Outsourced Bits

A research blog on cloud computing, cryptography, security, privacy, …

Applying Fully-Homomorphic Encryption (Part 1)

This entry was posted on June 26, 2012, in Cloud Computing, Crypto Design, Crypto Theory and tagged FHE. Bookmark the permalink. 5 Comments

In

(https://cloudcrypto.files.wordpress.com/2012/06/holygrail.jpg)
The Holy Grail

2009, Craig Gentry published a paper showing—for the first time—how to construct a fully-homomorphic encryption (FHE) scheme. This was a landmark event in cryptographic research that will eventually have huge practical implications for security and privacy. An often cited (especially by the press) application of FHE is cloud computing. Unfortunately, few (if any) details are usually given as to how exactly FHE is useful for cloud computing.

This is the first part in a series of posts that will focus on *applications* of FHE. What it can and can't do (and why) and how exactly it might be used in the cloud. But before we can discuss applications, we first have to understand exactly what FHE is.

What is FHE?

Throughout, I'll assume familiarity with public-key encryption (http://en.wikipedia.org/wiki/Public-key_cryptography).

A homomorphic encryption (HE) scheme encrypts data in such a way that computations can be performed on the encrypted data without knowing the secret key. So, given two encryptions $c_1 = E_{pk}(m_1)$ and $c_2 = E_{pk}(m_2)$ of messages m_1 and m_2 under public key pk, a HE scheme allows anyone to compute an encryption $E_{pk}(m_1 \otimes m_2)$ without needing to decrypt either c_1 or c_2. Here $\otimes$ denotes some arbitrary operation (as we'll see there are many HE schemes each supporting different operations over the ciphertexts).

As an example, consider the RSA (http://en.wikipedia.org/wiki/RSA_(algorithm)) public-key encryption scheme which produces ciphertexts of the form $m^e \mod N$, where m is the message, e is the public key and N is the product of two primes. Given two encryptions $c_1 = m_1^e \mod N$ and $c_2 = m_2^e \mod N$ it is easy to compute an encryption of $m_1 \times m_2$ by computing:

$$c_1 \times c_2 = m_1^e \times m_2^e = (m_1 \times m_2)^e \mod N.$$

This example shows that RSA supports multiplications over encrypted data, i.e., given the encryptions of two messages anyone can compute the encryption of their product [1]. We therefore say that RSA is *multiplicatively homomorphic*. It turns out that many public-key encryption schemes are homomorphic including El Gamal which is also multiplicatively homomorphic, Goldwasser-Micali which is XOR (http://en.wikipedia.org/wiki/Exclusive_or) homomorphic (i.e., supports XORs), and the Benaloh and Paillier schemes which are both additively homomorphic (i.e., support additions). If you' re curious about these schemes and how they work see this (http://en.wikipedia.org/wiki/Privacy_homomorphism) page.

So we've known how to do addition, multiplications and XOR over encrypted data for a long time and even being able to perform these simple operations has been tremendously useful (e.g., for doing things like electronic voting (http://web.mit.edu/6.857/OldStuff/Fall02/handouts/L15-voting.pdf), secure multi-party computation (http://en.wikipedia.org/wiki/Secure_multi-party_computation) and private information retrieval (http://en.wikipedia.org/wiki/Private_information_retrieval)). Notice, however, that all these schemes support only a *single* operation. For example, the Benaloh scheme only supports additions but not multiplications or XOR. Similarly, El Gamal supports multiplications but not additions.

It goes without saying that ideally we would like to support as many operations as possible so that we can perform a varied set of computations on encrypted data. It turns out, however, that as long as we can support two specific operations—namely, addition and multiplication—we can support any operation! [2]

And this is exactly what a fully-homomorphic encryption scheme is: an encryption scheme that supports *any* computation on encrypted data.

Cryptographers have long wondered whether it was possible to construct a FHE scheme. In 2005, Boneh, Goh and Nissim came pretty close by constructing a scheme that could support any number of additions and *one* multiplication. This was a real breakthrough and gave some hope that perhaps FHE could be achieved. Note that while this scheme was not fully-homomorphic it already enabled a host of interesting applications.

