Lattigo: a Multiparty Homomorphic Encryption Library in Go

Christian Mouchet
christian.mouchet@epfl.ch
EPFL

Jean-Philippe Bossuat
jean-philippe.bossuat@epfl.ch
EPFL

Juan Troncoso-Pastoriza
juan.troncoso-pastoriza@epfl.ch
EPFL

Jean-Pierre Hubaux
jean-pierre.hubaux@epfl.ch
EPFL

ABSTRACT

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_1} to an external party P_2, which can compute any polynomial function $f(x_{P_1})$ 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.

WAHC'20, 15 Dec. 2020, Online
.

Table 1: The `github.com/ldsec/lattigo/v2` Go module

`lattigo/ring`	provides the RNS (Residue Number System) modular arithmetic over the ring $\mathbb{Z}_Q[X]/(X^d+1)$ with d a power of two. This includes: RNS basis extension, RNS division, number theoretic transform (NTT), and uniform, Gaussian and ternary sampling.
`lattigo/bfv`	implements the Full-RNS, scale-invariant scheme of Brakerski, Fan and Vercauteren (BFV) [5, 12, 14]; it supports $\mathbb{Z}_p^d$ arithmetic.
`lattigo/ckks`	implements the Full-RNS scheme of Cheon et al. [10, 11] (CKKS, a.k.a. HEAAN), that supports approximate arithmetic over $\mathbb{C}^{d/2}$. This package features a bootstrapping procedure.
`lattigo/dbfv`	implements the local operations for multiparty key-generation and key-switching functionalities for the BFV scheme [19].
`lattigo/dckks`	implements the local operations for multiparty key-generation and key-switching functionalities for the CKKS scheme [19].

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 applied 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 α 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 α 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 α to 4, we get a 2× increase in throughput and a 5× 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,

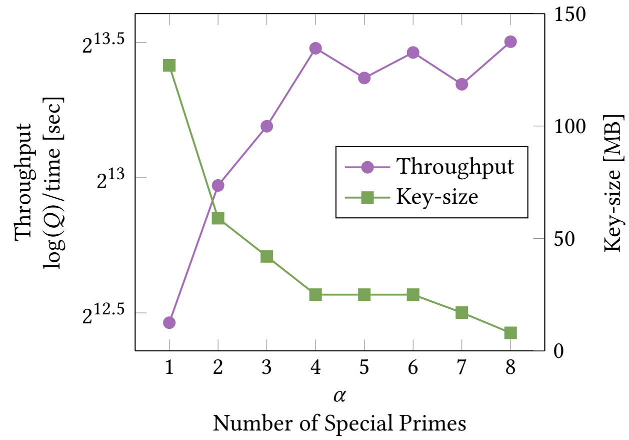

Figure 1: Comparison of the public key-switch operation throughput (in ciphertext-bits/sec.) and public switching-key size in Lattigo v2.1.0 for $d = 2^{15}$ and variable $1 \le \alpha \le 8$ and $L = 16 - \alpha$.

Table 2: CKKS Bootstrapping Parameters. d is the ring degree, h is the number of non-zero coefficients in the secret-key, $\log(Q)$ bit-size of the ciphertext modulus, $\log(P)$ is the bit-size of the key-switching decomposition-basis (the security is based on $\log(QP)$) and C the ciphertext modulus consumption by bootstrapping (in bits).

Set	d	h	$\log(Q)$	$\log(P)$	C
Best of [8]	2^{16}	64	1240	1240	1057
Best of [15]			1270	182	900
II		192	1248	305	743
III		2^{15}	1416	366	956

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

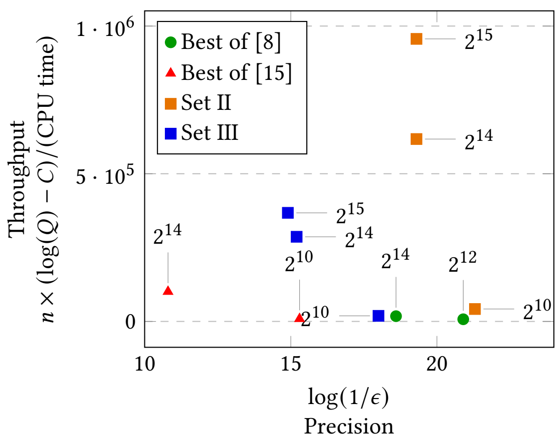

Figure 2: (From [7]) Comparison of the bootstrapping utility. See Table 2 for the parameters. We plot the results for our best performing parameter set against the state of the art. Nodes are labeled with n, the number of plaintext slots, $\log(Q) - C$ is the residual homomorphic capacity (in bits) after the bootstrapping, and the precision $\log(1/\epsilon)$ is defined as the negative log of the mean error across all the slots.

