Manipulating Data while It Is Encrypted

Craig Gentry

IBM Watson

Lattice Crypto Day, May 2010

The Goal

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

Application: Private Google Search

I want to delegate processing of my data, without giving away access to it.

- Do a private Google search
  - You encrypt your query, so that Google cannot "see" it
- Somehow Google processes your encrypted query
  - You get an encrypted response, and decrypt it

Application: Cloud Computing

I want to delegate processing of my data, without giving away access to it.

- You store your files on the cloud
  - Encrypt them to protect your information
- Later, you want to retrieve files containing "cloud" within 5 words of "computing".
  - Cloud should return only these (encrypted) files, without knowing the key
- Privacy combo: Encrypted query on encrypted data

Outline

- Fully homomorphic encryption (FHE) at a high level
- A construction
- Known Attacks
- Performance / Implementation

Can we separate processing from access?

Actually, separating processing from access even makes sense in the physical world…

An Analogy: Alice's Jewelry Store

- Workers assemble raw materials into jewelry
- But Alice ... d about theft
  - How ca ... ers <u>process</u> the raw materi ... having <u>access</u> to the ... ?

An Analogy: Alice’s Jewelry Store

- Alice puts materials in locked glovebox
  - For which only she has the key
- Workers assemble jewelry in the box
- Alice unlocks box to get “results”

An Encryption Glovebox?

- Alice delegated processing without giving away access.
- But does this work for encryption?
  - Can we create an “encryption glovebox” that would allow the cloud to process data while it remains encrypted?

Public-key Encryption

- Three procedures: KeyGen, Enc, Dec
  - (sk,pk) ← KeyGen(λ)
    - Generate random public/secret key-pair
  - c ← Enc(pk, m)
    - Encrypt a message with the public key
  - m ← Dec(sk, c)
    - Decrypt a ciphertext with the secret key

Homomorphic Public-key Encryption

- Another procedure: Eval (for Evaluate)
  - $c \leftarrow \text{Eval}(pk, f, c_1, \ldots, c_t)$

Encryption of $f(m_1, \ldots, m_t)$.
I.e., $\text{Dec}(sk, c) = f(m_1, \ldots m_t)$

function

Encryptions of
inputs $m_1, \ldots, m_t$ to f

  - No info about $m_1, \ldots, m_t$, $f(m_1, \ldots m_t)$ is leaked
  - $f(m_1, \ldots m_t)$ is the "ring" made from raw materials $m_1, \ldots, m_t$ inside the encryption box

Concept due to Rivest, Adleman, Dertouzous (1978)

Fully Homomorphic Public-key Encryption

- Another procedure: Eval (for Evaluate)
  - $c \leftarrow \text{Eval}(pk, f, c_1,\ldots,c_t)$

Encryption of $f(m_1,\ldots,m_t)$.
I.e., $\text{Dec}(sk, c) = f(m_1, \ldots m_t)$

function

Encryptions of inputs $m_1,\ldots,m_t$ to f

  - FHE scheme should:
    - Work for *any* well-defined function f
    - Be *efficient*

Back to Our Applications

$c \leftarrow$ Eval(pk, f, $c_1,\ldots,c_t$),
Dec(sk, c) = f($m_1, \ldots, m_t$)

- Private Google search
  - Encrypt bits of my query: $c_i \leftarrow$ Enc(pk, m_i)
  - Send pk and the c_i's to Google
  - Google expresses its search algorithm as a boolean function f of a user query
  - Google sends $c \leftarrow$ Eval(pk, f, $c_1,\ldots,c_t$)
  - I decrypt to obtain my result f($m_1, \ldots, m_t$)

Back to Our Applications

$c \leftarrow$ Eval(pk, f, $c_1,\ldots,c_t$),
Dec(sk, c) = f(m_1, …, m_t)

- Cloud Computing with Privacy
  - Encrypt bits of my files $c_i \leftarrow$ Enc(pk, m_i)
  - Store pk and the c_i's on the cloud
  - Later, I send query :"cloud" within 5 words of "computing"
  - Let f be the boolean function representing the cloud's response if data was unencrypted
  - Cloud sends $c \leftarrow$ Eval(pk, f, $c_1,\ldots,c_t$)
  - I decrypt to obtain my result f(m_1, …, m_t)

FHE: What does "Efficient" Mean?

- $c \leftarrow$ Eval(pk, f, $c_1,\ldots,c_t$) is efficient:
  - runs in time $g(\lambda) \cdot T_f$, where g is a polynomial and T_f is the Turing complexity of f

- KeyGen, Enc, and Dec are efficient:
  - Run in time polynomial in λ
    - Alice's work should be *independent* of the complexity of f
      - In particular, ciphertexts output by Eval should look "normal"
    - The point is to *delegate* processing!!

We had "somewhat homomorphic" schemes in the past

- Eval only works for some functions f
  - RSA works for MULT gates (mod N)
  - Paillier, GM, work for ADD, XOR
  - BGN05 works for quadratic formulas
  - MGH08 works for low-degree polynomials
    - size of c $\leftarrow$ Eval(pk, f, $c_1,\ldots,c_t$) grows exponentially with degree of polynomial f.
  - Before 2009, no efficient FHE scheme

A Construction of FHE…

Not my original STOC09 scheme. Rather, a simpler scheme by Marten van Dijk, me, Shai Halevi, and Vinod Vaikuntanathan

Smart and Vercauteren described an optimization of the STOC09 scheme in PKC10.

Step 1: Construct a Useful "Somewhat Homomorphic" Scheme

Why a somewhat homomorphic scheme?

