We present and demo of Lattigo, a multiparty homomorphic encryption library in Go. After a brief introduction of the origin and history of the library, we dive into the most relevant technical aspects that differentiate Lattigo from other existing libraries. From the cryptographic research perspective, we describe our realization of the keyswitch and CKKS bootstrapping operations. We also present our approach to multiparty homomorphic encryption and its importance for Lattigo use-cases. From the software perspective, we elaborate on the choice of the Go language and the benefits it brings to application developers who use the library. We then present performance benchmarks and the main use-case applications the library had so far. The last part of the presentation comprises a tutorial on how to use Lattigo to build a ‘toy’ use-case: a privacy-preserving web-application for scheduling meetings.

## 1 INTRODUCTION

Homomorphic Encryption (HE) techniques are becoming increasingly popular. This is reflected in a growing number of cryptographic libraries that implement efficient instantiations, and in the current process for standardization of HE [3]. Traditionally, HE schemes are used in a two-party setting comprising a data-holder party $P_1$ that sends its encrypted input data $x_P$ to an external party $P_2$, which can compute any polynomial function $f(x_P)$ over the scheme’s plaintext space, and then sends the encrypted result back to party $P_1$ for decryption. In the passive-adversary model, this simple protocol can achieve secure two-party computations.

The aforementioned setting can be extended to $N$ parties through the use of Multiparty Homomorphic Encryption (MHE) techniques such as multi-key-HE (MKHE) [9, 18] and threshold-HE (Th-HE) [4, 19]. In such schemes, the involved parties collectively (hence, interactively) enforce the access control to the data by distributing the scheme’s decryption circuit. Mouchet et al. proposed a threshold version of BFV and showed that its use as a secure-multiparty-computation (MPC) solution is, for several generic circuits, faster and has less communication overhead than LSSS-based MPC in the same adversary model [19]. Thus, there is a great interest in building concrete MPC systems that can employ MHE schemes.

Such systems, by nature, are highly interactive, concurrent and cross-platform. For this reason, implementing them may represent a significant investment in terms of time and effort when using C++, which most of the state-of-the-art HE libraries are using. More recent languages, such as Go [1], greatly reduce this effort, notably by featuring built-in concurrency primitives, extensive standard libraries and comprehensive toolchains for building, testing and analyzing code. In this demo, we present the Lattigo library, a Go module for R-LWE-based multiparty homomorphic encryption.

## 2 LIBRARY OVERVIEW

Lattigo is a Go module that contains the packages listed in Table 1.

### Genesis

The development of Lattigo started in March 2019 as a part of our research on multiparty homomorphic encryption (MHE) and secure multiparty computation. In addition to the scientific interest in being able to quickly integrate our research results into a code-base for their empirical evaluation, we saw an opportunity to benefit the community by bringing HE to a new programming language: Go. Our group currently uses Go for the implementation of several research projects. As these systems transitioned from proof-of-concept implementations to real-world prototypes deployed in operational settings, the need for a cryptographic layer supporting MHE became essential.

### Scope and interface principles

For each scheme, the corresponding package implements the cryptographic objects and the local operations on these objects. These local operations are defined as exported Go interface types (e.g., bfv.Encryptor) for which implementations are provided as methods (e.g., Encrypt(*)) of context-specific objects (e.g., skEncryptor, pkEncryptor) that
encapsulate the cryptographic parameters, temporary buffers and pre-computations. As of version v2.1.0, Lattigo provides a single-threaded implementation of its API and all types assume single-threaded use. Therefore, the API user controls the concurrency aspects of its application.

Support for Multiparty Access Structures. At the time of writing, the dbfv and dckks packages implement the $N$-out-of-$N$ Threshold access structure of Mouchet et al. [19] (we elaborate on this scheme in Section 2.2).

2.1 Cryptographic Optimizations and Features

We summarize the features in Lattigo that are relevant from a cryptographic-research standpoint.

Standalone Arithmetic Layer. The library exposes most of its polynomial arithmetic layer in the lattigo/ring sub-package. This package is implemented in pure Go and features a wide range of low-level optimized algorithms, with a minimal, unexported use of the unsafe package (that enables pointer arithmetic) and without any dependency on external numerical libraries. This includes Montgomery-form arithmetic, ring operations, Number Theoretic Transforms (NTT), evaluation of automorphisms, RNS bases extensions and scaling, and sampling of Gaussian, uniform and ternary distributions. Hence, it can be used to build and evaluate other R-LWE based FHE schemes and primitives.

