POSTER: An Implementation of the Fully Homomorphic Smart-Vercauteren Crypto-System

Henning Perl, Michael Brenner and Matthew Smith
Distributed Computing Security Group
Gottfried Wilhelm Leibniz Universität
Hannover, Germany
{brenner,perl,smith}@dcsec.uni-hannover.de

ABSTRACT
Since the discovery of a fully homomorphic cryptographic scheme by Gentry, a number of different schemes have been proposed that apply the bootstrap technique of Gentry’s original approach. However, to date no implementation of fully homomorphic encryption has been publicly released. This poster presents a working implementation of the Smart-Vercauteren scheme that will be freely available and gives substantial implementation hints.

Categories and Subject Descriptors
E.3 [Data Encryption]: Public Key Cryptosystems

General Terms
Security

Keywords
homomorphic encryption, implementation

1. LIBRARY IMPLEMENTATION
In this section we present the implementation of the Smart-Vercauteren homomorphic scheme. We modified the recrypt operation, specifically we improved the circuit to calculate the Hamming weights to make it shallower.

Helper Libraries
For large integers, the GNU Multiple Precision Arithmetic Library (GMP) was used. Much of the number theoretical calculation additionally utilizes the Fast Library for Number Theory (FLINT), version 1.6.

1.1 Somewhat Homomorphic Scheme
In the somewhat homomorphic encryption scheme as defined by Smart et al. decryption works only as long as the cipher text noise (more precisely the coefficients of the C(x)) is within certain bounds r. These bounds depend mainly on the parameter N. For our implementation and choice of parameters we get \( r = 2^n/\sqrt{2N} \), see [2] for a more generic calculation. Besides this restriction, the general idea is the same as with the fully homomorphic scheme. G(x) generates an ideal p = \( (G(x)) \) in \( \mathbb{Z}[x]/F[x] \), along with the dual element description p = \( \langle p, x - a \rangle \) (which is used in the encrypt function). This gives us the homomorphism

\[
q_p : \mathbb{Z}[x] \rightarrow (\mathbb{Z}[x]/F(x))/p \\
C(x) \mapsto C(a) \mod p
\]

which we will use for the encryption.

1.1.1 Key generation

Notation.
We write polynomials in uppercase roman letters. For a given polynomial \( G(x) \) of degree n, \( (g_0, \ldots, g_n) \) denote the coefficients such that \( G(x) = \sum_{i=0}^{n} g_i x^i \).

1: keygen(pk, sk)
2: F(x) = x^n + 1 // monic, irreducible of degree n
3: \( \text{do} \)
4: G(x) = \text{random polynomial in } B_{n,n}^{\text{even}}(\mu)
5: \( \text{while } p \text{ is not prime} \)
6: D(x) = \text{Fmpz}\_\text{gcd}_\text{euclidean}(G(x), F(x))
7: a = \text{random coefficient odd}
8: p = \text{Fmpz}_\text{poly}_{\text{resultant}}(G(x), F(x))
9: \( \text{while } p \text{ is not prime} \)
10: (r, Z(x), t) = \text{Fmpz}_\text{poly}_{\text{gcd}}(G(x), F(x))
11: pk.p = p // pk, sk are simply structs
12: sk.p = p // \( \text{gcd}( \text{Fmpz}(p), B) = z_0 \mod 2p \)
13: \( \text{while } p \text{ is not prime} \)
14: 

Discussion.
Line 3 \( x^n + 1 \) is always a safe choice for a monic, irreducible polynomial of degree n as it has no roots in \( \mathbb{Z}[x] \).

Line 5 [2] defines

\[
B_{n,n}(r) := \left\{ \sum_{i=0}^{n-1} a_i x^i : a_i \in [-r, r] \right\}
\]

Analogously, we define

\[
B_{n,n}^{\text{even}}(r) := \left\{ \sum_{i=0}^{n-1} 2a_i x^i : a_i \in \left[ -\frac{r}{2}, \frac{r}{2} \right] \right\}
\]

In the algorithm, we randomly choose coefficients in \( [-\frac{r}{2}, \frac{r}{2}] \) and then multiply them by 2.

Line 8 the Miller-Rabin prime number test is used here. \( p \neq 0 \) prime implies that \( F(x) \) and \( G(x) \) are coprime and further that \( G(x) \) is irreducible (as we already know...
that $F(x)$ is irreducible. Therefore $G(x)$ generates a principal ideal $p$ in $\mathbb{Z}[x]/F(x)$.

**Line 9** the GCD-algorithm used here is just a variation of euclid's classic algorithm working on polynomials modulo $p$. Here, we find the dual element representation $(p, x - a)$ of $p$ with $p$ being the norm of $p$ and $a$ a root of $F(x) \mod p$. The root of $D(x)$, of course, is also a root of $F(x)$ and $G(x)$.

