Fully Homomorphic Encryption

Craig Gentry
IBM Watson
Outline for Today

- Homomorphic Encryption Basics
- Somewhat homomorphic encryption (SWHE) schemes
Homomorphic Encryption Basics
A way to delegate **processing** of your data, without giving away **access** to it.

**Example App:** Cloud computing on encrypted data

Do you really think it’s safe to store your data in the cloud *unencrypted*?

“Where the sensitive information is concentrated, that is where the spies will go. This is just a fact of life.” – Ken Silva, former NSA official
I want 1) the cloud to process my data 2) even though it is encrypted.

Run

\[ \text{Evaluate}[ f, \text{Enc}_k(x) ] = \text{Enc}_k[f(x)] \]
An Analogy: Alice’s Jewelry Store

- Alice wants workers to assemble raw materials into jewelry
- But Alice is worried about theft: She wants her workers to process the raw materials without having access to them.

- Alice puts raw materials in locked glovebox.
- Workers assemble jewelry inside glovebox, using the gloves.
- Alice unlocks box to get “results”.

Alice wants workers to assemble raw materials into jewelry. But Alice is worried about theft: she wants her workers to process the raw materials without having access to them. Alice puts raw materials in a locked glovebox. Workers assemble jewelry inside the glovebox, using the gloves. Alice unlocks the box to get “results.”
Homomorphic Encryption Basics

Homomorphic Encryption [RAD78]:

Compactness: Size of Eval’d ciphertext independent of f
Homomorphinc Encryption Basics

“Fully” means it works for all functions $f$

Fully Homomorphic Encryption (FHE) [RAD78, Gen09]:

- **Compactness**: Size of Eval’d ciphertext independent of $f$
Somewhat Homomorphic Encryption (SWHE):

“Somewhat” means it works for some functions $f$.

Compactness: Size of Eval’d ciphertext independent of $f$.
Homomorphic Encryption Basics

A way to delegate processing of your data, without giving away access to it.

- Fully Homomorphic Encryption (FHE):
  - Arbitrary processing
  - But computationally expensive.

- Somewhat Homomorphic Encryption (SWHE):
  - Limited processing
  - Cheaper computationally.
Homomorphic Encryption Basics: Functionality
Forget encryption for a moment…

How does your computer compute a function?

Basically, by working on *bits*, 1’s and 0’s.

And by using bit operations – for example,

- \( \text{AND} (b_1, b_2) = 1 \) if \( b_1 = b_2 = 1 \); otherwise, equals 0.
  - \( \text{AND} (b_1, b_2) = b_1 \times b_2 \).
- \( \text{XOR} (b_1, b_2) = 0 \) if \( b_1 = b_2 \); equals 1 if \( b_1 \neq b_2 \).
  - \( \text{XOR} (b_1, b_2) = b_1 + b_2 \) (modulo 2)

*Any* function can be computed bit-wise – with only \text{AND}s and \text{XOR}s – if it can be computed at all.
Unencrypted String Matching

- Still forget encryption for now...
- Example: How do you detect whether a string is in a file?

Step 1: Match string against subsequences of file

ZeroString(100010) = 0
(not the zero string! not a match!)

The ZeroString function itself can be computed from basic bit operations.
Unencrypted String Matching

- Still forget encryption for now...
- Example: How do you detect whether a string is in a file?

Step 1: Match string against subsequences of file

01100111101100100100010001
XOR
111011

```
ZeroString(000000) = 0
(is the zero string! a match!)
```
Unencrypted String Matching

- Still forget encryption for now...
- Example: How do you detect whether a string is in a file?

