A multiparty homomorphic encryption library in Go

Demo @ WAHC 2020
Christian Mouchet, Jean-Philippe Bossuat, Juan Troncoso-Pastoriza, Jean-Pierre Hubaux
Laboratory for Data Security
EPFL
Dec. 15, 2020
1. Homomorphic Encryption & Multiparty Homomorphic Encryption
2. The Lattigo Library
   a. Purpose
   b. Components
   c. Performance
3. Demo: a private web-application for scheduling meetings
4. Applications using Lattigo
In the passive-adversary model, Single-party HE can achieve **secure two-party computation**: Can this be extended to N-parties?
“Computing on multiparty data without seeing them using cryptographic protocols”

- “Early” semi-homomorphic constructions [FH 96, CDB 01]
- (R-)LWE-based instantiations in the
  - Threshold model [AJLT12, MTBH20] and
  - Multi-key model [LTV12, CDKS19, CCS19]

“out-of-the-box”: secure multiparty function evaluation in the passive-adversary model
This is where the Go language shines!

Variety of system models

Cloud-based MPC

Peer-to-peer MPC

- Networked
- Concurrent
- Cross-platform

This is where the Go language shines!
The Go language

Go is a serious candidate for implementing (M)HE-based systems

Pros:

• Designed for concurrent and networked systems
• Extremely simple syntax
• Extensive toolchain (dependency management, unit-testing, benchmarking…)
• Possible to call C/C++ code (cgo)
• Can be called from C/C++ (compilation in a .so library)

Cons:

• Compiler not as mature as “older” languages
• Can discourage newcomers to the language
The library

Purpose

• Allows the materialization and open-sourcing of FHE-based system research
• Brings FHE to a new language: Go

Licence: Apache 2.0

Versions

• v1.0.0 : 08/2019 first public-release
• v1.1.0 : 10/2019 improved API design and documentation (godoc)
• v1.3.0 : 12/2019 hybrid key-switching algorithm [HK 19]
• v2.0.0 : 10/2020 full-RNS bootstrapping [BMTH 20] & Wasm builds
• v2.1.0 : 12/2020 current version
• Lattigo is a Go module (https://blog.golang.org/using-go-modules)
  • Implements several packages:
    • A math layer for polynomial ring operations
    • A cryptographic layer for two HE schemes:
      BFV [FV 12]           CKKS [CKKS 17]
      vectorized integer arithmetic    vectorized complex arithmetic
    • The local operations of the multiparty variants [MTBH 20]
      DBFV                        DCKKS
  • Does not implement
    • Network-layer functionality
    • Multi-core processing
  \} ongoing implementation projects in separate code-bases
The **lattigo/ring** package

Math layer of the Lattigo library:

- Arithmetic on power-of-two degree polynomial rings with coefficients in the RNS domain
- Full-RNS arithmetic (avoids multi-precision arithmetic as much as possible)
- Number-Theoretic-Transform for efficient nega-cyclic convolution
- Sampling (Gaussian, ternary, uniform)
The \texttt{lattigo/ckks} package

Implements the full-RNS CKKS scheme [CHKKS 18]
- Vectorized approximate complex-arithmetic

Notable features
- Hybrid parameterizable key-switching
  - Tweaked from [HK 19]
  - Let the user choose \( \alpha \), the number moduli dedicated to key-switching
- Depth-optimal, scale-managed, polynomial evaluation algorithm
  - Evaluate non-linear functions on encrypted inputs
- Full-RNS bootstrapping with non-sparse keys [BMTH 20]
The lattigo/bfv package

Implements the full-RNS variant of the BFV scheme [BEH 17]

- Vectorized integer modular arithmetic
- Scale invariant (quantization)

Notable features:

- Special full-RNS Quantization based on the full-RNS tools of [CHKKS 18]
  - Faster homomorphic multiplications than the current state of the art
- Tweaked hybrid parameterizable key-switching of [HK 19]
The \texttt{dbfv} and \texttt{dckks} packages

Implement the local operation for the \textit{N-out-of-N} threshold BFV and CKKS [MTBH 20]

- Collective public/evaluation key-generation
- Collective key-switching/re-encryption operation to arbitrary secret-key
- Bridge with Linear-Secret-Sharing-based MPC:
  - Encryption-to-additive-shares
  - Additive-shares-to-encryption