- Can't we construct a FHE scheme directly?
  - If I knew how, I would tell you.
  - Later…

somewhat hom. + bootstrappable $\rightarrow$ FHE

A homomorphic symmetric encryption

- Shared secret key: odd number p
- To encrypt a bit m in {0,1}:
  - Choose at random small r, large q
  - Output c = m + 2r + pq

    The "noise"

    Noise much smaller than p
    - Ciphertext is close to a multiple of p
    - m = LSB of distance to nearest multiple of p
- To decrypt c:
  - Output m = (c mod p) mod 2
    - m = c – p • [c/p] mod 2

      = c – [c/p] mod 2

      = LSB(c) XOR LSB([c/p])

A homomorphic symmetric encryption

- Shared secret key: odd number 101
- To encrypt a bit m in {0,1}:
  - Choose at random small r, large q
  - Output c = m + 2r + pq

    The "noise"

    Noise much smaller than p
    - Ciphertext is close to a multiple of p
    - m = LSB of distance to nearest multiple of p
- To decrypt c:
  - Output m = (c mod p) mod 2
    - m = c – p • [c/p] mod 2

      = c – [c/p] mod 2

      = LSB(c) XOR LSB([c/p])

A homomorphic symmetric encryption

- Shared secret key: odd number 101
- To encrypt a bit m in {0,1}: (say, m=1)
  - Choose at random small r, large q
  - Output c = m + 2r + pq (The "noise": m + 2r)
    - Ciphertext is close to a multiple of p
    - m = LSB of distance to nearest multiple of p

Noise much smaller than p

- To decrypt c:
  - Output m = (c mod p) mod 2
    - $m = c - p \cdot [c/p] \bmod 2$
      $= c - [c/p] \bmod 2$
      $= \mathrm{LSB}(c)\ \mathrm{XOR}\ \mathrm{LSB}([c/p])$

A homomorphic symmetric encryption

- Shared secret key: odd number 101
- To encrypt a bit m in {0,1}: (say, m=1)
  - Choose at random small r (=5), large q (=9)
  - Output c = m + 2r + pq
    - The "noise" (m + 2r)
    - Noise much smaller than p
    - Ciphertext is close to a multiple of p
    - m = LSB of distance to nearest multiple of p
- To decrypt c:
  - Output m = (c mod p) mod 2
    - m = c – p • [c/p] mod 2
      = c – [c/p] mod 2
      = LSB(c) XOR LSB([c/p])

A homomorphic symmetric encryption

- Shared secret key: odd number 101
- To encrypt a bit m in {0,1}: (say, m=1)
  - Choose at random small r (=5), large q (=9)
  - The "noise"
  - Output $c = m + 2r + pq = 11 + 909 = 920$
    - Ciphertext is close to a multiple of p
    - m = LSB of distance to nearest multiple of p
- To decrypt c:
  - Output $m = (c \bmod p) \bmod 2$
    - $m = c - p \cdot [c/p] \bmod 2$
      $= c - [c/p] \bmod 2$
      $= \text{LSB}(c) \text{ XOR } \text{LSB}([c/p])$

A homomorphic symmetric encryption

- Shared secret key: odd number 101
- To encrypt a bit m in {0,1}: (say, m=1)
  - Choose at random small r (=5), large q (=9)
  - Output c = m + 2r + pq = 11 + 909 = 920 (The "noise": m + 2r)
    - Ciphertext is close to a multiple of p
    - m = LSB of distance to nearest multiple of p
- To decrypt c:
  - Output m = (c mod p) mod 2 = 11 mod 2 = 1
    - m = c – p • [c/p] mod 2
      = c – [c/p] mod 2
      = LSB(c) XOR LSB([c/p])

Homomorphic Public-Key Encryption

- Secret key is an odd p as before
- Public key is many "encryptions of 0"
  - $x_i = [q_i p + 2r_i]_{x0}$ for $i=1,2,\ldots,n$
- $\text{Enc}_{pk}(m) = [\text{subset-sum}(x_i\text{'s})+m+2r]_{x0}$
- $\text{Dec}_{sk}(c) = (c \bmod p) \bmod 2$

Quite similar to Regev's '04 scheme. Main difference: we use much more aggressive parameters…

Security of E

- Approximate GCD (approx-gcd) Problem:
  - Given many $x_i = s_i + q_i p$, output p
  - Example params: $s_i \sim 2^{\lambda}$, $p \sim 2^{\lambda^2}$, $q_i \sim 2^{\lambda^5}$, where λ is security parameter
    - Best known attacks (lattices) require 2^{λ} time
- I'll discuss attacks on approx-gcd later
- Reduction:
  - if approx-gcd is hard, E is semantically secure

Why is E homomorphic?

- Basically because:
  - If you add or multiply two near-multiples of p, you get another near multiple of p…

Why is E homomorphic?

- $c_1 = m_1 + 2r_1 + q_1 p$, $c_2 = m_2 + 2r_2 + q_2 p$

Noise: Distance to nearest multiple of p

- $c_1 + c_2 = (m_1 + m_2) + 2(r_1 + r_2) + (q_1 + q_2)p$
  - $(m_1 + m_2) + 2(r_1 + r_2)$ still much smaller than p

  $\rightarrow c_1 + c_2 \bmod p = (m_1 + m_2) + 2(r_1 + r_2)$

  $\rightarrow (c_1 + c_2 \bmod p) \bmod 2 = m_1 + m_2 \bmod 2$

- $c_1 \times c_2 = (m_1 + 2r_1)(m_2 + 2r_2) + (c_1 q_2 + q_1 c_2 - q_1 q_2)p$
  - $(m_1 + 2r_1)(m_2 + 2r_2)$ still much smaller than p

  $\rightarrow c_1 x c_2 \bmod p = (m_1 + 2r_1)(m_2 + 2r_2)$

  $\rightarrow (c_1 x c_2 \bmod p) \bmod 2 = m_1 x m_2 \bmod 2$

Why is E homomorphic?

- $c_1 = m_1 + 2r_1 + q_1 p$, …, $c_t = m_t + 2r_t + q_t p$
- Let f be a multivariate poly with integer coefficients (sequence of +'s and x's)
- Let c = Eval$_E$(pk, f, c_1, …, c_t) = f(c_1, …, c_t)
  - Suppose this noise is much smaller than p
  - $f(c_1, \ldots, c_t) = f(m_1+2r_1, \ldots, m_t+2r_t) + qp$
  - Then (c mod p) mod 2 = $f(m_1, \ldots, m_t)$ mod 2

That's what we want!