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 $\mathsf{sk}_1, ...\mathsf{sk}_N$ and let $\tilde{\mathsf{sk}} = \sum_{i=1}^{N} \mathsf{sk}_N$. This scheme comprises the following multiparty protocols:

EncKeyGen	**Collective encryption-key generation**. It generates a public encryption-key pk for the secret-key $\tilde{\mathsf{sk}}$, in a single round.
RelinKeyGen	**Relinearization-key generation**. It generates a public relinearization-key rlk for the secret-key $\tilde{\mathsf{sk}}$, 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 $\tilde{\mathsf{sk}}'$, it computes the re-encryption of ct from $\tilde{\mathsf{sk}}$ to $\tilde{\mathsf{sk}}'$, in a single round. A decryption protocol is obtained from the special case $\tilde{\mathsf{sk}}' = 0$.
PubColKeySwitch	**Collective Public-key-switching**. Given a ciphertext ct and a target public-key pk′, it computes the re-encryption of ct from $\tilde{\mathsf{sk}}$ 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

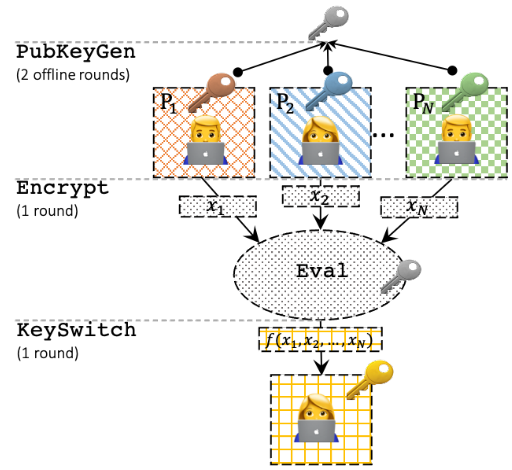

Figure 3: The MHE-based MPC protocol

(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 in 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

Table 3: BFV Timings in μs for $2^{10} \leq d \leq 2^{13}$.

Op	$d = 2^{11}, L = 1$		$d = 2^{12}, L = 2$		$d = 2^{13}, L = 4$	
	SEAL	Lattigo	SEAL	Lattigo	SEAL	Lattigo
Encode	29	26	60	55	122	123
Decode	29	30	73	56	129	112
Encrypt	803	226	2085	936	5711	2935
Decrypt	110	64	358	284	1374	1251
Add	7	3	28	11	126	46
Mul-Pt	129	90	482	380	2084	1652
Mul-Ct	1146	476	3721	2065	14987	9123
Square	816	368	2693	1569	10918	7234
KeySwitch	-	-	775	745	3933	3781

Table 4: CKKS Timings in μs for $2^{10} \leq d \leq 2^{13}$.

Op	$d = 2^{11}, L = 1$		$d = 2^{12}, L = 2$		$d = 2^{13}, L = 4$	
	SEAL	Lattigo	SEAL	Lattigo	SEAL	Lattigo
Encode	112	103	305	247	854	668
Decode	63	82	345	174	1385	382
Encrypt	548	392	1816	1115	5329	3680
Decrypt	18	6	71	27	272	108
Add	7	3	28	10	124	46
Mul-Pt	14	7	52	27	210	126
Mul-Ct	45	15	187	61	795	242
Square	27	13	124	54	496	224
Rescale	-	-	203	222	861	857
KeySwitch	-	-	807	731	3927	3619

