "load persistent" version loads indices into RAM for the lifetime of the program.

5 Experimental Results

In order to demonstrate the practicality of our library and evaluate its performance, we implement an example encryption flow on two embedded microcontrollers: an Azure Sphere MT3620 with ARM Cortex-A7, and the Nordic nRF52840 with ARM Cortex-M4. We give the specifications for each in Table 2. For all experiments, we compile our library with `-O3` optimizations using gcc 8.2 (which accompanies the Azure Sphere SDK) and ARM gcc 9.2.1 for the Azure Sphere and Nordic nRF52840, respectively. As shown in later sections, the Cortex-A7 and Cortex-M4 produce dramatically different runtimes of

Table 2: Cortex-A7 (MT3620) and Cortex-M4 (nRF52840) specifications

Processor	OS	Clock Speed	Cache (KB)	RAM	Flash
Cortex-A7	Linux	494 MHz	L1: 64/32 I/D, L2: 256	256 KB*	1 MB RO + 64 KB R/W
Cortex-M4	none	64 MHz	none	256 KB	1 MB

* The Azure Sphere MT3620 reserves a large portion of RAM for the OS, leaving only 256 KB remaining for each individual user application.

SEAL-Embedded function calls, which can be attributed to differences in clock frequencies, caching mechanisms, and RAM/flash speeds between the platforms.

5.1 Choosing Parameters

The memory requirement of SEAL-Embedded is linear with respect to the degree n. After numerous experiments, we found that the largest value of n that fits in 256 KB of RAM for our implementation is 4096 (or 8192 for our extremely memory-efficient library configuration). According to the Homomorphic Encryption security standard [ACC+18], for $n = 4096$, the product of all prime moduli $q_0 q_1 \cdots q_{L-1}$ must not exceed 109 bits for 128-bit classical security. Furthermore, Microsoft SEAL requires a prime to be reserved for the purposes of noise reduction. This prime does not need to be added during encryption, but is automatically added during key generation for evaluation keys. Therefore, an encryption with 3 primes on SEAL-Embedded corresponds to an evaluation with 4 primes in Microsoft SEAL. We limit moduli to 3 primes with 30 or fewer bits, leaving at least 19 bits of room for Microsoft SEAL to add a fourth modulus.

Using the above parameters, it is possible to use the CKKS scheme to build real-life applications with sufficient accuracy. For example, one can build an application for privacy-preserving inference of encrypted features using a trained linear model as follows: First, build an inference service application with Microsoft SEAL using four 25-bit primes. Then, on a client device, use SEAL-Embedded to encode a vector of real-valued features with scale $\Delta = 2^{25}$ and encrypt the result with three 25-bit primes. Send the resulting ciphertext to the server, which then performs an inner product of the encrypted features with a vector of plaintext weights. The server additionally performs a “rescale” operation on the resulting ciphertext to prevent exponential growth of precision in the underlying values, consuming one prime modulus in the process. The final ciphertext is left with two 25-bit primes and is sent to the client for decryption. The decrypted messages (again, a vector of real numbers) should roughly preserve 25-bit accuracy for the integral parts of the messages and 10-bit accuracy for the fractional parts (both of which can be adjusted by changing Δ and the sizes of the primes). In [KSK+18], the authors show that this resulting accuracy was sufficient for logistic regression training.

Comparison with Existing HE Libraries Several existing libraries also provide an implementation of CKKS homomorphic encryption; however, these libraries target full-fledged CPUs and cannot not be run on embedded devices. To demonstrate this, we create test programs that perform CKKS encoding and encryption of 2048 single-precision values with the same or smaller parameters as in our experiments ($n = 4096$ and 3 primes with 30 or fewer bits) using three popular open-source HE libraries: Microsoft SEAL [SEA20] v3.6.1, HElib [HEl] v1.3.1, and PALISADE [PAL] v1.10.6. We compile all tests with clang-10.0.0 with `-O3` optimizations on Ubuntu 20.04. We focus on the memory requirements of these libraries since this requirement is fundamental to the operation of embedded devices and is the main barrier to the ability of the libraries to be ported to the embedded domain. To ensure a fair comparison against our library, we load parameters, keys, and any pre-

Table 3: Memory usage for CKKS encoding and encryption of 2048 single-precision values for $n = 4096$ and 3 primes with 30 or fewer bits for various open-source HE libraries.

