6.889: New Developments in Cryptography February 15, 2011

Fully Homomorphic Encryption - Part II

Instructor: Boaz Barak *Scribe: Elette Boyle*

1 Overview

We continue our discussion on the fully homomorphic encryption scheme of van Dijk, Gentry, Halevi, and Vaikuntanathan [vDGHV10]. Last week we constructed a weakly homomorphic encryption scheme such that for any pair of properly generated ciphertexts $c \leftarrow \mathsf{Enc}(b)$, $c' \leftarrow \mathsf{Enc}(b')$, the sum or product of c, c' yields a value that decrypts to the sum (resp, product) of the original underlying plaintexts. That is, $\mathsf{Dec}(\mathsf{Add}(c, c')) = b \oplus b'$ and $\mathsf{Dec}(\mathsf{Mult}(c, c')) = b \cdot b'$. However, the scheme suffered from two problems: 1) the cipertext formed as the output of the Add or Mult operation does not have the same distribution as a ciphertext properly generated via the Enc algorithm, and 2) if we try to perform too many Add and Mult operations, we are no longer guaranteed decryption to the correct value (indeed, in the scheme, the noise in the ciphertext grows larger in each operation and will eventually lead to a nonempty intersection of the set of encryptions of 0 and 1).

As we discussed last week, these two problems will be solved if we can construct the following two operations:

- $\mathsf{ReRand}(X)$: takes as input any encryption of a bit b with noise $\leq 2^{n^{0.4}}$ (for example, in our scheme this includes the bit b itself!) and outputs a ciphertext with noise $\leq 2^{n^{0.5}}$ with the correct distribution $\mathsf{ReRand}(X) \approx_s \mathsf{Enc}^{2^{n^{0.5}}}(b)$.
- $\mathsf{Clean}(X)$: takes as input a ciphertext with noise $\leq 2^{n^{0.9}}$ and outputs a ciphertext for the same message with noise $\leq 2^{n^{0.3}}$. That is, it reduces the noise of the ciphertext.

Further, when combined with the original scheme, these operations will actually give us a *fully* homomorphic encryption scheme, by cleaning and rerandomizing the resulting ciphertext after every operation.

We showed how to do ReRand last week. Today, our focus will be on the more complicated operation: Clean.

2 Review from last week

We briefly review some information from last lecture. For formal definitions and discussions of these items, we refer the reader to the preceding set of scribe notes.

2.1 Notation

1. We use lowercase letters (such as t) for values that are polynomial in the security parameter n, and capital letters (such as N, P, Q, and R) for large numbers that have $\mathrm{poly}(n)$ digits (and hence are exponential in the security parameter n).
2. $\mathbb{Z}_N$: We denote by $\mathbb{Z}_N$ the set $\{0, \ldots, N-1\}$ and by $\mathbb{Z}_N^*$ the set $\{X \in \mathbb{Z}_N : \gcd(X, N) = 1\}$.

3. $\mathcal{P}(C)$: For a circuit C, we define $\mathcal{P}(C)$ to be the polynomial $\mathbb{Z}^m \to \mathbb{Z}$ that C computes.

4. $|f|_M$: For a polynomial $f : \mathbb{Z}^m \to \mathbb{Z}$ and $M \geq 0$, we define $|f|_M$ to be the maximum value attained by f over all possible inputs of size $\leq M$:

$$|f|_M = \max_{\substack{x_1,...,x_m \\ |x_i| \leq M}} |f(x_1, ..., x_m)|.$$

5. $\mathcal{E}^{\mathbf{E}}_{N,P}(b)$: We denote by $\mathcal{E}^{\mathbf{E}}_{N,P}(b) = \{X : X = RP + 2E + b \pmod N,\ R \in \mathbb{Z}_Q,\ E \in [-\mathbf{E}, +\mathbf{E}]\}$ the set of possible encryptions of b with parameter $\mathbf{E}$, public parameter N, and secret key P.

