

Homomorphic Evaluation of the AES Circuit

Craig Gentry, Shai Halevi, Nigel P. Smart

IBM Research
and
University Of Bristol.

August 22, 2012

Executive Summary

We present a working implementation of the (leveled) somewhat-HE scheme of BGV.

The implementation can evaluate (in reality) upto about 60 levels.

- ▶ Essentially circuits of degree at least 2^{60} .
- ▶ Due to extra tricks the effective degree is much larger

We use this to evaluate the AES circuit homomorphically

- ▶ Establishing a benchmark against which other implementations can be measured.

More importantly

- ▶ **On the way we develop some general optimization techniques**

Why Evaluate AES?

First Answer: **Why Not? It is as good as any other function**

Second Answer: Homomorphically decrypting AES-encrypted content could be important in some **future** applications

- ▶ Virus checking encrypted emails at a gateway

Third Answer: It presents a good design space to investigate FHE techniques

- ▶ Various implementation techniques known
- ▶ Parallel nature of the computation
- ▶ Algebraic nature of the computation

Fourth Answer: Used as a bench mark in MPC

- ▶ Allows us to see how far off FHE is, compared to Yao and general MPC.

Why Evaluate AES?

First Answer: **Why Not? It is as good as any other function**

Second Answer: Homomorphically decrypting AES-encrypted content could be important in some **future** applications

- ▶ Virus checking encrypted emails at a gateway

Third Answer: It presents a good design space to investigate FHE techniques

- ▶ Various implementation techniques known
- ▶ Parallel nature of the computation
- ▶ Algebraic nature of the computation

Fourth Answer: Used as a bench mark in MPC

- ▶ Allows us to see how far off FHE is, compared to Yao and general MPC.

Why Evaluate AES?

First Answer: **Why Not? It is as good as any other function**

Second Answer: Homomorphically decrypting AES-encrypted content could be important in some **future** applications

- ▶ Virus checking encrypted emails at a gateway

Third Answer: It presents a good design space to investigate FHE techniques

- ▶ Various implementation techniques known
- ▶ Parallel nature of the computation
- ▶ Algebraic nature of the computation

Fourth Answer: Used as a bench mark in MPC

- ▶ Allows us to see how far off FHE is, compared to Yao and general MPC.

Why Evaluate AES?

First Answer: **Why Not? It is as good as any other function**

Second Answer: Homomorphically decrypting AES-encrypted content could be important in some **future** applications

- ▶ Virus checking encrypted emails at a gateway

Third Answer: It presents a good design space to investigate FHE techniques

- ▶ Various implementation techniques known
- ▶ Parallel nature of the computation
- ▶ Algebraic nature of the computation

Fourth Answer: Used as a bench mark in MPC

- ▶ Allows us to see how far off FHE is, compared to Yao and general MPC.

Why BGV?

First Answer: **Why Not?**

- ▶ Differences between BGV and (say) Brakerski's scheme or the NTRU based scheme are minor
- ▶ BGV/Brakerski/NTRU seem significantly better than the older Integer/Ideal-Lattice based schemes.

Second Answer: Conceptually simpler

- ▶ NTRU and Brakerski schemes were not around when we started the work.

It is **not clear** which of BGV, NTRU and Brakerski is more efficient **in practice**.

- ▶ Each have different tradeoffs
- ▶ Need to duplicate the work in this paper for the other schemes to determine the exact comparisons.

Why BGV?

First Answer: **Why Not?**

- ▶ Differences between BGV and (say) Brakerski's scheme or the NTRU based scheme are minor
- ▶ BGV/Brakerski/NTRU seem significantly better than the older Integer/Ideal-Lattice based schemes.

Second Answer: Conceptually simpler

- ▶ NTRU and Brakerski schemes were not around when we started the work.

It is **not clear** which of BGV, NTRU and Brakerski is more efficient **in practice**.

- ▶ Each have different tradeoffs
- ▶ Need to duplicate the work in this paper for the other schemes to determine the exact comparisons.

Why BGV?

First Answer: **Why Not?**

- ▶ Differences between BGV and (say) Brakerski's scheme or the NTRU based scheme are minor
- ▶ BGV/Brakerski/NTRU seem significantly better than the older Integer/Ideal-Lattice based schemes.

Second Answer: Conceptually simpler

- ▶ NTRU and Brakerski schemes were not around when we started the work.

