An Implementation of KFHE With Faster Homomorphic Bitwise Operations

Kryptnostic
Matthew Tamayo-Rios & Nicholas Jian Hao Lai

April 25, 2016

Abstract: We will describe a new fully homomorphic cryptosytem, which we will call V2-KFHE in this article, which is based on KFHE with the explicit intention of improving homomorphic computations efficiencies.

1 Introduction

In this article, we will describe a modification of KFHE, which we temporarily (and unimaginatively) call V2-KFHE. V2-KFHE is practically demonstrated to be much faster than the original KFHE, performing homomorphic XOR and AND at speeds of a significant magnitude faster than KFHE.

2 Description of V2-KFHE

We describe now the underlying cryptosystem of KFHE. Readers who are familiar with KFHE can skip this section almost entirely, unless the reader wishes to be familiar with the notations employed in this article.

The cryptosystem is paramterised by $N \in \mathbb{N}$, where N is the size of the plaintext vector. We define the cryptosystem $\mathcal{C} = (\mathcal{E}, \mathcal{D})$ as follows:

- Secret key generation: Given N, we will randomly choose a $2N \times 2N$ invertible matrix

  $$M \in \mathrm{SL}(2N, \mathbb{F}_2)$$

  and random (multivariate) polynomials

  $$f_i \in \mathbb{F}_2[x_1, \cdots, x_N]$$

  for $i = 1, \cdots, N$. Let $f = (f_1, \cdots, f_N)$, i.e. for any $x \in \mathbb{F}_2^n$,

  $$f(x) = (f_1(x), \cdots, f_N(x))$$

  The secret key is exactly $\mathrm{sk} = (M, f)$. We will call f a *multivariate polynomial tuple.*

- Encryption: Given a message $m \in \mathbb{F}_2^N$, we randomly choose $r \in \mathbb{F}_2^N$, and define

  $$\mathcal{E} : \mathbb{F}_2^N \times \mathbb{F}_2^N \to \mathbb{F}_2^{2N}$$

  by

  $$\mathcal{E}(m, r) = M \begin{bmatrix} m + f(r) \\ r \end{bmatrix}$$

- Decryption: Given a ciphertext $c \in \mathbb{F}_2^{2N}$, we define

$$\mathcal{D} : \mathbb{F}_2^{2N} \to \mathbb{F}_2^N$$

by

$$\mathcal{D}(c) = \pi_1(M^{-1}c) + f\left(\pi_2(M^{-1}c)\right)$$

where for $i \leqslant k$,

$$\pi_i : \mathbb{F}_2^{kN} \to \mathbb{F}_2^N$$

is defined by

$$\pi_i(x_1, \cdots, x_{2N}) = (x_{(i-1)N+1}, \cdots, x_{iN})$$

The correctness of this scheme is easy to see, that is for all $m \in \mathbb{F}_2^N$, we get that for all $r \in \mathbb{F}_2^N$,

$$\mathcal{D}(\mathcal{E}(m, r)) = m$$

To make the above cryptosystem fully homomorphic, we will provide the following homomorphic binary operations: First, let $R_1, R_2 \in \mathrm{SL}(N, \mathbb{F}_2)$ and $K_1, K_2 \in \mathrm{SL}(3N, \mathbb{F}_2)$. Given the secret key (M, f), we define

$$H = K_1 \begin{bmatrix} I & 0 \\ 0 & I \\ R_1 & R_2 \end{bmatrix} \begin{bmatrix} \pi_2 \circ M^{-1} & \pi_2 \circ M^{-1} \end{bmatrix}$$

and

$$G = K_2 \left(\begin{bmatrix} f \\ f \\ f \end{bmatrix} \circ K_1^{-1} \right)$$

where I is the $N \times N$ identity matrix. Note that $\pi_2 \circ M^{-1}$ is equivalent to the bottom half rows of M^{-1}.

- Homomorphic XOR: Given two ciphertexts $x, y \in \mathbb{F}_2^{2N}$, we define the binary operation

$$\oplus : \mathbb{F}_2^{2N} \times \mathbb{F}_2^{2N} \to \mathbb{F}_2^{2N}$$

by

$$x \oplus y = M \begin{bmatrix} \mathcal{D}(x) + \mathcal{D}(y) + \pi_3 \left(K_2^{-1} G \left(H \begin{bmatrix} x \\ y \end{bmatrix} \right) \right) \\ R_1 \pi_2(M^{-1}x) + R_2 \pi_2(M^{-1}y) \end{bmatrix}$$

Specifically, if we define

$$\phi_{\mathrm{XOR}}(R) = M \begin{bmatrix} I & 0 \\ 0 & R \end{bmatrix} M^{-1}, \qquad Y_{\mathrm{XOR}} = M \begin{bmatrix} I & I & I \\ 0 & 0 & 0 \end{bmatrix} K_2^{-1}$$

Then, we have that

$$x \oplus y = \phi_{\mathrm{XOR}}(R_1)x + \phi_{\mathrm{XOR}}(R_2)y + Y_{\mathrm{XOR}} G \left(H \begin{bmatrix} x \\ y \end{bmatrix} \right)$$