Table 5: BFV Timings in ms for $2^{14} \leq d \leq 2^{16}$

Op	$d = 2^{14}, L = 8$		$d = 2^{15}, L = 15$		$d = 2^{16}, L = 31$	
	SEAL	Lattigo	SEAL	Lattigo	SEAL	Lattigo
Encode	0.2	0.2	0.5	0.5	1.1	1.2
Decode	0.2	0.2	0.6	0.6	1.2	1.3
Encrypt	18.5	10.5	65.4	39.2	253.5	153.0
Decrypt	5.6	5.3	23.5	22.4	115.7	95.6
Add	0.4	0.2	1.7	1.0	7.1	4.5
Mul-Pt	8.8	7.4	34.1	32.1	149.5	133.2
Mul-Ct	65.7	44.9	400.3	205.7	2822.6	1186.3
Square	48.4	35.0	306.8	155.8	2185.1	946.6
KeySwitch	24.3	24.1	147.0	154.5	1183.8	1235.5

Table 6: CKKS Timings in ms for $2^{14} \leq d \leq 2^{16}$

Op	$d = 2^{14}, L = 8$		$d = 2^{15}, L = 15$		$d = 2^{16}, L = 31$	
	SEAL	Lattigo	SEAL	Lattigo	SEAL	Lattigo
Encode	3.2	1.9	14.3	6.3	58.0	22.9
Decode	6.4	0.8	31.7	3.1	230.7	6.5
Encrypt	19.0	13.0	71.6	50.0	295.7	211.4
Decrypt	1.1	0.4	4.7	2.3	19.2	9.1
Add	0.4	0.2	1.7	0.9	7.2	4.5
Mul-Pt	0.8	0.5	3.1	2.2	13.2	9.1
Mul-Ct	3.1	1.1	12.0	4.6	49.2	19.4
Square	2.1	0.9	8.4	4.0	35.3	17.1
Rescale	3.6	3.4	14.6	13.8	64.2	65.7
KeySwitch	23.4	22.8	146.5	143.5	1178.5	1205.8

