Fully Homomorphic Encryption (part II)

Notes by Alexandra Henzinger

MIT - 6.5610
Lecture 9 (March 4, 2024)

Warning: This document is a rough draft, so it may contain bugs. Please feel free to email me with corrections.

Logistics

- Pset 3 published tomorrow
- Project check-in due on 3/8
- Exciting speaker (Eric Rescorla, former CTO of Mozilla) at the security lunch on 3/7 at 12pm (in 32-D463)!

Outline

Today, we will cover:

- Review of the GSW levelled FHE scheme
    - Recap of construction
    - What depth circuits can this scheme evaluate?
- Bootstrapping GSW to support arbitrary-depth computations
- **Stretch break**
- FHE applications
- Open questions in FHE

Review: The GSW construction

We will begin by recapping the GSW construction, described in detail in last week's lecture notes. This construction gives a *levelled* FHE scheme: it can only evaluate bounded-depth Boolean circuits on encrypted data.

Specifically, given ciphertexts with B-bounded error, GSW encryption has the following error growth:

1. Homomorphic additions: after each add gate, the error roughly doubles in magnitude (i.e., it goes from falling in the range $[-B, \dots, B]$ to falling in the range $[-2B, \dots, 2B]$).
2. Homomorphic multiplications: after each multiplication gate, the error is multiplied by roughly $\ell = (n+1) \cdot \log q$, on LWE security parameter n and LWE modulus q. That is, the error goes from falling in the range $[-B, \dots, B]$ to falling in the range $[-\ell B, \dots, \ell B]$.

Brakerski and Vaikuntanathan show that the error growth incurred by homomorphic multiplications can be made even smaller when evaluating branching programs.

So, if we take the LWE error distribution χ to be B_0-bounded (i.e., the error in fresh ciphertexts falls in the range $[-B_0, \dots, B_0]$), then after homomorphically evaluating a depth-d Boolean circuit on ciphertexts, the error can at most fall in the range $[-\ell^d \cdot B_0, \dots, \ell^d \cdot B_0]$.

We can still decrypt GSW ciphertexts as long as the (absolute value of the) error is much smaller than $q/4$—this ensures that checking whether a value is "big" or "small" as part of Dec lets us decrypt correctly. As a result, we can still correctly decrypt ciphertexts after evaluating any Boolean circuit of depth up to d, as long as

$$\ell^d \cdot B_0 \ll q/4.$$

Or, equivalently:

$$((n+1) \cdot \log q)^d \cdot B_0 \ll q/4.$$

In other words, when working with a large enough modulus q, the depth we can support is roughly $d \approx n^{0.99}$. So, given any predetermined circuit of depth d, we can pick our LWE parameters to be large enough to let us homomorphically evaluate the circuit.

However, perhaps surprisingly, it is possible to do even better: in the rest of this lecture, we will see an absolutely beautiful idea to boost the GSW encryption scheme to compute arbitrary-depth circuits! As a result, the size of our LWE parameters will *not* have to grow with the complexity of the Boolean circuits that we want to evaluate.

Bootstrapping GSW to support arbitrary-depth computations

In his original FHE construction, Gentry introduced a brilliant technique to *refresh* ciphertexts, which lets us go from a ciphertext encrypting a message μ with a lot of noise (as a result of performing homomorphic computations) to a separate ciphertext encrypting μ with only a little noise (namely, the baseline level of noise needed for security) [1]. This technique is referred to as "bootstrapping."