- Homomorphic AND: Given two ciphertexts $x, y \in \mathbb{F}_2^{2N}$, we define the binary operation

$$\odot : \mathbb{F}_2^{2N} \times \mathbb{F}_2^{2N} \to \mathbb{F}_2^{2N}$$

by

$$x \odot y = M \begin{bmatrix} \mathcal{D}(x)\mathcal{D}(y) + \pi_3 \left(K_2^{-1} G \left(H \begin{bmatrix} x \\ y \end{bmatrix} \right) \right) \\ R_1 \pi_2(M^{-1}x) + R_2 \pi_2(M^{-1}y) \end{bmatrix}$$

Specifically, if we define

$$\phi_{\text{AND}}(R) = M \begin{bmatrix} 0 & 0 \\ 0 & R \end{bmatrix} M^{-1}, \qquad Y_{\text{AND}} = M \begin{bmatrix} I & \pi_3 \circ K_2^{-1} \\ 0 & 0 \end{bmatrix}$$

Then, we have that

$$x \odot y = \phi_{\text{AND}}(R_1)x + \phi_{\text{AND}}(R_2)y + Y_{\text{AND}} \begin{bmatrix} \pi_1(M^{-1}x)\pi_1(M^{-1}y) + \pi_1(M^{-1}x)\pi_2\left(G\left(H\begin{bmatrix}x\\y\end{bmatrix}\right)\right) + \pi_1(M^{-1}y)\pi_1\left(G\left(H\begin{bmatrix}x\\y\end{bmatrix}\right)\right) + \pi_1\left(G\left(H\begin{bmatrix}x\\y\end{bmatrix}\right)\right)\pi_2\left(G\left(H\begin{bmatrix}x\\y\end{bmatrix}\right)\right) \\ G\left(H\begin{bmatrix}x\\y\end{bmatrix}\right) \end{bmatrix}$$

In fact, given any binary operation $g : \mathbb{F}_2^N \times \mathbb{F}_2^N \to \mathbb{F}_2^N$, we can construct a homomorphic version of it via figuring out the form of

$$M \begin{bmatrix} g(\mathcal{D}(x), \mathcal{D}(y)) + \pi_3\left(K_2^{-1} G\left(H\begin{bmatrix}x\\y\end{bmatrix}\right)\right) \\ R_1\pi_2(M^{-1}x) + R_2\pi_2(M^{-1}y) \end{bmatrix}$$

Also, just like the original KFHE, we can perform homomorphic left matrix multiplication: First, let $R \in \text{SL}(N, \mathbb{F}_2)$ and $K_1, K_2 \in \text{SL}(2N, \mathbb{F}_2)$. Given the secret key (M, f), we define

$$H = K_1 \begin{bmatrix} I \\ R \end{bmatrix} (\pi_2 \circ M^{-1})$$

and

$$G = K_2 \left(\begin{bmatrix} f \\ f \end{bmatrix} \circ K_1^{-1} \right)$$

where I is the $N \times N$ identity matrix.

- Homomorphic left matrix multiplication: Given a matrix $T \in \text{Mat}(N, \mathbb{F}_2)$, we define $\mathcal{H}(T)$ by for all $x \in \mathbb{F}_2^{2N}$,

$$\mathcal{H}(T)(x) = M \begin{bmatrix} T\mathcal{D}(x) + \pi_2(K_2^{-1}G(Hx)) \\ R\pi_2(M^{-1}x) \end{bmatrix}$$

Specifically, if we define

$$\phi(T, R) = M \begin{bmatrix} T & 0 \\ 0 & R \end{bmatrix} M^{-1}, \qquad , \varphi(T) = M \begin{bmatrix} T & I \\ 0 & 0 \end{bmatrix} K_2^{-1}$$

Then we have that

$$\mathcal{H}(T)(x) = \begin{bmatrix} \phi(T, R) & \varphi(T) \end{bmatrix} \begin{bmatrix} x \\ G(Hx) \end{bmatrix}$$

The correctness of the homomorphic operations is a (brutal) exercise in computational linear algebra, the kind that some (evil) professors gives to poor students, but some details can be found in our CRYPTO paper (but don't be lazy, you should do the exercise! ** EVIL LAUGHS **).

3 Modifications & Steroids

In this section, we detail the modifications and difference of V2-KFHE compared to the original KFHE.

3.1 Monomial Matrix

The first of the major difference is the implementation of polynomial in KFHE. In the original implementation, polynomial tuples are represented by the `MQT` object, which are matrix forms of multivariate *quadratic* polynomial tuples. For more information, please see the implementation paper for details.