Then, in 2009, Gentry finally figured out how to construct a FHE scheme. It's hard to overstate how important this was for cryptography. FHE was one of those crazy ideas that cryptographers dreamed about but that always remained out of reach. I don't think I' m exaggerating when I say that almost anyone who has ever tried to design a cryptographic primitive or protocol and failed thought at some point: "I'd be able to make it work if only I had a FHE scheme". FHE is such a powerful primitive that it has forced researchers to re-think almost every area of cryptography in order to figure out what is now possible. As Barak and Brakerski put it in their overview (http://windowsontheory.org/2012/05/01/the-swiss-army-knife-of-cryptography/) of FHE: FHE is the swiss-army knife of cryptography.

How does it Work?

Current FHE schemes are relatively complex [3] so I won't go into the details of how they work. Also, there are very good high-level descriptions if you are interested in more details. This (http://windowsontheory.org/2012/05/02/building-the-swiss-army-knife/) blog post by Barak and Brakerski is highly recommended. Once you've read that you might want to check out Gentry' s overview (http://crypto.stanford.edu/craig/easy-fhe.pdf) of his FHE construction and Vaikuntanathan' s survey (http://www.cs.toronto.edu/~vinodv/FHE-focs-survey.pdf) on the more recent developments in the area. Also, Gentry's thesis (http://crypto.stanford.edu/craig/) provides a very good introduction to HE, FHE and the high level ideas he developed to construct the scheme.

Here, I just want to give a high-level overview of what is referred to as Gentry's blueprint for constructing FHE schemes.

Somewhat homomorphic encryption (SHE). The first step is to design a SHE scheme which is a scheme that supports *some* computations over encrypted data. Gentry then showed that if you can manage to design a SHE scheme that supports the evaluation of its own decryption algorithm (and a little more) then there is a general technique to transform the SHE scheme into a FHE scheme. A SHE that can evaluate its own decryption algorithm homomorphically is called *bootstrappable* and the technique that transforms a bootstrappable SHE scheme into a FHE scheme is called *bootstrapping*.

Bootstrapping. So how does bootstrapping work and why is bootstrappability such a useful property? To understand this, you first have to know how the currently-known SHE schemes work. Roughly speaking, the ciphertexts of all these schemes have noise inside of them and unfortunately this noise gets larger as more and more homomorphic operations are performed. At some point there is so much noise that the encryptions becomes useless (i.e., they do not decrypt correctly). This is the main limitation of SHE schemes and this is the reason that they can only perform a restricted set of computations. Bootstrapping allows us to control this noise [5].

The idea is to take a ciphertext with a lot of noise in it and an encryption of the secret key and to homomorphically decrypt the ciphertext. Note that this can only work if the SHE scheme has enough homomorphic capacity to evaluate its own decryption algorithm which is why we need the SHE scheme to be bootstrappable. This homomorphically computed decryption will result in a new encryption of the message but without the noise (or at least with less noise than before). More concretely, say we have two ciphertexts

$$c_1 = E_{pk}(m_1) \text{ and } c_2 = E_{pk}(m_2)$$

with noise n_1 and n_2, respectively. We can multiply these encryptions using the homomorphic property of the SHE scheme to get an encryption

$$c_3 = E_{pk}(m_1 \times m_2)$$

of $m_1 \times m_2$ under key pk but c_3 will now have noise $n_1 \times n_2$. The idea behind bootstrapping is to get rid of this noise as follows. First, we encrypt c_3 and sk under pk [4]. This results in two new ciphertexts

$$c_4 = E_{pk}(c_3) = E_{pk}(E_{pk}(m_1 \times m_2)) \text{ and } c_5 = E_{pk}(sk).$$

Given c_4 and c_5 we now *homomorphically* decrypt c_4 using c_5. In other words, we compute the following operation over c_4 and c_5: “decrypt $c_3 = E_{pk}(m_1 \times m_2)$ using sk “. This is allowed since the scheme has enough homomorphic capacity to evaluate its own decryption algorithm.

By using this technique throughout a computation whenever the ciphertexts get too noisy, we can remove the main limitation of the SHE scheme and turn it into a FHE scheme.