It is **not clear** which of BGV, NTRU and Brakerski is more efficient **in practice**.

- ▶ Each have different tradeoffs
- ▶ Need to duplicate the work in this paper for the other schemes to determine the exact comparisons.

BGV Basics

Ring: $R = \mathbb{Z}[X]/\Phi_m(X)$, where m is a parameter to fix later.

Reduction: $R_q = (R \text{ mod } q)$ for integer q (not necessarily prime).

Secret key is element $s \in R$ which is “small”

- ▶ The associated public key is an Ring-LWE tuple based on s
- ▶ This will not bother us here

We define a sequence of moduli (a.k.a. levels) $q_0 < q_1 < \dots < q_{L-1}$

BGV Basics

A ciphertext is a tuple $c = (c_0, c_1, t)$

- ▶ $c_0, c_1 \in R_{q_t}$

Decryption via

$$(c_0 - s \cdot c_1 \pmod{q_t}) \pmod{2}$$

to obtain message $m \in R_2$.

Addition, multiplication, modulus switching etc as per normal BGV

- ▶ See later for optimizations though

SIMD Operations

The parameter m is chosen so that $\Phi_m(X)$ splits into ℓ factors of degree d modulo 2

- ▶ For “sufficiently large” ℓ .

Following Smart-Vercauteren R_2 acts as ℓ copies of the finite field \mathbb{F}_{2^d} .

- ▶ Implies SIMD addition and multiplication operations on ciphertexts

Following [LPR10, BGV12, GHS12a] we can also homomorphically apply Galois automorphisms to the ciphertexts

- ▶ Squaring is “for free” (Frobenius action)
- ▶ Can move data from one plaintext slot to another “for free”

SIMD Operations

The parameter m is chosen so that $\Phi_m(X)$ splits into ℓ factors of degree d modulo 2

- ▶ For “sufficiently large” ℓ .

Following Smart-Vercauteren R_2 acts as ℓ copies of the finite field \mathbb{F}_{2^d} .

- ▶ Implies SIMD addition and multiplication operations on ciphertexts

Following [LPR10, BGV12, GHS12a] we can also homomorphically apply Galois automorphisms to the ciphertexts

- ▶ Squaring is “for free” (Frobenius action)
- ▶ Can move data from one plaintext slot to another “for free”

SIMD Operations

The parameter m is chosen so that $\Phi_m(X)$ splits into ℓ factors of degree d modulo 2

- ▶ For “sufficiently large” ℓ .

Following Smart-Vercauteren R_2 acts as ℓ copies of the finite field \mathbb{F}_{2^d} .

- ▶ Implies SIMD addition and multiplication operations on ciphertexts

Following [LPR10, BGV12, GHS12a] we can also homomorphically apply Galois automorphisms to the ciphertexts

- ▶ Squaring is “for free” (Frobenius action)
- ▶ Can move data from one plaintext slot to another “for free”

Data Representation

Elements in R_{q_t} can be held in many ways.

- ▶ e.g. as coefficients of a polynomial of degree $\phi(m) - 1 \bmod q_t$

We pick $q_t = \prod_{i=0}^t p_i$ for small primes p_i .

- ▶ Means mapping from mod q_t to mod q_{t-1} is trivial
- ▶ Hold anything modulo q_t via a CRT representation

We also pick p_i so that m divides $p_i - 1$.

- ▶ Means \mathbb{F}_{p_i} has an m th root of unity ζ_{p_i} in it.

Then hold a polynomial modulo p_i as the evaluation vector of the polynomial evaluated at $\zeta_{p_i}^j$.

- ▶ Basically polynomial-CRT representation.

Combining both together an element in R_{q_t} is held in a **double-CRT** representation.

Data Representation

Advantages:

In double-CRT multiplication (and addition) takes linear time

- ▶ Multiplication in polynomial representation is quadratic time.

Disadvantages:

Moving from double-CRT representation to polynomial representation (resp. vice-versa) is more expensive and is performed via

- ▶ FFT algorithm modulo p (resp. inverse-FFT)
- ▶ CRT (resp. polynomial reduction).

But polynomial representation seems necessary in some sub-procedures of BGV

- ▶ Encryption, Decryption, Modulus Switching, Key Switching

We adapt sub-procedures to reduce the number of conversions.

