Outsourced Bits

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

Applying Fully-Homomorphic Encryption (Part 2)

This entry was posted on September 29, 2012, in Cloud Computing, Crypto Design, Crypto Theory and tagged FHE. Bookmark the permalink. 6 Comments

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

This is the second part of a series on applying fully-homomorphic encryption. In the first (http://outsourcedbits.org/2012/06/26/applying-fully-homomorphic-encryption-part-1/) post we went over what fully-homomorphic encryption (FHE) and shomewhat-homomorphic encryption (SHE) were and how they relate. In this post we' ll discuss actual applications.

To structure the discussion, I' ll refer to some applications as direct and others as indirect. Indirect applications will refer to applications where FHE is used as a building block—usually with other components—to construct something else of interest. Direct applications, on the other hand, will refer to applications where FHE is used (almost) "as-is". These are coarse and imprecise distinctions and are not particularly meaningful but they will be useful for organizational purposes. Roughly speaking, you can think about an indirect application as something mostly cryptographers would be excited about and a direct application as something everyone else might be excited about.

Quoting Barak and Brakerski (http://windowsontheory.org/2012/05/02/building-the-swiss-army-knife/), FHE is viewed by cryptographers as a swiss-army knife. This is because FHE has all kinds of indirect applications and can be used to construct and improve many cryptographic systems ranging from relatively simple things like encryption schemes to more complex things like secure multi-party computation (http://en.wikipedia.org/wiki/Secure_multi-party_computation) (MPC) protocols [1]. Gentry's thesis provides a good overview of the indirect applications of FHE, including to the design of one-time programs (http://www.mit.edu/~rothblum/papers/otp.pdf), public-key obfuscation (http://www.cs.ucla.edu/~rafail/PUBLIC/Ostrovsky-Skeith.html), proxy re-encryption (http://en.wikipedia.org/wiki/Proxy_re-encryption), software protection (http://www.cs.ucla.edu/~rafail/PUBLIC/09.pdf) and secure multi-party computation.

Here I' ll only cover a few possible applications of FHE. The first one is an indirect application of FHE to MPC while the rest will be direct applications.

Secure Multi-Party Computation

A MPC protocol allows mutually distrustful parties to cooperate securely. By this I mean that the parties—each with private data—can use a MPC protocol to evaluate any function f over their *joint* data without having to share the data with each other. Note that MPC can achieve this without hardware and without trusted third parties. MPC is a very general cryptographic technology and you can frame a huge number of security and privacy problems as instances of MPC. When trying to convey how general and useful it is I usually say that, roughly speaking, MPC is useful in any situation where you might think of using an NDA. Of course it is *much* more useful than NDAs because the parties will never see any of the data!

MPC is a very active area of research that started in the 80's and it is still going strong. In fact, recent work suggests that it could soon be practical. I would even venture to say that over the last few years, MPC has steadily moved from an area of theoretical cryptography research to an area of *applied* cryptography research.

So what does this have to do with FHE? It turns out that using FHE one can design very efficient MPC protocols (asymptotically). The protocols require little communication and interaction (the number of communication rounds between parties). But to see why this is the case, we first have to understand a little bit of how MPC protocols are typically designed.

Most protocols are based on (abstract) circuits (http://en.wikipedia.org/wiki/Circuit_(computer_theory)) as opposed to the more familiar computational models like random-access machines (http://en.wikipedia.org/wiki/Random-access_machine) (this is similar to how FHE works as discussed in the previous post). To use these protocols, the parties first construct a circuit that evaluates the function they wish to compute. They then use various techniques to jointly evaluate this circuit on their joint inputs. So, for example, if we consider the case of two-party computation where Alice and Bob wish to compute a function f on their respective inputs x and y, they first construct a circuit C that evaluates f on two inputs and then run the MPC protocol in order to securely compute $f(x, y)$.

By secure here what I mean is that Alice will not learn any information about y and Bob will not learn any information about x [2]. Now, there are many ways of performing this secure computation but they all have the following characteristics: either (1) they require Alice and Bob to send $\Omega(|C|)$ bits to each other, where $|C|$ denotes the size of C, i.e., its total number of gates; or Alice and Bob will need $\Omega(|C|)$ rounds of communication. This matters because when we' re working with circuits, the size of a circuit reflects the complexity of the function it computes. In other words, for "complicated" functions, Alice and Bob will have to exchange *a lot* of data and/or interact a large number of times.

With FHE, on the other hand, it is very easy to construct a MPC protocol without these limitations. To do this, Alice just needs to generate a public/private key pair for the FHE scheme. She then encrypts her input x and sends the resulting ciphertext

$$c_x = E_{pk}(x)$$

to Bob. Bob encrypts his input y, resulting in a ciphertext

$$c_y = E_{pk}(y)$$

and evaluates the circuit C on the encryptions on c_x and c_y, resulting in an encryption $c^\star$ of $f(x, y)$. Bob then sends $c^\star$ back to Alice who decrypts it and sends the result back to Bob [3].