Generalized Keyswitch Procedure. For both the BFV and CKKS schemes, Lattigo uses a generalization of the keyswitch procedure proposed by Han and Ki [15], which lets the user specify the norm of the decomposition-basis $P$ used during the key-switching. Hence, the parameters can be represented as a triplet $\{d, L, \alpha\}$, where $d$ is the ring degree, $L$ is the number of ciphertext moduli (prime factors of $Q$) and $\alpha$ is the number of special primes for the key-switching (prime factors of $P$). Even though it introduces an additional (yet optional) parameter, we observed that giving the user the ability to tune the trade-off (indeed, the size of $QP$ is capped by the security parameter) between homomorphic-capacity and keyswitch complexity results in great throughput gains. We compared the homomorphic throughput of the keyswitch procedure along with the size of the public switch-key for several values of $\alpha$ using the parameters $\{2^{15}, 16 - \alpha, \alpha\}$, i.e. for a fixed modulus size $QP$, the number of primes between $Q$ and $P$ varies. Figure 1 shows that, by increasing $\alpha$ to 4, we get a 2x increase in throughput and a 5x decrease in the key-size. This shows that, in terms of throughput, the loss in homomorphic capacity is more than compensated by the run-time reduction.

We also further optimized the keyswitch-key format and keyswitch algorithm for the evaluation of automorphisms such as rotations, as proposed by Bossuat et al. [7].

Novel BFV Quantization. Even in its RNS variant, [5, 14], the BFV homomorphic multiplication is an expensive operation because it requires the use of a secondary and temporary basis [12]. Lattigo takes a novel approach to this operation, by adapting the RNS-friendly quantization techniques proposed in original full-RNS variant of the CKKS scheme [10]. See Section 4 for benchmark comparisons.

CKKS Bootstrapping. The Lattigo library comprises an implementation of the CKKS bootstrapping from Bossuat et al. [7]. Hence, Lattigo is the second library to feature an open-source implementation of a bootstrapping circuit for the CKKS scheme and the first one to make such implementation available for the Full-RNS variant of the scheme. Compared to the current state-of-the-art, the procedure is both more efficient and more precise (as shown in Figure 2), and it does not require the use of sparse secret keys.

Homomorphic Polynomial Evaluation. The lattigo/ckks package provides a scale-invariant and depth-optimal polynomial evaluation algorithm, for both the standard and the Chebyshev bases. It allows the user to provide the clear-text polynomial coefficient and a desired output scale, and it recursively back-propagates it to ensure that all rescalings in the evaluation are exact (as described in more details by Bossuat et al. [7]).

2.2 Multiparty Homomorphic Encryption

MHE has a great potential as a generic secure multiparty computation (MPC) solution thanks to its low communication requirements and versatility. However, whereas traditional Linear Shamir Secret Sharing (LSSS)-based generic MPC protocols are implemented in several, well-established libraries, MHE-based solutions have been implemented mainly for specific computations. One of the main
As the negative log of the mean error across all the slots.

Hence, apart from the above functionalities purposes of the Lattigo library is to support the development of MHE-based MPC protocols.

Lattigo implements the scheme of Mouchet et al. [19] for both BFV and CKKS. We summarize its protocols from a high-level and refer the reader to [19] for the scheme details. Let \( P_1, ..., P_N \) be \( N \) parties holding their respective secret keys \( sk_1, ..., sk_N \) and let \( sk = \sum_{i=1}^{N} sk_i \). This scheme comprises the following multiparty protocols:

- **EncKeyGen**: **Collective encryption-key generation.** It generates a public encryption-key \( pk \) for the secret-key \( sk \), in a single round.
- **RelinKeyGen**: **Relinearization-key generation.** It generates a public relinearization-key \( rlk \) for the secret-key \( sk \), in two rounds.
- **RotKeyGen**: **Rotation-key generation.** Given an integer \( k \), it generates a public rotation-key \( rot_k \) enabling homomorphic plaintext-slots rotation by \( k \), in a single round.
- **ColKeySwitch**: **Collective Key-switching.** Given a ciphertext \( ct \) and a target secret-key \( sk' \), it computes the re-encryption of \( ct \) from \( sk \) to \( sk' \), in a single round. A decryption protocol is obtained from the special case \( sk' = 0 \).
- **PubColKeySwitch**: **Collective Public-key-switching.** Given a ciphertext \( ct \) and a target public-key \( pk' \), it computes the re-encryption of \( ct \) from \( sk \) to \( pk' \), in a single round.