Modulus Switching

A modulus switch operation is to take a ciphertext modulo Q and replace it with a ciphertext modulo Q' .

- ▶ Assume $Q > Q'$

At the same time we scale the noise by a down by factor of Q/Q'

This allows noise control and enables us to evaluate large degree circuits.

We (basically) use the BGV modulus switch operation

- ▶ Modified to cope with our double-CRT representation
- ▶ Need to avoid as many FFT and inverse-FFT operations as possible

New KeySwitching

In various operations we have a ciphertext (d_0, d_1, d_2, t) , which decrypts via,

$$d_0 - s \cdot d_1 - s' \cdot d_2 \pmod{q_t}.$$

We would like to return it to decrypting via

$$c_0 - s \cdot c_1 \pmod{q_t}$$

Usual method is to hold lots of data in the public key and apply an expensive binary decomposition step

- ▶ In practice memory is a problem
- ▶ Want to hold one set of data for all modulo q_t

New trick:

- ▶ mod-switch upwards (increase the noise)
- ▶ Then do the keyswitch
- ▶ Then do a modulus switch to reduce the noise

New KeySwitching: Public Key Data

Pick a large modulus P and in the public key put a quasi-encryption of $P \cdot s'$ modulo $P \cdot q_{L_1}$

$$(b_{s,s'}, a_{s,s'}) \in R_{P \cdot q_{L-1}}^2$$

where

- ▶ $a_{s,s'} \in R_{P \cdot q_{L-1}}$
- ▶ Pick $e_{s,s'}$ from a small distribution
- ▶ $b_{s,s'} = a_{s,s'} \cdot s + 2 \cdot e_{s,s'} + P \cdot s'$

Note this is also can be interpreted as an encryption of $P \cdot s'$ modulo $P \cdot q_t$ for any $0 \leq t < L$.

- ▶ So we use the same data for every level

New KeySwitching: Operation

Input : (d_0, d_1, d_2)

To KeySwitch we set, modulo $P \cdot q_t$,

- ▶ $c'_0 = P \cdot d_0 + b_{s,s'} \cdot d_2$
- ▶ $c'_1 = P \cdot d_1 + a_{s,s'} \cdot d_2$.

The pair $c' = (c'_0, c'_1)$ is an encryption under s' of the message m with respect to the modulus $P \cdot q_t$.

- ▶ The noise is about P times what the original ciphertext noise was

Now reduce modulus back to q_t , and rescale the noise, by applying a modulus switch to q_t .

KeySwitching Application

We use KeySwitching in two places:

Mult: An encryption of $m \cdot m'$ is given by the ciphertext

- ▶ $d_0 = c_0 \cdot c'_0$
- ▶ $d_1 = c_0 \cdot c'_1 + c_1 \cdot c'_0$
- ▶ $d_2 = -c_1 \cdot c'_1$.

with respect to the keys s and $s' = s^2$.

Conjugation: For $\sigma \in \mathcal{G}$ an encryption of $\sigma(m)$ is given by the ciphertext

- ▶ $d_0 = \sigma(c_0)$
- ▶ $d_1 = 0$
- ▶ $d_2 = \sigma(c_1)$

with respect to the keys s and $s' = \sigma(s)$.

Level Switching

Each ciphertext also carries around a measure of how much noise it has

This is updated on each operation

We switch a level when this becomes too large

- ▶ See paper for details

Mainly this happens just **before** the input to a multiplication gate.

We also do a Modswitch from level $L - 1$ down to level $L - 2$ on encryption

- ▶ Useful to make sure invariants wrt noise estimates are consistent

Parameter Selection

We select the parameters for the various distributions and use the Lindner-Peikert analysis of ring-LWE to fix key sizes.

We aim for 80-bit security levels and come up with the following (rough) estimates for sizes:

L	$\phi(m)$	$\log_2(p_0)$	$\log_2(p_i)$	$\log_2(p_{L-1})$	$\log_2(P)$
10	9326	37.1	17.9	7.5	177.3
20	19434	38.1	18.4	8.1	368.8
30	29749	38.7	18.7	8.4	564.2
40	40199	39.2	18.9	8.6	762.2
50	50748	39.5	19.1	8.7	962.1
60	61376	39.8	19.2	8.9	1163.5
70	72071	40.0	19.3	9.0	1366.1
80	82823	40.2	19.4	9.1	1569.8
90	93623	40.4	19.5	9.2	1774.5

Picking Finite Fields

The **exact** choice of the lattice dimension $\phi(m)$ is going to depend on what finite fields \mathbb{F}_{2^n} one wants to represent in ones application

Recall we want to implement AES.