**Step 2: Aggregate info about the subsequences**

}```01100111101100100100010001
111011
00000010...
```

\[ \text{OR} (00000010...) = 1 \text{ (string is in the file!)} \]

OR also can be decomposed into \textsc{ands} and \textsc{xors}. ```
Let’s Do This Encrypted…

- Let $b$ denote a valid encryption of bit $b$.
- Suppose we have a (homomorphic) encryption scheme with public functions $E$-ADD, $E$-MULT where:

$$
E\text{-MULT}(b_1, b_2) = b_1 \times b_2 \quad \text{and} \quad E\text{-ADD}(b_1, b_2) = b_1 + b_2
$$

for any $b_1$ and $b_2$.
- Then we can AND and XOR encrypted bits.
- Proceeding bit-wise, we can compute any function on encrypted data.
\( b \) denotes an encryption of bit \( b \).

**Step 1:** Match string against subsequences of file

\[
\text{E-ZeroString}(100010) = 0
\]

(not the zero string! not a match!)

E-ZeroString function itself can be computed from basic bit operations.
Encrypted String Matching

\( b \) denotes an encryption of bit \( b \).

1. **Bit-wise encrypted file**

2. **Step 2: Aggregate info about the subsequences**

\[
\begin{align*}
01100111101100100100010001 & \quad \text{E-OR (00000010..) = 1} \\
11100000000011 & \quad \text{(string is in the encrypted file!)}
\end{align*}
\]

E-OR can also be computed from basic bit operations.
Computing General Functions

- Can you add and multiply (mod 2) and remember stuff?
  - Congratulations, then you can compute any efficiently computable function.
  - If you only can add and multiply mod 3, no worries.
- \{ADD,MULT\} are Turing-complete (over any ring).
  - Take any (classically) efficiently computable function. Express it as a poly-size circuit of ADD and MULT gates.
- Circuits vs. Turing machines (about the same):
  - Circuit size = \(O(T_f \log T_f)\)
  - \(T_f = \text{time to compute } f \text{ on a TM}\)
FHE Defined

Can your cryptosystem encrypt 0 and 1, and ADD and MULT encrypted data efficiently?

Functionality: Let $S_{sk}$ be set of “valid” ciphertexts for (any) $sk$. For $c_1, c_2 \in S_{sk}$, set $c_{ADD} = ADD(c_1, c_2)$, $c_{MULT} = MULT(c_1, c_2)$. Then:

- $\text{DEC}_{sk}(c_{ADD}) = \text{DEC}_{sk}(c_1) + \text{DEC}_{sk}(c_2)$, and
- $\text{DEC}_{sk}(c_{MULT}) = \text{DEC}_{sk}(c_1) \cdot \text{DEC}_{sk}(c_2)$

Also, $c_{ADD}$ and $c_{MULT}$ are in $S_{sk}$.

Efficiency: For security parameter $k$,

- All ops (KEYGEN, ENC, DEC, ADD, MULT) take $\text{poly}(k)$ time.
- All valid ciphertexts have $\text{poly}(k)$ size.

CPA Security: Best known attacks have complexity $2^k$.

Congratulations, you have a (fully) homomorphic encryption scheme!
Homomorphic Encryption Basics: Security
Security of Homomorphic Encryption

- **Semantic security** [GM’84]: For any $m_0 \neq m_1$, 
  $$(pk, \text{Enc}_{pk}(m_0)) \approx (pk, \text{Enc}_{pk}(m_1))$$
  - $\approx$ means indistinguishable by efficient algorithms.
  - $pk$ is a public key, if there is one.
  - Any semantically secure encryption scheme must be probabilistic – i.e., many ciphertexts per plaintext.

- What about IND-CCA1 and IND-CCA2 security?
- IND-CCA2 is impossible for HE, since the adversary can homomorphically tweak the challenge ciphertext.
- IND-CCA1 FHE is open.
- [LMSV10] IND-CCA1 SWHE
Function Privacy

- **Function–privacy**: $c^* = \text{Eval}(f, \text{Enc}_{pk}(x))$ hides $f$.
  - **Statistical** (when Eval is randomized): $c^*$ has the same distribution as $\text{Enc}(f(x))$.
  - **Computational**: $c^*$ may not look like a “fresh” ciphertext as long as it decrypts to $f(x)$. 
HE Security: A Paradox?

- Cloud stores my encrypted files: pk, $\text{Enc}_{pk}(f_1), \ldots, \text{Enc}_{pk}(f_n)$.
- Later, I want $f_3$, but want to hide “3” from cloud.
- I send $\text{Enc}_{pk}(3)$ to the cloud.
- Cloud runs $\text{Eval}_{pk}(f, \text{Enc}_{pk}(3), \text{Enc}_{pk}(f_1), \ldots, \text{Enc}_{pk}(f_n))$, where $f(n, \{\text{files}\})$ is the function that outputs the nth file.
- It sends me the (encrypted) $f_3$.
- Paradox?: Can’t the cloud just “see” it is sending the 3rd encrypted file? By just comparing the stored value $\text{Enc}_{pk}(f_3)$ to the ciphertext it sends?

Resolution of paradox:

Semantic security implies:
- Many encryptions of $f_3$,
- Hard to tell when two ciphertexts encrypt the same thing.
Homomorphic Encryption Basics: Limitations
Circuits vs. RAMs:
- **Circuits are powerful**: For all functions, circuit-size $\approx$ TM complexity.
- **But** random-access machines compute some functions much faster than a TM or circuit (Binary search)
- Can’t do “random access” on encrypted data without leaking some information (not surprising)

What we can do:
- [GKKMRV11]: “Secure Computation with Sublinear Amortized Work”
- After setup cost quasi-linear in the size of the data, client and cloud run oblivious RAM on the client’s encrypted data.
FHE Doesn’t Do Obfuscation

- **Obfuscation:**
  - I give the cloud an “encrypted” program \( E(P) \).
  - For any input \( x \), cloud can compute \( E(P)(x) = P(x) \).
  - Cloud learns “nothing” about \( P \), except \( \{x_i, P(x_i)\} \).

