Implementing the FV FHE Scheme

Mark Schultz

University of California San Diego

10 May 2022

Contents

Who am I?

- 4th year PhD Student at U.C.S.D

Who am I?

- 4th year PhD Student at U.C.S.D
  - Working with Daniele Micciancio in Lattice-based Cryptography

Who am I?

- 4th year PhD Student at U.C.S.D
  - Working with Daniele Micciancio in Lattice-based Cryptography
- Want to get industry experience this summer

Who am I?

- 4th year PhD Student at U.C.S.D
  - Working with Daniele Micciancio in Lattice-based Cryptography
- Want to get industry experience this summer
  - Especially on implementations

The Challenge

- Implement a Fully Homomorphic Encryption Scheme!

The Challenge

- Implement a Fully Homomorphic Encryption Scheme!
  - The Fan-Vercauteren Scheme (FV) in particular

The Challenge

- Implement a Fully Homomorphic Encryption Scheme!
  - The Fan-Vercauteren Scheme (FV) in particular
- Implemented everything but Bootstrapping

Contents

What is FHE?

- PKE that supports *privacy homomorphism*.

What is FHE?

- PKE that supports *privacy homomorphism*.
  - Method to compute

$$(f, \mathsf{Enc}_{pk}(m_0), \mathsf{Enc}_{pk}(m_1)) \mapsto \mathsf{Enc}_{pk}(f(m_0, m_1))$$

What is FHE?

- PKE that supports *privacy homomorphism*.
  - Method to compute

$$(f, \mathsf{Enc}_{pk}(m_0), \mathsf{Enc}_{pk}(m_1)) \mapsto \mathsf{Enc}_{pk}(f(m_0, m_1))$$

  - $f \in \{+, \times\}$ is enough

What is FHE?

- PKE that supports *privacy homomorphism*.
  - Method to compute

$$(f, \mathsf{Enc}_{pk}(m_0), \mathsf{Enc}_{pk}(m_1)) \mapsto \mathsf{Enc}_{pk}(f(m_0, m_1))$$

  - $f \in \{+, \times\}$ is enough.
- Many early cryptosystems are *Partially Homomorphic*

What is FHE?

- PKE that supports *privacy homomorphism*.
    - Method to compute

$$(f, \mathsf{Enc}_{pk}(m_0), \mathsf{Enc}_{pk}(m_1)) \mapsto \mathsf{Enc}_{pk}(f(m_0, m_1))$$

    - $f \in \{+, \times\}$ is enough.
- Many early cryptosystems are *Partially Homomorphic*
- First FHE: Gentry 2009

What is FHE?

- PKE that supports *privacy homomorphism*.
  - Method to compute

$$(f, \mathrm{Enc}_{pk}(m_0), \mathrm{Enc}_{pk}(m_1)) \mapsto \mathrm{Enc}_{pk}(f(m_0, m_1))$$

  - $f \in \{+, \times\}$ is enough.
- Many early cryptosystems are *Partially Homomorphic*
- First FHE: Gentry 2009
  - Combines *Somewhat Homomorphic Encryption* and Bootstrapping.

What is FHE?

- PKE that supports *privacy homomorphism*.
  - Method to compute

$$(f, \mathsf{Enc}_{pk}(m_0), \mathsf{Enc}_{pk}(m_1)) \mapsto \mathsf{Enc}_{pk}(f(m_0, m_1))$$

  - $f \in \{+, \times\}$ is enough.
- Many early cryptosystems are *Partially Homomorphic*
- First FHE: Gentry 2009
  - Combines *Somewhat Homomorphic Encryption* and Bootstrapping.
- Also *Leveled FHE*.

What is FHE?

- PKE that supports *privacy homomorphism*.
  - Method to compute

$$(f, \mathsf{Enc}_{pk}(m_0), \mathsf{Enc}_{pk}(m_1)) \mapsto \mathsf{Enc}_{pk}(f(m_0, m_1))$$

  - $f \in \{+, \times\}$ is enough.
- Many early cryptosystems are *Partially Homomorphic*
- First FHE: Gentry 2009
  - Combines *Somewhat Homomorphic Encryption* and Bootstrapping.
- Also *Leveled FHE*.
  - FV is Leveled FHE.