There are two natural choices for the underlying finite field \mathbb{F}_{2^n}

- ▶ \mathbb{F}_{2^8}
- ▶ \mathbb{F}_2

To realise one of these settings we require n to divide d and m to divide $2^d - 1$.

- ▶ A small number of prime factors of m are preferred.
- ▶ Want to maximise the number of SIMD slots $\ell = \phi(m)/d$.

Example Parameters : $n = 8$

L	m	$N = \phi(m)$	(d, ℓ)
10	11441	10752	(48,224)
20	34323	21504	(48,448)
30	31609	31104	(72,432)
40	54485	40960	(64,640)
50	59527	51840	(72,720)
60	68561	62208	(72,864)
70	82603	75264	(56,1344)
80	92837	84672	(56,1512)
90	124645	98304	(48,2048)

Example Parameters : $n = 1$

L	m	$N = \phi(m)$	(d, ℓ)
10	11023	10800	(45,240)
20	34323	21504	(48,448)
30	32377	32376	(57,568)
40	42799	42336	(21,2016)
50	54161	52800	(60,880)
60	85865	63360	(60,1056)
70	82603	75264	(56,1344)
80	101437	85672	(42,2016)
90	95281	94500	(45,2100)

AES Implementation

We developed three implementations:

- ▶ Packed Representation: One AES state packed into a single ciphertext (byte wise)
- ▶ Byte-Sliced: 16 ciphertexts needed to represent one AES state
- ▶ Bit-Sliced: 128 ciphertexts needed to represent one AES state

In all variants we could process **multiple** AES states in one operation due to the SIMD operations.

For the Bit-Sliced implementation used the low depth circuit of Boyar-Peralta

For the two Byte oriented implementations used the algebraic structure of the S-Box.

- ▶ The only non-linear component

Byte Oriented S-Box

Recall Frobenius is essentially for free (in terms of noise/levels).

Following Rivain and Prouff (CHES 2010) one S-Box application can be implemented via:






Input: ciphertext c	Level t	
// Compute $c_{254} = c^{-1}$		
1. $c_2 \leftarrow c \ggg 2$	t	// Frobenius $X \mapsto X^2$
2. $c_3 \leftarrow c \times c_2$	$t + 1$	// Multiplication
3. $c_{12} \leftarrow c_3 \ggg 4$	$t + 1$	// Frobenius $X \mapsto X^4$
4. $c_{14} \leftarrow c_{12} \times c_2$	$t + 2$	// Multiplication
5. $c_{15} \leftarrow c_{12} \times c_3$	$t + 2$	// Multiplication
6. $c_{240} \leftarrow c_{15} \ggg 16$	$t + 2$	// Frobenius $X \mapsto X^{16}$
7. $c_{254} \leftarrow c_{240} \times c_{14}$	$t + 3$	// Multiplication
// Affine transformation over \mathbb{F}_2		
8. $c'_{2j} \leftarrow c_{254} \ggg 2^j$ for $j = 0, 1, 2, \dots, 7$	$t + 3$	// Frobenius $X \mapsto X^{2^j}$
9. $c'' \leftarrow \gamma + \sum_{j=0}^7 \gamma_j \times c'_{2j}$	$t + 3.5$	// Linear combination over \mathbb{F}_{2^8}

Note: Level is an estimate as levels are consumed dynamically

Results

Run on [BlueCrystal](#), IBM machine owned by Uni Bristol

- ▶ Run on one core with 256GB RAM

	Packed	Byte-Sliced	Bit-Sliced
Number Levels Needed	60	50	60
Key Generation	43mn	22mn	20mn
FHE Encript AES State	2mn	25mn	1h
FHE Encrypt AES Key Schedule	23mn	4h	150h
Evaluate AES Round 1	7h	12h	
Evaluate AES Round 9	2h	5h	
Evaluate AES Round 10	28mn	4h	
Evaluate AES Encrypt	34h	65h	
Number SIMD Blocks	54	720	1056
Time Per Block	37mn	5mn	

Any Questions ?