2.2 The weakly homomorphic scheme

The weakly homomorphic scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Add}, \mathsf{Mult})$ is defined as follows.

Key Generation: Draw a random n-bit prime P and a random n^4-bit prime Q. Set $N = PQ$, keep P as the secret key, and publish N as a public parameter.

Encryption: For $b \in \{0, 1\}$, we let $\mathsf{Enc}_{N,P}(b) = \mathsf{Enc}^{2^{\sqrt{n}}}_{N,P}(b)$, where $\mathsf{Enc}^{\mathbf{E}}_{N,P}(b)$ is the distribution defined as follows: choose $R \leftarrow_{\mathrm{R}} \mathbb{Z}_Q$ and $E \leftarrow_{\mathrm{R}} [-\mathbf{E}, +\mathbf{E}]$, and output $X = RP + 2E + b \pmod N$.

Decryption: To decrypt X, output $X - \lfloor X/P \rceil\, P \pmod 2$. (Where $\lfloor x \rceil$ denotes the integer closest to x.) In other words, find the nearest multiple of P to X, subtract it away to get the noise, and then the parity of the noise is the decrypted bit.

Addition: Given ciphertexts X, X', output $\mathsf{Add}_N(X, X') = X + X' \pmod N$.

Multiplication: Given ciphertexts X, X', output $\mathsf{Mult}_N(X, X') = X \cdot X' \pmod N$.

We refer the reader to last week's notes for a proof of semantic security of the scheme based on the Learning Divisor with Noise assumption (in addition to the definition of this assumption), and proof that the scheme satisfies a "noisy" homomorphism property.

3 Bootstrapping to a Fully Homomorphic Scheme

3.1 Clean and ReRand operations

As we discussed, to make the scheme fully homomorphic, it will suffice to add the following two operations.

- $\mathsf{Clean}(X)$ will take as input a ciphertext in $\mathcal{E}^{2^{n^{0.9}}}(b)$ and output a ciphertext in $\mathcal{E}^{2^{n^{0.3}}}(b)$. That is, it reduces the noise of the ciphertext.

- $\mathsf{ReRand}(X)$ will take as input a ciphertext in $\mathcal{E}^{2^{n^{0.4}}}(b)$ and output a ciphertext that distributed statistically close to the uniform distribution over $\mathcal{E}^{2^{\sqrt{n}}}(b)$, that is, $\mathsf{ReRand}(X) \approx_s \mathsf{Enc}(b)$.

Fully homomorphic encryption Together Clean and ReRand imply a fully homomorphic encryption scheme: we just change the definition of Mult and Add to apply Clean and ReRand as follows:

- $\mathsf{Add}(X, X') = \mathsf{ReRand}(\mathsf{Clean}(X + X' \pmod N))$.
- $\mathsf{Mult}(X, X') = \mathsf{ReRand}(\mathsf{Clean}(X \cdot X' \pmod N))$.

Note that in an actual implementation, we can increase efficiency by performing several additions and multiplications back-to-back, running a Clean only when the noise gets too large. The ReRand operation also only needs to be run a single time at the very end of the calculation.

3.2 Teaser: Lucas's Theorem

We will use the following result later in the proof of correctness of Clean. To simplify the discussion, we present the proof now and simply make reference to it later.

Theorem 3.1 (Lucas)**.** *If $a = \sum a_i p^i$ and $b = \sum b_i p^i$ for some integer p and $a_i, b_i \in [0, p) \cap \mathbb{Z}$, then*

$$\binom{a}{b} \equiv \prod \binom{a_i}{b_i} \pmod p.$$

Proof. We show the claim by equating the coefficient of x^b in the following two expansions of the formal polynomial $(x+1)^a \pmod p$:

$$\begin{aligned}\sum_k \binom{a}{k} x^k = (x+1)^a &= (x+1)^{\sum_i a_i p^i} \\ &= \prod_i (x+1)^{a_i p^i} \\ &\equiv \prod_i (x^{p^i} + 1)^{a_i} \pmod p.\end{aligned}$$