The total amount of data exchanged between Alice and Bob in this protocol is

$$|c_x| + |c^\star| + |f(x, y)|,$$

where $|c_x|$, $|c^\star|$ and $|f(x, y)|$ refer to the bit length of c_x, $c^\star$ and $f(x, y)$, respectively. What's important to note here is that this is *independent* of the size/complexity of the circuit C! Also, the total number of rounds needed is 1.5, i.e., one round of communication plus one message. So, in other words, the FHE-based MPC protocol will have the same efficiency in terms of data exchanged and rounds, no matter how complicated the function is.

Another nice application of FHE in the context of MPC is that one can use it to design server-aided (or cloud-assisted) MPC protocols. A server-aided MPC protocol is like a regular MPC protocol except that the parties can make use of an untrusted party to outsource some of their computations. The point here is to decrease the computational burden of the parties at the expense of the server but, of course, without having to trust it.

This can greatly improve the efficiency of MPC as explored recently in a paper (http://eprint.iacr.org/2012/542) I co-authored with Mohassel and Riva. But as shown by Asharov et al in this paper (http://delta-apache-vm.cs.tau.ac.il/~tromer/papers/tfhe-mpc.pdf), if one uses FHE (plus some additional machinery) it is possible to design server-aided MPC protocols that are even more efficient from an asymptotic point of view (but unfortunately not from a concrete/practical point of view).

Delegated Computation

Of course, the most direct application of FHE is to the problem of outsourced computation. Here, Alice wants to evaluate a function f over her input x but she doesn't have enough resources to do the evaluation herself.

Because of this she wishes to outsource the computation to another party but she doesn't trust that party with her data.

FHE provides a perfect solution to this problem. Alice encrypts x and sends the ciphertext to the server who evaluates f on it and returns an encryption of $f(x)$. Alice can then decrypt it to recover $f(x)$. Note that similarly to the case of MPC this approach is only secure in the semi-honest model, where we assume the server will indeed evaluate the function f. What happens if the server evaluates some other function $f' \neq f$?

Fortunately, there are ways of handling this problem.

Search on Encrypted Data

Another application of FHE that is often cited is to the problem of searching on encrypted data. This is a special case of the delegated computation problem mentioned above where the client just wants to search through an encrypted file collection stored at the server. The idea is that Alice stores an encryption of her dataset (e.g., a collection of emails) on the server. Whenever she wants to search over her data, she sends an encryption of her keyword and the server homomorphically evaluates a search algorithm on the encrypted data and keyword.

While this obviously works, as far as I know, all the ways of using FHE to search on encrypted data require linear time in the length of the data. In other words, the keyword will have to be checked (homomorphically) against every word in the dataset. In practice, of course, linear-time search is inconceivable for many practical scenarios since more often than not search is a latency-sensitive operation. Just imagine if a search engine or a desktop search application used linear time search. In future posts, we' ll see how one can achieve *sub-linear* and even output-sensitive (http://en.wikipedia.org/wiki/Output-sensitive_algorithm) time search over encrypted data (albeit with a weaker security guarantee).

Since Gentry proposed his construction and blueprint in 2009, there has been a huge effort to make FHE more practical. While a lot of progress has been made, unfortunately, we' re still some way from truly practical FHE. So the natural question is: "where are the bottlenecks?".

As we discussed in the last post, most FHE schemes are based on Gentry's blueprint which consists of first constructing a SHE and then using Gentry's bootstrapping technique to turn it into a FHE scheme. It turns out that bootstrapping is a major bottleneck and that SHE is actually reasonably efficient. So if we care about practical applications, then it may be worthwhile to explore what exactly we can do with SHE instead.

Private Healthcare and Online Ads

This question was explored recently by Lauter, Naehrig and Vaikuntanathan in a paper (http://eprint.iacr.org/2011/405) where they consider various classes of applications and argue that SHE is enough for many of them. The first class is a multi-source and single-reader scenario where: (1) encrypted data is uploaded to the cloud by many different entities; (2) the cloud does some computation over the data; and (3) the encrypted answer is returned to the data owner/reader. One example of such a scenario is in healthcare where medical data pertaining to a given patient is sent to the cloud by many sources. These sources could be, for example, doctors or medical devices owned by the patient. All the data is sent encrypted and the cloud processes it homomrphically (e.g., it could run some statistical algorithm or classifier) and return the encrypted answer to the patient.

Another example explored in the paper is online ads. The idea here is that your mobile device could use SHE to encrypt and send private information to the cloud about your location, browsing history, emails etc. The cloud would store a set of ads encrypted under your key and then homomorphically run targeting ad algorithm on your data and the ads to figure out which ads to display to you. The result would be an encrypted ad that it could then return to you. Because your data is encrypted, the cloud will obviously never see your emails, location etc.