It turns out that constructing a bootstrappable SHE scheme is difficult. To do this, Gentry had to build his scheme using sophisticated techniques [6] so a lot of the recent work in FHE has tried to figure out how to design simpler bootstrappable SHE schemes. Vaikuntanathan’s survey (http://www.cs.toronto.edu/~vinodv/FHE-focs-survey.pdf) gives a good overview of the latest results in this direction.

In the next posts, we’ ll start discussing applications.

Notes

[1] Note that as described here RSA is technically not really secure (see here (http://en.wikipedia.org/wiki/RSA_algorithm#Attacks_against_plain_RSA) for more details) and if we were to describe the secure version of RSA then it would not be homomorphic anymore.

[2] Matt Green’s post (http://blog.cryptographyengineering.com/2012/01/very-casual-introduction-to-fully.html) gives a good overview of why this is the case.

[3] Actually current schemes are not that complex in the sense that their correctness and security can be verified relatively easily. Developing an intuition as to *why* they work and why they were designed the way they were, however, is not as easy.

[4] For typical encryption schemes, encrypting the secret key sk under the public key pk is not secure. There are specially-designed schemes for which this is allowed and Gentry provides in his thesis an argument as to why his construction is likely to satisfy this requirement.

[5] It's important to note that the noise is crucial to the security of the schemes. In other words, with bootsrapping we don't want to get rid of the noise completely, just reduce it once it becomes too difficult to deal with.

[6] Gentry's construction is lattice-based. Lattice-based schemes have several nice properties, one of them being that their decryption algorithms tend to be "simple" in the sense that they require a relatively small number of multiplications and additions. This makes them ideal for building bootstrappable SHE schemes. Unfortunately, using lattices was not enough to achieve bootstrappability so Gentry introduced a general technique to simplify the decryption algorithm further. This technique is called *squashing* and relies on non-standard assumptions so recent work has tried to remove the need for squashing altogether.

5 thoughts on "Applying Fully-Homomorphic Encryption (Part 1)"

1. **Leo says:**
   July 1, 2012 at 6:19 am
   This part only describes the definition of FHE and the general idea about construction. Waiting for part two!

   Reply
2. Pingback: Applying Fully-Homomorphic Encryption (Part 2) « Outsourced Bits

3. Pingback: How to Search on Encrypted Data (Part 4): Oblivious RAMs | Outsourced Bits

4. Pingback: Applied Crypto Highlights: Searchable Encryption with Ranked Results | Outsourced Bits

5. **hsk81 says:**
   December 2, 2015 at 12:08 am
   Can ElGamal made be additive using following complex numbers: M=EXP{i*c*m}, where c is some coefficients where c is e.g. (A) pi or (B) pi/2. Now you'd feed M1=EXP{i*c*m} and M2=EXP{i*c*m} ElGamal, receiving M1*M2=EXP{i*c*[m1+m2]}. Now you essentiall just take the logarithm and divide by i*c to get m1+m1=LOG{M1*M2}/(i*c).

   To avoid real numbers, and make the LOG{M1*M2} feasible, using c=pi or c=pi/2 helps since EXP{i*pi/2*(m=0)}=1 and M=EXP{i*pi/2*(m=1)}=i, or M=EXP{i*pi*(m=0)}=+1 and EXP{i*pi*(m=1)}=-1.

   Using these alternative encoding we can then do a reverse lookup of get from M1*M2 to m1+m2:

   (A) For M1*M2 in {+1=1*1,i=1*i,i=i*1,-1=i*i} we could deduce m1+m2 to be {0, 1, 1, 0=2(mod2)}, or

   (B) For M1+M2 in {+1=1*1,-1=1*(-1),-1=(-1)*1,+1=(-1)*(-1)} we could deduce m1+m2 to be {0, 1, 1, 0=2(mod2)}.

   In the former case (A) we would retain the information about the carry bit when both m1=1 and m2=1, but would be required to extend ElGamal over integer valued complex numbers (Gaussian integers).

In the latter case (B) we would loose the carry bit information, but could directly apply ElGamal: m1+m2 would then not designate an addition but simply a (bitwise) OR operation, which in combination with proper multiplication m1*m2 in standard ElGamal could consist of a FHE system.

Reply