On the left-hand side, the coefficient of x^b is simply $\binom{a}{b}$. On the right-hand side, since each $a_i < p$, the only way to form x^b is to take the $x^{b_i p^i}$ term from the ith factor $(x^{p^i} + 1)^{a_i}$. The coefficient of $x^{b_i p^i}$ in the ith factor is precisely $\binom{a_i}{b_i}$. Thus, the coefficient of x^b in the entire right-hand expansion will be $\prod_i \binom{a_i}{b_i}$, as desired. □

3.3 Getting Clean using "wishful thinking"

Last week we started discussing how one could go about implementing the Clean functionality. Obtaining Clean appears to be very challenging, because, up until this point, any operation that we performed on ciphertexts (such as adding, multiplying, or re-randomizing) only *increased* the noise. In fact, it seems somewhat counterintuitive that we could decrease the noise of a ciphertext without knowing the secret key, since if you could decrease *too much* then you would be able to identify the underlying plaintext. Nevertheless, we saw that if the decryption algorithm of the weakly homomorphic scheme is sufficiently simple, then we can use the bootstrapping trick of Gentry [Gen09a, Gen09b, Gen10] to achieve our goal.

We can view the decryption algorithm Dec as a function from the bits of the secret key and the ciphertext into the message space $\{0, 1\}$. Since the function is efficient, it can be computed

by a polynomial-size Boolean circuit. Suppose further that we were lucky and it was also the case that $|\mathcal{P}(C)|_{2^{n^{0.1}}} < 2^{n^{0.3}}$; that is, for any input $(x_1, ..., x_m)$ to $\mathcal{P}(C)$ with $|x_i| \leq 2^{n^{0.1}}$, the value of $|\mathcal{P}(C)(x_1, ..., x_m)|$ is bounded by $2^{n^{0.3}}$. For example, this will hold if the polynomial $\mathcal{P}$ describing the decryption circuit has $\{0, 1\}$ coefficients and degree at most $n^{0.1}$. Then we could implement Clean as follows:

Recall that Clean is given as input $X = RP + 2E + b$ where $|E| \leq 2^{n^{0.9}}$ and must output a ciphertext $X' = R'P + 2E' + b$ such that $|E'| \leq 2^{n^{0.3}}$.

1. At the key generation stage, we publish additional public parameters corresponding to encryptions of the bits of the secret key P, each with small noise. Explicitly, let $Y_i = \mathsf{Enc}_{N,P}^{2^{n^{0.1}}}(P_i)$, where P_i is the i-th bit of the secret key. Publish $Y_1, ..., Y_n$. The noise value $2^{n^{0.1}}$ is smaller than the standard encryption parameter ($2^{\sqrt{n}}$) but is still large enough to ensure security.

2. Now, suppose we are a third party (without the secret key) who wishes to run Clean on some ciphertext X.

   (a) We take $Y_1, ..., Y_n$ from the public parameters; then define $Y_{n+1}, \ldots, Y_m$ (where $m = n + n^5$) according to the bits of the ciphertext X. That is, Y_{n+1} is 1 if the first bit of X is 1 and 0 otherwise, and so on.

   Note that we can think of the number 1 also as an encryption of 1 (after all, $1 = 0 \cdot P + 2 \cdot 0 + 1$) and similarly we can think of the number 0 also as an encryption of 0. We thus have ciphertexts $Y_1, ..., Y_m$ that are encryptions of the bits of the string $P||X$ (where $||$ denotes concatenation). Moreover, these ciphertexts Y_i have very low noise—indeed, each has noise at most $2^{n^{0.1}}$.

   (b) Now, we know that $\mathsf{Dec}(P||X) = b$, and so if we run the circuit C *homomorphically* on the ciphertexts $Y_1, ..., Y_m$ of $P||X$, then we should get a ciphertext X' encrypting b, which is exactly what we wanted! (Very cool!)