As for several threshold schemes, the multiparty scheme emulates a single-key setting and preserves the structure of the ciphertexts and keys. Hence, apart from the above functionalities (those that depend on the secret-key), the single-party scheme operations can be directly used in a multi-party setting (and are non-interactive).

The multiparty scheme of Mouchet et al. enables generic 2+2-rounds MPC protocols (illustrated in Figure 3) in the passive-adversary, dishonest-majority setting [19]. In the first 2 rounds, the parties run a PubKeyGen protocol, which is a parallel composition of EncKeyGen, RelinKeyGen and RotKeyGen, to generate the necessary public-key material. This can be done in an off-line phase and only once for a given set of parties and cryptographic parameters. In the next round, the parties provide their inputs, encrypted under the generated public-key. The evaluation of the circuit is done using the usual homomorphic operations for the scheme. The last round corresponds to the output phase: Depending on the setting, the parties use either the ColKeySwitch or the PubColKeySwitch to re-encrypt or decrypt the final result. An interesting feature of the MHE-MPC protocol is that its transcript is entirely public, so it does not require private channels between the parties. Hence, in addition to the traditional peer-to-peer system-model for MPC, the MHE-based protocols can work in cloud-assisted models for which the parties communicate solely with a central server.

Go is an ideal choice for implementing networked systems and web-services, thanks to its natural concurrency model, rich API and ease of deployment. Hence, building on these features using the Lattigo library makes developing HE-based privacy-preserving applications considerably easier.

### 3 SOFTWARE FEATURES

We provide an overview on the features that distinguish Lattigo from other homomorphic encryption libraries. Whereas most state-of-the-art HE libraries use C++, Lattigo is written in Go.

**The Go language.** The Go language was designed for multi-core, concurrent and networked systems, which makes it ideal for implementing multiparty computation. It features a minimal set of fundamental concepts and associated syntax, which makes learning Go easy. The Go run-time is highly efficient and Go programs can match and even outperform C++ programs, if the overhead of
garbage collection is taken into account. We found that, by implementing allocation-free API methods, this overhead is negligible.

The Go toolchain. As for most modern languages, Go provides a complete toolchain for building programs. In addition to the compiler, this toolchain comprises a dependency resolver and integrates unit-tests and benchmarks. This makes Lattigo easy to download, compile, test and benchmark.

To simply explore the library and run the examples programs, the easiest way is to clone the repository at github.com/ldsec/lattigo. From the library root directory, example programs can be run using the go run command. For example,

```
$ go run ./examples/dbfv/psi
```

runs the multiparty-BFV-based PSI example. Benchmarking and testing Lattigo is equally easy:

```
$ go test -run=X -bench=. ./...
```

(ckks tests only)

```
$ go test -run=X -bench=. ./ckks
```

(ckks benchmarks)

```
$ go test -run=X -bench=. /Encrypt ./Decrypt
```

(encrypt only)

The Go toolchain also makes it easy to import Lattigo as a dependency. From within the directory of another Go module, running

```
$ go test -run=X -bench=./Encrypt ./...
```

(ckks tests only)

```
$ go test -run=X -bench=. ./ckks
```

(ckks benchmarks)

```
$ go test -run=X -bench=. /Encrypt ./Decrypt
```

(encrypt only)

Installs the latest released version of Lattigo as a dependency and runs its unit tests.

The downsides of Go. The Go compiler, while being constantly improved, is not as mature as C++ compilers. We found that it does not optimize arithmetic and does not use SIMD or vectorized instructions when available. Hence we implemented several low-level optimizations, such as loop-unrolling and pointer arithmetic, to obtain performance figures comparable to C++. However, this complexity is not exposed to the user. The garbage collection introduces a slight overhead, which can however be reduced to negligible by writing allocation-free code.

## 4 Performance comparison

We provide performance benchmarks for the single-party and multiparty primitives implemented in Lattigo v2.1.0. We used SEAL v3.6 [21] as a baseline for the single-party schemes. All experiments were conducted single-threaded on an i5-6600k at 3.5 GHz with 32 GB of RAM running Windows 10. We used Go version 1.14.2 for building Lattigo and the MSVC++ compiler version 14.28 to compile the SEAL library and its examples.