What is FV?

- RLWE-based version of Brakerski's "Scale-Invariant" FHE scheme

What is FV?

- RLWE-based version of Brakerski's "Scale-Invariant" FHE scheme
- Alternatively, variant of the LPR Cryptosystem that uses relinearization-based multiplication

What is FV?

- RLWE-based version of Brakerski's "Scale-Invariant" FHE scheme
- Alternatively, variant of the LPR Cryptosystem that uses relinearization-based multiplication
  - I will explain things from this perspective.

Ring LWE

- Throughout, fix $R_q = \mathbb{Z}_q[x]/(2^d + 1)$ for $d = 2^n$, and $q \in \mathbb{N}$.

Ring LWE Assumption

For random $s \in R_q$, and a distribution χ supported on R_q, the RLWE assumption is that for $a(x) \leftarrow R_q, u(x) \leftarrow R_q$, $e(x) \leftarrow \chi$:

$$(a(x), a(x)s(x) + e(x)) \approx_c (a(x), u(x)).$$

Ring LWE

- Throughout, fix $R_q = \mathbb{Z}_q[x]/(2^d + 1)$ for $d = 2^n$, and $q \in \mathbb{N}$.

Ring LWE Assumption

For random $s \in R_q$, and a distribution χ supported on R_q, the RLWE assumption is that for $a(x) \leftarrow R_q, u(x) \leftarrow R_q$, $e(x) \leftarrow \chi$:

$$(a(x), a(x)s(x) + e(x)) \approx_c (a(x), u(x)).$$

- χ a discrete Gaussian of parameter σ.

LPR Cryptosystem

- KeyGen: $sk \leftarrow \chi$, and $pk = (a(x), a(x)s(x) + b(x))$ is an RLWE sample

LPR Cryptosystem

- KeyGen: $sk \leftarrow \chi$, and $pk = (a(x), a(x)s(x) + b(x))$ is an RLWE sample
- Encryption: Sample $u(x) \leftarrow R_q$, $e_0(x), e_1(x) \leftarrow \chi$, and outputs

$$u(x)\begin{pmatrix} a(x)s(x) + e(x) \\ -a(x) \end{pmatrix} + \begin{pmatrix} e_0(x) \\ e_1(x) \end{pmatrix} + \begin{pmatrix} \Delta m(x) \\ 0 \end{pmatrix}.$$

LPR Cryptosystem

- KeyGen: $sk \leftarrow \chi$, and $pk = (a(x), a(x)s(x) + b(x))$ is an RLWE sample
- Encryption: Sample $u(x) \leftarrow R_q$, $e_0(x), e_1(x) \leftarrow \chi$, and outputs

$$u(x)\begin{pmatrix} a(x)s(x) + e(x) \\ -a(x) \end{pmatrix} + \begin{pmatrix} e_0(x) \\ e_1(x) \end{pmatrix} + \begin{pmatrix} \Delta m(x) \\ 0 \end{pmatrix}.$$

  - Easily supports additive homomorphism

LPR Cryptosystem

- KeyGen: $sk \leftarrow \chi$, and $pk = (a(x), a(x)s(x) + b(x))$ is an RLWE sample
- Encryption: Sample $u(x) \leftarrow R_q$, $e_0(x), e_1(x) \leftarrow \chi$, and outputs

$$u(x)\begin{pmatrix} a(x)s(x) + e(x) \\ -a(x) \end{pmatrix} + \begin{pmatrix} e_0(x) \\ e_1(x) \end{pmatrix} + \begin{pmatrix} \Delta m(x) \\ 0 \end{pmatrix}.$$

  - Easily supports additive homomorphism
- Decryption: For a ciphertext $\begin{pmatrix} c_0(x) \\ c_1(x) \end{pmatrix}$, compute $c_0(x) + s(x)c_1(x)$, and perform simple error-correction procedure.