Bootstrapping intuition. At a high level, bootstrapping is based on the following observation: by construction, the decryption algorithm $\mathsf{Dec}(\mathbf{s}, \mathbf{C})$ removes the noise from a ciphertext $\mathbf{C}$, given the corresponding secret-key vector $\mathbf{s}$. So, if the server was given the secret key $\mathbf{s}$, the server could run $\mathsf{Dec}(\mathbf{s}, \mathbf{C})$ on each ciphertext matrix $\mathbf{C}$ to recover the underlying message μ, then the server could build a new ciphertext $\mathbf{C}'$ that encrypts μ with the baseline level of noise by running $\mathsf{Enc}(\mathbf{s}, \mu)$, and finally the server could continue performing homomorphic operations on ciphertext $\mathbf{C}'$. Clearly though, this would be insecure: we cannot let the server learn the secret-key vector $\mathbf{s}$, otherwise it could decrypt all of the ciphertexts (exactly as we've suggested) and this would completely break security.

However, we can play a trick that is very similar to this approach: namely, we can have the user send the server the *encryption* of its secret-key vector. Then, the server can *homomorphically*—i.e., under encryption—evaluate the Dec algorithm on a ciphertext encrypting μ, as long as Dec has sufficiently low circuit complexity. The output of this computation will be an encryption of the same message μ, but with *low* error. We will now discuss this bootstrapping technique, as well as the assumption that is needed to make it work (circular security), in more detail.

Bootstrapping construction. To perform bootstrapping, we will think of the decryption algorithm, Dec, as a Boolean circuit, $\mathcal{C}_{\mathsf{Dec}}$. This circuit takes as input two parameters: the secret-key vector $\mathbf{s} \in \mathbb{Z}_q^{n+1}$ and a ciphertext matrix $\mathbf{C} \in \mathbb{Z}_q^{\ell \times (n+1)}$. Then, the circuit outputs the bit $\mu \in \{0, 1\}$ encrypted by the ciphertext $\mathbf{C}$.

Here, we are using the fact that we can write any polynomial-time computation, such as Dec, as a Boolean circuit that is also requires polynomial time to evaluate.

Now, in the setting we are working in, the server knows the full ciphertext $\mathbf{C}$ that it wants to bootstrap. So, we can think of this ciphertext $\mathbf{C}$ as *hard-coded* into the circuit $\mathcal{C}_{\mathsf{Dec}}$ that we are trying to evaluate. That is, we will denote $\mathcal{C}_{\mathsf{Dec},\mathbf{C}}$ as the circuit $\mathcal{C}_{\mathsf{Dec}}$ in which the input ciphertext $\mathbf{C}$ has been fixed. In other words, the modified circuit $\mathcal{C}_{\mathsf{Dec},\mathbf{C}}$ now takes as input *only* the secret-key vector $\mathbf{s} \in \mathbb{Z}_q^{n+1}$, and it spits out the message bit $\mu \in \{0, 1\}$ encrypted by the fixed ciphertext $\mathbf{C}$.

This new circuit $\mathcal{C}_{\mathsf{Dec},\mathbf{C}}$ is exactly what we will homomorphically evaluate. Namely, the user will include the *encryption* of each bit of its secret-key vector, $\mathbf{s} \in \mathbb{Z}_q^{n+1}$, as part of its evaluation key, ek. We can write this evaluation key as:

We are being slightly sloppy with notation here and directly encrypting $\mathbb{Z}_q$ values. In reality, the evaluation key here consists of the encryption of each bit in the bit-decomposition of each entry of $\mathbf{s}$.

$$\mathsf{ek} = (\mathsf{ct}_{\mathbf{s}_1}, \ldots, \mathsf{ct}_{\mathbf{s}_n}) = (\mathsf{Enc}(\mathbf{s}, \mathbf{s}_1), \ldots, \mathsf{Enc}(\mathbf{s}, \mathbf{s}_n)) .$$

Then, given a ciphertext $\mathbf{C}$ that has the maximal amount of allowable noise in it (such that it is still decryptable), the server will:

1. build the Boolean circuit $\mathcal{C}_{\mathsf{Dec},\mathbf{C}}$, which correspond to the decryption circuit with the ciphertext $\mathbf{C}$ hard-coded in it, and
2. homomorphically evaluate this decryption circuit to get the resulting ciphertext $\mathbf{C}'$:

$$\mathbf{C}' \leftarrow \mathsf{Eval}\left(\mathcal{C}_{\mathsf{Dec},\mathbf{C}}, \mathsf{ct}_{\mathbf{s}_1}, \ldots, \mathsf{ct}_{\mathbf{s}_n}\right) .$$

Here, we observe two crucial properties:

- We designed the circuit $\mathcal{C}_{\mathsf{Dec},\mathbf{C}}$ to output the message bit $\mu \in \{0, 1\}$ encrypted in the ciphertext $\mathbf{C}$. So, the new ciphertext $\mathbf{C}'$, which is the result of homomorphically evaluating $\mathcal{C}_{\mathsf{Dec},\mathbf{C}}$, will then be an encryption of μ.
- The amount of noise contained in the new ciphertext $\mathbf{C}'$ depends *only* on
  1. the noise contained in the encryptions $\mathsf{ct}_{\mathbf{s}_1}, \ldots, \mathsf{ct}_{\mathbf{s}_n}$ given as part of the evaluation key ek, and
  2. the depth of the decryption circuit $\mathcal{C}_{\mathsf{Dec},\mathbf{C}}$.

  So, since the encryptions given in the evaluation key ek are fresh (in that they have the minimum amount of required noise), the output ciphertext $\mathbf{C}'$ can also have low noise!

As a result, the ciphertext $\mathbf{C}'$ is an encryption of the same message as the ciphertext $\mathbf{C}$, but $\mathbf{C}'$ is guaranteed to have low noise! This is exactly what we set out to construct. However, there are two important caveats that we need to verify to make this technique work:

1. Decryption circuit complexity. For bootstrapping to be useful, the underlying levelled FHE scheme must be powerful enough to homomorphically evaluate (a) the decryption circuit $\mathcal{C}_{\mathsf{Dec},\mathbf{C}}$ and (b) at least one extra addition or multiplication gate. In other words, we need the decryption circuit to be "shallow" enough to fit into the function class supported by our levelled FHE scheme. Concretely, if the decryption circuit $\mathcal{C}_{\mathsf{Dec},\mathbf{C}}$ has depth d, we need the levelled FHE scheme to support computations of depth at least $d + 1$.

When this is the case, we can homomorphically evaluate any arbitrary-depth boolean circuit as follows:

1. *Homomorphic addition:* we replace each "ADD" gate by a homomorphic addition (i.e., $\mathbf{C} \leftarrow \mathsf{Eval}(``+", \mathbf{C}_1, \mathbf{C}_2)$ in the notation of last lecture), followed by a homomorphic decryption (i.e., $\mathbf{C}' \leftarrow \mathsf{Eval}(\mathcal{C}_{\mathsf{Dec},\mathbf{C}}, \mathsf{ek})$).

   At the end of this procedure, we have a ciphertext $\mathbf{C}'$ with noise in the range $[-\ell^d \cdot B_0, \cdots, \ell^d \cdot B_0]$, where d is the depth of the decryption circuit.

2. *Homomorphic multiplication:* we replace each "MUL" gate by a homomorphic multiplication (i.e., $\mathbf{C} \leftarrow \mathsf{Eval}(``\times", \mathbf{C}_1, \mathbf{C}_2)$), followed by a homomorphic decryption (i.e., $\mathbf{C}' \leftarrow \mathsf{Eval}(\mathcal{C}_{\mathsf{Dec},\mathbf{C}}, \mathsf{ek})$).

   At the end of this procedure, we again have a ciphertext $\mathbf{C}'$ with noise in the range $[-\ell^d \cdot B_0, \cdots, \ell^d \cdot B_0]$, where d is the depth of the decryption circuit.

By alternating computing on and refreshing the ciphertexts in this way, the noise in our ciphertexts will never grow too large. No matter the degree of our computation, we will always be able to decrypt.