- **[BGIRSVY01]: “On the (Im)possibility of Obfuscating Programs”**

- **Difference between obfuscation and FHE:**
  - In FHE, cloud computes \( E(P(x)) \), and it can’t decrypt to get \( P(x) \).
Multi-Key FHE
- Different clients encrypt data under different FHE keys.
- Later, cloud “combines” data encrypted under different keys: \(\text{Enc}_{pk_1,\ldots,pk_t}(f(m_1,\ldots,m_t)) \leftarrow \text{Eval}(pk_1,\ldots, pk_t, f, c_1, \ldots, c_t)\).

FHE doesn’t do this “automatically”.

But, [LATV12]: “On-the-fly Multiparty Computation on the Cloud via Multikey FHE”:
- They have a scheme that does this.
Now, all we need is an encryption scheme that:
- Given any encryptions $E(b_1)$ and $E(b_2)$,
- can output encryptions $E(b_1 + b_2)$ and $E(b_1 \times b_2)$,
- forever,
- without using the secret key of course.

Pre–2009 schemes were *somewhat homomorphic*.
- They could do ADD or MULT, not both, indefinitely.
- Analogous to a glovebox with “clumsy” gloves.
Somewhat Homomorphic Encryption (SWHE)
SWHE: What’s it Good For?

I thought we were doing FHE...
Why Somewhat HE?

- **Performance!**
  - For many somewhat simple functions, the “overhead” of SWHE is much less than overhead of FHE
  - “Overhead” = (time of encrypted computation)/(time of unencrypted computation)

- **Stepping-stone to FHE**
  - Most FHE schemes are built “on top of” a SWHE scheme with special properties.
SWHE: Performance
FHE Implementations

- First attempt [Smart–Vercauteren 2010]
  - Implemented (a variant of) the underlying SWHE
  - But parameters too small to get bootstrapping

- Second attempt [Gentry–Halevi 2011a]
  - Implemented a similar variant
  - Many more optimizations, tradeoffs
  - Could implement the complete FHE for 1st time
Using NTL/GMP

Run on a “strong” 1-CPU machine
- Xeon E5440 / 2.83 GHz (64-bit, quad-core) 24 GB memory

Generated/tested instances in 4 dimensions:
- Toy($2^9$), Small($2^{11}$), Med($2^{13}$), Large($2^{15}$)

Details at [https://researcher.ibm.com/researcher/view_project.php?id=1548](https://researcher.ibm.com/researcher/view_project.php?id=1548)
## Performance: SWHE

<table>
<thead>
<tr>
<th>Dimension</th>
<th>KeyGen</th>
<th>Enc amortized</th>
<th>Mult / Dec</th>
<th>degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048 800,000-bit integers</td>
<td>1.25 sec</td>
<td>.060 sec</td>
<td>.023 sec</td>
<td>~200</td>
</tr>
<tr>
<td>8192 3,200,000-bit integers</td>
<td>10 sec</td>
<td>.7 sec</td>
<td>.12 sec</td>
<td>~200</td>
</tr>
<tr>
<td>32768 13,000,000-bit integers</td>
<td>95 sec</td>
<td>5.3 sec</td>
<td>.6 sec</td>
<td>~200</td>
</tr>
</tbody>
</table>

PK is 2 integers, SK one integer
## Performance: FHE

<table>
<thead>
<tr>
<th>Dimension</th>
<th>KeyGen</th>
<th>PK size</th>
<th>ReCrypt</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048</td>
<td>40 sec</td>
<td>70 MByte</td>
<td>31 sec</td>
</tr>
<tr>
<td>8192</td>
<td>8 min</td>
<td>285 MByte</td>
<td>3 min</td>
</tr>
<tr>
<td>32768</td>
<td>2 hours</td>
<td>2.3 GByte</td>
<td>30 minute</td>
</tr>
</tbody>
</table>
Can HE Be Practical? [LNV11]

- Implementation of [BV11a] SWHE scheme.
- For lattice dim. 2048, Mult takes 43 msec.
  - Comparable to 23 msec of [GH10]
  - They use Intel Core 2 Duo Processor at 2.1 GHz.
- Shows lattice-based SWHE can compute quadratic functions more efficiently than [BGN05].
SWHE: Applications
Rule of Thumb: If your function $f$ can be expressed as a low-degree polynomial, SWHE might be sufficient.
Private information retrieval

- Client wants bit $B_i$ of database $B_1 \ldots B_n$, w/o revealing $i$.
- The PIR function has degree only log $n$.
- Easily achievable with SWHE.
Keyword Search / String Matching

- Client wants to know whether encrypted string $s = s_1 \ldots s_m$ is in one of its encrypted files
- Comparison of two $m$-bit strings is a $m$-degree poly.
- OR of $n$ comparisons is a $n$-degree poly.
- “Smolensky trick”: in both cases we can reduce the degree to $k$, with a $2^{-k}$ probability of error.
SWHE: Stepping–Stone to FHE

Tomorrow, we’ll see how SWHE helps construct FHE...
SWHE: Older Schemes

RSA, ElGamal, Paillier, Boneh–Goh–Nissim, Ishai–Paskin, …

I won’t cover these.
SWHE: To The Constructions!!!
Polly Cracker: An Early Attempt at SWHE

And perhaps the most “natural” way to do it…
ADD and MULT, Naturally...

Most Natural Approach
Ciphertexts live in a “ring”.
ADDING ciphertexts (as ring elements)
adds underlying plaintexts.
Some for MULT.

Definition of (commutative) ring:
- Like a field, without inverses.
- It has $+$, $\times$, 0 and 1,
  additive and multiplicative closure.

Examples: integers $\mathbb{Z}$,
polynomials $\mathbb{Z}[x,y,\ldots]$, …
Polly Cracker

Main Idea
Encryptions of 0 are polynomials that evaluate to 0 at the secret key.

- **KeyGen**: Secret = some point \((s_1, \ldots, s_n) \in \mathbb{Z}_q^n\).
  Public key: Polys \(\{f_i(x_1,\ldots,x_n)\}\) s.t. \(f_i(s_1,\ldots,s_n) = 0 \mod q\).

- **Encrypt**: From \(\{f_i\}\), generate random polynomial \(g\) s.t. \(g(s_1,\ldots,s_n) = 0 \mod q\). Ciphertext is: \(c(x_1,\ldots,x_n) = m + g(x_1,\ldots,x_n) \mod q\).