Library	Valgrind (KB)	RSS (KB)	VM (KB)
Microsoft SEAL v3.6.1	4,982	8,992	12,156
HElib v1.3.1	n/a*	14,416	19,276
PALISADE v1.10.6	2,359	14,840	2,372,060

* Valgrind was unable to measure HElib due to unrecognised instructions.

computed values from memory rather than generating them at runtime. We measured the peak stack and heap allocation using Valgrind v3.16.1 [NS07], the resident set size (RSS), which indicates the portion of memory occupied by a process that is held in main memory, and virtual memory (VM), which includes the memory usage of any shared libraries, as listed in Table 3. These results show that the tested libraries require orders of magnitude more memory than 256 KB, and therefore are not suitable for embedded devices.

5.2 SEAL-Embedded Modules

We report the overhead of independent modules in SEAL-Embedded in Table 4 (see Table 1 and Section 4.2 for descriptions of the various modes). `CKKS.Encode` is executed only once irrespective of the number of primes in the encode-and-encrypt workflow. The vast majority of the encoding module's runtime is consumed by the IFFT. The remaining overhead includes evaluating π^{-1}, multiplying by the scale Δ, and rounding to the nearest integer, all of which are insignificant compared to the IFFT computation runtime.

`CKKS.SymEncrypt` / `CKKS.AsymEncrypt` is performed once for each prime modulus. The majority of overhead for both symmetric and asymmetric modules are caused by application of the NTT and random sampling. In symmetric mode, encryption includes $2 \times L$ NTTs, L polynomial samples from uniform distribution over R_{q_i}, and 1 polynomial sample from $\mathcal{B}_{21}[x]$, as well as an $O(n)$ amount of integer modular multiplications, additions, and negations. In asymmetric mode, encryption includes $3 \times L$ NTTs, L polynomial samples from ternary distribution R_3, and 2 polynomial samples from $\mathcal{B}_{21}[x]$, plus an $O(n)$ amount of integer modular multiplications, additions, and negations. Note that the runtimes of the NTT module provided in Table 4 are given with respect to a single prime.

For random sampling from all distributions, we generate a 64-byte random seed using a call to `/dev/urandom` through the Azure Sphere's Linux subsystem on the Cortex-A7 and the RNG peripheral on nRF52480. We then use the SHAKE-256 XOF to expand the seed to generate random bytes, which dominates the overhead of random sampling. For the uniform distribution, we fill a pre-allocated buffer with n 32-bit coefficients using a single call to SHAKE-256. For the other distributions, we allocate a 96-byte buffer on the stack and fill it as needed using SHAKE-256 until all coefficients of the polynomial have been sampled. We implement both the uniform and ternary distributions using rejection sampling, and therefore make extra calls to SHAKE-256 when rejection occurs.

5.3 Encoding and Encryption

In Table 5, we list the total memory and flash data requirements and performance overhead of encoding and encryption using $n = 4096$ and 3 primes, for both symmetric and asymmetric modes. In particular, we provide evaluation results for the most memory efficient and highest performance configurations, as well as a configuration that provides a balance between the two. For all configurations, we find that encoding and encryption is $10\times$ faster on the Cortex-A7 than the Cortex-M4. This can be attributed to the $7.72\times$ difference in frequency between the cores as well as the presence of caches on the Cortex-A7.

Table 4: Performance and data memory requirements of SEAL-Embedded modules on the Cortex-M4 and Cortex-A7 for $n = 4096$.

Module	Mode/Type	RAM	FLASH	Cortex-M4	Cortex-A7
IFFT	compute on-the-fly	0 KB	0 KB	751,745 µs	7,524 µs
	load	64 KB	64 KB	408,063 µs	6,040 µs
NTT	compute on-the-fly	0 KB	0 KB	65,134 µs	15,870 µs
	compute one-shot	16 KB	0 KB	28,240 µs	4,684 µs
	load	16 KB	48 KB	25,495 µs	4,976 µs
	load fast	32 KB	96 KB	15,336 µs	3,221 µs
Sample	uniform	16 KB	0 KB	34,294 µs	3,888 µs
	ternary	1 KB	0 KB	24,474 µs	1,749 µs
	CBD	4 KB	0 KB	77,155 µs	9,367 µs

Table 5: SEAL-Embedded runtimes and data memory requirements for CKKS encode-and-encrypt on the Cortex-M4 and Cortex-A7 for $n = 4096$ and 3 primes. Configurations for the IFFT, NTT, and π^{-1} (see Table 1) are as follows: "memory efficient" uses "compute on-the-fly" for all; "balanced" uses "load" for all; "high performance" uses "load", "load fast", and "load persistent", respectively.