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 all tests)
$ go test ./ckks                               (ckks tests only)
$ go test -run=X -bench=. ./...                (run all benchmarks)
$ go test -run=X -bench=. ./ckks               (ckks bench only)
$ go test -run=X -bench=./Encrypt ./...        (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 get github.com/ldsec/lattigo/v2
$ go test github.com/ldsec/lattigo/v2/...
```

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 α 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 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).

Table 7: DBFV Timings [ms]: Total cost per party for generation (Gen), aggregation (Agg) and output (Out) local operations. Aggregated over the two rounds for the RelinKeyGen.

Op		$d = 2^{13}$	$d = 2^{14}$	$d = 2^{15}$
EncKeyGen	Gen	0.77	2.59	10.45
	Agg	0.02	0.08	0.32
	Out	0.09	0.34	1.28
RelinKeyGen	Gen	8.79	30.79	159.48
	Agg	0.25	1.12	6.18
	Out	0.36	1.81	9.72
RotKeyGen	Gen	2.46	8.73	45.24
	Agg	0.06	0.25	1.53
	Out	0.11	0.67	3.88
ColKeySwitch	Gen	1.15	4.43	19.31
	Agg	0.02	0.06	0.25
	Out	0.02	0.07	0.32
PubColKeySwitch	Gen	3.01	11.98	48.74
	Agg	0.03	0.12	0.55
	Out	0.03	0.12	0.68

Table 8: DBFV Share Sizes [MB]: Total amount sent per party. Aggregated over the two rounds for the RelinKeyGen.

Op	$d = 2^{13}$	$d = 2^{14}$	$d = 2^{15}$
EncKeyGen	0.26	1.05	3.93
RelinKeyGen	3.15	12.58	62.91
RotKeyGen	0.79	3.15	15.73
ColKeySwitch	0.2	0.79	3.15
PubColKeySwitch	0.39	1.57	6.29

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 ($\mathsf{sk}, \mathsf{pk}, \mathsf{rlk}$) and sending a `POST /createpoll` request containing pk and rlk). 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 $\mathsf{ct} = \mathsf{Encrypt}_{\mathsf{pk}}(\mathsf{pt})$. 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

[1] [n.d.]. The Go Programming Language. http://golang.org

[2] [n.d.]. IDASH PRIVACY & SECURITY WORKSHOP 2019 - secure genome analysis competition. http://www.humangenomeprivacy.org/2019/

[3] 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. 2018. *Homomorphic Encryption Security Standard.* Technical Report. HomomorphicEncryption.org, Toronto, Canada.

[4] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. 2012. Multiparty computation with low communication, computation and interaction via threshold FHE. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 483–501.

[5] Jean-Claude Bajard, Julien Eynard, M Anwar Hasan, and Vincent Zucca. 2016. A full RNS variant of FV like somewhat homomorphic encryption schemes. In *International Conference on Selected Areas in Cryptography*. Springer, 423–442.

[6] Carsten Baum, Daniel Escudero, Alberto Pedrouzo-Ulloa, Peter Scholl, and Juan Ramón Troncoso-Pastoriza. 2020. *Efficient Protocols for Oblivious Linear Function Evaluation from Ring-LWE.* 130–149. https://doi.org/10.1007/978-3-030-57990-6_7

[7] Jean-Philippe Bossuat, Christian Mouchet, Juan R. Troncoso-Pastoriza, and Jean Pierre Hubaux. 2019. Efficient Bootstrapping for ApproximateHomomorphic Encryption with Non-Sparse Keys. In *https://eprint.iacr.org/2020/1203.*

[8] Hao Chen, Ilaria Chillotti, and Yongsoo Song. 2019. Improved bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 34–54.

[9] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. 2019. Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 395–412.

[10] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. A full RNS variant of approximate homomorphic encryption. In *International Conference on Selected Areas in Cryptography*. Springer, 347–368.

[11] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 409–437.

[12] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. In *https://eprint.iacr.org/2012/144.*

[13] David Froelicher, Juan R. Troncoso-Pastoriza, Apostolos Pyrgelis, Sinem Sav, Joao Sa Sousa, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. 2020. Scalable Privacy-Preserving Distributed Learning. arXiv:2005.09532 [cs.CR]

[14] Shai Halevi, Yuriy Polyakov, and Victor Shoup. 2019. An improved RNS variant of the BFV homomorphic encryption scheme. In *Cryptographers' Track at the RSA Conference*. Springer, 83–105.

[15] Kyoohyung Han and Dohyeong Ki. 2020. Better bootstrapping for approximate homomorphic encryption. In *Cryptographers' Track at the RSA Conference*. Springer, 364–390.

[16] Duhyeong Kim and Yongsoo Song. 2019. Approximate Homomorphic Encryption over the Conjugate-Invariant Ring. In *Information Security and Cryptology – ICISC 2018*, Kwangsu Lee (Ed.). Springer International Publishing, Cham, 85–102.

[17] Miran Kim, Arif Harmanci, Jean-Philippe Bossuat, Sergiu Carpov, Jung Hee Cheon, Ilaria Chillotti, Wonhee Cho, David Froelicher, Nicolas Gama, Mariya Georgieva, Seungwan Hong, Jean-Pierre Hubaux, Duhyeong Kim, Kristin Lauter, Yiping Ma, Lucila Ohno-Machado, Heidi Sofia, Yongha Son, Yongsoo Song, Juan Troncoso-Pastoriza, and Xiaoqian Jiang. 2020. Ultra-Fast Homomorphic Encryption Models enable Secure Outsourcing of Genotype Imputation. *bioRxiv* (2020). https://doi.org/10.1101/2020.07.02.183459 arXiv:https://www.biorxiv.org/content/early/2020/07/04/2020.07.02.183459.full.pdf

[18] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. 2012. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*. 1219–1234.

[19] Christian Mouchet, Juan R. Troncoso-Pastoriza, Jean-Philippe Bossuat, and Jean Pierre Hubaux. 2019. Multiparty Homomorphic Encryption: From Theory to Practice. In *https://eprint.iacr.org/2020/304.*

[20] Sinem Sav, Apostolos Pyrgelis, Juan R. Troncoso-Pastoriza, David Froelicher, Jean-Philippe Bossuat, Joao Sa Sousa, and Jean-Pierre Hubaux. 2020. POSEIDON: Privacy-Preserving Federated Neural Network Learning. arXiv:2009.00349 [cs.CR]

[21] SEAL 2020. Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.