**Line 11** the extended GCD-algorithm sets the output such that $Z(x) \cdot G(x) = p \mod F(x)$. Here we generate the subsecret key. The decryption algorithm requires us to calculate $\frac{Z(x)}{G(x)}$. In the concrete implementation of the decrypt function we only need to round to the nearest integer. Therefore, only $z_0$ is relevant here as the subsecret key.

### 1.1.2 Encrypt, Decrypt, Add, Mult

The code for encrypt, decrypt, add and mult largely resembles the pseudocode given in [2] as all of the operations translate well to GMP calls. In the encryption function we generate a polynomial $C(x)$ with the parity of the constant coefficient depending on the message to be encrypted. Then we use eq. 1 to transform the polynomial to the crypto space by evaluating $C(a) \mod p$ with `fmpz_poly_evaluate()`. After an addition, if the input cyphertexts were bound by $b_1$ and $b_2$, the result will be bound by $b_1 + b_2$ (the coefficients simply add up). After a multiplication however, the result will be bound by $b_1 + b_2 + b_1 \cdot b_2$. In the following we will examine how many multiplications are possible while the cypher text is still decryptable. After $d$ multiplications, the output will be bound by $\mu^d = 4^d$.

$$4^{2^d} = \frac{2^{384}}{16} \Leftrightarrow 2^d = 190 \Leftrightarrow d = \lfloor \log_2 190 \rfloor = 7$$

This will be enough for our `recrypt()` to work as well as leaving some operations for homomorphic gates.

### 1.2 Fully Homomorphic Scheme

In this section we present a version of the decryption algorithm consisting only of xor and and-gates. This allows us to apply the decryption to a cyphertext, returning a cleaner version. For this recrypt to be effective, the depth of the circuit has to be shallow enough. We present a minimal rounding function to account for this.

#### 1.2.1 Key generation revised

In addition to $B$, $a$ and $p$ constructed in the somewhat homomorphic version of the keygen, we now also have to construct a hint $\{c_i, B_i\}_{i=1}^{t_1}$ such that $\sum_{i=1}^{t_1} \text{decrypt}(c_i)B_i = B$. Since we rely on hiding the value of $B$ in the array $B_t$, the security of this hint depends largely on the size of $B_t$. We modified our algorithm to account for this. The security of the hint can then be reduced to the Subset-Sum problem. We call the tuple $\{\{a\}, S_1, S_2\}$ the key geometry.

```
1: keygen() // continued . . .
2: $B^* = \lfloor \frac{B}{T} \rfloor$ // Step 1: Distribute
3: for $(i = 0; i < S_2; i++)$
4:     $pk.B_i = B^*$
5:     $pk.c_i = 1$
6: }

```

**Table 1:** Initial distribution of the hint

<table>
<thead>
<tr>
<th>pk.c</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>pk.B</td>
<td>$\lfloor \frac{B}{T} \rfloor$</td>
<td>...</td>
<td>$\lfloor \frac{B}{T} \rfloor$</td>
<td>$B - 4\lfloor \frac{B}{T} \rfloor$</td>
</tr>
<tr>
<td>rand</td>
<td>rand</td>
<td>...</td>
<td>rand</td>
<td>rand</td>
</tr>
</tbody>
</table>

#### Discussion:

**Step 1** In the distribution step the array is initialized as depicted in table 1.

**Step 2** Addition and subtraction of random values only operates on the first $S_2 (= 5)$. Note that in each step the invariant $\sum_{i=1}^{t_1} \text{decrypt}(c_i)B_i = B$ holds.

**Step 3** In order to randomize the hint, we finally shuffle the array.

#### 1.2.2 Recrypt

The purpose of the `recrypt()` function is to generate a “cleaner” cryptotext from a “dirty” one with the same cleartext value. The homomorphic operations add and mult add to the dirtyness of the cryptotext, making it eventually unrecoverable. Therefore, without a `recrypt()` function, we can only compute circuits of a fixed depth. Beyond that, the results are not correctly decryptable.