The last thing to check is that the noise of X' is not too large. But, since we started from ciphertexts with small noise, this will be the case if the decryption circuit is simple as hoped. Specifically, the final noise level will be at most $|\mathcal{P}(C)|_{2 \cdot 2^{n^{0.1}}+1}$.

The issue. The issue is that we unfortunately have no reason to believe our decryption circuit has small norm. Generally a circuit over $\{0, 1\}^m$ is expected to have polynomial degree about $m \sim n^5$, and indeed it seems that one can verify that the decryption circuit actually computes a polynomial of degree at least $n/100$, and will satisfy $|\mathcal{P}(C)|_1 > 2^{n/100}$. So we're off by a polynomial factor in the exponent.

We will tackle this issue by making an additional tweak to the encryption scheme, intended to "squash" the decryption circuit and make it of smaller degree.

A divergence— is decryption *inherently* complex? One can wonder whether it's an accident that we were unlucky, or there is something inherently fishy about trying to get an encryption scheme with a very simple decryption function. Consider the adversary's task in breaking an encryption scheme: he gets access to labeled ciphertexts $(X_1, \mathsf{Dec}(X_1)), \ldots, (X_{\mathrm{poly}(m)}, \mathsf{Dec}(X_m))$ and then gets a new ciphertext X^* and needs to guess $\mathsf{Dec}(X^*)$. This is exactly the task of *learning* the decryption function. But this should raise a red flag, since low degree polynomials are easy to

learn. Fortunately, this depends on the definition of "low"— the algorithms to learn polynomials of degree d take roughly time $\binom{n}{d}$, which means that for degree $n^{0.1}$ polynomials we can still hope for non-trivial security. This does mean however that we should not expect breaking the decryption to take more than $2^{n^{0.1}}$ time, even if we use a large noise parameter. Since no such algorithms are known for our decryption algorithm, this suggests we need to tweak our scheme to become *less secure* if we want to ensure it can be implemented by a low degree polynomial. This is indeed what we'll do.

4 Getting the Clean operation—squashing decryption

While at a high level, what we do amounts to squashing the decryption circuit, and in the papers it is described in this way, our goal is just to get the Clean operation in some way. Thus, we will just show what we do to implement Clean. We will not change the actual encryption and decryption algorithms at all, just add some public parameters. Thus, all the properties of our scheme (correctness, homomorphism, and rerandomization) will be preserved.

4.1 Sparse Subset Sum (SSS) Assumption

In order to accomplish our goal, we will need to rely on an additional cryptographic assumption, the Sparse Subset Sum assumption. This is a variant of the subset sum problem.

Subset Sum Problem The *subset sum* problem is the question, given $m' = \text{poly}(m)$ m-bit numbers $\alpha_1, \ldots, \alpha_{m'}$ and a target number $\beta \in [2^m]$, whether there exists a subset $T \subseteq [m']$ such that $\sum_{i \in T} \alpha_i = \beta \pmod{2^m}$. When m' is sufficiently large as a function of m, this is considered a hard problem. We will assume hardness of a average-case decision variant of this problem where the set T is relatively small (of size m^ϵ for some $\epsilon > 0$).

Sparse Subset Sum (SSS) Assumption For any $m, \epsilon > 0$ and $\beta \in [M = 2^m]$, we assume there is some $m' = \text{poly}(m)$ such that the following two distributions on $\alpha_1, \ldots, \alpha_{m'}$ are computationally indistinguishable:

1. $\alpha_1, \ldots, \alpha_{m'}$ are chosen randomly and independently in $[M]$.
2. We choose a set $T \subseteq [m']$ of size m^ϵ at random, for $i \notin T$ choose α_i randomly and independently from $[M]$, while on the other hand we ensure that $\sum_{i \in T} \alpha_i = \beta \pmod{M}$. (Technically we can do so by selecting $\alpha_i \in_R [M]$ randomly for $i \in T \setminus \{i^*\}$ and then setting $\alpha_{i^*} = \beta - \sum_{i \in T \setminus \{i^*\}} \alpha_i \pmod{M}$ for some chosen index $i^* \in T$).