Relinearization

- Idea: view decryption $\mathrm{Dec}_s(c_0, c_1) = c_0 + c_1 s = f_{c_0,c_1}(s)$ as a degree-1 polynomial in s, where

$$c_0 + c_1 s = \Delta m + e(x).$$

Relinearization

- Idea: view decryption $\mathrm{Dec}_s(c_0, c_1) = c_0 + c_1 s = f_{c_0,c_1}(s)$ as a degree-1 polynomial in s, where

$$c_0 + c_1 s = \Delta m + e(x).$$

- For two ciphertexts, one can multiply these polynomials

$$\mathrm{Dec}_s(c_0, c_1)\mathrm{Dec}_s(c_0', c_1') = \Delta^2 mm' + \Delta \times E$$

Relinearization

- Idea: view decryption $\mathrm{Dec}_s(c_0, c_1) = c_0 + c_1 s = f_{c_0,c_1}(s)$ as a degree-1 polynomial in s, where

$$c_0 + c_1 s = \Delta m + e(x).$$

- For two ciphertexts, one can multiply these polynomials

$$\mathrm{Dec}_s(c_0, c_1)\mathrm{Dec}_s(c_0', c_1') = \Delta^2 mm' + \Delta \times E$$

- After scaling down by Δ, one obtains an encryption of mm'

Relinearization Difficulties

1. The polynomial is now of degree two in s

Relinearization Difficulties

1. The polynomial is now of degree two in s
   - additionally, E may be large.

Relinearization Difficulties

1. The polynomial is now of degree two in s
   - additionally, E may be large.
2. Relinearization is a technique to address both of these issues

Relinearization Difficulties

1. The polynomial is now of degree two in s
   - additionally, E may be large.
2. Relinearization is a technique to address both of these issues
   - I will focus on discussing the first issue

Reducing Degrees

- The goal is to linearize the degree 2 polynomial, i.e. find c_0', c_1' such that

$$c_0 + c_1 s + c_2 s^2 = c_0' + c_1' s$$

Reducing Degrees

- The goal is to linearize the degree 2 polynomial, i.e. find c_0', c_1' such that

$$c_0 + c_1 s + c_2 s^2 = c_0' + c_1' s$$

- Done by utilizing certain encrypted forms of s^2

Reducing Degrees

- The goal is to linearize the degree 2 polynomial, i.e. find c_0', c_1' such that

$$c_0 + c_1 s + c_2 s^2 = c_0' + c_1' s$$

- Done by utilizing certain encrypted forms of s^2
  - Called the *relinearization key rk*

Reducing Degrees

- The goal is to linearize the degree 2 polynomial, i.e. find c_0', c_1' such that

$$c_0 + c_1 s + c_2 s^2 = c_0' + c_1' s$$

- Done by utilizing certain encrypted forms of s^2
  - Called the *relinearization key rk*
    - In particular, it satisfies $rk_0 + rk_1 s = s^2 + e$

Reducing Degrees

- The goal is to linearize the degree 2 polynomial, i.e. find c_0', c_1' such that

$$c_0 + c_1 s + c_2 s^2 = c_0' + c_1' s$$

- Done by utilizing certain encrypted forms of s^2
  - Called the *relinearization key rk*
    - In particular, it satisfies $rk_0 + rk_1 s = s^2 + e$
    - Can multiply by c_2 and subtract

Reducing Degrees

- The goal is to linearize the degree 2 polynomial, i.e. find c_0', c_1' such that

$$c_0 + c_1 s + c_2 s^2 = c_0' + c_1' s$$

- Done by utilizing certain encrypted forms of s^2
  - Called the *relinearization key* rk
    - In particular, it satisfies $rk_0 + rk_1 s = s^2 + e$
    - Can multiply by c_2 and subtract
    - Issue: $c_2 e$ may be large

Making c_2 smaller

- Idea is to take a base T decomposition $c_2 = \sum_i c_{2,i} T^i$

Making c_2 smaller

- Idea is to take a base T decomposition $c_2 = \sum_i c_{2,i} T^i$
- Each $c_{2,i}$ small now