- **Decrypt**: Evaluate ciphertext at the secret: \(c(s_1,\ldots,s_n) = m \mod q\).

- **ADD and MULT**: Output sum or product of ciphertext polynomials.
Polly Cracker

Main Idea
Encryptions of 0 are polynomials that evaluate to 0 at the secret key.

- Semantic Security (under chosen plaintext attack): Given two ciphertexts $c_0$ and $c_1$, can you distinguish whether:
  - $c_0$ and $c_1$ encrypt the same message?
  - $c_0 - c_1$ encrypts 0?
  - $c_0 - c_1$ evaluates to 0 at the secret key?
  - Solve “Ideal Membership” Problem?
Ideals: Definition and Examples

- Ideal: Subset I of a ring R that is:
  - Additively closed: \( i_1, i_2 \in I \rightarrow i_1 + i_2 \in I \).
  - Closed under mult with R: \( i \in I, r \in R \rightarrow i \cdot r \in I \).

- Example:
  - \( R = \mathbb{Z} \), the integers. \( I = (5) \), multiples of 5.
  - \( R = \mathbb{Z}[x,y] \). \( I = \{ f(x,y) \in \mathbb{Z}[x,y] : f(7,11) = 0 \} \).
    - \( I = (x-7,y-11) \). These “generate” the ideal.

- “Modulo”
  - 7 modulo (5) = 2, or \( 7 \in 2+(5) \)
  - \( g(x,y) \) modulo (\( x-7,y-11 \)) = \( g(7,11) \).
Back to Polly Cracker...

**Main Idea**
Encryptions of 0 are polynomials that evaluate to 0 at the secret key.

- **Semantic Security: Ideal Membership Problem:**
  - Given ciphertext polys $c_1(x_1,\ldots,x_n)$ and $c_2(x_1,\ldots,x_n)$,
  - Distinguish whether $c_1(x_1,\ldots,x_n) - c_2(x_1,\ldots,x_n)$ is in the ideal $(x_1-s_1, \ldots, x_n-s_n)$. 
[AFFP11] Sadly, Polly Cracker is typically easy to break, using just linear algebra.

Public key: polys \{f_i\} such that \( f_i(s_1, \ldots, s_n) = 0 \).

Computing Grobner bases is hard, \textit{in general}.

\textit{In practice}, only a small (polynomial #) of monomials can be used in the ciphertexts.
Polly Cracker Cryptanalysis

An Attack:

- Collect lots of encryptions \( \{c_i\} \) of 0.
  - (These are elements of an ideal I.)
- The \( c_i \)'s generate a lattice \( L \) (over the multivariate monomials). Compute Hermite Normal Form (HNF) of \( L \).
- To break semantic security, reduce \( c_1 - c_2 \mod \text{HNF}(L) \): the result will be 0 if \( m_1 = m_2 \).
Noisy Polly Cracker

Adding noise to Polly Cracker to defeat attacks…
Main Idea
Encryptions of 0 are polynomials that evaluate to 0 at the secret key.
Main Idea
Encryptions of 0 are polynomials that evaluate to something small and even (smeven) 0 at the secret key.

- **KeyGen**: Secret = some point \((s_1, \ldots, s_n) \in \mathbb{Z}_q^n\).
  Public key: \(\{f_i(x_1,\ldots,x_n)\}\) s.t. \(f_i(s_1,\ldots,s_n) = 2e_i \mod q\), \(|e_i| \ll q\).

- **Encrypt**: Generate random poly \(g\) s.t. \(g(s_1,\ldots,s_n) = \text{smeven from } \{f_i\}\). Ciphertext is \(c(x_1,\ldots,x_n) = m + g(x_1,\ldots,x_n) \mod q\) for message \(m \in \{0,1\}\).

- **Decrypt**: \(c(s_1,\ldots,s_n) = m + \text{smeven mod q. Reduce mod } 2\).

- **ADD and MULT**: Output sum or product of ciphertext polys.
Noisy Polly Cracker [AFFP11]

Main Idea

Encryptions of 0 are polynomials that evaluate to something small and even (smeven) 0 \( \text{modulo} \) a secret ideal.

- **KeyGen**: Secret ideal = \( (x_1-s_1, \ldots, x_n-s_n) \).
  Public key: \( \{f_i(x_1,\ldots,x_n)\} \) s.t. \( f_i(s_1,\ldots,s_n)=2e_i \mod q, \ |e_i| \ll q \).

- **Encrypt**: Generate random poly \( g \) s.t. \( g(s_1,\ldots,s_n)= \text{smeven} \) from \( \{f_i\} \).
  Ciphertext is \( c(x_1,\ldots,x_n) = m + g(x_1,\ldots,x_n) \mod q \) for message \( m \in \{0,1\} \).

- **Decrypt**: \( c(s_1,\ldots,s_n) = m+\text{smeven} \mod q \). Reduce mod 2.

- **ADD and MULT**: Output sum or product of ciphertext polys.
Noisy Polly Cracker [AFFP11]

Main Idea
Encryptions of 0 are polynomials that evaluate to something small and even (smeven) 0 \textit{modulo} a secret ideal.

- **KeyGen**: Secret ideal = \((x_1-s_1, \ldots, x_n-s_n)\).
  Public key: \(\{f_i(x_1,\ldots,x_n)\} \text{ s.t. } f_i(s_1,\ldots,s_n) = 2e_i \mod q\), \(|e_i| \ll q\).

- **Encrypt**: Generate random poly \(g\) s.t. \(g(s_1,\ldots,s_n) = \text{smeven}\) from \(\{f_i\}\). Ciphertext is \(c(x_1,\ldots,x_n) = m + g(x_1,\ldots,x_n) \mod q\) for message \(m \in \{0,1\}\).

- **Decrypt**: \(c(s_1,\ldots,s_n) = m + \text{smeven} \mod q\). Reduce mod 2.

- **ADD and MULT**: Output sum or product of ciphertext polys.

We call \(c(s_1,\ldots,s_n)\) the “noise” of the ciphertext. ADDs and MULTs make the “noise” grow.
Noisy Ciphertexts

- Each ciphertext has some noise that hides the message.
- Think: “hidden” error correcting codes…
- If error is small, Alice can use knowledge of “hidden” code, or a (hidden) good basis of a known code to remove the noise.
- If noise is large, decryption becomes hopeless even for Alice.
Adding and Multiplying Noise

- Message “hides” in the noise.
- Adding ciphertexts adds the noises.
- Multiplying ciphertexts multiplies the noises.
- The ciphertext noisiness grows!
  - Eventually causes a decryption error!
SWHE over the Integers $[\nuDGHV10]$,