Note that the SSS assumption implies that one can view the numbers $\alpha_1, \ldots, \alpha_{m'}$ as an "encryption" of the number β such that $\sum_{i \in T} \alpha_i = \beta \pmod{M}$, which can be recovered by someone one knows the secret set T.

4.2 Adding Public Parameters

We now return to the weakly homomorphic scheme. We want to include additional public parameters that won't break the security of the scheme (based on our new assumption), but that will help simplify the computation of the decryption circuit to the point where we can use the bootstrapping technique described in Section 3.3.

Recall the decryption algorithm takes a ciphertext X and computes $X - \lfloor X/P \rceil P \pmod 2$. The computationally expensive part of this evaluation is calculating $\lfloor X/P \rceil$. Thus, we will try to

"help it out" by giving it a value closely related to $1/P$.[1] Now, obviously we cannot publish this value in the clear, since it would allow anyone to decrypt. What we will instead do is *hide* it in the form of a subset sum within a list of random values. The location of the special set of indices T where the value is hidden will become an additional part of the secret key.

More explicitly, let $M = 2^{100n}$ and $\alpha_1, ..., \alpha_{m'}$ be random elements of $[M]$ subject to the constraint

$$\sum_{i \in T} \alpha_i = \lfloor M/P \rceil \pmod{M},$$

for a randomly selected subset $T \subseteq [m']$ of size $|T| = n^{1/100}$. (The number of random elements $m' = \text{poly}(n)$ that we need to hide T is dictated by the sparse subset sum assumption).

Publish: $\boxed{M \mid \alpha_1, ..., \alpha_{m'}}$

Note that by the SSS assumption, the list of elements α_i is indistinguishable from a list of completely random numbers, and so the scheme remains secure.

For each $i \in [m']$, define

$$t_i = \begin{cases} 1 & i \in T \\ 0 & i \notin T \end{cases}.$$

Let $T_i = \mathsf{Enc}_P^{2^{n^{0.1}}}(t_i)$ be an encryption of t_i with the small noise parameter $2^{n^{0.1}}$, and make the additional security assumption that it is safe to publish $T_1, ..., T_{m'}$ as a public parameter.

Publish: $\boxed{T_1 = \mathsf{Enc}(t_1) \mid \cdots \mid T_{m'} = \mathsf{Enc}(t_{m'})}$

The notion that the scheme remains secure even given an encryption of (part of) the secret key is a property known as *circular security*, and is a specific case of the more general notion of *key-dependent message (KDM) security*. See last week's notes for a brief discussion.

4.3 Implementation of Clean

We are now ready to implement the Clean operation. Again, we are given a ciphertext $X = RP + 2E + b$ where $|E| \leq 2^{n^{0.9}}$ and our goal is to come up with $X' = R'P + 2E' + b$ such that $|E'| \leq 2^{n^{0.3}}$. We can assume without loss of generality that the input X itself is even; otherwise, we can always add 1 to the input, run Clean, and then add 1 back to the output.

The Clean algorithm. Given input X, perform the following steps:

1. Take $\alpha_1, ..., \alpha_{m'}$ from the public parameters; for $i = 1, \ldots, m'$, compute $\tilde{\alpha}_i = X \cdot \alpha_i$.
2. Construct a mod 2 circuit $C(t_1, \ldots, t_{m'})$ that computes the following function:

$$C(t_1, \ldots, t_{m'}) = \left\lfloor \frac{\sum_{i=1}^{m'} \tilde{\alpha}_i \cdot t_i}{M} \right\rceil \pmod{2}. \tag{1}$$

   for any sparse input $(t_1, ..., t_{m'})$ (i.e., with Hamming weight $\leq |T| = n^{1/100}$).
3. Invoke C on the ciphertexts $T_1, \ldots, T_{m'}$ using Add, Mult in the place of addition and multiplication mod 2 to obtain the ciphertext X'.