### Parameters

We define the benchmarked parameters as the triplet \(\{d, L, \alpha\}\), where \(d\) is the ring degree, \(L\) is the number of ciphertext moduli (prime factors of \(Q\)) and \(\alpha\) is the number of special primes for the key-switching (prime factors of \(P\)). These are indeed the most relevant factors when comparing the library performance, as each individual modulus fits within one machine limb. Both Lattigo and SEAL propose several default parameter sets for 128-bit security (according to the standardization document [3]) and varying homomorphic capacity. However, SEAL does not yet support the use of multiple moduli in the extended-basis \(P\) (it enforces \(\alpha \leq 1\)), so the default parameters proposed by Lattigo cannot be directly compared. Hence, we performed our benchmarks with the default parameters of SEAL. We generated custom parameters for the ring degree \(d = 2^{16}\); these parameters use 31 moduli, for an

### Table 3: BFV Timings in \(\mu s\) for \(2^{10} \leq d \leq 2^{13}\).

<table>
<thead>
<tr>
<th>Op</th>
<th>(d = 2^{11}, L = 1)</th>
<th>(d = 2^{12}, L = 2)</th>
<th>(d = 2^{13}, L = 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encode</td>
<td>29</td>
<td>60</td>
<td>122</td>
</tr>
<tr>
<td>Decode</td>
<td>29</td>
<td>73</td>
<td>129</td>
</tr>
<tr>
<td>Encrypt</td>
<td>803</td>
<td>2085</td>
<td>5711</td>
</tr>
<tr>
<td>Decrypt</td>
<td>110</td>
<td>358</td>
<td>1374</td>
</tr>
<tr>
<td>Add</td>
<td>7</td>
<td>3</td>
<td>126</td>
</tr>
<tr>
<td>Mul-Pt</td>
<td>129</td>
<td>482</td>
<td>2084</td>
</tr>
<tr>
<td>Mul-Ct</td>
<td>1146</td>
<td>3721</td>
<td>14987</td>
</tr>
<tr>
<td>Square</td>
<td>816</td>
<td>2693</td>
<td>10918</td>
</tr>
<tr>
<td>KeySwitch</td>
<td>-</td>
<td>-</td>
<td>775</td>
</tr>
</tbody>
</table>

### Table 4: CKKS Timings in \(\mu s\) for \(2^{10} \leq d \leq 2^{13}\).

<table>
<thead>
<tr>
<th>Op</th>
<th>(d = 2^{11}, L = 1)</th>
<th>(d = 2^{12}, L = 2)</th>
<th>(d = 2^{13}, L = 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encode</td>
<td>112</td>
<td>305</td>
<td>854</td>
</tr>
<tr>
<td>Decode</td>
<td>63</td>
<td>345</td>
<td>1385</td>
</tr>
<tr>
<td>Encrypt</td>
<td>548</td>
<td>1816</td>
<td>5329</td>
</tr>
<tr>
<td>Decrypt</td>
<td>18</td>
<td>71</td>
<td>272</td>
</tr>
<tr>
<td>Add</td>
<td>7</td>
<td>28</td>
<td>124</td>
</tr>
<tr>
<td>Mul-Pt</td>
<td>14</td>
<td>52</td>
<td>210</td>
</tr>
<tr>
<td>Mul-Ct</td>
<td>45</td>
<td>187</td>
<td>795</td>
</tr>
<tr>
<td>Square</td>
<td>27</td>
<td>124</td>
<td>496</td>
</tr>
<tr>
<td>Rescale</td>
<td>-</td>
<td>203</td>
<td>861</td>
</tr>
<tr>
<td>KeySwitch</td>
<td>-</td>
<td>807</td>
<td>3927</td>
</tr>
</tbody>
</table>

### Table 5: BFV Timings in ms for \(2^{14} \leq d \leq 2^{16}\).

<table>
<thead>
<tr>
<th>Op</th>
<th>(d = 2^{14}, L = 8)</th>
<th>(d = 2^{15}, L = 15)</th>
<th>(d = 2^{16}, L = 31)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encode</td>
<td>0.2</td>
<td>0.5</td>
<td>1.1</td>
</tr>
<tr>
<td>Decrypt</td>
<td>18.5</td>
<td>65.4</td>
<td>253.5</td>
</tr>
<tr>
<td>Add</td>
<td>0.4</td>
<td>1.7</td>
<td>7.1</td>
</tr>
<tr>
<td>Mul-Pt</td>
<td>8.8</td>
<td>34.1</td>
<td>149.5</td>
</tr>
<tr>
<td>Mul-Ct</td>
<td>65.7</td>
<td>400.3</td>
<td>2822.6</td>
</tr>
<tr>
<td>Square</td>
<td>48.4</td>
<td>306.8</td>
<td>2185.1</td>
</tr>
<tr>
<td>KeySwitch</td>
<td>24.3</td>
<td>147.0</td>
<td>1183.8</td>
</tr>
</tbody>
</table>

