Somewhat Homomorphic Encryption

Michael Belland, William Xue, Mohammed Kurdi, Weilian Chu

May 18, 2017

1 Introduction

Homomorphic Encryption (HE) is a way that encrypted data can be processed without being decrypted first. An encoded message is sent to a third-party, who performs an operation on the received message and sends back the result. The original requester can decode that result to get the result of their original query, while the third-party who processed the data has no idea what the query or result actually was.

2 Problem Statement

2.1 How can we compute on encrypted data securely?

A common problem that comes up in security literature is as follows: Suppose a hospital has patient data with personal information about their patients, and they wish to perform data analysis on aggregated data from their dataset. Is it possible to do this without any individual's personal data being revealed in the process?

It turns out that this is possible—with encrypted data $E(x)$ and $E(y)$ but without the original values x and y, it's possible to compute values like $E(x + y)$ and $E(x * y)$, which can then be decrypted so the values of x and y never have to be decrypted. This property of performing computations on values by using their encrypted representations is called *homomorphism*. Some cryptosystems have homomorphic propertes, albeit as a negative aspect—for example, the basic version of RSA is weak to chosen ciphertext attacks because ciphertexts are multiplicative—but HE focuses on using these properties beneficially. In particular, HE relies on having an encryption scheme with both additive and multiplicative properties (and getting both properties isn't trivial - for example, RSA is multiplicative but not additive) so it can perform a number of useful operations.

HE is just a part of systems, such as CryptDB, that were designed to address the hospital problem proposed above. However, due to implementation limitations its use in such systems was minimized [5]. We wish to explore implementations of HE that may have expanded potential in such systems, as discussed more in section 3.2.

2.2 Project Goal

Our goal is to implement a HE encryption scheme proposed in literature [3] and analyze its performance. To our knowledge, no such scheme has been implemented and released publicly.

In section 5, we will analyze and discuss the tradeoff between security and speed, and we will evaluate what limits potential operations in such an HE application.

3 Background and Theory

In this section we will further discuss the particular HE scheme from [3] that we are using.

3.1 BGV encryption

CryptDB uses a HE scheme known as the Paillier cryptosystem to perform homomorphic adds on encrypted data [5, 8]. The encrypted value $Enc(PK, x+y)$ is simply $Enc(PK, x) * Enc(PK, y)$ in this scheme, making it quick to calculate, but this scheme has no way of determining $Enc(PK, x*y)$ without the system's secret key. Although this scheme may have worked for CryptDB's purposes, HE with both multiplicative and additive properties can perform more operations. Operations that CryptDB currently processes with other methods could also be processed by HE.

The BGV encryption scheme is not unlike the HE we discussed in class - it relies on lattice problems (over a polynomial ring of $\mathbb{Z}[x]/(x^d+1)$) on the hardness of Learning with Errors problem. Under the assumption the lattice, or system of equations, with errors (and an appropriate error distribution) is indistinguishable from a random distribution, this system acheives semantic security for encrypted values. This alone doesn't make BGV special, but BGV is special because of how it makes "Somewhat Homomorphic Encryption (SwHE)" more viable.

3.2 Somewhat Homomorphic Encryption

The noise that makes SwHE schemes like BGV work is both a blessing and a curse; while this error makes the system semantically secure, it also grows with addition operations and multiplies itself with each multiplication operations. Once this noise grows too large, the underlying encrypted value is lost. Before then, steps must be taken to decrease the noise if more operations must be performed on the data.

The traditional approach, and the only approach at the time of the CryptDB paper, was a "Fully Homomorphic Encryption (FHE)" technique called "bootstrapping." Bootstrapping reduces noise by recreating the initial equations; this process takes exponential time in the polynomial degree, and is a major factor in why CryptDB stated SwHE is cripplingly slow.

The BGV scheme, at a high level, takes a modulus switching approach that lets us decrease the absolute size of the error by switching our computations to a smaller modulus. By decreasing the absolute size of the error, we can replace the previous noise dependence on the degree of the polynomials being operated on into a dependence on the depth of the *circuits* (a set of homomorphic operations applied to our encrypted data for some computational purpose, such as those described in section 3.3) [6]. This lets us perform a number more operations on our data without the need to restore it from error. The BGV paper also discusses how to make this scheme FHE, but the SwHE approach is all we need for most practical applications.

3.3 Using SwHE to make circuit primitives

Given homomorphic addition and multiplication, we can come up with primitive circuits that allow us to perform a number of operations that are useful in the context of encrypted stored data.

These operations are equality, greater-than comparison, and integer addition. We provide the mathematical details of these operations below:

3.3.1 Equal

$$equal(E(x), E(y)) = \prod_{i=0}^{\mu-1} (1 + E(x)_i + E(y)_i)$$

This checks for encrypted data $E(x)$ and $E(y)$, whether plaintext values x and y are the same.