Why is E *somewhat* homomorphic?

- 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 $q' \neq q$
  - $(c \bmod p) = f(m_1+2r_1, \ldots, m_t+2r_t) + (q-q')p$
  - $(c \bmod p) \bmod 2$

    $= f(m_1, \ldots, m_t) + (q-q') \bmod 2$

    $= ???$
- We say E can <u>handle</u> 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_E

Example of a Function that E Handle

- Elementary symmetric poly of degree d:

  $f(x_1, \ldots, x_t) = x_1 \cdot x_2 \cdot x_d + \ldots + x_{t-d+1} \cdot x_{t-d+2} \cdot x_t$

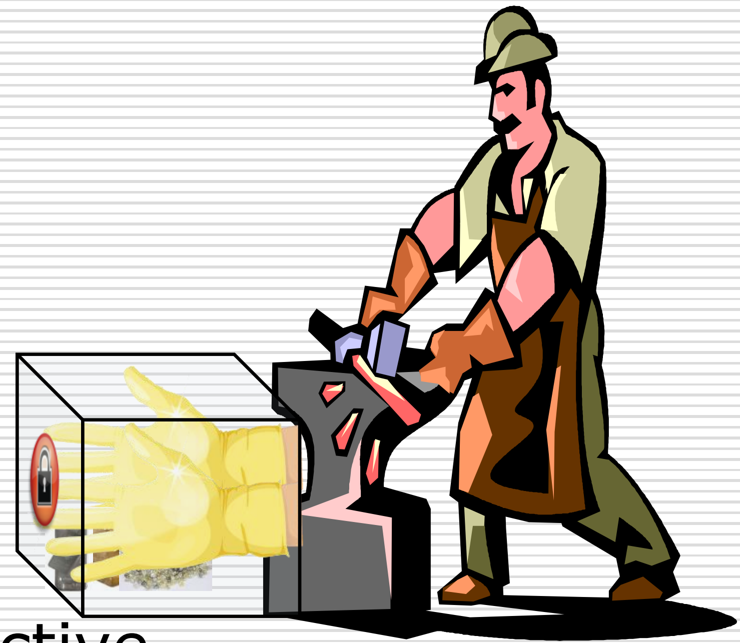
  - Has (t choose d) < t^d monomials: a lot!!
- If $|x_i|<B$, then $|f(x_1, \ldots, x_t)|<t^d \cdot B^d$
- E can handle f if:

  $t^d \cdot B^d < p/4 \;\rightarrow\;$ basically if: $d < (\log p)/(\log tB)$

- Example params: $B \sim 2^\lambda$, $p \sim 2^{\lambda^2}$
  - Eval_E can handle an elem symm poly of degree approximately λ.

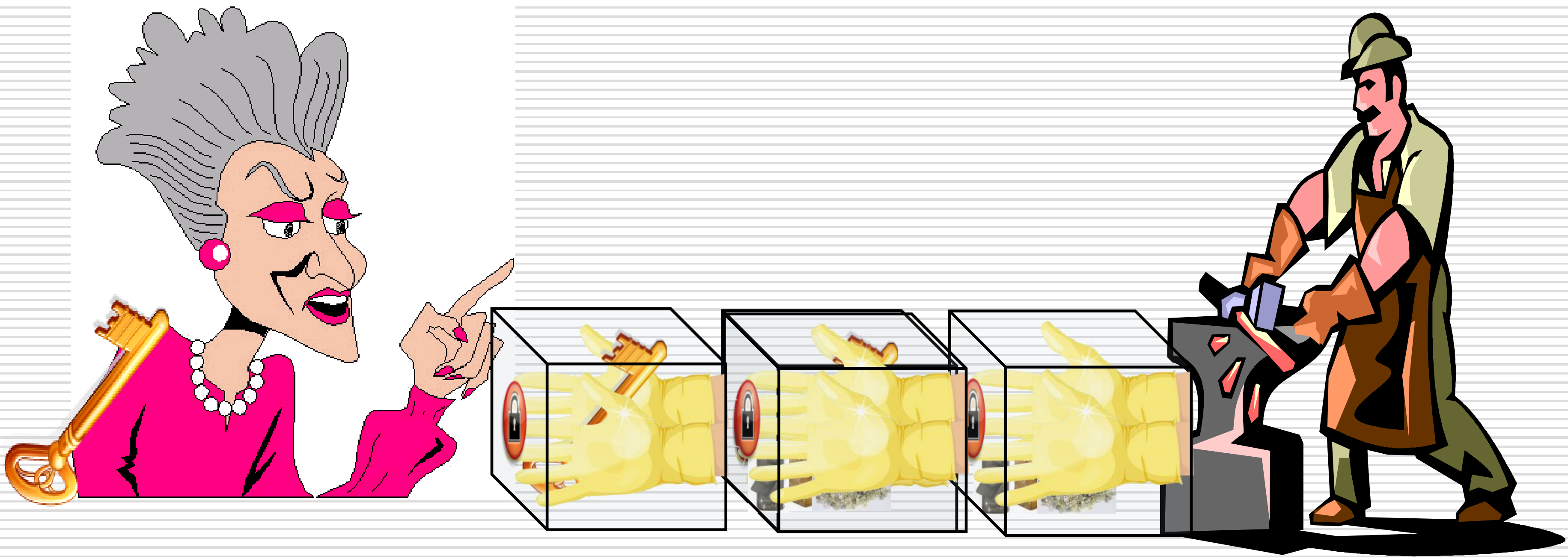
Step 2: Somewhat Homomorphic + Bootstrappable $\rightarrow$ FHE