Interactivity enables a fast, low-depth and exact bootstrapping

- Relying on the conversion between encryption and secret-shares
• Setup:
  • Windows 10 @3.5 GHz and 32GB of RAM
    • Go v1.14.2
    • MSVC++ v14.28

• We use Microsoft SEAL 3.6 as a baseline in the single-party setting

• Goal: show that we can write fast cryptographic code in Go
Benchmarks – BFV

Benchmark parameters:
- \( d \) the polynomial modulus degree
- \( L \) the coefficient modulus size in # of word-sized moduli
- \( \alpha \) the # of special primes (fixed to 1 for baseline comparison)

### 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>28</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>775</td>
<td>3933</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>Decode</td>
<td>0.2</td>
<td>0.6</td>
<td>1.2</td>
</tr>
<tr>
<td>Encrypt</td>
<td>18.5</td>
<td>65.4</td>
<td>253.5</td>
</tr>
<tr>
<td>Decrypt</td>
<td>5.6</td>
<td>23.5</td>
<td>115.7</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>
Benchmarks – CKKS

Benchmark parameters:
- \(d\) the polynomial modulus degree
- \(L\) the coefficient modulus size in # of word-sized moduli
- \(\alpha\) the # of special primes (fixed to 1 for baseline comparison)

### 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></td>
<td>SEAL</td>
<td>Lattigo</td>
<td>SEAL</td>
</tr>
<tr>
<td>Encode</td>
<td>112</td>
<td>103</td>
<td>305</td>
</tr>
<tr>
<td>Decode</td>
<td>63</td>
<td>82</td>
<td>345</td>
</tr>
<tr>
<td>Encrypt</td>
<td>548</td>
<td>392</td>
<td>1816</td>
</tr>
<tr>
<td>Decrypt</td>
<td>18</td>
<td>6</td>
<td>71</td>
</tr>
<tr>
<td>Add</td>
<td>7</td>
<td>3</td>
<td>28</td>
</tr>
<tr>
<td>Mul-Pt</td>
<td>14</td>
<td>7</td>
<td>52</td>
</tr>
<tr>
<td>Mul.Ct</td>
<td>45</td>
<td>15</td>
<td>187</td>
</tr>
<tr>
<td>Square</td>
<td>27</td>
<td>13</td>
<td>124</td>
</tr>
<tr>
<td>Rescale</td>
<td>-</td>
<td>-</td>
<td>203</td>
</tr>
<tr>
<td>KeySwitch</td>
<td>-</td>
<td>-</td>
<td>807</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></td>
<td>SEAL</td>
<td>Lattigo</td>
<td>SEAL</td>
</tr>
<tr>
<td>Encode</td>
<td>3.2</td>
<td>1.9</td>
<td>14.3</td>
</tr>
<tr>
<td>Decode</td>
<td>6.4</td>
<td>0.8</td>
<td>31.7</td>
</tr>
<tr>
<td>Encrypt</td>
<td>19.0</td>
<td>13.0</td>
<td>71.6</td>
</tr>
<tr>
<td>Decrypt</td>
<td>1.1</td>
<td>0.4</td>
<td>4.7</td>
</tr>
<tr>
<td>Add</td>
<td>0.4</td>
<td>0.2</td>
<td>1.7</td>
</tr>
<tr>
<td>Mul-Pt</td>
<td>0.8</td>
<td>0.5</td>
<td>3.1</td>
</tr>
<tr>
<td>Mul.Ct</td>
<td>3.1</td>
<td>1.1</td>
<td>12.0</td>
</tr>
<tr>
<td>Square</td>
<td>2.1</td>
<td>0.9</td>
<td>8.4</td>
</tr>
<tr>
<td>Rescale</td>
<td>3.6</td>
<td>3.4</td>
<td>14.6</td>
</tr>
<tr>
<td>KeySwitch</td>
<td>23.4</td>
<td>22.8</td>
<td>146.5</td>
</tr>
</tbody>
</table>
Demo: Private Polls

A privacy-preserving web-application for scheduling meetings

- Very simple BFV-based circuit
- A guided tour more than a full-fledged application
- Code is available: https://github.com/ldsec/lattigo-polls-demo
Demo: Private Polls – Setting