Maybe the simplest SWHE scheme you could imagine…
A Symmetric SWHE Scheme [vDGHV10]

- Shared secret key: odd number $p$
- To encrypt a bit $m$ in \{0,1\}:
  - Choose at random small $r \ll p$, large $q$
  - Output $c = m + 2r + pq$
    - Ciphertext is close to a multiple of $p$
    - $m = \text{LSB of distance to nearest multiple of } p$
- To decrypt $c$:
  - Output $m = (c \mod p) \mod 2 = [c]_p$
- ADD, MULT: Output $c \leftarrow c_1 + c_2$
  or $c \leftarrow c_1 \times c_2$. 

What could be Simpler?
A Symmetric SWHE Scheme [vDGHV10]

- Shared secret key: odd number $p$
- To encrypt a bit $m$ in $\{0,1\}$:
  - Choose at random small $r \ll p$, large $q$
  - Output $c = m + 2r + pq$
    - Ciphertext is close to a multiple of $p$
    - $m = \text{LSB of distance to nearest multiple of } p$
- To decrypt $c$:
  - Output $m = (c \mod p) \mod 2 = [[c]_p]_2$
- ADD, MULT: Output $c \leftarrow c_1 + c_2$
  or $c \leftarrow c_1 \times c_2$. 

$(p)$ is our secret ideal.

An encryption of 0 is small and even modulo our ideal.

To decrypt, evaluate $c$ modulo the ideal. Then reduce mod 2.
Asymmetric SWHE [$\nu$DGHV10]

- Secret key is an odd $p$ as before
- Public key $pk$ has “encryptions of 0” $x_i = 2r_i + q_ip$
  - Actually $x_i = [2r_i + q_ip]x_0$ for $i = 1, \ldots, n$.
- $Enc(pk, m) = m + \text{subset-sum}(x_i’s)$
  - Actually, $Enc(pk, m) = [m + \text{subset-sum}(x_i’s) + 2r]x_0$.
- $Dec(sk, c) = [[c]_p]_2$

Making a public key out of “encryptions of 0” formalized by Rothblum (“From Private Key to Public Key”, TCC’11).
Asymmetric SWHE [vDGHV10]

- Secret key is an odd $p$ as before
- Public key $pk$ has “encryptions of 0” $x_i = 2r_i + q_i p$
  - Actually $x_i = [2r_i + q_i p]_{x_0}$ for $i = 1, \ldots, n$.
- $\text{Enc}(pk, m) = m + \text{subset}-\text{sum}(x_i’s)$
  - Actually, $\text{Enc}(pk, m) = [m + \text{subset}-\text{sum}(x_i’s) + 2r]_{x_0}$.
- $\text{Dec}(sk, c) = [[c]_p]_2$

Quite similar to Regev’s ’03 scheme. Main difference: SWHE uses much more aggressive parameters…
Security

- Approximate GCD (approx\text{-}gcd) Problem:
  - Given many $x_i = s_i + q_i p$, output $p$
  - Example params: $s_i \sim 2^{O(\lambda)}$, $p \sim 2^{O(\lambda^2)}$, $q_i \sim 2^{O(\lambda^5)}$, where $\lambda$ is security parameter
    - Best known attacks (lattices) require $2^\lambda$ time

- Reduction:
  - If approx\text{-}gcd is hard, scheme is semantically secure
Hardness of Approximate-GCD

- Several lattice-based approaches for solving approximate-GCD
  - Studied in [Howgrave-Graham01], more recently in [vDGV10, CH11, CN11]
  - All run out of steam when $|q_i| \gg |p|^2$, where $|p|$ is number of bits of $p$
  - In our case $|p| = O(\lambda^2)$, $|q_i| = O(\lambda^5) \gg |p|^2$
Relation to Simultaneous Diophantine Approximation

- \( x_i = q_ip + r_i \) (\( r_i \ll p \ll q_i \)), \( i = 0,1,2,... \)
  - \( y_i = x_i/x_0 = (q_i+s_i)/q_0 \), \( s_i \sim r_i/p \ll 1 \)
  - \( y_1, y_2, \ldots \) is an instance of SDA
    - \( q_0 \) is a good denominator for all \( y_i \)'s

Use Lagarias’s algorithm:

- Consider the rows of this matrix:
- Find a short vector in the lattice that they span
- \( \langle q_0,q_1,\ldots,q_t \rangle \cdot L \) is short
- Hopefully we will find it.
Relation to SDA (cont.)

- When will Lagarias’ algorithm succeed?
  - $\langle q_0, q_1, \ldots, q_t \rangle \cdot L$ should be shortest in lattice
    - In particular shorter than $\sim \det(L)^{1/t+1}$
  - This only holds for $t > \log Q / \log P$
  - The dimension of the lattice is $t+1$
  - Rule of thumb: takes $2^{t/k}$ time to get $2^k$ approximation of SVP/CVP in lattice of dim $t$.
    - $2 |q_0|^2 / |p|^2 = 2^\lambda$ time to get $2 |p| \gg 2^\lambda$ approx.
  
  Minkowski bound

- Bottom line: no known efficient attack on approx–gcd
How Homomorphic Is It?

- Suppose $c_1 = m_1 + 2r_1 + q_1p$, ..., $c_t = m_t + 2r_t + q_tp$
- **ADD**: $c = c_1 + c_2$.
  - Noise of $c$ is $[c]_p = (m_1 + m_2 + 2r_1 + 2r_2)$, sum of noises
- **MULT**: $c = c_1 \times c_2$.
  - Noise of $c$ is $[c]_p = (m_1 + 2r_1) \times (m_2 + 2r_2)$, product of noises.
- $f$: $c = f(c_1, \ldots, c_t) = f(m_1 + 2r_1, \ldots, m_t + 2r_t)$, the function $f$ applied to the noises.
Claim: If $|f(m_1+2r_1, ..., m_t+2r_t)| < p/2$ for all possible “fresh” noises $m_i+2r_i$, the SWHE scheme can Eval $f$ correctly.

Proof:
- Set $c = f(c_1, ..., c_t)$.
- Then, $[c]_p = f(m_1+2r_1, ..., m_t+2r_t)$ by assumption.
- Then, $[[c]_p]_2 = f(m_1, ..., m_t) \mod 2$.

That’s what we want!
How Homomorphic Is It?

- What if $|f(m_1 + 2r_1, \ldots, m_t + 2r_t)| > p/2$?
  - $c = f(c_1, \ldots, c_t) = f(m_1 + 2r_1, \ldots, m_t + 2r_t) + qp$
    - Nearest $p$-multiple to $c$ is $q'p$ for $q' \neq q$
  - $(c \mod p) = f(m_1 + 2r_1, \ldots, m_t + 2r_t) + (q-q')p$
  - $(c \mod p) \mod 2$
    - $= f(m_1, \ldots, m_t) + (q-q') \mod 2$
    - $= ???$

- We say the scheme can handle $f$ if:
  - $|f(x_1, \ldots, x_t)| < p/4$
  - Whenever all $|x_i| < B$, where $B$ is a bound on the noise of a fresh ciphertext output by Enc.