For our purpose, due to our choice of the form of secret polynomial tuple, we require higher degree polynomial, typically of degree five or seven, to achieve the same security. For that, we need a new way of representing polynomials, which lead us to the formulation of monomial matrices. We first define the following:

Definition 3.1. *A vector v is said to* represent monomial $m \neq 0$ *if for* $i = 1, \cdots, N$,

$$v_i = \begin{cases} 0, & \text{if } x_i \in m \\ 1, & \text{otherwise} \end{cases}$$

Here is an example: Suppose you have the monomial $x_1x_2x_3x_4x_5$. Then, the vector representing this monomial is

$$(0, 0, 0, 0, 0, 1, \cdots, 1)$$

i.e. the vector with the first five entry being zero, and the other entries being one.

To evaluate a monomial m at a point p, i.e. the value $m(p)$, if v is the vector representing m, we define

$$v(p) = \prod_{\substack{i=1 \\ p_i=0}}^{N} v_i$$

Note that

$$\begin{aligned} v(p) = 1 &\iff \forall i \text{ such that } p_i = 0,\ v_i = 1 \iff \{\, i;\ p_i = 0 \,\} \subseteq \{\, i;\ v_i = 1 \,\} \\ &\iff \{\, i;\ x_i \in m \,\} = \{\, i;\ v_i = 0 \,\} \subseteq \{\, i;\ p_i = 1 \,\} \iff m(p) = 1 \\ v(p) = 0 &\iff \exists i \text{ such that } p_i = 0 \text{ and } v_i = 0 \iff \exists i \text{ such that } x_i \in m \text{ and } p_i = 0 \iff m(p) = 0 \end{aligned}$$

so indeed $v(p) = m(p)$. In the special case that $m = 0$, we let the zero vector 0 represent it, and note that

$$0(p) = 0$$

for all $p \in \mathbb{F}_2^N$ except when $p = \mathbb{1}$, the all one vector, in which case we define $0(\mathbb{1}) = 0$.

With this, given a multivariate polynomial tuple $f = (f_1, \cdots, f_N)$, we write each f_i as

$$f_i = c_i + \sum_{j=1}^{l_i} m_{ij}$$

where l_i is the number of nonconstant monomials in f_i, each m_{ij} are monomials and c_i a constant, a representation of f will be of will be of form

$$\left(c, \sum_{j=1}^{l} V_j \right)$$

where $l = \max_{i=1}^{N} l_i$, $c = (c_1, \cdots, c_N) \in \mathbb{F}_2^N$, and each

$$V_j = \begin{bmatrix} v_{1j} & v_{2j} & \cdots & v_{Nj} \end{bmatrix}$$

where each v_{ij} represents monomial m_{ij}. Note that if $j > l_i$, then m_{ij} is taken to be a zero monomial, and so v_{ij} is the zero vector. To evaluate f at a point p, simply perform

$$c + \sum_{j=1}^{l} V_j(p)$$

where each $V_j(p)$ is defined as

$$V_j(p) = \begin{bmatrix} v_{1j}(p) & \cdots & v_{Nj}(p) \end{bmatrix}$$

For implementation details, see later section.

3.2 Smarter Choice of Public Obfuscated Polynomial

Another major difference is the choice of the multivariate polynomial tuple. In the original KFHE, a collection of l multivariate quadratic polynomial tuples were chosen as the polynomial tuples, where the usual choice is $l = 2$. This means that the key involves a choice of a product of l arbitrary multivariate quadratic polynomial tuples.

The problem with this choice of the form (or lack thereof) is that a typical multivariate quadratic polynomial may take a long time to compute the evaluation at a point. Compounded by the fact that one needs to perform l of these evaluations (of *obfuscated* polynomial tuples) every time a homomorphic binary operation is performed, the homomorphic computations can get extremely expansive. Indeed, usually the most expansive routine in a homomorphic binary operation is the evaluation of these multivariate polynomial tuple, and so the speed at which these homomorphic computations can be performed is significantly affected by the speed at which these evaluations can be performed.

To combat this issue, we elected to a choice of form for the multivariate polynomial tuple. First, we reduce the number of multivariate polynomial tuples to just one, of a high degree. This will reduce the number of multivariate polynomial evaluations performed in a homomorphic binary operation to just one, while still retaining a suitable amount of security. However, an evaluation of an arbitrary high degree multivariate polynomial tuple can be exponentially expansive, due to the number of possible monomials which appears as its terms, which is part of the reason a chain of l multivariate quadratic polynomial tuples is used as one component of the key in the original KFHE.

This leads us to the second modification: We will instead choose a multivariate polynomial tuple where each component polynomial is a product of linear factors, each of which has at most two terms, that is of Hamming weight 2. This reduces the total number of possible monomials to 2^d, where d is the number of linear factors and hence the degree of the polynomial. This is significantly less than $\binom{N}{d}$ if d is small, which is the number of possible monomials in a degree d polynomial.