Machine Learning on Encrypted Data

In a more recent paper (http://eprint.iacr.org/2012/323), Graepel, Lauter and Naehrig study the more specific problem of machine learning over encrypted data. The setting here is the same as the healthcare scenario described above (i.e., the multi-source single-reader one) where we have doctors or perhaps medical devices sending encrypted data pertaining to a patient, say Alice, to the cloud. Here, Graepel et al. consider the setting of supervised learning (http://en.wikipedia.org/wiki/Supervised_learning) where, given a set of labeled training data, we want to derive a function that will accurately label future data, i.e., a classifier. As one can imagine, supervised learning has many applications including to bioinformatics, spam detection and speech recognition just to name a few.

So in this setting, the data sources (i.e., doctors or medical devices) send encrypted labeled data to the cloud under Alice's public key. The cloud then runs a machine learning algorithm homomorphically over all the data. This results in an encrypted classifier which Alice can now use in the following way. Whenever Alice wants to classify some data, she encrypts it under her public key and sends it to the cloud which evaluates the classifier on her data homomorphically, resulting in an encrypted labeling of her data. The cloud returns this to Alice who can then decrypt it using her secret key.

The high level idea is sound, but remember that in this work the authors are only interested in using SHE since it is more practical than FHE. Unfortunately, this turns out to be a serious limitation as it requires the algorithms to be low-degree polynomials and many are not. What is meant here is that if one views the learning and classifying algorithms as a single function, then that function has to be a low-degree polynomial.

This is pretty restrictive, but Graepel et al. still manage to show how one can homomorphically evaluate interesting and useful machine learning algorithms under this constraint. One example is Fisher's linear discriminant (http://en.wikipedia.org/wiki/Linear_discriminant_analysis) classifier [4]. As suggested in the paper, this this poses an interesting challenge for machine learning research: "is it possible to design good machine learning algorithms that can be computed using low-degree polynomials".

That's it for applications. In the next and final post in the series we' ll discuss the limitations of FHE and SHE and some alternatives.

Notes

[1] By simple and complex, here I am not referring to the complexity of constructing the objects (e.g., many encryption schemes, FHE included, are very difficult to construct) but to the complexity of the object's functionality or "behavior"

[2] Technically this is not true because the fact that Alice and Bob learn $f(x, y)$ means that they learn something about each other's input—namely they learn $f(x, y)$. But for the purposes of MPC this is not a problem because it is the best we can do. Indeed, if the goal is to construct a protocol where Alice and Bob will learn $f(x, y)$ then we cannot do so without them learning $f(x, y)$. So when we talk about the security of MPC, we usually say that Alice and Bob do not learn anything about each other's input that they cannot learn from $f(x, y)$

[3] I should stress that, as described, the protocol is only secure in a very restricted adversarial model known as the *semi-honest* model, where it is assumed that all the parties will follow the protocol but will try to learn whatever they can from the execution. This may seem like a weak and strange adversarial model at first but it is reasonable to consider for the following reasons. First, a protocol that is secure in this model can be safely used in any situation where the adversary is passive, i.e., where it only sees transcripts of the messages sent between parties. One could imagine an adversary that tries to recover information about the parties' inputs after the protocol was executed, for example, from a log of the messages. Second, there are general techniques that can transform any protocol that is secure in this model into a protocol that is secure in stronger and more natural models (e.g., where the adversaries does not have to follow the protocol).

[4] A note that SHE cannot handle division which may seem necessary for the classifiers considered in this work. Graepel et al., however, get around this by observing that the classifiers can be made to work as well by multiplying through with the denominators. [**Note**: in a previous version of this post I erroneously claimed that the client had to do the division.]

6 thoughts on "Applying Fully-Homomorphic Encryption (Part 2)"

1. Pingback: Applying Fully-Homomorphic Encryption (Part 1) « Outsourced Bits

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

3. **Wi says:**
   May 8, 2014 at 9:54 am
   It is very nice article, thank you. And I am interested in “Delegated Computation” application, can you refer me to some other web sites with more details and concrete examples.

   Reply

   senykam says:
   May 12, 2014 at 3:53 pm
   I don’t really know of other web sites that discuss FHE and it’s applications (including delegated computation) other than the Barak-Brakerski post on Windows on Theory that is linked. For more resources on FHE you can check out the links in the Reading List: http://outsourcedbits.org/reading-list/

   Reply
4. **vuonghv says:**
   April 7, 2015 at 7:59 pm
   I’m studying FHE for my graduation project, so this post is very useful and interesting for me. I have to bookmark this blog. Thank you.

   Reply
5. **Paradox (@KaiZhou5) says:**
   September 21, 2015 at 9:10 am
   Thank you for this nice illustration. I read your recommended blog on how to construct FHE. It seems that the essence of FHE is additive and multiplicative homomorphic encryption of a single bit. And the basic operation is a set of gates {AND, OR}. I am wondering can we build FHE on additive and multiplicative HE of element in a finite field and the basic operations are {addition, multiplication,inversion}. I thought it might increase the efficiency since we can operate on integers instead of bits.

   I don’t know if there are already some work on this idea. If there are, can you give me some reference?

   Reply