3.3.2 Greater-than Comparison

$$comp(E(x), E(y)) = E(c)_{\mu-1}$$
$$E(c)_i = (1 + E(x)_i) \cdot E(y)_i + (1 + E(x)_i + E(y)_i) \cdot E(c)_{i-1}$$
$$E(c)_0 = (1 + E(x)_0) \cdot E(y)_0$$

This checks for encrypted μ-bit data $E(x)$ and $E(y)$, whether plaintext value x is greater than plaintext value y. We can express the above equations in the following closed form:

$$E(c)_{\mu-1} = (1+x_{\mu-1}) \cdot E(y)_{\mu-1} + \sum_{i=0}^{\mu-2} (1+E(x)_i) \cdot E(y)_i \cdot \prod_{j=i+1}^{\mu-1} (1+E(x)_j+E(y)_j)$$

We can then make an optimization by parallelizing operations using the SIMD technique [7]. Particularly, through this parallelization, we can compute for all i the product $\prod_{j=i+1}^{\mu-1}(1+E(x)_j+E(y)_j)$ via $2\mu-4$ homomorphic multiplications. We can also compute for all i the product $(1+E(x)_i) \cdot E(y)_i$ via just one homomorphic multiplication.

3.3.3 Integer Addition

$$fadd_\mu(E(x), E(y)) = (E(s)_0, E(s)_1, \ldots, E(s)_{\mu-1})$$

$$E(s)_i = E(x)_i + E(y)_i + \sum_{j=0}^{i-1} (E(x)_j \cdot E(y)_j) \prod_{k=j+1}^{i-1} (E(x)_k + E(y)_k)$$

We observe that this is bitwise addition with carry bits, computed as shown by the sum with the nested product above, and returns $E(c)$ such that $E(c) = E(x) + E(y)$.

We can see that for a naive implementation of this circuit, we must perform $O(\mu)$ homomorphic additions and $O(\mu^3)$ homomorphic multiplications, so our complexity is $O(\mu \cdot A + \mu^3 \cdot M)$.

We observe that by parallelizing via SIMD, we can evaluate the entire circuit with $3\mu - 5$ homomorphic multiplications.

4 Implementation

We used HElib, a C++ open source HE primitive library, to handle the underlying HE scheme (BGV, as discussed in section 3.1). This way we could focus on implementing the higher-level circuits as described in [3] and section 3.3, which are what expand the potential data operations HE can perform on encrypted data.

The code samples below are from our implementation. These functions are all implemented as described in [3], so please consult section 3.3 for detailed discussion of the theory behind why they work.
None of the operations does not implement SIMD (single instruction, multiple data) techniques, although the Ctxt library does make such optimizations possible for performance enhancements.

4.1 Equality Test code

```
// a fuction that compares two ctxts,
// encrypted as arrays of binary ctxts
// returns 1 if equal, 0 otherwise
Ctxt equal_c(vector<Ctxt>  c1, vector<Ctxt> c2){
    // intialize
    Ctxt equal =  c1[0];
    equal.addCtxt(c2[0]);
    equal.addConstant(to_ZZX(1));

    // loop through all bits
    for(int i = 1 ; i < BITS; i++) {
    Ctxt temp =  c1[i];  //+ c2[0] + onecv[0];
     temp.addCtxt(c2[i]);
    temp.addConstant(to_ZZX(1));
    equal.multiplyBy(temp);
    }
    return equal;
}
```

4.2 Comparison Test code

```
Ctxt cmpr_c(vector<Ctxt>  x, vector<Ctxt> y){
    // intialize
    vector<Ctxt> cv;
    Ctxt c = x[0];
    c.addConstant(to_ZZX(1));
    c.multiplyBy(y[0]);
    cv.push_back(c);

    // loop through all bits
    for(int i = 1 ; i < BITS; i++) {
        Ctxt temp = x[i];
        temp.addConstant(to_ZZX(1));
        temp.multiplyBy(y[i]);

        Ctxt temp2 = x[i];
        temp2.addConstant(to_ZZX(1));
        temp2.addCtxt(y[i]);
        temp2.multiplyBy(cv[i-1]);

        temp.addCtxt(temp2);

        cv.push_back(temp);
    }
    return cv[BITS-1];
}
```