Now, note that we need to speed up the *public* polynomial tuple, i.e. the obfuscated polynomial tuple, since that polynomial is the one that is evaluated in homomorphic computations. However, if we choose an arbitrary secret polynomial tuple f, the public obfuscated polynomial tuple, given by

$$G = K_2 \left(\begin{bmatrix} f \\ f \\ f \end{bmatrix} \circ K_1^{-1} \right)$$

while is a multivariate polynomial tuple in which each component polynomial is a product of linear factors as longs as f is, there is no guarantee that each linear factors of G will have Hamming weight 2, for an arbitrary choice of K_1 and K_2. In particular, it turns out that solving for an appropriate K_1, K_2 given f and G is equivalent to solving for

$$G_i = K_2 \left(\begin{bmatrix} f_i \\ f_i \\ f_i \end{bmatrix} \circ K_1^{-1} \right)$$

for $i = 1, \cdots, d$, where each G_i and f_i are linear factors of G and f respectively. There is no obvious way how such a system can be solved, at least to the authors. Indeed, this seems to be the problem of obfuscated isomorphism of polynomials, which is the basis of the security of the original KFHE.

Instead, we perform the following: During the secret key generation, we generate one linear factor f_1 of f, of arbitrary Hamming weight. We also generate one linear factor G_1 of G, of Hamming weight 2. We choose K_2 arbitrarily (not quite, see remark at the end of this section for details). We then calculate the K_1 such that

$$\begin{bmatrix} f_1 \\ f_1 \\ f_1 \end{bmatrix} \circ K_1^{-1} = K_2^{-1} G_1$$

How will one go about solving this system? Now, note that since f_1 and G_1 is a linear factor, hence a proper linear function, thus we can represent them in matrix form. So, in proper matrix form, the above system is

actually

$$\begin{bmatrix} f_1 & 0 & 0 \\ 0 & f_1 & 0 \\ 0 & 0 & f_1 \end{bmatrix} K_1^{-1} = K_2^{-1} G_1$$

Now, if we require f_1 and G_1 to be invertible, K_1 can be easily solved for. Finally, for the other factors of f and G, we just obtain them by performing the same permutation of both f_1 and G_1. i.e.

$$f_i = f_1 P_i, \qquad G_i = G_1 \begin{bmatrix} P_i & 0 & 0 \\ 0 & P_i & 0 \\ 0 & 0 & P_i \end{bmatrix}$$

where each P_i is a permutation matrix. Note that for each i,

$$G_i \begin{bmatrix} f_i^{-1} & 0 & 0 \\ 0 & f_i^{-1} & 0 \\ 0 & 0 & f_i^{-1} \end{bmatrix} = G_1 \begin{bmatrix} f_1^{-1} & 0 & 0 \\ 0 & f_1^{-1} & 0 \\ 0 & 0 & f_1^{-1} \end{bmatrix}$$

and so the choice of K_1 agrees over all linear factors of f and G.

Finally, it turns out that revealing the linear factors of G can reveal enough information to deduce f, so G must be presented without its linear factors. In our implementation, we present G in its monomial matrix representation whilst keeping f in its linear factorisation.

Remark 3.1: Note that if K_2 is chosen to be arbitrary, the form of the result $K_2 G$ for G a multivariate polynomial tuple is extremely complicated in its linear factorisation. For our implementation, we require K_2 to be a permutation matrix, so that the linear factorisation of $K_2 G$ is more predictable. Hence, for all choice of matrix K that is required to be multiplied to a multivariate polynomial tuple from the right, it is understood that the matrix is a permutation matrix.

4 SEARCH

In this section, we describe the encrypted search employed by V2-KFHE. The high level pipeline and structure remains the same as the original encrypted search in KFHE, but due to our modifications, there are minor change in the computation of ClientHashFunctions.

Given the cryptosystem as explained above, initialised at N bits, we define the following encrypted searching procedure for client Ω (so everything belonging to client Ω will have the subscript), which contains the following subroutine:

- Search private key generation: Given N, we will randomly choose two $N \times N$ invertible matrix

  $$K_\Omega, R_\Omega \in \mathrm{SL}(N, \mathbb{F}_2)$$

  The search private key is then spk $= (K, R)$.

- ClientHashFunction generation: Given the search private key spk $= (K_\Omega, R_\Omega)$, we define the *ClientHash-Function* to be the function

  $$h_\Omega : \mathbb{F}_2^{4N} \to \mathbb{F}_2^N$$

  defined by

  $$h_\Omega(x, y) = K_\Omega \begin{bmatrix} I & R_\Omega \end{bmatrix} \begin{bmatrix} \mathcal{D}(x) \\ \mathcal{D}(y) \end{bmatrix}$$

  for all $x, y \in \mathbb{F}_2^{2N}$. Specifically, if we define the *hash matrix*

  $$H_\Omega = K_\Omega \begin{bmatrix} I & R_\Omega \end{bmatrix} \begin{bmatrix} \pi_1 \circ M_\Omega^{-1} & 0 \\ 0 & \pi_1 \circ M_\Omega^{-1} \end{bmatrix}$$

and for some random $(C_1)_\Omega, (C_2)_\Omega \in \mathrm{SL}(2N, \mathbb{F}_2)$, define the *augmented matrix*

$$\mathrm{augmentedK}_\Omega = K_\Omega \begin{bmatrix} I & R_\Omega \end{bmatrix} (C_2)_\Omega^{-1}$$

define the *concealing matrix*

$$C_\Omega = (C_1)_\Omega \begin{bmatrix} \pi_2 \circ M_\Omega^{-1} & 0 \\ 0 & \pi_2 \circ M_\Omega^{-1} \end{bmatrix}$$

and define the *augmented multivariate polynomial tuple*

$$F = (C_2)_\Omega \left(\begin{bmatrix} f \\ f \end{bmatrix} \circ (C_1)_\Omega^{-1} \right)$$