Back to Alice's Jewelry Store



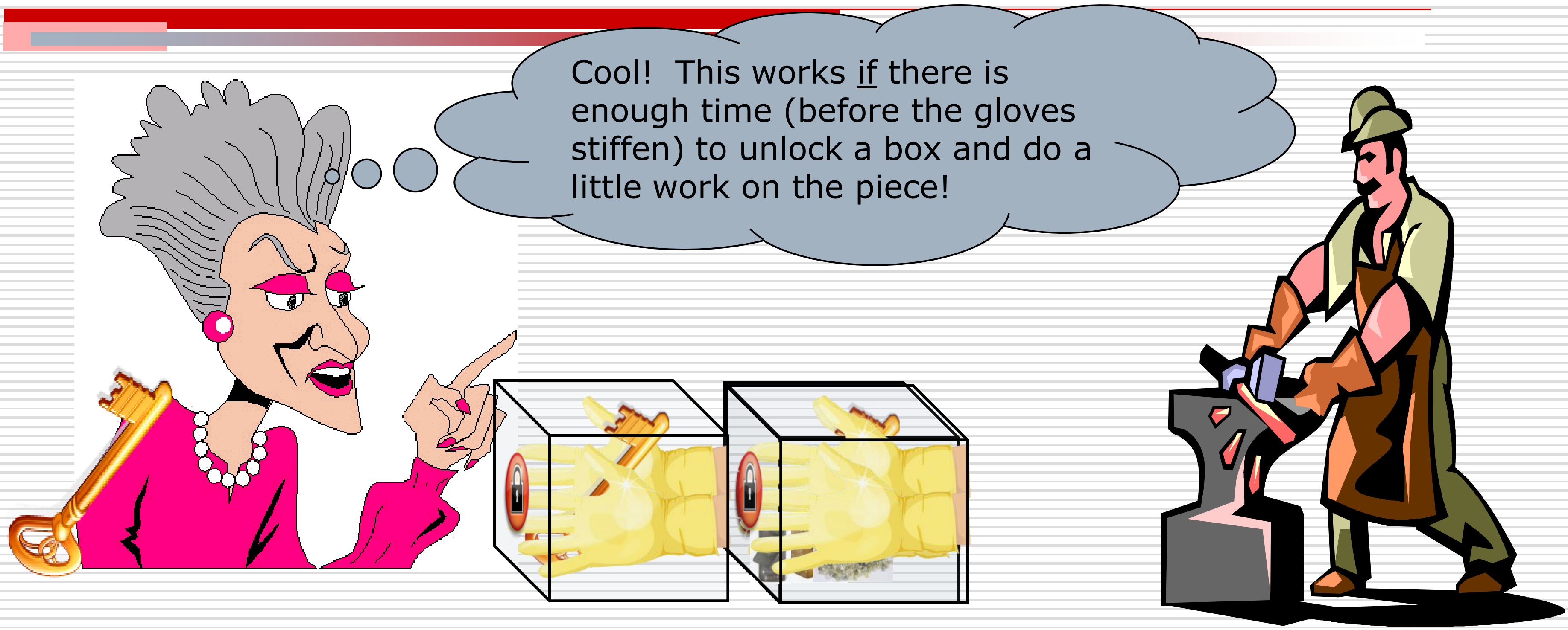
- Suppose Alice's boxes are defective.
  - After the worker works on the jewel for 1 minute, the gloves stiffen!
- Some complicated pieces take 10 minutes to make.
- Can Alice still use her boxes?
- Hint: you can put one box inside another.

Back to Alice's Jewelry Store



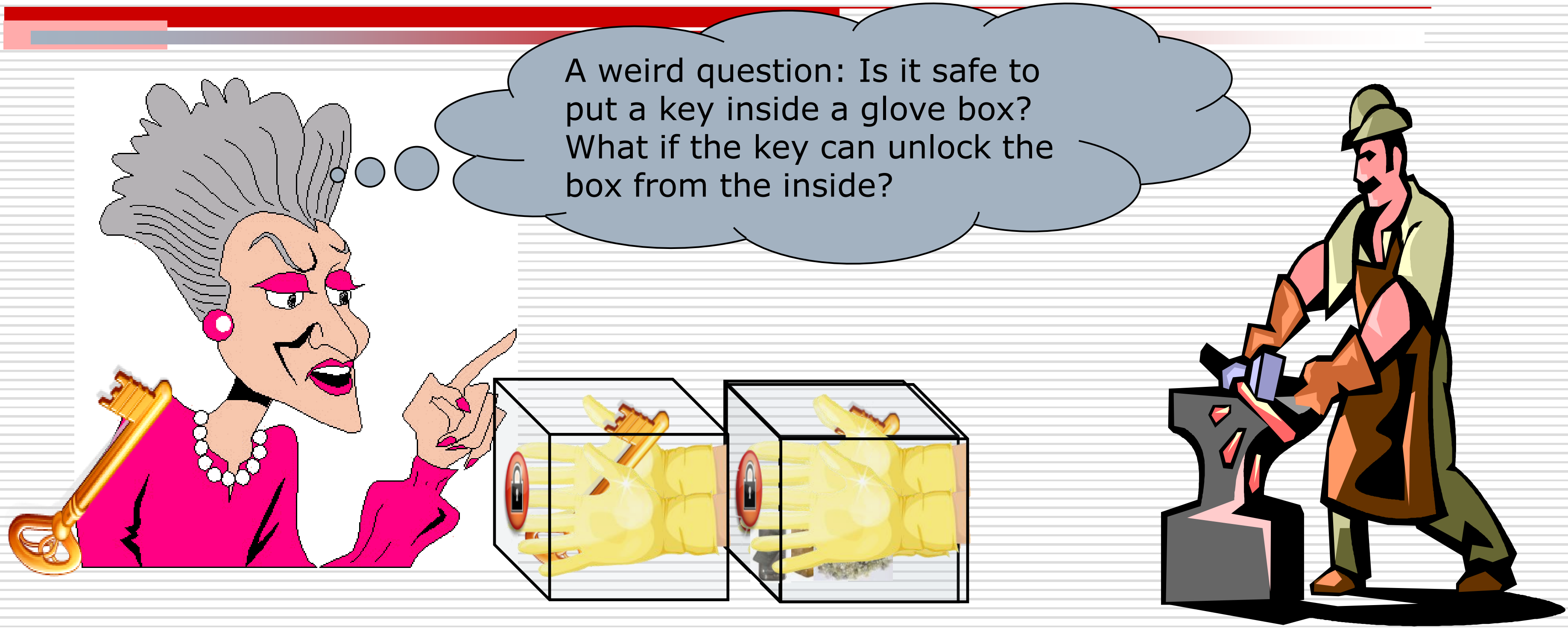
- Yes! Alice gives worker more boxes with a copy of her key
- Worker assembles jewel inside box #1 for 1 minute.
- Then, worker puts box #1 inside box #2!
- With box #2's gloves, worker opens box #1 with key, takes jewel out, and continues assembling till box #2's gloves stiffen.
- And so on…