### Table 6: CKKS Timings in ms for \(2^{14} \leq d \leq 2^{16}\).

<table>
<thead>
<tr>
<th>Op</th>
<th>(d = 2^{14}, L = 8)</th>
<th>(d = 2^{15}, L = 15)</th>
<th>(d = 2^{16}, L = 31)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encode</td>
<td>3.2</td>
<td>14.3</td>
<td>58.0</td>
</tr>
<tr>
<td>Decrypt</td>
<td>6.4</td>
<td>31.7</td>
<td>230.7</td>
</tr>
<tr>
<td>Add</td>
<td>1.1</td>
<td>4.7</td>
<td>19.2</td>
</tr>
<tr>
<td>Mul-Pt</td>
<td>0.8</td>
<td>3.1</td>
<td>13.2</td>
</tr>
<tr>
<td>Mul-Ct</td>
<td>3.1</td>
<td>12.0</td>
<td>49.2</td>
</tr>
<tr>
<td>Square</td>
<td>2.1</td>
<td>8.4</td>
<td>35.3</td>
</tr>
<tr>
<td>Rescale</td>
<td>3.6</td>
<td>14.6</td>
<td>64.2</td>
</tr>
<tr>
<td>KeySwitch</td>
<td>23.4</td>
<td>146.5</td>
<td>1178.5</td>
</tr>
</tbody>
</table>
equivalent $\log(QP)$ of 1782 bits. For all parameter sets, we used a number of plaintext slots $n = d$ for BFV and $n = d/2$ for CKKS.

**Results.** Tables 3, 4, 5, and 6 summarize the timings of local operations for BFV and CKKS in a single-key setting, along with the corresponding baseline-system timings. We observe that, within the scope of these benchmarks, it is possible to produce Go cryptographic code that matches the performance of C++.

The timings for the local operations for DBFV are presented in Table 7 (the timings for DCKKS would be identical). These timings reflect the per-party cost of generating its share in the protocol (Gen), the cost of aggregating two received shares (Agg) and the cost of computing the protocol output from its transcript (Out). Table 8 reports the size of one share, which corresponds to the total amount of data sent by one party in each protocol. For the RelinKeyGen protocol, the values represent the aggregation between the two rounds. Note that, thanks to the properties of the multiparty scheme, none of these values actually depend on the number of parties $N$. Indeed, the system-wide network load and number of calls to the share aggregation operations (Agg) grows with $N$ and depends on the system model and network topology. We refer the reader to [19] for an analysis of these costs in concrete instances of the MHE-based MPC protocol.

The good performance of Lattigo can be attributed to the efficiency of the package lattigo/ring, which heavily leverages on low-level Go-friendly optimizations (e.g. Montgomery and pointer arithmetic, lazy-reduction, loop unrolling) as well as scheme-specific high-level algorithmic optimization (e.g. a novel BFV quantization, operation-specific plaintext encoding).

### 5 APPLICATIONS

Lattigo has been successfully used in the implementation of complex application workflows involving both client-server applications and large-scale multiparty settings.

**Client-server applications.** A paradigmatic case of a secure service that works on encrypted sensitive client data was proposed in the 2019 iDash challenge [2], involving a problem of secure genotype imputation. Lattigo was used for developing one of the three winning solutions, implementing a multinomial logistic regression with CKKS-encrypted data, that performs a batch prediction (1,000 patients with 80,000 to-be-imputed variants each) in seconds, and has memory requirements and prediction accuracy comparable to clear-text state-of-the-art genotype imputation tools [17].

Lattigo was also used for implementing building blocks for MPC protocols, such as a passively-secure oblivious linear function evaluation (OLE) protocol [6]. This protocol generalizes oblivious transfer to linear functions, and its Lattigo implementation (on top of the ring package) is able to evaluate more than 1 million OLEs per second over the ring $\mathbb{Z}_m$, for a 120-bit $m$ on standard hardware.