[1] Indeed, for $M = 2^k$ a large power of 2, the value $\lfloor M/P \rfloor$ can be viewed as k bits of precision of the fraction $1/P$.

Analysis of Clean. We need to prove the following lemmas:

Lemma 4.1. *Given $\tilde{\alpha}_1, \ldots, \tilde{\alpha}_{m'}$ we can find a polynomial time circuit C satisfying (1) such that $|\mathcal{P}(C)|_{2^{n^{0.1}}} < 2^{n^{0.3}}$.*

Lemma 4.2. *Applying the circuit C to the true secret values $t_1, ..., t_{m'}$ gives*

$$b = \left\lfloor \frac{\sum_{i=1}^{m'} \tilde{\alpha}_i \cdot t_i}{M} \right\rceil \pmod 2.$$

That is, C serves the functionality of a decryption circuit.

We begin by proving Lemma 4.2, which will actually be the simpler of the two.

Proof of Lemma 4.2. We know that

$$\sum_{i=1}^{m'} \tilde{\alpha}_i \cdot t_i = \sum_{i=1}^{m'} X \cdot \alpha_i \cdot t_i = X \sum_{i \in T} \alpha_i = X(KM + \lfloor M/P \rceil)$$

for some integer multiple K. Now, if we divide the RHS by M we get

$$(*) = XK + X \lfloor M/P \rceil / M.$$

Since X is even XK is an even integer, and since $M \gg P$ (M is $100n$ bits while P is n bits), $X \lfloor M/P \rceil / M$ is within 0.01 distance to $\lfloor X/P \rceil$, meaning that $\lfloor (*) \rceil \pmod 2 = \lfloor X/P \rceil \pmod 2 = b$. □

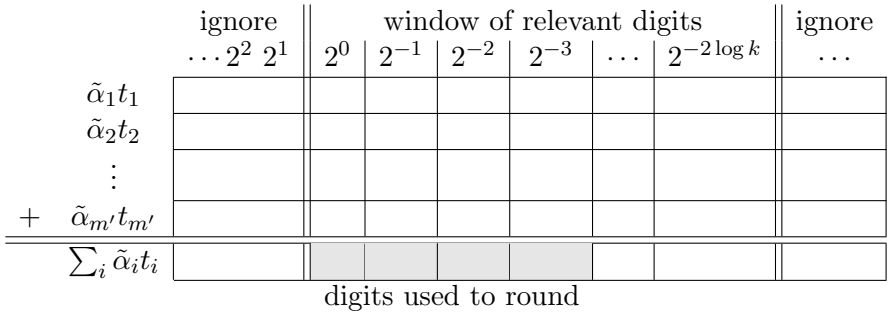
Proof of Lemma 4.1. We now argue that we can construct such a circuit with small norm. Consider the following observations.

1. Since M is a power of 2, dividing by M will simply shift the input by $\log M$ bits.
2. Multiplying $\tilde{\alpha}_i$ by t_i is just the identity function or 0, since $t_i \in \{0, 1\}$.
3. In the proof of Lemma 4.2 above, we showed that the number we end up rounding will be within distance 0.02 to an integer. So to determine whether it comes out to being odd or even, we just need to look at, say, four binary digits of the sum $\sum_i \tilde{\alpha}_i t_i$. (That is, strictly speaking, we won't prove Lemma 1 as stated but rather we'll come up with a circuit that agrees with the rounding operation when the number involved is within 0.02 distance from an integer, and we don't care what it does otherwise.)
4. Furthermore, since we know at most $k = |T| = n^{0.01}$ of the numbers in this sum are nonzero (since $t_i = 0 \ \forall i \notin T$), it suffices to look at a "window" of, say, $2 \log k$ digits within this sum. Indeed, digits corresponding to 2^1 and higher are irrelevant since we are considering the value mod 2, and digits below a certain point cannot make make any significant difference on the distance of the resulting sum from an integer.