Back to Alice's Jewelry Store



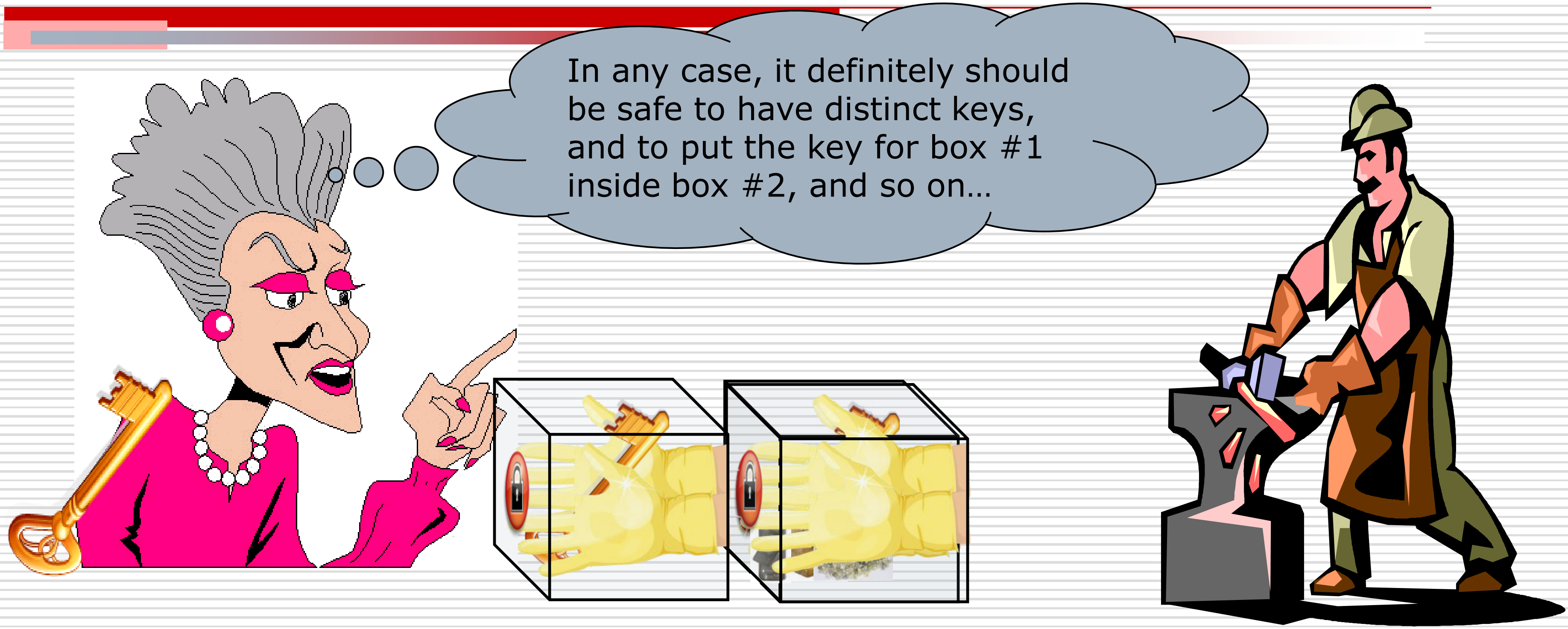
- Yes! Alice gives worker a boxes with a copy of her key
- Worker assembles jewel inside box #1 for 1
- Then, worker puts box #1 inside box #2!
- With box #2's gloves, worker opens box #1 with key, takes jewel out, and continues assembling till box #2's gloves stiffen.

Back to Alice's Jewelry Store



- Yes! Alice gives worker a boxes with a copy of her key
- Worker assembles jewel inside box #1 for 1
- Then, worker puts box #1 inside box #2!
- With box #2's gloves, worker opens box #1 with key, takes jewel out, and continues assembling till box #2's gloves stiffen.

Back to Alice’s Jewelry Store



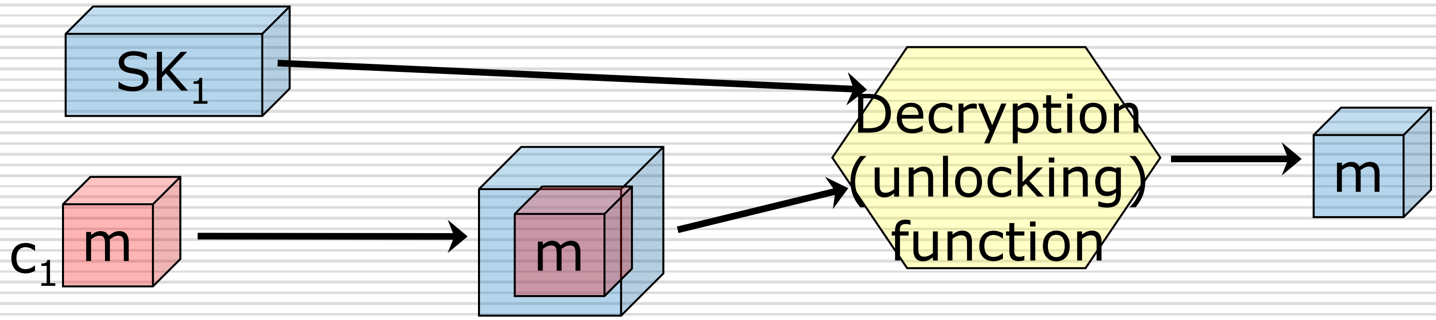
- Yes! Alice gives worker a boxes with a copy of her key
- Worker assembles jewel inside box #1 for 1
- Then, worker puts box #1 inside box #2!
- With box #2’s gloves, worker opens box #1 with key, takes jewel out, and continues assembling till box #2’s gloves stiffen.