- Indexing: Given a document i with token τ, we index the document by:
  1. Generate the metadata $\alpha_{\tau,i}$.
  2. Generate the object search key $d_i \in \mathbb{F}_2^N$.
  3. Generate the object address matrix $L_{i\Omega} \in \mathrm{SL}(N, \mathbb{F}_2)$.
  4. Generate the object conversion matrix $C_{i\Omega} = L_{i\Omega} K_\Omega^{-1}$.
  5. The server puts $\alpha_{\tau,i}$ at address
  $$L_{i\Omega} \begin{bmatrix} I & R_\Omega \end{bmatrix} \begin{bmatrix} \tau \\ d_i \end{bmatrix}$$
  6. Store $(\mathcal{E}(d_i), C_{i\Omega})$ on the server.
  7. The client stores $L_{i\Omega}$.
- Searching: Given an encrypted token $\mathcal{E}(\tau)$, the server searches for all instances of the search token appearing across all documents by: For each pair $(\mathcal{E}(d_i), C_{i\Omega})$ corresponding to the client Ω, do
  1. With the ClientHashFunction h_Ω, compute the address given by
  $$C_{i\Omega} h_\Omega(\mathcal{E}(\tau), \mathcal{E}(d_i))$$
  2. Returns the metadatum $\alpha_{\tau,i}$ in address, or nothing if it does not exists.
- Sharing: Given a document i which is to be shared with another client Γ, we share the document by:
  1. Client Ω sends $(R_\Omega d_i, L_i)$ to client Γ.
  2. Client Γ encrypts $(\mathcal{E}_\Gamma(R_\Gamma^{-1} R_\Omega d_i), C_{i\Gamma})$ and sends it to the server.

Note that here, we suppress the mention of the second input of $\mathcal{E}$, for brevity.

Again, the correctness of this encrypted search and sharing procedure is just a simple but tedious exercise in computation and symbol chasing, which is left to the interested reader.

5 C++ Implementation: krypto-V2-lib

In this section, we will describe the C++ implementation, krypto-V2-lib, in detail. We are assuming you have a Unix-like system.

5.1 Classes & Objects

There are several classes and objects at three different levels: interface, algorithm, and computation.

5.1.1 Interface-Level Objects

The interface level consists of high-level classes that are generated and/or exported by the classes at the API level.

- **PrivateKey**: and object with encryption and decryption capabilities that generates and store the parameters used in all of the KFHE processes. Generated by the client. It contains:
  - Secret matrix $M \in \mathrm{SL}(2N, \mathbb{F}_2)$.
  - Secret multivariate polynomial tuple f.
  - $K_1^u, K_2^u \in \mathrm{SL}(2N, \mathbb{F}_2)$ for the homomorphic left matrix multiplication.
  - $K_1^b, K_2^b \in \mathrm{SL}(3N, \mathbb{F}_2)$ for the homomorphic binary operations.

  which are accessible only by BridgeKey, ClientHashFunction, SearchPrivateKey and KryptnosticClient, and provides access to the following functions:
  - `encrypt`, an encryption function
  - `decrypt`, a decryption function
- **BridgeKey**: an obejct that uses a client's PrivateKey to generate objects that are used in PublicKey for the various homomorphic operations without exposing the PrivateKey. Generated by the client. It contains:
  - Random matrices $R, R_x, R_y \in \mathrm{SL}(N, \mathbb{F}_2)$, where R is used in homomorphic left matrix multiplication, and R_x, R_y are used in homomorphic binary operations.
  - Secret matrix $M \in \mathrm{SL}(2N, \mathbb{F}_2)$.
  - Secret multivariate polynomial tuple f.
  - $K_1^u, K_2^u \in \mathrm{SL}(2N, \mathbb{F}_2)$ for the homomorphic left matrix multiplication.
  - $K_1^b, K_2^b \in \mathrm{SL}(3N, \mathbb{F}_2)$ for the homomorphic binary operations.

  which are all private and provides access to the structures:
  - `H_XOR`, a structure containing the necessary coefficients for homomorphic XOR, and provides access for homomorphic XOR function.
  - `H_AND`, a structure containing the necessary coefficients for homomorphic AND, and provides access for homomorphic AND function.