**Poll creator**
- Passive adversary
- No collusion with Server
- Runs a web-browser

**Poll participants**
- Passive adversaries
- Run a web-browser

**Server**
- Passive adversary
- No collusion with Poll creator
Demo: Private Polls – Basic (crypto) protocol

\[ \text{sk, pk, rlk} = \text{KeyGen} \]
\[ [0, 1, 0, 0, 0, 1, 0] = \text{Decrypt} (\text{sk, ct}_{\text{res}}) \]

\[ \text{ct}_1 = \text{Encrypt} (\text{pk}, [1, 1, 1, 0, 1, 0]) \]
\[ \text{ct}_2 = \text{Encrypt} (\text{pk}, [0, 1, 0, 0, 0, 1]) \]
\[ \text{ct}_3 = \text{Encrypt} (\text{pk}, [0, 1, 0, 0, 1, 1]) \]

\[ \text{ct}_{\text{res}} = \text{Mult} (\text{rlk, ct}_1, \text{ct}_2, \text{ct}_3) \]

\[ \text{ct}_{\text{res}} \]
Demo: Private Polls – Software Stack

**Client application**

- **client.go**
  - Performs cryptographic ops: KeyGen, Encrypt, Decrypt
  - Interface with the JS runtime
  - GOARCH=wasm go build client.go

- **client.wasm**

- **JavaScript**

**Server application**

- **server.go**
  - Performs cryptographic ops: Mult(ct1, ct2,...ctN)
  - Manages polls state: StoreResponse, Close
  - Serves the web-app & client.wasm
    - POST, GET

- **server**

- **go build client.go**

- **HTTP**

- **lattigo/bfv**

- **syscall/js**

- **net/http**
We define a PollClient as the union type between the following Lattigo types:

```go
type KeyGenerator interface {
    GenKeyPair() (*SecretKey, *PublicKey)
    GenRelinKey(*SecretKey, uint64) (*EvolutionKey)
    // [...]}
}

type Encoder interface {
    EncodeUint(coeffs []uint64, pt *Plaintext)
    DecodeUint(pt interface{}, coeffs []uint64)
    // [...]}
}

type Encryptor interface {
    Encrypt(plaintext *Plaintext, ciphertext *Ciphertext)
    // [...]}
}

type Decryptor interface {
    Decrypt(ciphertext *Ciphertext, plaintext *Plaintext)
    // [...]}
}
```

Client functionalities:
- Initialize with parameters
- Generate cryptographic keys
- Encode/Encrypt inputs
- Decrypt/Decode outputs

```go
import {
    "syscall/js"
    "github.com/ldsec/lattigo/v2/bfv"
}

// PollClient represents a client in the PrivatePoll application
type PollClient struct {
    bfv.KeyGenerator
    bfv.Encoder
    bfv.Encryptor
    bfv.Decryptor
    params *bfv.Parameters
}
```
Demo: Private Polls – client.go

Client functionalities:
- Initialize with parameters
- Generate cryptographic keys
- Encode/Encrypt inputs
- Decrypt/Decode outputs

Given the params and a secret or public key as a JS object.

Depending on the type of the key:
- deserialze the key
  - Lattigo has (de-)serializers for all cryptographic objects
- instantiate the encryptor and/or decryptor
  - Encryption from a sk is more efficient than from the pk

```go
func NewPollClient(params *bfv.Parameters, keyObj js.Value) *PollClient {
    pc := new(PollClient)
    pc.params = params
    pc.KeyGenerator = bfv.NewKeyGenerator(params)
    pc.Encoder = bfv.NewEncoder(params)

    switch keyObj.Get("type").String() {
        case "sk":
            sk := bfv.NewSecretKey(params)
            utils.UnmarshalFromBase64(sk, keyObj.Get("key").String())
            pc.Encryptor = bfv.NewEncryptorFromSk(params, sk)
            pc.Decryptor = bfv.NewDecryptor(params, sk)
        case "pk":
            pk := bfv.NewPublicKey(params)
            utils.UnmarshalFromBase64(pk, keyObj.Get("key").String())
            pc.Encryptor = bfv.NewEncryptorFromPk(params, pk)
    }

    return pc
}
```
Demo: Private Polls – client.go

Client functionalities:

- Initialize with parameters
- Generate cryptographic keys
- Encode/Encrypt inputs
- Decrypt/Decode outputs

```go
func main() {
    var pollClient *PollClient
    var params = bfv.DefaultParams[1]
    initClientFunc := js.FuncOf(func(this js.Value, args []js.Value) interface{} {
        keyObj := args[0]
        pollClient = NewPollClient(params, keyObj)
        return nil
    })
    js.Global().Set("initClient", initClientFunc)
    //...
}
```

main is the entry point of a Go program

Wraps a Go function in a “JS” object

Define the function JS global context:

```html
<html>
<script>
    let sk = localStorage.getItem("sk");
    initClient(key);
</script>
</html>
```
Demo: Private Polls – client.go

Client functionalities:
- **Initialise with parameters**
- **Generate cryptographic keys**
- **Encode/Encrypt inputs**
- **Decrypt/Decode outputs**

```go
func main() {
    // ...
    genKeysFunc := js.FuncOf(func(this js.Value, args []js.Value) interface{} {
        jsObj := args[0]
        sk, pk := pollClient.GenKeyPair()
        rlk := pollClient.GenRelinKey(sk, 1)
        jsObj.Set("sk", utils.MarshalToBase64String(sk))
        jsObj.Set("pk", utils.MarshalToBase64String(pk))
        jsObj.Set("rlk", utils.MarshalToBase64String(rlk))
        return jsObj
    })
    js.Global().Set("genKeys", genKeysFunc)
    // ...
}
```

Generates the key from the bfv.KeyGenerator interface

Write the key as base-64 string in the fields of a JS object
Demo: Private Polls – client.go

Client functionalities:
- Initialize with parameters
- Generate cryptographic keys
- Encode/Encrypt inputs
- Decrypt/Decode outputs

func main() {
    // ...
    encryptFunc := js.FuncOf(func(this js.Value, args []js.Value) interface{} {
        coeffs := make([]int64, 7)
        for i := range coeffs {
            coeffs[i] = int64(args[0].Index(i).Int())
        }
        pt := bfv.NewPlaintext(params)
        pollClient.EncodeInt(coeffs, pt)
        ct := pollClient.EncryptNew(pt)
        return
    }, js.FuncTypeAny)
    js.Global().Set("encrypt", encryptFunc)
    // ...
}
func main() {
    // ...

    decryptFunc := js.FuncOf(func(this js.Value, args []js.Value) interface{} {
        ctObj := args[0]

        if pollClient.Decryptor != nil {
            ct := bfv.NewCiphertext(params, 1)
            utils.UnmarshalFromBase64(ct, ctObj.String())
            pt := pollClient.DecryptNew(ct)

            coeffs := make([]interface{}, 7)
            for i, v := range pollClient.DecodeIntNew(pt)[:7] {
                coeffs[i] = v
            }

            return coeffs
        }

        return nil
    })

    js.Global().Set("decrypt", decryptFunc)
    // ...
}
Client functionalities:

- Initialize with parameters  ⇒ The initClient JS function
- Generate cryptographic keys ⇒ The genKeys JS function
- Encode/Encrypt inputs    ⇒ The encrypt JS function
- Decrypt/Decode outputs   ⇒ The decrypt JS function

Remaining tasks for the client:

- Create the UI (in HTML+CSS+JS)
- Compile client.go in WebAssembly:
  - $ GOOS=js GOARCH=wasm go build client.go
- Serve the UI and Wasm files from the server

115 Lines of code
Poll is the main data type for a Poll

A PollServer is a union type between a Lattigo Evaluator and an HTTP router.

The state of a PollServer is a collection of Polls

Server functionalities:
- Create a new poll from a `http.Request`
- Register a new response from a `http.Request`
- Compute the final result
- Serve the UI and Wasm files

```go
import {
    "net/http"
    "github.com/gorilla/mux"
    "github.com/ldsec/lattigo/v2/bfv"
}

type Poll struct {
    ID   string
    Closed  bool
    Participants map[string]*bfv.Ciphertext
    pk   bfv.PublicKey
    rlk  bfv.EvaluationKey
    responses []*bfv.Ciphertext
    result *bfv.Ciphertext
}

type PollServer struct {
    *mux.Router
    bfv.Evaluator
    Polls map[string]*Poll
}
```
Demo: Private Polls – server.go

Server functionalities:
- Create a new poll from a `http.Request`
- Register a new response from a `http.Request`
- Compute the final result
- Serve the UI and Wasm files

Unmarshalling of the pk and rlk into the Poll object fields

The “ID” of the Poll is the SHA256 hash of the public key.

```go
func (ps *PollServer) NewPoll(r *http.Request) *Poll {
    p := new(Poll)
    ps.Polls[p.ID] = p
    return p
}
```
Server functionalities:
- Create a new poll from a `http.Request`
- Register a new response from a `http.Request`
- Compute the final result
- Serve the UI and Wasm files

```go
func (p *Poll) RegisterResponse(r *http.Request) error {
    name := r/FormValue("name")
    ct, update := p.Participants[name]
    if !update {
        ct = new(bfv.Ciphertext)
        p.responses = append(p.responses, ct)
        p.Participants[name] = ct
    }
    return utils.UnmarshalFromBase64(ct, r.FormValue("ct"))
}
```

If no response was provided yet, create a new `bfv.Ciphertext` object
Deserializes the ciphertext from the request
func (ps *PollServer) Close(p *Poll) {
  p.Closed = true
  if len(p.Participants) > 0 {
    agg := p.responses
    for len(agg) > 1 {
      agg = append(agg[2:],
        ps.RelinearizeNew(
          ps.MulNew(agg[0], agg[1]), &p.rlk),
      )
    }
    p.result = agg[0]
  }
}

Aggregates the responses recursively in a low-multiplicative-depth circuit

The multiplicative-depth is log_2(len(p.responses))

Server functionalities:
- ✓ Create a new poll from a http.Request
- ✓ Register a new response from a http.Request
- ❏ Compute the final result
- ❏ Serve the UI and Wasm files
func main() {
    ps := NewPollServer(bfv.DefaultParams[1])
    pollTpl := template.Must(template.ParseFiles("poll.gohtml"))
    ps.HandleFunc("/polls", func(rw http.ResponseWriter, r *http.Request) {
        if r.Method == "POST" {
            p := ps.NewPoll(r)
            rw.Write([]byte(p.ID))
        } else {
            pollTpl.Execute(rw, nil)
        }
    })
    // ... [handler function for the /polls/<poll_id> route] ...
    http.Handle("/", ps)
    http.ListenAndServe("*:8080", nil)
}
Server functionalities:

- Create a new poll from a http.Request
- Display a Poll page for a given poll ID
- Register a new response from a http.Request
- Compute the final result
- Serve the UI and Wasm files

⇒ The POST /polls route
⇒ The GET /polls/<poll_id> route
⇒ The POST /polls/<poll_id> route
⇒ The GET /polls/<poll_id>?closing=true route
⇒ The GET /static/* route

155 Lines of code

Remaining tasks for the server:

- Input checking and error handling
- Concurrent request handling
Demo: Private Polls – Live Demo

https://github.com/ldsec/lattigo-polls-demo
Lattigo Applications

- **Single-party applications**
    - Batch predictions of 80k variants/patient for 1000 patients in less than 10 seconds
      - [https://doi.org/10.1101/2020.07.02.183459](https://doi.org/10.1101/2020.07.02.183459)
  - Secure oblivious linear function evaluation (vOLE) in passive and active adversarial settings
    - 1 million 120-bit OLEs/s in passive adversary model
      - SCN’20, [https://eprint.iacr.org/2020/970](https://eprint.iacr.org/2020/970)
  - Verifiable computation of encrypted certified inputs with MPC-in-the-head proofs
    - Application to smart metering, disease susceptibility and location-based activity tracking
Demo: why MHE?

- **Poll creator**
  - Passive adversary
  - No collusion with Server

- **Poll participants**
  - Passive adversaries

- **Server**
  - Passive adversary
  - No collusion with Poll creator

Participant can be unwilling to contribute their data without privacy guarantees. E.g.: **Federated Learning** among medical institutions
Lattigo Applications

- Multi-party applications in private federated learning
  
  - Secure training and evaluation of generalized linear models
    - Logreg with 1 million samples and 32 features across 160 data providers in 3 minutes
  
  - Secure training and evaluation of neural networks
    - 3-layer NN on MNIST across 10 data providers in <2h
Ongoing & Future work

- **Fully-Threshold MHE**: $T$-out-of-$N$ threshold variant of [MTBH 20], allows to set a tradeoff between security and availability.

- **Real-CKKS [KS 18]**: Variant of the CKKS scheme able to encrypt up to $d$ real numbers instead of $d/2$ complex numbers.

