This section briefly discusses the fundamentals of the CKKS HE scheme described by Cheon et al. [CKKS], and how it is implemented in this project. Also, to provide an investigation into specialising HE schemes, a bespoke CKKS implementation, called MeKKS, is detailed.

# CKKS Primitives

Fundamentally, the CKKS scheme encrypts plaintext polynomials into ciphertext polynomials. Addition and multiplication operations can then be performed on the data, introducing uncertainty. Ciphertext polynomials can be decrypted to retrieve approximations of plaintext polynomials, with the accuracy depending on the number of operations performed.

To encrypt vectors of real values, they must first be encoded as polynomials in the ring $\mathcal{R} = \mathbb{Z}[X] / (X^N + 1)$, where $N$ is a power of 2. During encoding, the real values are rounded, so precision is preserved by multiplying by a \textit{scaling factor}, $\Delta$. Once the vector has been encoded, it can be encrypted into a \textit{pair} of polynomials: the ciphertext.

Consider two vectors $\vec{v}$ and $\vec{w}$. Before encoding, the vectors are scaled to $\Delta \vec{v}$ and $\Delta \vec{w}$. They can then be encrypted and multiplied to produce ciphertext equivalents of $\Delta^2 (\vec{v} \bigodot \vec{w})$. This implies that a sequence of multiplications will continuously increase the scale factor. To overcome this, CKKS introduces a \textit{rescaling} procedure, which can be understood as dividing the ciphertext by $\Delta$.

However, rescaling cannot be repeatedly applied to allow unlimited multiplications. CKKS is a levelled HE scheme, so each ciphertext resides at a discrete level. Each level has a coefficient modulus $q$ that dictates a ciphertext’s coefficients must be in $\mathbb{Z}\_q$. When first encrypted, a polynomial exists in the maximum level, $L$, with coefficient modulus $Q = q\_0 \dot \Delta^L$, for a \textit{base modulus} $q\_0$. When a ciphertext is rescaled, the coefficient modulus is divided by $\Delta$, reducing it to $q\_0 \cdot \Delta^{L-1}$. Hence, the ciphertext is lowered to the next level. If a ciphertext reaches level 0, no further multiplications can be applied to preserve the encrypted values. To ensure decryption produces correct results, a polynomial’s coefficients cannot exceed $q\_0$. Therefore, practical implementations will use a $q\_0$ much larger than $\Delta$.

## Encoding and Decoding

The plaintext message space is the \textit{cyclotomic polynomial ring} $\mathcal{R} = \mathbb{Z}[X] / \Phi\_M(X)$, for the $M$-th cyclotomic polynomial, where $M$ is a power of two. Encoding describes mapping a complex vector to an element in $\mathcal{R}$, and decoding is the process of reversing it.

Another way of defining decoding is as a mapping of an element $r \in \mathcal{R}$ to a vector in $\mathbb{C}^N$ using the embedding $\sigma : \frac{\mathbb{R}[X]}{X^N + 1} \rightarrow \mathbb{C}^N$. This is applied to each root of $\Phi\_M(X)$ through the evaluation of $r$ as

EQUATION

where $\xi = e^{\frac{2 \pi i}{M}}$ is a primitive $M$-th root of unity.

However, encoding requires the input to be \textit{scaled} and \textit{rounded}, and decoding requires half of the complex vector to be \textit{discarded} to produce a one-to-one mapping.

Hence, the CKKS encoding function is given by Equation \ref{eq:encode}.

## Operations

A complete list of the operations supported by CKKS and their definitions is included in Appendix \ref{app:operations}.

## Rescaling

The rescaling function introduced above is defined by equation \ref{eq:rescale}.

EQUATION

Rescaling truncates some of the least-significant bits of a ciphertext by dividing by a power of the scaling factor. Practically, this is performed after every multiplication to scale $\Delta^2$ down to $\Delta$.

## Rotations

An advantage of RLWE based HE schemes over others is the ability to \textit{rotate} ciphertext slots. Given a vector $\vec{V} = [v\_0, v\_1, v\_2, \ldots, v\_n]$ and an offset $d$. A rotation would cyclically shift each element by $d$ indices\footnote{For example, $\vec{v} = [1, 2, 3, 4, 5]$ and $d = 2$ would produce $\vec{v}’ = [4, 5, 1, 2, 3]$.}. This is achieved by performing a \textit{Galois automorphism} on the ciphertext using standard results from \textit{Galois theory}. However, these operations are the most expensive operations offered by RLWE-based schemes [ROTATIONBAD].