**Large-scale multi-party applications.** The main use-case of Lattigo is the development of multi-party secure protocols where the input confidential data is partitioned among several entities. These entities impose an access structure on the computation results, by leveraging the MHE solution enabled by Lattigo. The achieved security guarantees are much stronger than traditional federated learning approaches, which leak intermediate computation results. Lattigo has been used for implementing distributed training and evaluation of several machine learning models, including generalized linear models [13] and feed-forward neural networks [20]. The systems built with Lattigo are capable of efficiently scaling up to thousands of parties and achieve a high training throughput, while closing the accuracy gap with respect to centralized clear-text systems. Examples of its performance include training a logistic regression model on a dataset of 1 million samples with 32 features distributed among 160 data providers in less than three minutes [13], and training a 3-layer neural network on the MNIST dataset with 784 features and 60,000 samples distributed among 10 parties in less than 2 hours.

### 6 DEMO

The final part of the presentation is a tutorial demonstrating the use of Lattigo for building a simple Doodle-like web-application for privacy-preserving scheduling. The code for this demo is available at https://github.com/ldsec/lattigo-polls-demo.

#### 6.1 Example Web-Application

In this application, we consider a web-service provider and $D$ clients willing to find an intersection in their availabilities, while revealing only this intersection to the creator of the poll. We assume that the clients will not cheat on their input and that the web-service provider is honest-but-curious (passive adversaries). For the sake of this demo, we assume that the creator of the poll does not collude...
with the web-service provider and we instantiate it in single-key setting. This assumption can be relaxed by using MHE.

The server is a Go program using the net/http package to serve the web-application. It implements different routes to create the poll, collect the answers and finally compute and serve the result to the poll creator.

The client is a web browser that makes requests to the server and presents the UI to the user. When loading the page, the client fetches and runs a Go executable compiled in WebAssembly (Wasm). This program exposes procedures that can be called through javascript, and is used by the browser client to call Lattigo functions.

The protocol. A client creates a new poll by generating a new key-triple (sk, pk, rkl) and sending a POST /creategpoll request containing pk and rkl. The participants can join this poll by retrieving the associated pk, encoding their availabilities as binary vector pt (0 meaning unavailable and 1 meaning available for this option) and sending a POST <poll ID>/availability containing their name and ct = Encrypt_{pk}(pt). Upon the poll closing by the creator, the server computes the product between all submitted ciphertexts, and serves the resulting ciphertext to the poll creator.

6.2 Hands-On Tutorial

During the tutorial, we will review the Go code for the server and client and their respective use of the Lattigo library. Thanks to Go’s rich API and simplicity, the whole application requires less than 300 lines of Go code. We will also show how the client program can be called by the client through Javascript. Finally, we will show a complete polling scenario, from the client (browser window) and server (terminal output) perspectives.

7 CONCLUSIONS & ON-GOING WORK

This demo presents the Lattigo library, a multiparty homomorphic encryption library written in Go. Lattigo greatly facilitates the development of new HE- and MHE-applications, by enabling the use of these primitives in a modern language: Go. By considerably reducing the development time of such applications, Lattigo can be a catalyst in both the cryptography research and the adoption of HE in real systems. Our ongoing work comprises:

- Fully-Threshold MHE. When the number of parties N is large, the risk of one party going offline for an indeterminate amount of time can become an issue. By relaxing the threshold to T-out-of-N (T < N), this risk can be mitigated. Full-threshold variants of BFV and CKKS are implemented in a development branch of Lattigo.

- Real-CKKS. Although the CKKS scheme encrypts complex numbers, most of its applications only use the real part of the plaintexts. Hence, only half of the homomorphic capacity is effectively used. Kim and Song proposed a variant of CKKS that encrypts d real numbers [16]. This scheme is implemented in a development branch of Lattigo.

Lattigo-Cloud/Lattigo-MP. Two generic network-layers implementing the MHE-based MPC protocol are currently in development. They provide the network and service layers for both the cloud-based (client+server) and peer-to-peer (client) settings.

ACKNOWLEDGMENTS

This work is partly funded by grant #2017-201 (DPPH) of the PHRT.

REFERENCES


[22] Microsoft Research, Redmond, WA.

[23] Christian Mouchet, Jean-Philippe Bossuat, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux