Secure Computation with HELib

Posted 13 May 2013 by Thomas M. DuBuisson

Homomorphic encryption is a hot new area of study and discussion in the world of cryptography. Using homormorphic encryption it is possible to produce ciphertexts on which an arbitrary third party (ex: Amazon) can perform computations despite the encryption. That is to say, if you have a secret (pt) and homomorphically encrypt it then an operation f can be performed on the cipher text such that $decrypt(f(encrypt(pt))) = f(pt)$.

This result is truly stunning, but singing the praises, perils, and history of fully, somewhat, semi, or leveled homomorphic encryption is not what this post is about. You can Google for the cool implications but today I'm going to talk about a particular open-source homomorphic encryption library (written in C++) called HELib.

What is HELib?

HELib implements a homormophic encryption scheme. This library is open source on github and, from a popularity perspective, has really taken off for an obscure crypto library - having over 500 watchers.

Unlike some earlier HE schemes, HELib uses a SIMD-like optimization known as ciphertext packing. As a result, each individual ciphertext element with which you can perform a computation (addition or multiplication) is conceptually a vector of encrypted plaintext integrals. Thus, this scheme is particularly effective with problems that can benefit from some level of parallel computation.

Example Use of HELib

Folks on github have asked Shai to produce a tutorial showing how to take HELib and perform basic computations such as $2 + 2$. As awesome as Shai and Victor are, I don't expect them to find time to fill this request. Hopefully this section of the blog will suffice for most people. This is more of an HELib quick start guide and absolutely not an FHE primer, there is lots of text on FHE but not much on getting started with HELib.

Lets start with an example that fills two vectors with integrals, encrypts them homomorphically, and performs component-wise addition and multiplication. This process is actually more involved, the real steps we will go through are: declare our parameters (plaintext space, levels, columns, secret key haming weight, security), generate a secret key, obtain an `EncryptedArray`, which is a class that aids in later computations, encrypt our vectors, perform addition and multiplication, decrypt the results and print.

Without further ado, lets dive in. NOTE: all this code is based on one of HELibs 'Test_*.cpp' examples.

First some boiler-plate that is already discussed or self explainitory.

```
#include "FHE.h"
#include "EncryptedArray.h"
#include <NTL/lzz_pXFactoring.h>
#include <fstream>
```

```
#include <sstream>
#include <sys/time.h>

int main(int argc, char **argv)
{
    /* On our trusted system we generate a new key
     * (or read one in) and encrypt the secret data set.
     */

    long m=0, p=2, r=1; // Native plaintext space
                        // Computations will be 'modulo p'
    long L=16;          // Levels
    long c=3;           // Columns in key switching matrix
    long w=64;          // Hamming weight of secret key
    long d=0;
    long security = 128;
    ZZX G;
    m = FindM(security,L,c,p, d, 0, 0);
```

Notice the parameter names are pretty consistent in HELib examples as well as the literature. In this case, I am building for GF(2) - so my homormorphic addition is XOR and multiplication is AND. Changing this is as easy as changing the value of `p`. Folks wanting to see `2+2=4` should set p to something that matches their desired domain, such as 257 to obtain 8 bit Ints.

NOTE: I ran into a bug with `p=65537` and had to manually set `m` to an acceptable value, 65539, because `FindM()` failed.

Now lets use these parameters to build a private key - HELib is evidently asymmetric in addition to being homomorphic, wow!

```
    FHEcontext context(m, p, r);
    // initialize context
    buildModChain(context, L, c);
    // modify the context, adding primes to the modulus chain
    FHESecKey secretKey(context);
    // construct a secret key structure
    const FHEPubKey& publicKey = secretKey;
    // an "upcast": FHESecKey is a subclass of FHEPubKey

    //if(0 == d)
    G = context.alMod.getFactorsOverZZ()[0];

   secretKey.GenSecKey(w);
   // actually generate a secret key with Hamming weight w

   addSome1DMatrices(secretKey);
   cout << "Generated key" << endl;
```

We have now generated a secret key. Notice the public key was extracted from the private key - this is C++, that isn't just an alias. Interestingly, a public key in this context need only be an encryption of "1".

Lets get our helper class instantiated:

```
   EncryptedArray ea(context, G);
   // constuct an Encrypted array object ea that is
   // associated with the given context and the polynomial G

   long nslots = ea.size();
```

I find having an `nslots` value around is useful. You can't pick the number of slots (plaintext elements) a ciphertext can hold, but you can learn what parameters control this value and try to optimize for your situation. Now for encryption.

```
    vector<long> v1;
    for(int i = 0 ; i < nslots; i++) {
        v1.push_back(i*2);
    }
    Ctxt ct1(publicKey);
    ea.encrypt(ct1, publicKey, v1);

    vector<long> v2;
    Ctxt ct2(publicKey);
    for(int i = 0 ; i < nslots; i++) {
        v2.push_back(i*3);
    }
    ea.encrypt(ct2, publicKey, v2);
```

Well that was just some verbose C++ with calls to `EncryptedArray::encrypt()` hidden within. Hopefully by this point the notion has sunk in that we are talking about a whole vector of plaintext values (`v1` , `v2`) that are placed into individual ciphertexts (`ct1` , `ct2`).

Lets see some SIMD style computation!

```
    // On the public (untrusted) system we
    // can now perform our computation

    Ctxt ctSum = ct1;
    Ctxt ctProd = ct1;

    ctSum += ct2;
    ctProd *= ct2;
```

Well damn, that was anti-climactic. When you think about it, it should be almost disappointingly easy if this library is as awesome as I hope I hyped you up to expect.

Finally, we will decrypt the sum and product results:

```
    vector<long> res;
    ea.decrypt(ctSum, secretKey, res);

    cout << "All computations are modulo " << p << "." << endl;
    for(int i = 0; i < res.size(); i ++) {
        cout << v1[i] << " + " << v2[i] << " = " << res[i] << endl;
    }

    ea.decrypt(ctProd, secretKey, res);
    for(int i = 0; i < res.size(); i ++) {
        cout << v1[i] << " * " << v2[i] << " = " << res[i] << endl;
    }

    return 0;
}
```

And that's it - under one hundred lines and you have a program that can add, multiply, subtract, and negate (see the header files) data without even knowing the values.

Performance

The performance of addition and multiplication varies by level, security, and plaintext characteristics

(ex: word size). For the example, I get timings of:

Modulus	Number of Slots	Time for addition (ms)	Time for multiplication (ms)
257	44	0.7	39
8209	22	0.7	38
65537	2	2.9	177

It is worth once-again pointing out that this is running on top of NTL, which isn't thread safe. While you get some "parallelism" from the ciphertext packing you can't take easy advantage of multiple cores without multiple processes.

Noise in the CipherText

Each operation, but most notably multiply, results in a ciphertext that is more "noisy" than its inputs. Too much noise and your decryption will fail (producing results that don't match your intended computation). Thus, FHE computations have a notion of depth much like a circuit depth. If you exceed a given depth then the noise inherent to the computation will become great enough that decryption can result in errors. The trick is to know your current depth, for any given ciphertext, and perform a `recrypt` operation that produces a ciphertext of identical meaning but with reduced noise. However, recrypt is computationally expensive so performing it only when needed is critical.

One way to reduce the number of recrypt operations is to increase the "levels" parameter (`L` in the above example). There is a trade-off with the depth and how often you recrypt - computing over ciphertexts that support more levels is itself more computationally expensive.

HELib doesn't currently support a recrypt operation, so you are simply not able to perform computations of significant depth - saving you from the misery of determining a good balance between levels and recryption. This does, however, make noise management critical as you simply are not able to perform computations that require significant depth.

Noise Management

I just told you that FHE operations produce ciphertexts that build up noise and this noise prevents decryption. Furthermore, in HELib (as it stands today) this causes a limit to the depth of the computations you can practically perform. As a result, you, the programmer, must be intelligent with respect to your algorithms and how they combine ciphertexts.

For example, if you wish to acquire the product of an array of ciphertexts (which are themselves arrays of slots) you could perform what is known in functional programming as a fold - a for loop taking a zero element and adding each element in the array. An implementation can be seen below, this is horrible as it causes $O(n)$ noise, where n is the length of the array.

```
    int mulEntireArray( Ctxt &res
                      , const vector<Ctxt> input
                      , const EncryptedArray &ea)
    {
        const int nrElem  = input.size();
        if(nrElem > 0) {
            res = input[0];
            for(int i = 1; i < nrElem; i++) {
                res.multiplyBy(input[i]);
            }
            return 0;
        }
```

```
        return (-1);
    }
```

Instead you should use a divide and conquer algorithm, cutting the array in half in each iteration and multiply the two haves, terminating with an array of length one. This results in $O(lg(n))$ noise. Below is a display version that uses tail calls, likely not using memory in anything approaching a wise manner in C++ but it makes for easy reading.

```
    int mulEntireArray_logNoise( Ctxt &res
                               , const vector<Ctxt> input
                               , const EncryptedArray &ea)
    {
        const size_t nrElem  = input.size();
        if(nrElem > 1) {
            vector<Ctxt> tmp( (nrElem+1)/2
                            , Ctxt(input[0].getPubKey()));
            const size_t sz = tmp.size();
            for(int i=0 ; i < sz; i++) {
                tmp[i] = input[i];
            }
            for(int i=0 ; i < sz && (sz + i) < nrElem; i++) {
                tmp[i].multiplyBy(input[sz + i]);
            }
            mulEntireArray_logNoise(res, tmp, ea);
        }
        else
            res = input[0];
        return 0;
    }
```

The important part of the above is that `tmp` copies the first half of input and multiplies by the second half, terminating when it has a one element vector.

Minor Details: Building and the Like

HELib currently exists only as a git repository, there is no distribution package or Windows installer. What's more, it depends on a new version of NTL (6.0) that is not packaged by many distributions. To get going, you are going to need to download and install NTL, git clone HELib, build HELib remembering to link it against the new NTL and then build your application.

For *nix users, the NTL portion should be as simple as:

```
    wget http://www.shoup.net/ntl/ntl-6.0.0.tar.gz
    tar xzf ntl-6.0.0.tar.gz
    cd ntl-6.0.0
    ./configure && make && sudo make install # or something like this
```

Then to get and build HELib:

```
    git clone git://github.com/shaih/HElib.git
    cd HElib/src
    # Edit the Makefile as needed
    # It already had the correct -I and -L paths for me
    make
```

Now you can write your application and compile it via:

```
    g++ App.cpp $HELib/src/fhe.a -I$HELib/src -o App \
                                 -L/usr/local/lib -lntl
```

Future

HELib is a young library in a rather young field of cryptography. It blows my mind that the library is so well formed at this point. Hopefully we will see serious contributors and an active community.

If anyone is interested in either making Haskell bindings or porting it to using something thread safe (not NTL) then I'd be interested and would likely contribute, so get in touch.