Making c_2 smaller

- Idea is to take a base T decomposition $c_2 = \sum_i c_{2,i} T^i$
- Each $c_{2,i}$ small now
  - Downside: Relinearization key needs encryptions of $T^i s^2$ for each i

Contents

High Level Implementation Overview

- Language: Rust

High Level Implementation Overview

- Language: Rust
- Used crates for Big-Int arithmetic + a PRG

High Level Implementation Overview

- Language: Rust
- Used crates for Big-Int arithmetic + a PRG
- Focused implementation on straightforward algorithms

High Level Implementation Overview

- Language: Rust
- Used crates for Big-Int arithmetic + a PRG
- Focused implementation on straightforward algorithms
- All of FV implemented except Bootstrapping

High Level Implementation Overview

- Language: Rust
- Used crates for Big-Int arithmetic + a PRG
- Focused implementation on straightforward algorithms
- All of FV implemented except Bootstrapping
    - Only implemented one of the two relinearization techniques.

High Level Implementation Overview

- Language: Rust
- Used crates for Big-Int arithmetic + a PRG
- Focused implementation on straightforward algorithms
- All of FV implemented except Bootstrapping
  - Only implemented one of the two relinearization techniques.

The Straightforward Algorithms

- Polynomial Arithmetic

The Straightforward Algorithms

- Polynomial Arithmetic
    - Power of Two Cyclotomics

The Straightforward Algorithms

- Polynomial Arithmetic
  - Power of Two Cyclotomics
  - Schoolbook Multiplication: $O(n^2)$

The Straightforward Algorithms

- Polynomial Arithmetic
  - Power of Two Cyclotomics
  - Schoolbook Multiplication: $O(n^2)$
    - Karatsuba $O(n^{1.6})$, Toom-Cook $O(n^{1+\epsilon})$

The Straightforward Algorithms

- Polynomial Arithmetic
  - Power of Two Cyclotomics
  - Schoolbook Multiplication: $O(n^2)$
    - Karatsuba $O(n^{1.6})$, Toom-Cook $O(n^{1+\epsilon})$
    - Number-Theoretic Transform: $O(n \log_2 n)$.
- Gaussian Sampling

The Straightforward Algorithms

- Polynomial Arithmetic
  - Power of Two Cyclotomics
  - Schoolbook Multiplication: $O(n^2)$
    - Karatsuba $O(n^{1.6})$, Toom-Cook $O(n^{1+\epsilon})$
    - Number-Theoretic Transform: $O(n \log_2 n)$.
- Gaussian Sampling
  - Inverse-CDT with f64's.

The Straightforward Algorithms

- Polynomial Arithmetic
  - Power of Two Cyclotomics
  - Schoolbook Multiplication: $O(n^2)$
    - Karatsuba $O(n^{1.6})$, Toom-Cook $O(n^{1+\epsilon})$
    - Number-Theoretic Transform: $O(n \log_2 n)$.
- Gaussian Sampling
  - Inverse-CDT with f64's.
    - Turns $[0, 1]$ samples to samples of *any* distribution

The Straightforward Algorithms

- Polynomial Arithmetic
  - Power of Two Cyclotomics
  - Schoolbook Multiplication: $O(n^2)$
    - Karatsuba $O(n^{1.6})$, Toom-Cook $O(n^{1+\epsilon})$
    - Number-Theoretic Transform: $O(n \log_2 n)$.
- Gaussian Sampling
  - Inverse-CDT with f64's.
    - Turns $[0, 1]$ samples to samples of *any* distribution
  - Precision a little low.