## RNS Optimisation

SEAL utilises a \textit{*residue number system*} (RNS) to overcome the slow computations caused by very large polynomial coefficients requiring arbitrary precision arithmetic [RNS]. The Chinese Remainder Theorem (CRT) is exploited to decompose the coefficients in $\mathbb{Z}\_q$ into $n$ smaller ones: $\mathbb{Z}\_{p\_1}, \ldots, \mathbb{Z}\_{p\_n}$. To do this, the CRT uses a \textit{*moduli switching chain*} $p\_1, \ldots, p\_n$ such that $\prod\_i p\_i = q$ and each $p\_i$ is a prime number requiring fewer than 64-bits to store. Hardware limitations mean it is faster to compute the $n$ separate multiplications in $p\_1, \ldots, p\_n$ than it is to compute a single multiplication in $q$. The results of the separate multiplications can be combined using the isomorphism between $\mathbb{Z}\_q$ and $\mathbb{Z}\_{p\_1}, \ldots, \mathbb{Z}\_{p\_n}$.

Unfortunately, as a result of the RNS optimisation, $q\_l = q\_0 \Delta^l$ cannot always be maintained due to the difficulty of finding every $q\_l$ as the product of $n$ primes. To overcome this, SEAL instead defines $q\_l = q\_0 \prod\_{i=1}^l p\_i$ where $q\_0$ must be prime. Where the original CKKS scheme divides by $\Delta$ to rescale, SEAL’s implementation divides by $p\_l$. Consequently, the scaling factor is not exactly equal to $\Delta$. In practice, the scaling factor must be manually reset to $\Delta$ after each rescaling.

# MeKKS

As an extension to the project, the CKKS scheme was reimplemented from first principles to create MeKKS. This allowed the investigation of a HE scheme specialised for this application through simplification of data structures and removal of unnecessary functionality.

## Limitations

The primary limitation of MeKKS was the language used for implementation. Traditionally, encryption schemes are written in C or C++ to expose low-level functionality. Consequently, the more computationally expensive operations, such as large prime number multiplications, would run much quicker than in Python.

Also, in contrast to CKKS, MeKKS is severely limited by optimisations that have been applied. This extension began by focussing on improving HE understanding, so the foundations of CKKS were implemented. However, CKKS has been significantly optimised since it was originally proposed. For example, using residue number systems, bootstrapping extensions, and reduced approximation error [RNS, BOOTSTRAPPINGHEAAN, RAE]. The project's time constraints meant that defining a clear implementation endpoint was essential for scheduling. Therefore, MeKKS is significantly less optimised than SEAL’s CKKS, limiting performance expectations.

## Implementation

The first iteration of the MeKKS implementation began with the scheme presented by Cheon et al. in 2016 [CKKS]. This allowed all of the functionality described in §\ref{sec:CKKS} to be implemented following the SEAL API, enabling easy integration into the core project.

However, this implementation’s performance meant it was infeasible for recording results. Therefore, the bootstrapping procedure from the next HEAAN iteration was implemented [BootstrappingHEAAN]. Bootstrapping takes advantage of the \textit{approximate computation} characteristic of HEAAN to evaluate the decryption formula approximately so a ciphertext can be obtained in a large modulus. Hence, an approximation of the \textit{modular reduction} formula was implemented to enable efficient evaluation using standard operations – where the error induced by the approximation is small enough to maintain precision.

Using the observation that the modular reduction function, $F(t) = [t]\_q$, is the identity near zero and periodic in $q$, bootstrapping uses a trigonometric function to approximate the function when $t = \langle \texttt{Ciphertext}, \texttt{SecretKey} \rangle$ is close to a multiple of $q$ (the ciphertext modulus). Specifically, using the \textit{sine} function given by Equation \ref{eq:bootstrapping}.

EQUATION

when $F(\langle \texttt{Ciphertext}, \texttt{SecretKey} \rangle) < \epsilon \cdot q$.

The Taylor polynomial can be used to approximate the trigonometric function to enable evaluation in the HE domain. The input, $t$, is bounded by $K \cdot q$ for some constant $K = O(\lambda)$, where $\lambda$ is the security parameter. The degree of the Taylor polynomial should be at least $O(K \cdot q)$ to ensure the error term is small enough on the interval $(-K \cdot q, K \cdot q)$. Cheon et al.\ proposed using the \textit{Paterson-Stockmeyer} method to reduce the calculation complexity [CHEON, PATERSON]. However, the complexity of recryption grows exponentially with the depth of the decryption circuit, which still has a substantial impact.

Figure \ref{fig:bootstrapping} provides a graphical representation of the bootstrapping approximation.

## Optimisations

The principal optimisations of MeKKS came through data representation. When integrating SEAL, a Python wrapper for a C++ implementation was used. Consequently, the application was forced to handle large C++ objects, abstracted through the wrapper. Python struggled to manipulate these objects efficiently, specifically when serialising data for transmission. However, since MeKKS was written in Python, object attributes and functionality were exposed, providing much more efficient manipulation.

Another optimisation came through the specialisation of the library. The project schedule ensured MeKKS was only implemented once the application’s core had been complete. Therefore, the required functionality had been finalised, so only necessary operations were implemented for MeKKS. This reduced the size of objects and removed any unused computation, making execution more efficient. For example, the most expensive operations offered by CKKS are rotations. However, since the application does not use them, they were not included in MeKKS.