  and provides access to the following function:
  - `getLMMZ`, a function which takes as input $T \in \mathrm{Mat}(N, \mathbb{F}_2)$ and outputs $\mathcal{H}(T)$.
  - `getLeftShiftMatrix`, a function which returns $\mathcal{H}(T)$, where T is the left shift matrix.
  - `getRightShiftMatrix`, a function which returns $\mathcal{H}(T)$, where T is the right shift matrix.
  - `getLeftColumnMatrix`, a function which returns $\mathcal{H}(T)$, where $T = \begin{bmatrix}\mathbb{1} & 0 & \cdots & 0\end{bmatrix}$.
  - `getRightColumnMatrix`, a function which returns $\mathcal{H}(T)$, where $T = \begin{bmatrix}0 & \cdots & 0 & \mathbb{1}\end{bmatrix}$.
- **PublicKey**: an object that uses a client's BridgeKey to perform various homomorphic operations on the server. Generated by the client for use by the server. It contains:
  - $\mathcal{H}(T)$, where T is the left shift matrix.
  - $\mathcal{H}(T)$, where T is the right shift matrix.
  - $\mathcal{H}(T)$, where $T = \begin{bmatrix}\mathbb{1} & 0 & \cdots & 0\end{bmatrix}$.
  - $H = K_1 \begin{bmatrix} I \\ R \end{bmatrix} (\pi_2 \circ M^{-1})$, for homomorphic left matrix multiplication.

– $G = K_2 \left(\begin{bmatrix} f \\ f \end{bmatrix} \circ K_1^{-1} \right)$, for homomorphic left matrix multiplication.

and provides access to the structures:

– `H_XOR`, a structure containing the necessary coefficients for homomorphic XOR, and provides access for homomorphic XOR function.

– `H_AND`, a structure containing the necessary coefficients for homomorphic AND, and provides access for homomorphic AND function.

- **SearchPrivateKey**: an object that uses a client's PrivateKey to generate the various private objects and functions used to index and search for objects. Generated by the client. It contains:

  – A random matrix $K \in \mathrm{SL}(N, \mathbb{F}^2)$.

  – ClientHashFunction auxilary matrix $R \in \mathrm{SL}(N, \mathbb{F}^2)$.

  which are all private and provides access to subroutines which are required to calculate encrypted and non-encrypted object address. For more information, see Section 4.

- **ClientHashFunction**: an object that describes the ClientHashFunction. See the description of ClientHashFunction in the next subsection.

5.1.2 Algorithm-Level Object

These are objects used in homomorphic computations and used in searching.

- `H_XOR`: a structure containing all the necessary coefficients for homomorphic XOR, and provides access for homomorphic XOR function. It contains:

  – Matrices `_Xx` and `_Xy` representing $\phi_{\mathrm{XOR}}(R_1), \phi_{\mathrm{XOR}}(R_2) \in \mathrm{SL}(2N, \mathbb{F}_2)$.

  – Matrix `_Y` representing $Y_{\mathrm{XOR}} \in \mathrm{Mat}(2N \times 3N, \mathbb{F}_2)$.

  – $H \in \mathrm{Mat}(3N \times 4N, \mathbb{F}_2)$, given by

  $$H = K_1 \begin{bmatrix} I & 0 \\ 0 & I \\ R_1 & R_2 \end{bmatrix} \begin{bmatrix} \pi_2 \circ M^{-1} & \pi_2 \circ M^{-1} \end{bmatrix}$$

  – Obfuscated multivariate polynomial tuple

  $$G = K_2 \left(\begin{bmatrix} f \\ f \\ f \end{bmatrix} \circ K_1^{-1} \right)$$

  which are all publicly accessible and provides access to the functions:

  – operator (), which takes as input two ciphertexts $x, y \in \mathbb{F}_2^{2N}$ and outputs $x \oplus y$.

- `H_AND`: a structure containing all the necessary coefficients for homomorphic AND, and provides access for homomorphic AND function. It contains:

  – BitMatrix `_z` representing the multivariate quadratic polynomial tuple

  $$W(x,y) = \pi_1 \left(M^{-1} \begin{bmatrix} \pi_1(x) \\ \pi_2(x) \end{bmatrix} \right) \pi_1 \left(M^{-1} \begin{bmatrix} \pi_1(y) \\ \pi_2(y) \end{bmatrix} \right) + \pi_1 \left(M^{-1} \begin{bmatrix} \pi_1(x) \\ \pi_2(x) \end{bmatrix} \right) \begin{bmatrix} \pi_3(y) \\ \pi_4(y) \\ \pi_5(y) \end{bmatrix} + \begin{bmatrix} \pi_3(x) \\ \pi_4(x) \\ \pi_5(x) \end{bmatrix} \pi_1 \left(M^{-1} \begin{bmatrix} \pi_1(y) \\ \pi_2(y) \end{bmatrix} \right) + \begin{bmatrix} \pi_3(x)\pi_3(y) \\ \pi_4(x)\pi_4(y) \\ \pi_5(x)\pi_5(y) \end{bmatrix}$$

  where $x, y \in \mathbb{F}_2^{5N}$. Note the dimensionality!!!