The general idea of this function is to decrypt the cyphertext in the cryptospace using homomorphic operations. Recall that the decrypt function computes

$$(c - \left\lfloor \frac{B}{T} \right\rfloor) \equiv_2 (c - \sum_{i} c_i B_i \frac{c}{p}) \equiv_2 \left( c - \sum_{i} c_i B_i \cdot (c \mod 2p / p) \right)$$

The right part of the equation we can compute using only the public key and the hint constructed in subsection 1.2.1. But since the $c_i$’s are encrypted, the final output will also still be encrypted. The result is a cleaner encryption of $c$.

```
1: for $(i = 0; i < S_2; i++)$
2:     $d = (B_i \cdot c \mod 2p)$ // Note that $d \in \{0, 2\}$
3:     for $(j = 0; j < S_2; j++)$
4:         $C_{ij} = \text{encrypt}(\lfloor d \rfloor) \cdot c_i \mod p$
5:     $d = (d - \lfloor d \rfloor) \cdot 2$ // This converts the fraction base 2
```
After running the algorithm above, each row of the matrix $(C_{ij})$ holds a binary representation of $(B_i \cdot c \mod 2p)$ with $T - 1$ bits of precision in the ciphertext. As an optimization in line 4 we can check whether $|d|_i$ is zero and then use either $c_i$ or encrypt(0). This saves an additional multiplication and provides a cleaner cryptotext matrix to start with.

![Figure 1: Shifting and merging of the rows](image)

Next, we sum up individual rows with a circuit as shallow as possible. Therefore, the addition of the rows is split up in separate steps:

1. Calculate the Hamming weights of the individual rows:
   The calculation of the Hamming weight is using (elementary) symmetric polynomials as suggested by [2]. With this approach we can get a much shallower circuit than using half- and fulladders, since the polynomial $e_k(X_1,\ldots,X_n) = \sum_{i\leq k<\ldots<i\leq n} X_{i_1}\cdots X_{i_k}$ directly gives the $k$-th bit of the Hamming weight of the input. For a pseudocode see [2, p. 15].

2. Shift and merge the Hamming weights according to the significance of the column: Once we computed the Hamming weight of the individual columns, we can shift and merge the values as shown in fig. 1.2.2. Note that we don’t need any circuit gates (or homomorphic operations) here as we just “rewire”.

3. Use a carry-save adder until there are only two rows left: The carry-save adder is simply a full-adder applied to three rows at a time. This allows for a constant circuit depth.

4. Use a ripple-carry adder for the final addition: Finally, we have to do one ripple-carry-addition, which has a linear circuit depth with respect to the length of the rows.

After this last step, we have reduced the matrix to one column holding a fixed precision floating point representation of $\sum_i c_i (B_i \cdot c \mod 2p)/p$.

### 1.2.3 A shallower rounding function

To produce the cleaner encryption of the input, [2] suggests a rounding using the last two floating point bits. However, in our initial implementation this resulted in a circuit that was overall too deep to compute the correct value. We suggest an alternative rounding using just the last bit to save circuit depth while still being able to correctly “clean” the value. Effectively we are able to reduce the depth by two multiplications. Table 2 compares our rounding function to the function proposed in [2].

<table>
<thead>
<tr>
<th>$c_0,c_1,c_2$</th>
<th>decimal</th>
<th>our rounding</th>
<th>rounding in [2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0.01</td>
<td>0.25</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0.10</td>
<td>0.50</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0.11</td>
<td>0.75</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.01</td>
<td>1.25</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1.10</td>
<td>1.50</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1.11</td>
<td>1.75</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 2: Comparison of rounding circuits**

The output of the algorithm is then given by

$$c = \left\lceil \sum_i c_i (B_i \cdot c \mod 2p)/p \right\rceil = (c + c_0 + c_1) \mod 2.$$  

**Performance.**

Due to space constraints, most performance figures for different key sizes and key configurations are available on the poster. Table 3 shows basic figures depending on the key geometry (see 1.2.1).

<table>
<thead>
<tr>
<th>key geometry</th>
<th>key size (kB)</th>
<th>keygen (s)</th>
<th>recrypt (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>384/16/5</td>
<td>1.8/30</td>
<td>17</td>
<td>263</td>
</tr>
<tr>
<td>384/32/16</td>
<td>1.8/60</td>
<td>14</td>
<td>307</td>
</tr>
<tr>
<td>384/64/16</td>
<td>1.8/120</td>
<td>15</td>
<td>684</td>
</tr>
<tr>
<td>2048/64/16</td>
<td>9.6/626</td>
<td>3180</td>
<td>1350</td>
</tr>
<tr>
<td>4096/64/16</td>
<td>18/1250</td>
<td>14278</td>
<td>3280</td>
</tr>
</tbody>
</table>

**Table 3: Key geometry and performance**

**Conclusion.**

While ([1]) gave some hints to implementing a fully homomorphic scheme, no source code was published and as was shown above there are significant implementation and parameter issues. Presenting and discussion these details is vital to the development of cryptographic systems. This implementation is a step towards making fully homomorphic encryption available for a broad audience of practitioners.

## 2. REFERENCES