Example of a Function It Can Handle

- Elementary symmetric poly of degree d:
  - \( f(x_1, ..., x_t) = x_1 \cdot x_2 \cdot x_d + ... + x_{t-d+1} \cdot x_{t-d+2} \cdot x_t \)
  - Has \((t \text{ choose } d) < t^d\) monomials: a lot!!

- If \(|x_i| < B\), then \( |f(x_1, ..., x_t)| < t^d \cdot B^d \)

- E can handle f if:
  - \( t^d \cdot B^d < p/4 \) → basically if: \( d < (\log p)/(\log tB) \)

- Example params: \( B \sim 2^\lambda, p \sim 2^{\lambda^2} \)
  - Eval can handle elem symm poly of degree about \( \lambda \).
An Optimization

- If $f$ has degree $d$, $c = f(c_1, \ldots, c_t)$ will have about $d$ times as many bits as the fresh $c_i$’s.
- Can we reduce the ciphertext length after multiplications?
An Optimization

- A heuristic:
  - Suppose $n$ is bit-length of normal ciphertext.
  - Put additional “encryptions of 0” $\{y_i = 2r_i + q_ip\}$ in pk.
    - Set $y_i$’s to increase geometrically up to square of normal ciphertext: $y_i \approx 2^{n+i}$, for $i$ up to $\approx n$.
  - Set $c = c_1 \times c_2 - \text{subsetsum}(y_i$’s), and $c$ will have normal size.
    - Subtract off $y_i$’s according to $c$’s binary representation.
Well, a little slow…

- Example parameters: a ciphertext is \( O(\lambda^5) \) bits.
- Least efficient SWHE scheme, asymptotically.

But Coron, Mandal, Naccache, Tibouchi have made impressive efficiency improvements.

- [CMNT Crypto ‘11]: FHE over the Integers with Shorter Public Keys
- [CNT Eurocrypt ‘12]: Public–key Compression and Modulus Switching for FHE over the Integers.
- Asymptotics are much better now.
SWHE Based on LWE
[BV11b]
The LWE Problem

- Traditional Version:
  - Let $\chi$ be an error distribution.
  - Distinguish these distributions:
    - Generate uniform $s \leftarrow \mathbb{Z}_q^n$. For many $i$, generate uniform $a_i \leftarrow \mathbb{Z}_q^n$, $e_i \leftarrow \chi$, and output $(a_i, [<a_i, s> + e_i]_q)$.
    - For many $i$, generate uniform $a_i \leftarrow \mathbb{Z}_q^n$, $b_i \leftarrow \mathbb{Z}_q$ and output $(a_i, b_i)$. 
The LWE Problem

- Noisy Polly Cracker Version:
  - Let $\chi$ be an error distribution.
  - Distinguish these distributions:
    - Generate uniform $s \leftarrow \mathbb{Z}_q^n$. For many $i$, generate $e_i \leftarrow \chi$ and a linear polynomial $f_i(x_1, \ldots, x_n) = f_0 + f_1 x_1 + \ldots + f_n x_n$ (from $\mathbb{Z}_q^{n+1}$) such that $[f_i(s_1, \ldots, s_n)]_q = e_i$.
    - For many $i$, generate and output a uniformly random linear polynomial $f_i(x_1, \ldots, x_n)$ (from $\mathbb{Z}_q^{n+1}$).
Regev LWE Encryption Revisited

- **Parameters**: $q$ such that $\gcd(q,2) = 1$.
- **KeyGen**: Secret = uniform $s \in \mathbb{Z}_q^n$. Public key: linear polys $\{f_i(x_1, \ldots, x_n)\}$ s.t. $[f_i(s)]_q = 2e_i$, $|e_i| \ll q$.
- **Encrypt**: Set $g(x_1, \ldots, x_n)$ as a random subset sum of $\{f_i(x_1, \ldots, x_n)\}$. Output $c(x_1, \ldots, x_n) = m + g(x_1, \ldots, x_n)$.
- **Decrypt**: $[c(s)]_q = m + \text{smeven}$. Reduce mod 2.

**Security**:
- Public key consists of an LWE instance, doubled.
- Leftover hash lemma.
SWHE Based on LWE [BV11b]

- **Parameters**: $q$ such that $\text{gcd}(q,2)=1$.
- **KeyGen**: Secret = uniform $s \in \mathbb{Z}_q^n$. Public key: linear polys $\{f_i(x_1,\ldots,x_n)\}$ s.t. $[f_i(s)]_q=2e_i$, $|e_i| \ll q$.
- **Encrypt**: Set $g(x_1,\ldots,x_n)$ as a random subset sum of $\{f_i(x_1,\ldots,x_n)\}$. Output $c(x_1,\ldots,x_n)=m+g(x_1,\ldots,x_n)$.
- **Decrypt**: $[c(s)]_q = m+s\text{meven}$. Reduce mod 2.
- **ADD and MULT**: Output sum or product of ciphertext polynomials.
Relinearization [BV11b]

- After MULT, we have ciphertext \( c(x) = c_1(x) \cdot c_2(x) \) that encrypts some \( m \) under key \( s \).
  - \( [c(s)]_q = m+s\text{meven} \)
  - \( c(x) \) is a quadratic poly with \( O(n^2) \) coefficients.

- What we want: a linear ciphertext \( d(y) \) that encrypts same \( m \) under some key \( t \in \mathbb{Z}_q^n \).

- Relinearization maps a long quadratic ciphertext under \( s \) to a normal linear ciphertext under \( t \).
Relinearization: From Quadratic to Linear (A Change of Variables)

- First step: View c(x) as a long linear ciphertext C(X).
  - Set the variables $X_{ij} = x_i \cdot x_j$.
  - Set the values $S_{ij} = s_i \cdot s_j$.
  - Set $C(X) = \sum c_{1i} c_{2j} X_{ij}$.
  - Then, $[C(S)]_q = [c(s)]_q = m + smeven$.
  - (This is only a change of perspective.)
Second Step: Key Switching

- **Input:** *Long* linear ciphertext $C(X)$ with $N > n$, where $[C(S)]_q = e = m+s\text{meven}$, and $S = (S_1, \ldots, S_N)$ is a long secret key.
- **Output:** *Normal–length* linear ciphertext $d(x)$, where $[d(t)]_q = e+s\text{meven} = m+s\text{meven}$, and $t = (t_1, \ldots, t_n)$ is a normal–length secret key.
- **Special case:** $N \approx n^2$. 

Key Switching Details

- **SwitchKeyGen($S,t$):** Output linear polys $\{h_i(x)\}$, $i \in \{1,\ldots,N\}$ such that:
  \[ [h_i(t)]_q = S_i + \text{smeven}_i \]
  (like an encryption of $S_i$ under $t$)

Add $\text{Aux}(S,t) = \{h_i(x)\}$ to $pk$.

- **SwitchKey($pk, C(X)$):** Set $d(x) = \sum_i C_i \cdot h_i(x)$.