Key Type	Configuration	RAM	FLASH	Cortex-M4	Cortex-A7
symmetric	memory efficient	65 KB	1 KB	1,348,602 µs	131,691 µs
	balanced	129 KB	121 KB	735,826 µs	68,959 µs
	high performance	137 KB	169 KB	680,790 µs	60,016 µs
asymmetric	memory efficient	85 KB	96 KB	1,550,116 µs	188,681 µs
	balanced	128 KB	216 KB	824,403 µs	91,267 µs
	high performance	136 KB	264 KB	737,342 µs	76,778 µs

6 Conclusion

In this work, we present SEAL-Embedded, the first Homomorphic Encryption library for the Internet of Things. SEAL-Embedded implements several optimizations that enable efficient operation of CKKS encoding and encryption, including optimized low-level arithmetic, fast multiplication, and finely tuned memory management. We also present an adapter component for the SEAL-Embedded library, which enables compatibility with the Microsoft SEAL HE library for an end-to-end solution for homomorphic encryption on embedded devices. To demonstrate the efficiency of our library, we implement an example encode-and-encrypt procedure on two different IoT platforms. Our results show that our solution is able to provide efficient performance and memory consumption for homomorphic encryption on embedded devices. This work is released open-source at https://github.com/microsoft/SEAL-Embedded.

References

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R,M}LWE schemes. *IACR TCHES*, 2020(3):336–357, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8593.

[ACC+18] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard.

Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[AGR+16] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 191–219. Springer, Heidelberg, December 2016.

[Bar87] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 311–323. Springer, Heidelberg, August 1987.

[BGGJ20] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. Chimera: Combining Ring-LWE-based fully homomorphic encryption schemes. *Journal of Mathematical Cryptology*, 14(1):316–338, 2020.

[BGP+20] Marcelo Blatt, Alexander Gusev, Yuriy Polyakov, Kurt Rohloff, and Vinod Vaikuntanathan. Optimized homomorphic encryption solution for secure genome-wide association studies. *BMC Medical Genomics*, 13(7):1–13, 2020.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012.

[CDKS20] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient homomorphic conversion between (ring) LWE ciphertexts. Cryptology ePrint Archive, Report 2020/015, 2020. https://eprint.iacr.org/2020/015.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020.

[CHK+19] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In Carlos Cid and Michael J. Jacobson Jr:, editors, *SAC 2018*, volume 11349 of *LNCS*, pages 347–368. Springer, Heidelberg, August 2019.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 409–437. Springer, Heidelberg, December 2017.

[DKRV19] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 617–640. Springer, Heidelberg, April 2015.

[DSES14] Yarkin Doröz, Aria Shahverdi, Thomas Eisenbarth, and Berk Sunar. Toward practical homomorphic evaluation of block ciphers using prince. In Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith, editors, *FC 2014*

Workshops, volume 8438 of *LNCS*, pages 208–220. Springer, Heidelberg, March 2014.

[Fär05] Tommy Färnqvist. Number theory meets cache locality – efficient implementation of a small prime FFT for the GNU multiple precision arithmetic library. Technical report, 2005.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. http://eprint.iacr.org/2012/144.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 850–867. Springer, Heidelberg, August 2012.

[Har14] David Harvey. Faster arithmetic for number-theoretic transforms. *Journal of Symbolic Computation*, 60:113–119, 2014.

[HEA] HEAAN. https://github.com/snucrypto/HEAAN. CryptoLab Inc.

[HEl] HElib. https://github.com/homenc/HElib. IBM Corp.

[JKLS18] Xiaoqian Jiang, Miran Kim, Kristin E. Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1209–1222. ACM Press, October 2018.

[KRR+18] Angshuman Karmakar, Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. Constant-time discrete Gaussian sampling. *IEEE Transactions on Computers*, 67(11):1561–1571, November 2018.

[KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[KSK+18] Andrey Kim, Yongsoo Song, Miran Kim, Keewoo Lee, and Jung Hee Cheon. Logistic regression model training based on the approximate homomorphic encryption. *BMC Medical Genomics*, 11(4):83, 2018.

[KSLM20] Miran Kim, Yongsoo Song, Baiyu Li, and Daniele Micciancio. Semi-parallel logistic regression for GWAS on encrypted data. *BMC Medical Genomics*, 13(7):1–13, 2020.