- Matrices `_Zx` and `_Zy` representing $\phi_{\text{AND}}(R_1), \phi_{\text{AND}}(R_2) \in \text{SL}(2N, \mathbb{F}_2)$.
- Matrix `_Y` representing the $Y_{\text{AND}} \in \text{Mat}(2N \times 3N, \mathbb{F}_2)$.
- $H \in \text{Mat}(3N \times 4N, \text{F}_2)$, given by

$$H = K_1 \begin{bmatrix} I & 0 \\ 0 & I \\ R_1 & R_2 \end{bmatrix} \begin{bmatrix} \pi_2 \circ M^{-1} & \pi_2 \circ M^{-1} \end{bmatrix}$$

- Obfuscated multivariate polynomial tuple

$$G = K_2 \left(\begin{bmatrix} f \\ f \\ f \end{bmatrix} \circ K_1^{-1} \right)$$

which are all publicly accessible and provides access to the functions:

- operator (), which takes as input two ciphertexts $x, y \in \mathbb{F}_2^{2N}$ and outputs $x \odot y$.

- **ClientHashFunction**: an object that describes the client hash function. It contains:
  - Hash matrix $H \in \text{Mat}(N \times 4N, \mathbb{F}^2)$.
  - Augmented matrix $K \in \text{Mat}(N \times 2N, \mathbb{F}^2)$.
  - Concealing matrix $C \in \text{Mat}(2N \times 4N, \mathbb{F}^2)$.
  - Augmented multivariate polynomial tuple F.

  and provides access to the functions:
  - operator (), which takes as input the search token and object search key, both of which are encrypted, and computes the address.

5.1.3 COMPUTATION-LEVEL OBJECT

These are objects used at the lowest level for underlying storage and computation, which are provided by our C++ libraries.

- **BitVector**: a vector of N bits stored in an array of 64 bit unsigned long longs. This is used to represent arithmetic in $\mathbb{F}_2^N$.
- **BitMatrix**: a matrix of bits stored as a collection of BitVectors as rows. As such, it is a row-major implementation of $\text{Mat}(\cdot, \mathbb{F}_2)$.
- **MonomialMatrixChain**: an implementation of MonomialMatrix, as described in Section 3.1. It contains:
  - `_node`, which is the monomial matrix representation of a monomial, in BitMatrix form as described in Section 3.1.
  - `_next`, which is a pointer to the next monomial, whose value will be a MonomialMatrixChain object.

  NOTE: Any struct or class which contains MonomialMatrixChain as one of its members must have a custom destructor, where `freeAllNext()` is called to properly delete the chain to prevent memory leaks.
- **ConstantHeaderChain**: an object which fully describes a multivariate polynomial tuple. It contains:
  - `_headConstant`, which is the constant part of the multivariate polynomial tuple, in BitVector form.
  - `_mmc`, which is the nonconstant monomials of the multivariate polynomial tuple, in MonomialMatrixChain form.

NOTE: Any struct or class which contains ConstantHeaderChain as one of its members must have a custom destructor, where `freeAllNext()` of MonomialMatrixChain is called to properly delete the chain to prevent memory leaks.

- **LinearFactorPoly**: a collection of BitMatrices, where each column is has Hamming weight at most 2. These BitMatrices are linear factors of a multivariate polynomial tuple with zero constant. The product of all these BitMatrices is the multivariate polynomial tuple. See Section 3.2.
- **SecretPolynomial**: an object which fully describes a multivariate polynomial tuple in its linear factorisation. It contains:
  - `_linearFactors`, which is the linear factorisation of the nonconstant monomials of the multivariate polynomial tuple, in LinearFactorPoly form.
  - `_constant`, which is the constant part of the multivariate polynomial, in BitVector form.

5.2 C++ Implementation Flow

In this section, we describe the enumerated steps of the implementation of V2-KFHE.

1. **Initialisation**: A client initialises by doing the following:
   (a) Call constructor of PrivateKey and feed N as the number of bits. This will generate PrivateKey for the client.
   (b) Call constructor of BridgeKey and feed PrivateKey. This will generate BridgeKey for the client.
   (c) Call constructor of PublicKey and feed BridgeKey. This will generate PublicKey for the client. Client sends PublicKey to server.
   (d) Call constructor of SearchPrivateKey and feed N as the number of bits. This will generate SearchPrivateKey for the client.
   (e) Generate ClientHashFunction by calling `getClientHashFunction` provided by SearchPrivateKey. Store the ClientHashFunction on the server.
2. **Encryption**: A client encrypts a message m by calling `encrypt`(m) of PrivateKey.
3. **Homomorphic Computations**: A server can perform binary operations for the ciphertexts $x, y \in \mathbb{F}_2^{2N}$ as follows:
   (a) $x \oplus y$ can be calculated by calling `homomorphicXOR`(x, y) of PublicKey.
   (b) $x \odot y$ can be calculated by calling `homomorphicAND`(x, y) of PublicKey.
   (c) Given $H(T)$, where $T \in \mathrm{Mat}(N \times N, \mathbb{F}_2)$, $\mathcal{E}(T\mathcal{D}(x))$ can be calculated by calling `homomorphicLMM`$(H(T), x)$ of PublicKey.
   (d) Left/Right shift of $\mathcal{D}(x)$ can be calculated by calling `homomorphicLEFTSHIFT`/`homomorphicRIGHTSHIFT`(x) of PublicKey.
   (e) Homomorphic integer addition in base 2, i.e. integer with carry over, of $\mathcal{D}(x)$ and $\mathcal{D}(y)$ can be calculated by calling `homomorphicADD`(x, y) of PublicKey.
   (f) Homomorphic integer multiplication in base 2 of $\mathcal{D}(x)$ and $\mathcal{D}(y)$ can be calculated by calling `homomorphicMULT`(x, y) of PublicKey.
4. **Encrypted Search**:
   (a) **Index**: Client Ω indexes the document i for future searching purposes by doing the following:
      i. Tokenise document and generate the metadata $\alpha_{\tau,i}$. This is handled externally and not by krypto-V2-lib.