  $d(t) = \sum_i C_i \cdot (S_i + \text{smeven}_i) = C(S) + \sum_i C_i \cdot \text{smeven}_i$

Oh wait, $\sum_i C_i \cdot \text{smeven}_i$ is not small and even...

Fix: Bit-decompose $C$ first so that it has small coefficients...
Key Switching: Bit Decomposition Interlude

- **BitDecomp:**
  - Let BitDecomp(C(X)) be the bit-decomposition of C(X).
  - (U_1(X), ..., U_{\log_2 q}(X)) ← BitDecomp(C(X)), where each U_j(X) has 0/1 coefficients and C(X) = \sum_j 2^j \cdot U_j(X).

- **Powerof2:**
  - (S, 2S, ..., 2^{\log_2 q} S) ← Powersof2(S).
  - Let C' = BitDecomp(C) and S' = Powerof2(S). Then, <C', S'> = <C, S>.
  - So, C'(S') = C(S) mod q.
Key Switching Details

- **SwitchKeyGen(S,t):** Output linear polys \( \{h_i(x)\} \),
  \( i \in \{1,\ldots,N\} \) such that:
  \[
  [h_i(t)]_q = S'_i + \text{smeven}_i
  \]
  (like an encryption of \( S'_i \) under \( t \))

  Add \( \text{Aux}(S',t) = \{h_i(x)\} \) to \( \text{pk} \).

- **SwitchKey(pk, C'(X)):** Set \( d(x) = \sum_i C'_i \cdot h_i(x) \).

  \[
  d(t) = \sum_i C'_i \cdot (S'_i + \text{smeven}_i) = C'(S') + \sum_i C'_i \cdot \text{smeven}_i
  \]

- Now, \( \sum_i C'_i \cdot \text{smeven}_i \) is small and even…
Key Switching: Summary

- **Functionality:**
  - Regev ciphertext under key $S \rightarrow$ Ciphertext under $t$.
  - Need to put $\text{Aux}(S,t)$ in pk.
  - Like proxy re-encryption.
  - Relinearization is only a special case.
    - Later, we will use key switching in a different context.

- **Effect on noise:** SwitchKey increases noise only additively.

- For depth L circuit, use a chain of L encrypted secret keys.
Follows Noisy Polly Cracker blueprint
- With a relinearization step.

Relinearization / key-switching
- Doesn’t increase the noise much.
- So noise analysis, and “homomorphic capacity” analysis, is similar to integer scheme.
- For L depth circuit, use a chain of L encrypted secret keys.
SWHE Based on Ideal Lattices [Gen09]

I’ll skip my 2009 scheme, and focus on RLWE- and NTRU-based schemes.
SWHE Based on RLWE [BV11a]
The Ring–LWE Problem

Traditional Version:

- Let $\chi$ be an error distribution over $R = \mathbb{Z}_q[y]/(y^n+1)$.
- Distinguish these distributions:
  - Generate uniform $s \leftarrow R$. For many $i$, generate uniform $a_i \leftarrow R$, $e_i \leftarrow \chi$, and output $(a_i, a_i \cdot s + e_i)$.
  - For many $i$, generate uniform $a_i \leftarrow R$, $b_i \leftarrow R$ and output $(a_i, b_i)$. 
Noisy Polly Cracker Version:
- Let \( \chi \) be an error distribution over \( R = \mathbb{Z}_q[y]/(y^n+1) \).
- Distinguish these distributions:
  - Generate uniform \( s \leftarrow R \). For many \( i \), generate \( e_i \leftarrow \chi \) and a linear polynomial \( f_i(x) = f_0 + f_1x \) (from \( R^2 \)) such that \( f_i(s) = e_i \).
  - For many \( i \), generate and output a uniformly random linear polynomial \( f_i(x) \) (from \( R^2 \)).
Parameters: $q$ with $\gcd(q, 2) = 1$, $R = \mathbb{Z}_q[y]/(y^n+1)$.

KeyGen: Secret = uniform $s \in R$. Public key: linear polys $\{f_i(x)\}$ s.t. $f_i(s) = 2e_i$, $|e_i| \ll q$.

Encrypt: Set $g(x)$ as a random subset sum of $\{f_i(x)\}$. Output $c(x) = m + g(x)$.

- $m$ can be a “polynomial”, an element of $\mathbb{Z}_2[y]/(y^n+1)$.

Decrypt: $c(s) = m + \text{smeven}$. Reduce mod 2.
SWHE from RLWE [BV11a]

- **Parameters**: q with $\gcd(q,2)=1, R = \mathbb{Z}_q[y]/(y^n+1)$.
- **KeyGen**: Secret $= \text{uniform } s \in R$. Public key: linear polys $\{f_i(x)\}$ s.t. $f_i(s) = 2e_i$, $|e_i| \ll q$.
- **Encrypt**: Set $g(x)$ as a random subset sum of $\{f_i(x)\}$. Output $c(x) = m + g(x)$.
  - $m$ can be a “polynomial”, an element of $\mathbb{Z}_2[y]/(y^n+1)$.
- **Decrypt**: $c(s) = m + sm\text{even}$. Reduce mod $2$.

- **ADD** and **MULT**: Add or multiply the ciphertext polynomials.
After MULT, we have ciphertext $c(x) = c_1(x) \cdot c_2(x)$ that encrypts some $m$ under key $s$.

- $c(s) = m + smeven$
- $c(x)$ is a quadratic poly with 3 coefficients.

What we want: a linear ciphertext $d(x)$ that encrypts same $m$ under some key $t \in R$.

Relinearization maps a long quadratic ciphertext under $s$ to a normal linear ciphertext under $t$. 
First step: View $c(x)$ as a \textit{long} linear ciphertext $C(X)$. 

- Set the variables $X_1 = x$ and $X_2 = x^2$.
- Set the values $S_1 = s$ and $S_2 = s^2$.
- Set $C(X) = (c_{11}x + c_{10})(c_{21}x + c_{20}) = c_{11}c_{21}X_2 + (c_{11}c_{20} + c_{10}c_{21})X + c_{10}c_{20}$.
- Then, $C(S) = c(s) = m + smeven$.
- (This is only a change of perspective.)
Second Step: Key Switching

- Input: *Long* linear ciphertext $C(X)$, where $C(S) = e = m+s\text{meven}$, and $S = (S_1,S_2)$ is a long secret key.
- Output: *Normal-length* linear ciphertext $d(x)$, where $d(t) = e+s\text{meven} = m+s\text{meven}$, and $t \in \mathbb{R}$. 
Key Switching Details

- **SwitchKeyGen(\(S,t\)):** Output linear polys \(\{h_i(x)\}\), \(i \in \{1,\ldots,N\}\) such that:
  \[h_i(t) = S_i + \text{smeven}_i\]
  (like an encryption of \(S_i\) under \(t\))