The Straightforward Algorithms

- Polynomial Arithmetic
  - Power of Two Cyclotomics
  - Schoolbook Multiplication: $O(n^2)$
    - Karatsuba $O(n^{1.6})$, Toom-Cook $O(n^{1+\epsilon})$
    - Number-Theoretic Transform: $O(n \log_2 n)$.
- Gaussian Sampling
  - Inverse-CDT with f64's.
    - Turns $[0, 1]$ samples to samples of *any* distribution
  - Precision a little low.
  - Big hit to make side-channel resistant

The Straightforward Algorithms

- Polynomial Arithmetic
  - Power of Two Cyclotomics
  - Schoolbook Multiplication: $O(n^2)$
    - Karatsuba $O(n^{1.6})$, Toom-Cook $O(n^{1+\epsilon})$
    - Number-Theoretic Transform: $O(n \log_2 n)$.
- Gaussian Sampling
  - Inverse-CDT with f64's.
    - Turns $[0, 1]$ samples to samples of *any* distribution
  - Precision a little low.
  - Big hit to make side-channel resistant
  - Space complexity that scales poorly with σ.
- Relinearization via Digit Decomposition

Parts of Implementation Working

- Everything but Bootstrapping!

Parts of Implementation Working

- Everything but Bootstrapping!
  - (And potentially extremely large σ Gaussian sampling)

Parts of Implementation Working

- Everything but Bootstrapping!
  - (And potentially extremely large σ Gaussian sampling)
- Concretely
  - All PKE algorithms supported

Parts of Implementation Working

- Everything but Bootstrapping!
  - (And potentially extremely large σ Gaussian sampling)
- Concretely
  - All PKE algorithms supported
  - Addition and Multiplication work

Parts of Implementation Working

- Everything but Bootstrapping!
  - (And potentially extremely large σ Gaussian sampling)
- Concretely
  - All PKE algorithms supported
  - Addition and Multiplication work
  - Relinearization works

Some Timings

- Due to Rust implementation, *extremely* efficient.

Some Timings

- Due to Rust implementation, *extremely* efficient.
- Opt-level $= 3$ (and full-size parameters) leads to multiplications in ≈ 5 seconds

Some Timings

- Due to Rust implementation, *extremely* efficient.
- Opt-level $= 3$ (and full-size parameters) leads to multiplications in ≈ 5 seconds
- Funny story: Initial timing *much* worse

Some Timings

- Due to Rust implementation, *extremely* efficient.
- Opt-level $= 3$ (and full-size parameters) leads to multiplications in ≈ 5 seconds
- Funny story: Initial timing *much* worse
  - ≈ 180s

Some Timings

- Due to Rust implementation, *extremely* efficient.
- Opt-level = 3 (and full-size parameters) leads to multiplications in ≈ 5 seconds
- Funny story: Initial timing *much* worse
  - ≈ 180s
  - Due to choice of base $T = 2$ for relinearization

Some Timings

- Due to Rust implementation, *extremely* efficient.
- Opt-level $= 3$ (and full-size parameters) leads to multiplications in ≈ 5 seconds
- Funny story: Initial timing *much* worse
  - ≈ 180s
  - Due to choice of base $T = 2$ for relinearization
    - Implementation was implicitly using *big int* arithmetic for *binary* computations

Some Timings

- Due to Rust implementation, *extremely* efficient.
- Opt-level = 3 (and full-size parameters) leads to multiplications in ≈ 5 seconds
- Funny story: Initial timing *much* worse
  - ≈ 180s
  - Due to choice of base $T = 2$ for relinearization
    - Implementation was implicitly using *big int* arithmetic for *binary* computations

Contents

Contents

Conclusion

- Plenty that could still be done

Conclusion

- Plenty that could still be done
  - Bootstrapping

Conclusion

- Plenty that could still be done
  - Bootstrapping
  - Faster polynomial arithmetic

Conclusion

- Plenty that could still be done
  - Bootstrapping
  - Faster polynomial arithmetic
  - Better Gaussian sampling

Conclusion

- Plenty that could still be done
  - Bootstrapping
  - Faster polynomial arithmetic
  - Better Gaussian sampling
- Fun Implementation

Conclusion

- Plenty that could still be done
  - Bootstrapping
  - Faster polynomial arithmetic
  - Better Gaussian sampling
- Fun Implementation
  - The easy parts were hard, and the hard parts were easy