How is it Analogous?

- Alice's jewelry store: Worker can assemble any piece if gloves can "handle" unlocking a box (plus a bit) before they stiffen
- Encryption:
  - If E can handle Dec_E (plus a bit), then we can use E to construct a FHE scheme E^{FHE}

Warm-up: Applying Eval to Dec_E

Blue means box #2.
It also means encrypted under key PK_2.



Red means box #1.
It also means encrypted under key PK_1.

Warm-up: Applying Eval to Dec_E

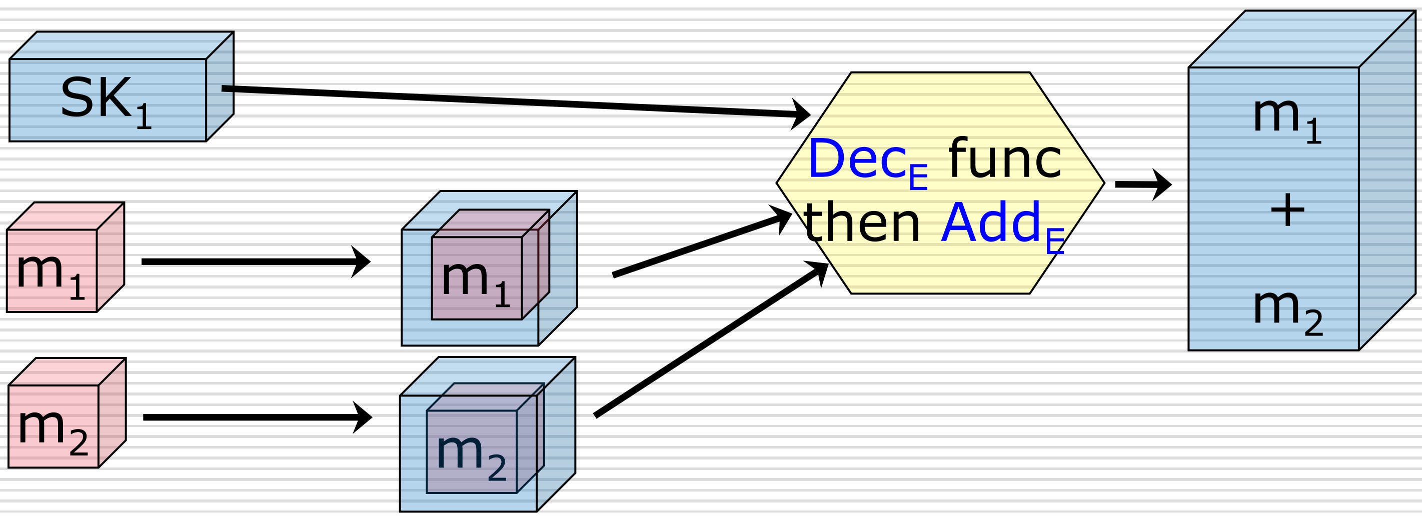
- Suppose c = Enc(pk, m)
- $\text{Dec}_E(sk_1^{(1)}, \ldots, sk_1^{(t)}, c_1^{(1)}, \ldots, c_1^{(u)}) = m$, where I have split sk and c into bits
- Let $sk_1^{(1)}$ and $c_1^{(1)}$, be ciphertexts that encrypt $sk_1^{(1)}$ and $c_1^{(1)}$, and so on, under pk_2.
- Then,

$$\text{Eval}(pk_2, \text{Dec}_E, sk_1^{(1)}, \ldots, sk_1^{(t)}, c_1^{(1)}, \ldots, c_1^{(1)}) = m$$

i.e., a ciphertext that encrypts m under pk_2.

Applying Eval to (Dec_E then Add_E)

Blue means box #2.
It also means encrypted under key PK_2.

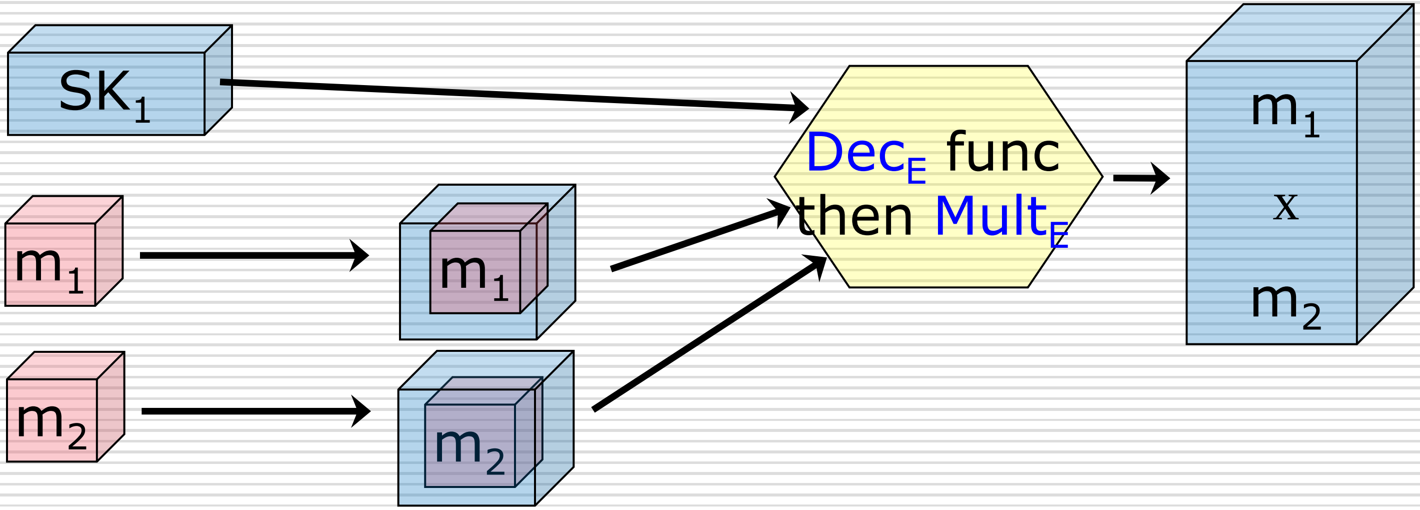


Red means box #1.
It also means encrypted under key PK_1.