Add \(\text{Aux}(S,t) = \{h_i(x)\}\) to \(pk\).

- **SwitchKey(pk, C(X)):** Set \(d(x) = \sum_i C_i \cdot h_i(x)\).
  \[d(t) = \sum_i C_i \cdot (S_i + \text{smeven}_i) = C(S) + \sum_i C_i \cdot \text{smeven}_i\]
Oh wait, \(\sum_i C_i \cdot \text{smeven}_i\) is not small and even...
Fix: Bit-decompose \(C\) first so that it has small coefficients...
Key Switching: Bit Decomposition Interlude

- **BitDecomp:**
  - Let $\text{BitDecomp}(C(X))$ be the bit–decomposition of $C(X)$.
  - $(U_1(X), \ldots, U_{\log q}(X)) \leftarrow \text{BitDecomp}(C(X))$, where each $U_j(X)$ has coefficients (in $\mathbb{R}$) that are 0/1 polynomials and $C(X) = \sum_j 2^j \cdot U_j(X)$.

- **Powerof2:**
  - $(S, 2S, \ldots, 2^{\log q} S) \leftarrow \text{Powersof2}(S)$.

- Let $C' = \text{BitDecomp}(C)$ and $S' = \text{Powerof2}(S)$. Then, $\langle C', S' \rangle = \langle C, S \rangle$.

- So, $C'(S') = C(S)$ in $\mathbb{R}$. 
Key Switching Details

- **SwitchKeyGen($S,t$):** Output linear polys $\{h_i(x)\}$, $i \in \{1,\ldots,N\}$ such that:
  \[ h_i(t) = S'_i + \text{smeven}_i \]
  (like an encryption of $S'_i$ under $t$)
  Add $\text{Aux}(S',t) = \{h_i(x)\}$ to $pk$.

- **SwitchKey($pk$, $C'(X)$):** Set $d(x) = \sum_i C'_i \cdot h_i(x)$.
  $d(t) = \sum_i C'_i \cdot (S'_i + \text{smeven}_i) = C'(S') + \sum_i C'_i \cdot \text{smeven}_i$

- Now, $\sum_i C'_i \cdot \text{smeven}_i$ is small and even...
RLWE Key Switching: Summary

- Functionality: as in LWE.
- Effect on noise: SwitchKey increases noise only additively, as in LWE.
- Performance: Better!
  - RLWE:
    - Key switching involves $O(\log q)$ multiplications in $R$.
    - We can use FFT for multiplication.
    - quasi-$O(n \log q)$ work
  - LWE:
    - Relinearization is $O(n^3 \log q)$ work.
NTRU–Based SWHE
NTRU–Based SWHE ([LATV12] and [GHLPSS12])

- **Parameters**: $q$ with $\gcd(q,2)=1$, $R = \mathbb{Z}_q[y](y^n+1)$.  
- **KeyGen**: Secret $= \text{uniform } s \in R$. Public key: linear polynomials $\{f_i(x)\}$ s.t. $f_i(s) = 2e_i$, $|e_i| \ll q$. More reqs:
  - $s$ is small and $1$ mod $2$ (smodd?)
  - $f_i(x)$ has no constant term – i.e., $f_i \cdot s = 2e_i$.
- **Encrypt**: Set $g(x)$ as a random subset sum of $\{f_i(x)\}$. Output $c(x) = m \cdot x + g(x)$.
  - $m$ can be a “polynomial”, an element of $\mathbb{Z}_2[y] / (y^n+1)$.
- **Decrypt**: $c(s) = m \cdot s + s \cdot \text{even}$. Reduce mod $2$.

- **Security**: NTRU Problem: Do $f_i$’s have form $f_i = 2e_i / s_i$; $e_i$, $s_i$ short?
NTRU–Based SWHE ([LATV12] and [GHLPPSS12])

- **Parameters**: \( q \) with \( \gcd(q,2) = 1 \), \( R = \mathbb{Z}_q[y]/(y^n+1) \).
- **KeyGen**: Secret = uniform \( s \in R \). Public key: linear polys \( \{ f_i(x) \} \) s.t. \( f_i(s) = 2e_i \), \( |e_i| \ll q \). More reqs:
  - \( s \) is small and \( 1 \mod 2 \) (smodd?)
  - \( f_i(x) \) has no constant term – i.e., \( f_{i1} \cdot s = 2e_i \).
- **Encrypt**: Set \( g(x) \) as a random subset sum of \( \{ f_i(x) \} \). Output \( c(x) = m \cdot x + g(x) \).
  - \( m \) can be a “polynomial”, an element of \( \mathbb{Z}_2[y]/(y^n+1) \).
- **Decrypt**: \( c(s) = m \cdot s + \text{smeven} \). Reduce mod 2.

- **ADD** and **MULT**: Add or multiply the ciphertext polynomials.
NTRU–Based SWHE: Multiplication Becomes Simpler

- **Multiplicands**: \( c_1(x) = c_{11} \cdot x \) and \( c_2(x) = c_{21} \cdot x \).
- **Product**: \( c(x) = c_1(x) \cdot c_2(x) = c_{11} \cdot c_{21} \cdot x^2 \).
- Can we forget key switching?
  - Just view \( t = s^2 \) as the new secret key.
  - \( c(t) = m_1 \cdot m_2 \cdot t + \text{smeven} = m_1 \cdot m_2 + \text{smeven} \).
- Not quite: What if we want to add a ciphertext under key \( s \) to another ciphertext under \( s^2 \)?
NTRU–Based SWHE: Key Switching Becomes Simpler

- **Multiplicands**: \( c_1(x) = c_{11} \cdot x \) and \( c_2(x) = c_{21} \cdot x \).

- **Product**: \( c(x) = c_1(x) \cdot c_2(x) = c_{11} \cdot c_{21} \cdot x^2 \).

- **Aux(S,t)**: Choose \( e^* \leftarrow \chi \), and set \( e_{S,t} = 2e^* + 1 \).
  Output \( a_{S,t} = S \cdot e_{S,t} \cdot t^{-1} \). \( (e_{S,t} \cdot t^{-1} \) should look random.\)

- **SwitchKey(c,a_{S,t})**:
  - Suppose \( c \cdot S = e = m+smeven \).
  - New ciphertext is \( c' = c \cdot a_{S,t} \).
  - Then, \( c' \cdot t = (c \cdot a_{S,t})t = c(a_{S,t} \cdot t) = c(S \cdot e_{S,t}) = e \cdot e_{S,t} = m+smeven \).

- Noise increases multiplicatively.
Two ciphertexts under different keys:
- $c_1(x) = c_{11} \cdot x$ and $c_2(x) = c_{21} \cdot x$.
- $c_1(s_1) = m_1 \cdot s_1 + smeven$, $c_2(s_2) = m_2 \cdot s_2 + smeven$.

Product: $c_{11}c_{21}s_1s_2 = m_1m_2s_1s_2 + smeven = m_1m_2 + smeven$.

 LATV12: Cloud can (noninteractively) combine data encrypted under different keys.
Other SWHE Schemes

Insert your scheme here!
Thank You! Questions?
Bibliography

Bibliography

- [GHLPPSS12] Craig Gentry, Shai Halevi, Vadim Lyubashevsky, Chris Peikert, Joseph Silverman, and Nigel Smart. Unpublished observation regarding NTRU-Based FHE.


[SV10] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. PKC 2010.