[KSW+18] Miran Kim, Yongsoo Song, Shuang Wang, Yuhou Xia, and Xiaoqian Jiang. Secure logistic regression based on homomorphic encryption: Design and evaluation. *JMIR Med Inform*, 6(2):e19, April 2018.

[LM20] Baiyu Li and Daniele Micciancio. On the security of homomorphic encryption on approximate numbers. Cryptology ePrint Archive, Report 2020/1533, 2020. https://eprint.iacr.org/2020/1533.

[LN14] Tancrède Lepoint and Michael Naehrig. A comparison of the homomorphic encryption schemes FV and YASHE. In David Pointcheval and Damien Vergnaud, editors, *AFRICACRYPT 14*, volume 8469 of *LNCS*, pages 318–335. Springer, Heidelberg, May 2014.

[LN16] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In Sara Foresti and Giuseppe Persiano, editors, *CANS 16*, volume 10052 of *LNCS*, pages 124–139. Springer, Heidelberg, November 2016.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Heidelberg, May / June 2010.

[MKV20] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. Time-memory trade-off in Toom-Cook multiplication. *IACR TCHES*, 2020(2):222–244, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8550.

[MW17] Daniele Micciancio and Michael Walter. Gaussian sampling over the integers: Efficient, generic, constant-time. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 455–485. Springer, Heidelberg, August 2017.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[PAA+19] Thomas Poppelmann, Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Peter Schwabe, Douglas Stebila, Martin R. Albrecht, Emmanuela Orsini, Valery Osheter, Kenneth G. Paterson, Guy Peer, and Nigel P. Smart. NewHope. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[PAL] PALISADE. https://gitlab.com/palisade. New Jersey Institute of Technology (NJIT).

[pqc] libpqcrypto. https://libpqcrypto.org/. PQCRYPTO project.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.

[SAB+19] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[SEA20] Microsoft SEAL (release 3.5). https://github.com/Microsoft/SEAL, April 2020. Microsoft Research, Redmond, WA.

[wol] wolfSSL/wolfssl. https://github.com/wolfSSL/wolfssl. wolfSSL Inc.

[ZCH+19] Zhenfei Zhang, Cong Chen, Jeffrey Hoffstein, William Whyte, John M. Schanck, Andreas Hulsing, Joost Rijneveld, Peter Schwabe, and Oussama Danba. NTRUEncrypt. Technical report, National Institute of Standards and Technology, 2019. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

A NTT and IFFT Algorithms

Algorithm 1 Inplace NTT for Negacyclic Convolution

Input: A vector $a \in \mathbb{Z}_q^n$ with dimension n a power of two, a prime modulus q congruent to 1 modulo $2n$, and ψ a table of precomputed 0 to $(n-1)$-th powers of a primitive $2n$-th root of unity in bit-reversed order.
Output: a in NTT domain

1: **function** NTT(a, ψ, n, q)
2: **for** $(m = 1;\ m < n;\ m = 2m)$ **do**
3: $t = n/2/m$
4: **for** $(i = 0;\ i < m;\ i{+}{+})$ **do**
5: $S = \psi[m + i]$
6: **for** $(j = 2it;\ j < (2i + 1)t;\ j{+}{+})$ **do**
7: $L = a[j]$
8: $R = a[j + t] \cdot S \bmod q$
9: $a[j] = (L + R) \bmod q$
10: $a[j + t] = L - R + q \bmod q$
11: **return** a

Algorithm 2 Inverse Fast Fourier Transform (IFFT) for the Canonical Embedding

Input: A vector $a \in \mathbb{C}^n$ with dimension n a power of two and ψ^{-1} a table of precomputed 0 to $(n-1)$-th inverse powers of a primitive $2n$-th root of unity in bit-reversed order.
Output: a in IFFT domain

1: **function** IFFT$(a,\ \psi^{-1},\ n)$
2: **for** $(m = n/2;\ m > 0,\ m = m/2)$ **do**
3: $t = n/2/m$
4: **for** $(i = 0;\ i < m;\ i{+}{+})$ **do**
5: $S = \psi^{-1}[m + i]$
6: **for** $(j = 2it;\ j < (2i + 1)t;\ j{+}{+})$ **do**
7: $L = a[j]$
8: $R = a[j + t]$
9: $a[j] = L + R$
10: $a[j + t] = (L - R) \cdot S$
11: **for** $(i = 0;\ i < n,\ i{+}{+})$ **do**
12: $a[i] = a[i] \cdot n^{-1}$
13: **return** a