Applying Eval to (Dec_E then $Mult_E$)

Blue means box #2.
It also means encrypted under key PK_2.

If E can evaluate (Dec_E then Add_E) and (Dec_E then $Mult_E$), then we call E "bootstrappable" (a self-referential property).

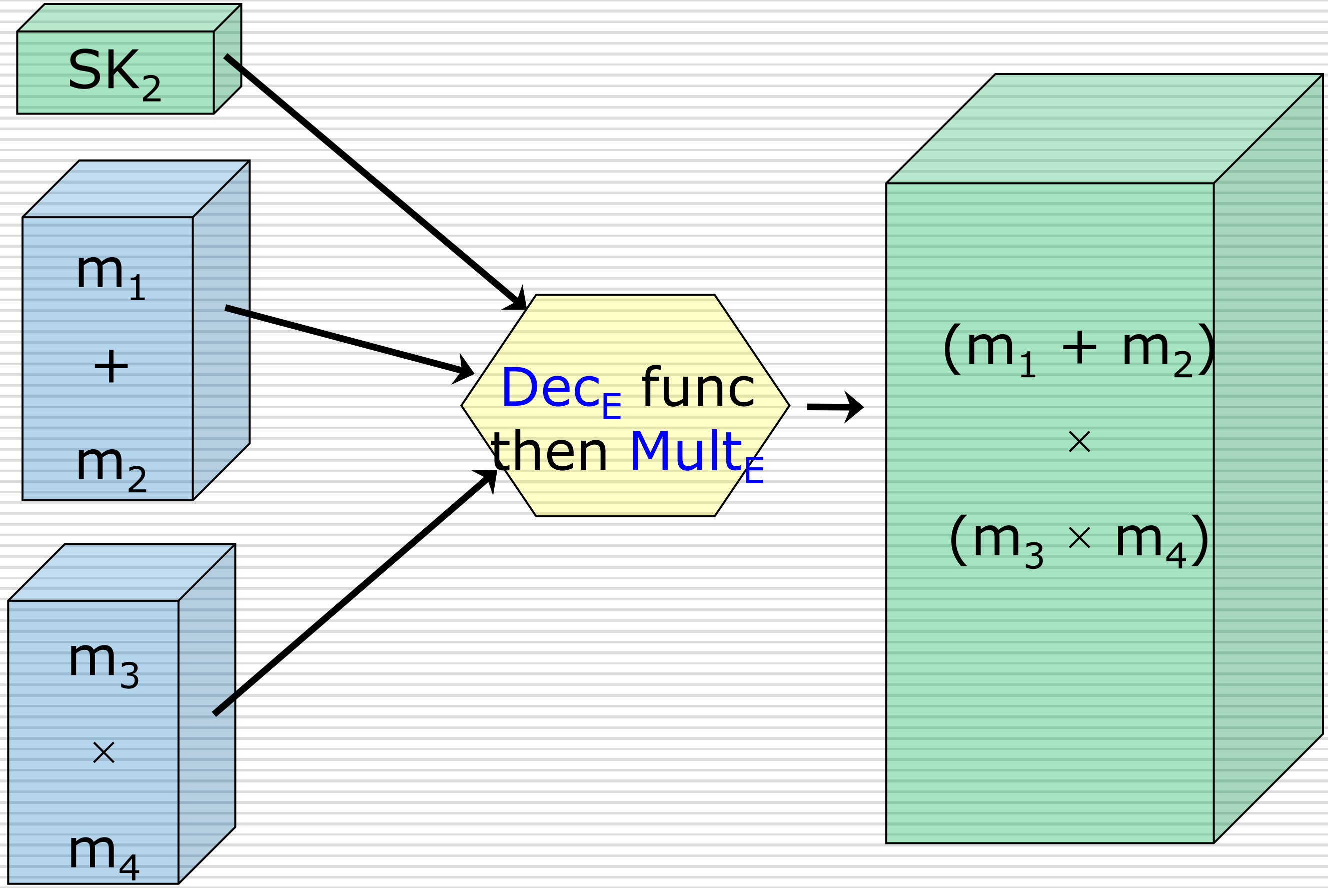


Red means box #1.
It also means encrypted under key PK_1.

And now the recursion…

Green means encrypted under PK_3.

Blue means encrypted under PK_2.



And so on…

Arbitrary Functions

- Suppose E is bootstrappable – i.e., it can handle Dec_E augmented by Add_E and $Mult_E$ efficiently.
- Then, there is a scheme E_d that evaluates arbitrary functions with d "levels".
- Ciphertexts: Same size in E_d as in E.
- Public key:
  - Consists of (d+1) E pub keys: $pk_0, \ldots, pk_d$
  - and encrypted secret keys: $\{Enc(pk_i, sk_{(i-1)})\}$
  - Size: linear in d. Constant in d, if you assume encryption is "circular secure."
    - The question of circular security is like whether it is "safe" to put a key for box i inside box i.

Step 2b: Is our Somewhat Homomorphic Scheme Already Bootstrappable?

No.

Why not?