- **Lattigo-Cloud/Lattigo-MP**: Network-layers implementing the MHE-based MPC protocols for both cloud-based (clients + server) and peer-2-peer (clients) settings.
What is new in Lattigo:
- The first HE library focusing on multiparty settings
- The first open-source Full-RNS-CKKS bootstrapping

What becomes easier with Lattigo:
- Developing HE-based multi-party systems
- Running HE on any platform (incl. browsers!)
- Experimenting/benchmarking new Multi-party HE-applications (e.g., in federated learning)

What is still hard:
- Mapping complex functionalities to efficient HE-circuits
- Parametrization of the schemes
- Composability of HE and Circuit-Privacy
https://github.com/ldsec/lattigo


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.

<table>
<thead>
<tr>
<th>Op</th>
<th>$d=2^{13}, L=4$</th>
<th>$d=2^{14}, L=8$</th>
<th>$d=2^{15}, L=15$</th>
</tr>
</thead>
<tbody>
<tr>
<td>EncKeyGen</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gen</td>
<td>0.77</td>
<td>2.59</td>
<td>10.45</td>
</tr>
<tr>
<td>Agg</td>
<td>0.02</td>
<td>0.08</td>
<td>0.32</td>
</tr>
<tr>
<td>Out</td>
<td>0.09</td>
<td>0.34</td>
<td>1.28</td>
</tr>
<tr>
<td>RelinKeyGen</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gen</td>
<td>8.79</td>
<td>30.79</td>
<td>159.48</td>
</tr>
<tr>
<td>Agg</td>
<td>0.25</td>
<td>1.12</td>
<td>6.18</td>
</tr>
<tr>
<td>Out</td>
<td>0.36</td>
<td>1.81</td>
<td>9.72</td>
</tr>
<tr>
<td>RotKeyGen</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gen</td>
<td>2.46</td>
<td>8.73</td>
<td>45.24</td>
</tr>
<tr>
<td>Agg</td>
<td>0.06</td>
<td>0.25</td>
<td>1.53</td>
</tr>
<tr>
<td>Out</td>
<td>0.11</td>
<td>0.67</td>
<td>3.88</td>
</tr>
<tr>
<td>ColKeySwitch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gen</td>
<td>1.15</td>
<td>4.43</td>
<td>19.31</td>
</tr>
<tr>
<td>Agg</td>
<td>0.02</td>
<td>0.06</td>
<td>0.25</td>
</tr>
<tr>
<td>Out</td>
<td>0.02</td>
<td>0.07</td>
<td>0.32</td>
</tr>
<tr>
<td>PubColKeySwitch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gen</td>
<td>3.01</td>
<td>11.98</td>
<td>48.74</td>
</tr>
<tr>
<td>Agg</td>
<td>0.03</td>
<td>0.12</td>
<td>0.55</td>
</tr>
<tr>
<td>Out</td>
<td>0.03</td>
<td>0.12</td>
<td>0.68</td>
</tr>
</tbody>
</table>

Benchmark parameters:
- $d$ the polynomial modulus degree
- $L$ the coefficient modulus size in # moduli
- $\alpha$ the # of special primes (set to $\sqrt{(L-1)}$)

Benchmarked protocol operations:
- Gen Generating a party share (indep. N)
- Agg Aggregating two shares (indep. N)
- Out Computing the protocol output from the view (indep. N)

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

<table>
<thead>
<tr>
<th>Op</th>
<th>$d=2^{13}, L=4$</th>
<th>$d=2^{14}, L=8$</th>
<th>$d=2^{15}, L=15$</th>
</tr>
</thead>
<tbody>
<tr>
<td>EncKeyGen</td>
<td>0.26</td>
<td>1.05</td>
<td>3.93</td>
</tr>
<tr>
<td>RelinKeyGen</td>
<td>3.15</td>
<td>12.58</td>
<td>62.91</td>
</tr>
<tr>
<td>RotKeyGen</td>
<td>0.79</td>
<td>3.15</td>
<td>15.73</td>
</tr>
<tr>
<td>ColKeySwitch</td>
<td>0.2</td>
<td>0.79</td>
<td>3.15</td>
</tr>
<tr>
<td>PubColKeySwitch</td>
<td>0.39</td>
<td>1.57</td>
<td>6.29</td>
</tr>
</tbody>
</table>
Design principles

- Cryptographic functions are provided by function-specific objects

  encryptor := NewEncryptor(params, key)
  encryptor.Encrypt(pt, ct)

- Allocation-free API with shortcuts

  evaluator.Add(ct1, ct2, ctOut)
  ctOut := evaluator.AddNew(ct1, ct2)

- Objects assumes single-threaded use

- Objects methods are single-threaded

- Cryptographic objects types are serializable