4.3 Full Adder Addition Test code

```
//The bitwise product encrypted cyphertexts x and y,
//as described in section 3.3 of [3].
Ctxt t_c(vector<Ctxt>  x, vector<Ctxt> y,int i, int j){
    Ctxt t_i_j = x[j];
    t_i_j.multiplyBy(y[j]);
    for(int k = j+1; k < i; k++)
    {
        Ctxt temp = x[k];
        temp.addCtxt(y[k]);
        t_i_j.multiplyBy(temp);
    }
    return t_i_j;
}

vector <Ctxt> fadd_c(vector<Ctxt>  x, vector<Ctxt> y){
    // initialize
    vector<Ctxt> cv;
    vector<vector <Ctxt>> t;
    for(int i = 0; i < BITS; i++){
        vector<Ctxt> temp;
        for(int j = 0; j < BITS; j++){
            Ctxt t_ij = t_c(x,y,i,j);
            temp.push_back(t_ij);
        }
        t.push_back(temp);
    }

    Ctxt s_0 = x[0];
    s_0.addCtxt(y[0]);
    cv.push_back(s_0);

    for(int i = 1 ; i < BITS; i++) {
        Ctxt s_i = x[i];
        s_i.addCtxt(y[i]);

        Ctxt t_i = t[i][0];
        for(int j = 1; j < i; j++){
            t_i.addCtxt(t[i][j]);
        }
        s_i.addCtxt(t_i);
        cv.push_back(s_i);
    }
    return cv;
}
```

5 Future Work

5.1 Potential circuit primitive uses

While it is out of the scope of the project discussed in this paper, we can use our circuit primitives to build more complicated functions that are more directly applicable to encrypted database operations.

For example, we can implement a simple selection query, where we search for all values from a series of N encrypted entries $A^(i)$ for which a certain property $A_0^{(i)}$ matches a desired α as follows. We go through our N entries, and perform the following operation with our equality primitive.

$$select(E(\alpha), E(A^{(i)}) = equal(E(A^{(i)})_0, \alpha) \cdot (E(A^{(i)})_1, E(A^{(i)})_2, \ldots)$$

In fact, we can directly build a variety of database operations directly, including, but not limited to conjunctive and disjunctive select queries, search-and-count queries, search-and-avg queries, and search-and-max queries [3].

As this is the case, it seems possible to use our primitives to build a SwHE-based encrypted database or even modify a current database such as CryptDB for increased security while maintining performance.

6 Conclusions and Reflections

Upon running some performance tests on our HE operations on 16 bit values (on a VM single core laptop), we obtained the following results:

Operation	Time in ms
Equality Circuit	$\sim 800ms$
Comparison Circuit	$\sim 2500ms$
Addition Circuit	$\sim 19000ms$

As it turns out, these three basic operations run a lot slower when you implement HE security (approximately three orders of magnitude slower than basic equality, comparision addition); this gives us an insight into what the tradeoff is between speed and security. We observe the difference especially in the addition circuit, as with our naive implementation, we must perform a polynomial number of multiplications. It's clear that homomorphic encryption (even in the somewhat homomorphic variety) is too slow to be used for practical applications.

However, as noted in Section 3.3, with SIMD (single instruction, multiple data), we can reduce comparison to a constant number of homomorphic multiplications, and addition to a linear number of homomorphic multiplications. Indeed, upon testing, we can decrease the execution time of the addition circuit to $\sim 300ms$ with the SIMD optimization. We see that in order for any form of homomorphic encryption to be viable in a practical application, SIMD will be crucial to its efficiency and execution.

In this paper we've reviewed the concept of homomorphic encryption, its benefits, and also how it can be applied to a database system to provide an extra level of security. We examined previous literature on the topic (in particular the BGV encryption that CryptDB uses), and settled on using Somewhat Homomortphic Encryption as a tradeoff between speed and security. Using the Somewhat Homomorphic operations of addition multiplication, we recreated the primitive circuits that form the basis of a database system, and implemented them using the C++ library, HElib.

While we have had moderate success in creating the basics of our own SwHE database, there is still much work out there to be done. The field of homomorphic encryption and its applications is still young, and much more dev is still required in this subject.

7 References

[1] HELib, a GitHub repo with many useful low-level homomorphic encyption functions (in C++): https://github.com/shaih/HElib

[2] Applications of SwHE: https://www.microsoft.com/en-us/research/wp-content/uploads/2011/05/ccs2011_submission_412.pdf

[3] A proposed SwHE implementation, including a good literature review and discussion of searchable encryption: https://eprint.iacr.org/2014/812.pdf

[4] Craig Gentry's 2009 PhD Thesis, which set the groundwork for HE beyond theory by creating the first functional (albeit inefficent) FHE: https://crypto.stanford.edu/craig/craig-thesis.pdf

[5] Popa et al. "CryptDB: Protecting Confidentiality with Encrypted Query Processing." http://web.cs.ucdavis.edu/˜franklin/ecs228/2013/popa_etal_sosp_2011.pdf

[6] Brakerski, Gentry, and Vaikuntanathan. "Fully Homomorphic Encryption without Bootstrapping." https://eprint.iacr.org/2011/277.pdf

[7] Smart and Vercauteren. "Fully Homomorphic SIMD Operations." http://eprint.iacr.org/2011/133/20110803:143250

[8] Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes." http://www.cs.tau.ac.il/ fiat/crypt07/papers/Pai99pai.pdf