In the case of GSW, the decryption circuit has depth $O(\log n)$. (This is depth stems from the comparison operations required to check whether each value is "big" or "small".) As we saw, the levelled version of GSW is powerful enough to support computations up to depth $n^{0.99}$. So we can indeed bootstrap!

2. Circular security. To preserve security, the server cannot learn any information about the secret key. Intuitively, the way we are achieving this is by *encrypting* the secret-key vector, under itself. Proving that is secure requires the additional assumption that GSW encryption is *circular secure*—that is, that publishing the ciphertexts that encrypt each bit of the secret key under itself, i.e.,

$$\mathsf{ct}_{\mathbf{s}_i} = \mathsf{Enc}(\mathbf{s}, \mathbf{s}_i) \text{ for } i \in [n],$$

hides the secret key.

In general, we do not know how to show that GSW is circular secure from just the LWE assumption. Building full FHE without the circular security assumption is still an open question and an active area of research!

In fact, there exist semantically-secure encryption schemes that are provably *not* circular secure.

Applications of FHE

Now that we have constructed FHE, we will discuss three particular applications that it lets us construct:

There are many, many more applications of FHE. Coming up with a new application might be a fun avenue for a class project!

1. *Private delegation.* In private delegation, a user wants to outsource the computation of some function to a powerful but untrusted server, without revealing its input to the computation.

Some examples of this are: a user may want to query a LLM on a sensitive prompt, without revealing her prompt. Or, a user may want to outsource the storage of her emails to a cloud server and to keep the contents of her emails hidden, while retaining the ability to search over them.

2. *Secure collaboration.* In secure collaboration, multiple users want to jointly evaluate a function on their hidden inputs, while revealing nothing but the output of the function.

   Some examples of this are: hospitals may want to collaborate to train machine learning models, while keeping sensitive patient data hidden. Or, banks may want to run an auction without revealing (or learning) each bidder's bid and each seller's price.

3. *Private database lookups.* Private database lookups are a generalization of private information retrieval (which we covered in lectures 5 and 6). In this setting, a user wants to make arbitrary queries to a remote database, while hiding her queries. With FHE, the server hosting the database can answer these queries under encryption.

   These private database queries could take many forms: for example, a user may want to make general SQL queries. Or, a user may want to make a private query to a web search engine (e.g., Google), without revealing her query string.

Open questions in FHE

The GSW scheme that we saw in these last two lectures is an incredibly powerful tool that can unlock an array of cryptographic applications. However, building and improving on know FHE constructions is an active and exciting area of research in which many interesting questions remain unanswered. Two such open questions are:

1. *Can we build FHE from assumptions other than lattices?*

   In cryptography, it is always good to construct any given primitive from a variety of assumptions—this gives us confidence that, even if any one assumption turns out to be broken, our primitive still exists. In the case of FHE, we (roughly) only know how to build it from lattice-based assumptions (e.g., LWE, ring-LWE). Building FHE from a number-theoretic assumption—for example, the hardness of factoring (e.g., RSA) or the hardness of discrete log (e.g., DDH)—would be a big research result!

2. *Can we make FHE concretely efficient and practical?*

While the construction of FHE that we covered today is possible in theory, it is still very far from concretely efficient in practice. In large part, this is due to the computational costs of

- representing computations as Boolean circuits, and
- incurring $O(\text{poly}(n))$ (or even just $O(\text{polylog}(n))$) overhead per gate.

The state-of-the-art in research today is that evaluating functions with low multiplicative degree (e.g., degree 2 or 3) and high additive degree can be concretely efficient. However, once we try to evaluate circuits with high multiplicative degree, the LWE parameters that we must use become large and computing on ciphertexts (and, in particular, bootstrapping them) becomes very expensive.

While FHE isn't used much in practice yet, building truly practical FHE would have an enormous impact on the world!

References

[1] Craig Gentry. A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University, 2009.