ii. Call `getObjectIndexPair` of SearchPrivateKey, which will produce the pair (d_i, L_i), which as a subroutine calls `getObjectSearchKey` of SearchPrivateKey for d_i and `getObjectAddressMatrix` of SearchPrivateKey for L_i. L_i is stored by the client.

iii. Generate object conversion matrix $C_{i\Omega}$ by calling `getObjectConversionMatrix`(L_i) of SearchPrivateKey.

iv. Computes the address $L_i \begin{bmatrix} I & R_\Omega \end{bmatrix} (\tau, d_i)$ by calling `getMetadataAdress`$((d_i, L_i), \tau)$ of SearchPrivateKey. Tells the server to store $\alpha_{\tau,i}$ at the address.

v. Call `getObjectSearchPairFromObjectIndexPair`$((d_i, L_i)$, PrivateKey) of SearchPrivateKey, which will produce the pair $(\mathcal{E}_\Omega(d_i), C_{i\Omega})$, which as subroutine calls `encrypt`(d_i) of PrivateKey for $\mathcal{E}_\Omega(d_i)$ and `getObjectConversionMatrix`(L_i) of SearchPrivateKey for $C_{i\Omega}$. Store the search pair in the server.

(b) **Search**: A server can perform encrypted search querry $\mathcal{E}_\Omega(\tau)$ of client Ω appearing across all documents by doing the following:

i. The server initialises KryptnosticServer by calling the constructor and feed ClientHashFunction h_Ω and $\mathcal{E}_\Omega(\tau)$. This will generate KryptnosticServer for the server.

ii. For each object search pair $(\mathcal{E}_\Omega(d_i), C_{i\Omega})$ associated to client Ω, the server computes the metadatum address by calling `getMetadataAddress`$(\mathcal{E}_\Omega(d_i), C_{i\Omega})$ of KryptnosticServer. The server returns the metadatum stored at the calculated address, if it exists.

(c) **Sharing**: Client Ω can share document i to client Γ by doing the following:

i. Get object search pairs $(\mathcal{E}_\Omega(d_i), C_{i\Omega})$ from the server.

ii. Call `getObjectSharePairFromObjectSearchPair`$(\mathcal{E}_\Omega(d_i), C_{i\Omega})$ of SearchPrivateKey, which will produce the pair $(R_\Omega d_i, L_i)$, and sends it to client Γ.

iii. Client Γ calls `getObjectSearchPairFromObjectSharePair`$(R_\Omega d_i, L_i)$ of SearchPrivateKey, which will produce the pair $(\mathcal{E}_\Gamma(R_\Gamma^{-1} R_\Omega d_i), C_{i\Gamma})$. Store the pair on the server.

5. **Decryption**: A client decrypts a ciphertext c by calling `decrypt`(c) of PrivateKey.

6 BENCHMARKS

This records the computation time of the homomorphic functions. Running this implementation on a system with

- Prcoessor: 2.5 GHz Intel Core i7 with 6MB shared L3 cache
- RAM: 16 GB of 1600 MHz DDR3 onboard memory

gives the following average runtimes over 1000 runs with random initialisations on 128-bits with degree of multivariate polynomial tuple being 5:

Operations	Times
Encryption	7.76μs
Decryption	327.33μs
Homomorphic LMM	58.359μs
Homomorphic LS	40.912μs
Homomorphic RS	38.805μs
Homomorphic XOR	122.265μs
Homomorphic AND	288.182μs
Homomorphic ADD	39.384 ms
Homomorphic MULT	5.39186 s
Encrypted Search	24.0393μs

Finally, running this implementation on the same system gives the following average runtime over 1000 runs with random initialisations on 128-bits with degree of multivariate polynomial tuple being 7:

Operations	Times
Encryption	6.734μs
Decryption	317.631μs
Homomorphic LMM	175.76μs
Homomorphic LS	138.387μs
Homomorphic RS	146.595μs
Homomorphic XOR	361.857μs
Homomorphic AND	489.514μs
Homomorphic ADD	104.99 ms
Homomorphic MULT	13.3297 s
Encrypted Search	85.4806μs