So, ignoring all binary digits outside the $2 \log k$ window, our task boils down to coming up with a circuit for the following problem:

> Let $\hat{\alpha}_1, \ldots, \hat{\alpha}_{m'}$ be m' numbers of $2 \log k$ bits, and $t_1, \ldots, t_{m'}$ be numbers in $\{0, 1\}$ such that at most k of them are nonzero, and $j \in [3 \log k]$. Compute the j^{th} bit of $\sum \hat{\alpha}_i t_i$.

Figure 1: Calculating $\lfloor \sum_i \tilde{\alpha}_i t_i \rceil \pmod 2$. Shaded squares: Since the sum is close to an integer, we only need a few bits of precision to accurately round. Window of relevant digits: Since few terms $\tilde{\alpha}_i t_i$ are nonzero, we can disregard low-order bits (and bits 2^1 and higher since working mod 2).



We'll show one can do this in a circuit of $\text{poly}(m', k)$ size that computes a polynomial with 0/1 coefficients and $< k^3$ degree, hence completing the proof. This will follow from the following two claims:

Claim 4.3. *Let $a_1, \ldots, a_m \in \{0, 1\}$. The j^{th} bit of $\sum_{i=1}^m a_i$ is equal modulo 2 to*

$$\sum_{S \subseteq [m] : |S| = 2^j} \prod_{i \in S} a_i. \tag{2}$$

Proof. Let $s = \sum_{i=1}^m a_i$. Now, the value of Expression (2) is precisely the number of subsets $S \subseteq [m]$ of size 2^j such that $a_i = 1\ \forall i \in S$. This can be counted equivalently by $\binom{s}{2^j}$, since the sum s equals the number of a_i equal to 1. By Lucas's Theorem 3.1, we have

$$\binom{s}{2^j} \equiv \binom{s_j}{1} \prod_{k \neq j} \binom{s_k}{0} \pmod 2,$$

where s_k denotes the k^{th} bit of s. Indeed, this is because the binary representation of 2^j will have a single 1 in the j^{th} bit and 0s elsewhere. But, since $\binom{s_k}{0} = 1\ \forall k$, this product is just $\binom{s_j}{1} = s_j$, which is the j^{th} bit of $s = \sum_{i=1}^m a_i$ as desired. □

Claim 4.4. *There is an arithmetic circuit of size* $\text{poly}(m)$ *that computes the polynomial of (2) over the integers.*

Proof. We can compute the polynomial efficiently using dynamic programming. For any subset $A \subseteq [m]$ and integer $\ell \leq |A|$, we define the partial sum

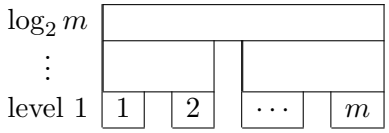
$$S_{A,\ell} = \sum_{\substack{S \subseteq A \\ |S| = \ell}} \prod_{i \in S} a_i.$$

Our goal is to calculate the value of this sum when $A = [m]$ and $\ell = 2^j$. To do so, we divide and conquer: we will calculate and maintain a table of partial sums for subsets of $[m]$, beginning with the singleton subsets, and then merging subsets in pairs until we reach $[m]$ itself. Explicitly, $S_{\{i\},1} = a_i$ for each i. For subsets $A, B \subseteq [m]$ with $|A| = |B|$ and $A \cap B = \emptyset$,

given the values of $S_{A,\ell'}$ and $S_{B,\ell'}$ $\forall \ell' \leq |A|$, we can calculate the value of $S_{A\cup B,\ell}$ using the following merge rule:

$$S_{A\cup B,\ell} = \sum_{\ell'=0}^{\ell} S_{A,\ell'} \cdot S_{B,\ell-\ell'}.$$

Storage: At each level i of $m/2^i$ different subsets of $[m]$ size 2^i; for each such subset A we have $|A|+1$ different partial sums $S_{A,\ell}$, corresponding to $\ell = 0, ..., |A|$. At any given time, we never need to store more than two levels of partial sums, since previous levels never appear again in calculation. Thus the total storage needed is $O(m)$.



Computation: Consider a single merge at level i. That is, let $A \subseteq [m]$ be the new merged subset of size 2^i. Using the merge rule above, for any given ℓ, the computation to calculate $S_{A,\ell}$ requires $O(\ell) \leq O(2^i)$ additions and multiplications. Since there are $|A| = 2^i$ different values of ℓ for this set A, and $m/2^i$ different choices for A at this level, the total computation to go from level $i-1$ to level i will be $O(m2^i) \leq O(m^2)$. Finally, there are $\log_2 m$ different levels, yielding an overall computation of $O(m^2 \log_2 m)$.

□

Completing the proof of Lemma 4.1 These two claims together allow us to compute the j-th digit of any collection of 0s and 1s. We can use this to find a digit in the sum of larger (namely, $2\log k$-bit) numbers by simply adding one column at a time in the bit representation and keeping track of carries, as in the standard long addition gradeschool algorithm.

Let a_i^ℓ be the ℓ-th digit of a_i. For $m < \ell$, we denote by $\mathsf{carry}_{m,\ell}$ the ℓ-th digit of the sum $\sum_i a_i^m$: that is, the contribution in location ℓ of the final sum that we get from adding the m-th column of 0/1 values. The j^{th} digit of the output $\sum_i a_i$ can be obtained by XORing the following values:

0. The j-th digit of $(\sum_i a_i^0)$
1. The $(j-1)$th digit of $\left(\sum_i a_i^1 + \mathsf{carry}_{0,1}\right)$
2. The $(j-2)$th digit of $\left(\sum_i a_i^2 + \mathsf{carry}_{0,2} + \mathsf{carry}_{1,2}\right)$
3. The $(j-3)$th digit of $\left(\sum_i a_i^3 + \mathsf{carry}_{0,3} + \mathsf{carry}_{1,3} + \mathsf{carry}_{2,3}\right)$

⋮

j. The 0th digit of $\left(\sum_i a_i^j + \mathsf{carry}_{0,j} + \mathsf{carry}_{1,j} + \mathsf{carry}_{2,j} + \cdots + \mathsf{carry}_{j-1,j}\right)$

From Claim 4.3, each term $\mathsf{carry}_{m\ell}$ can be computed by a polynomial with 0/1 coefficients and degree $2^{\ell-m}$. In turn, the $(j-i)$th digit of the appropriate sum of (values + carries) can be computed by a polynomial with 0/1 coefficients and total degree $2^{j-i} \cdot 2^i$, where the 2^i comes from the maximum degree of the terms within the sum (namely, from $\mathsf{carry}_{0,i}$). Further, by Claim 4.4, each of these steps can be implemented by an arithmetic circuit of

polynomial size. Since there are only polynomially many of these values to calculate, and $j \sim log|T| = \log n^{1/100}$, the corresponding polynomial will have sufficiently small norm.

Now, putting the final pieces together, the complete circuit to compute the value of (1) simply runs the above process 4 times to determine 4 digits of the sum needed for rounding, and then (since we know the value is close to an integer) implement a simplistic mod 2 rounding functionality that sends $0.000, 1.111 \mapsto 0$ and $0.111, 1.000 \mapsto 1$.

□

The Bottom line We have obtained a fully homomorphic private-key encryption! This is already good enough for our applications such as cloud computing, zero knowledge, and multiparty computation (where one side generates the keys and the other just applied Eval), but it is also very easy to transform it to a public-key encryption.

References

[Gen09a] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, 2009. `http://crypto.stanford.edu/craig`.

[Gen09b] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[Gen10] Craig Gentry. Computing arbitrary functions of encrypted data. *Communications of the ACM*, 53:97–105, March 2010.

[vDGHV10] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Proceedings of the 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT '10, pages 24–43, Berlin, Heidelberg, 2010. Springer-Verlag.