- The boolean function $\text{Dec}_E(p,c)$ sets:

$$m = \text{LSB}(c) \text{ XOR } \text{LSB}([c/p])$$

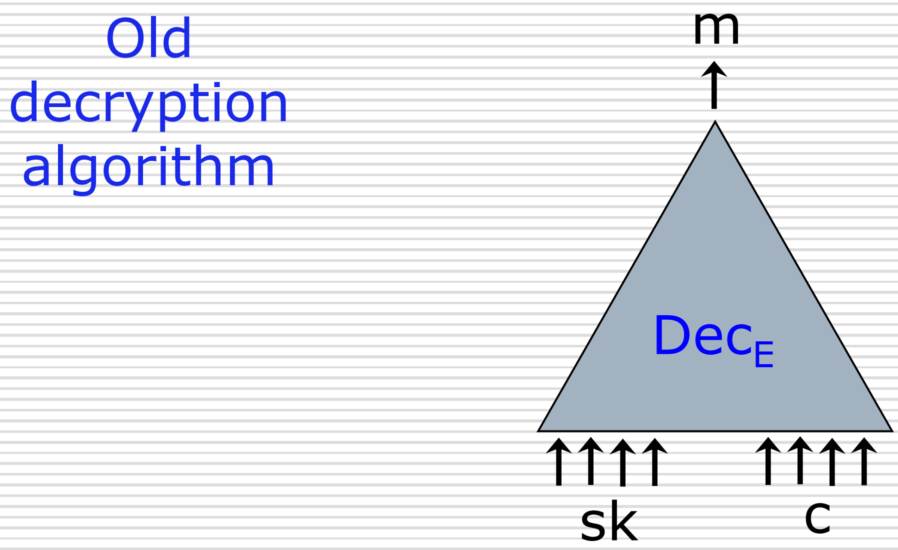
- Unfortunately, $f(c,p^{-1}) = \text{LSB}([c \times p^{-1}])$ is a high degree formula in the bits of c and p^{-1}.
  - If c and p each have t > log p bits, the degree is more than t.
  - But if f has degree > log p, then $|f(x_1, \ldots, x_t)|$ could definitely be bigger than p
    - And E can handle f only with guarantee that $|f(x_1, \ldots, x_t)| < p/4$
- E is not bootstrappable. ☹

Step 3 (Final Step): Modify our Somewhat Homomorphic Scheme to Make it Bootstrappable

The Goal

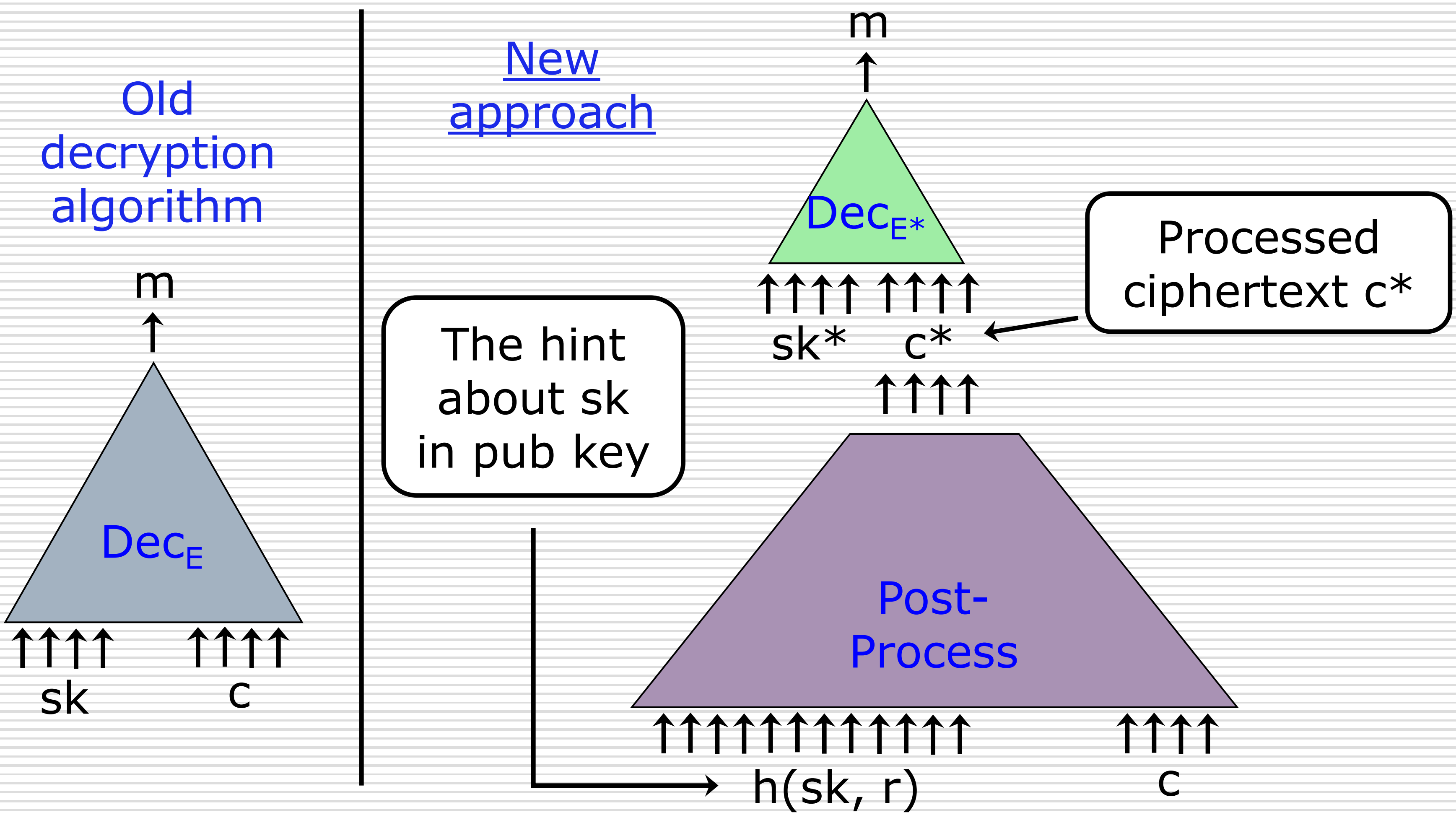
- Modify E $\rightarrow$ get E^* that is bootstrappable.
- Properties of E*
  - E* can handle any function that E can
  - Dec_{E*} is a lower-degree poly than Dec_E, so that E* can handle it

How do we "simplify" decryption?



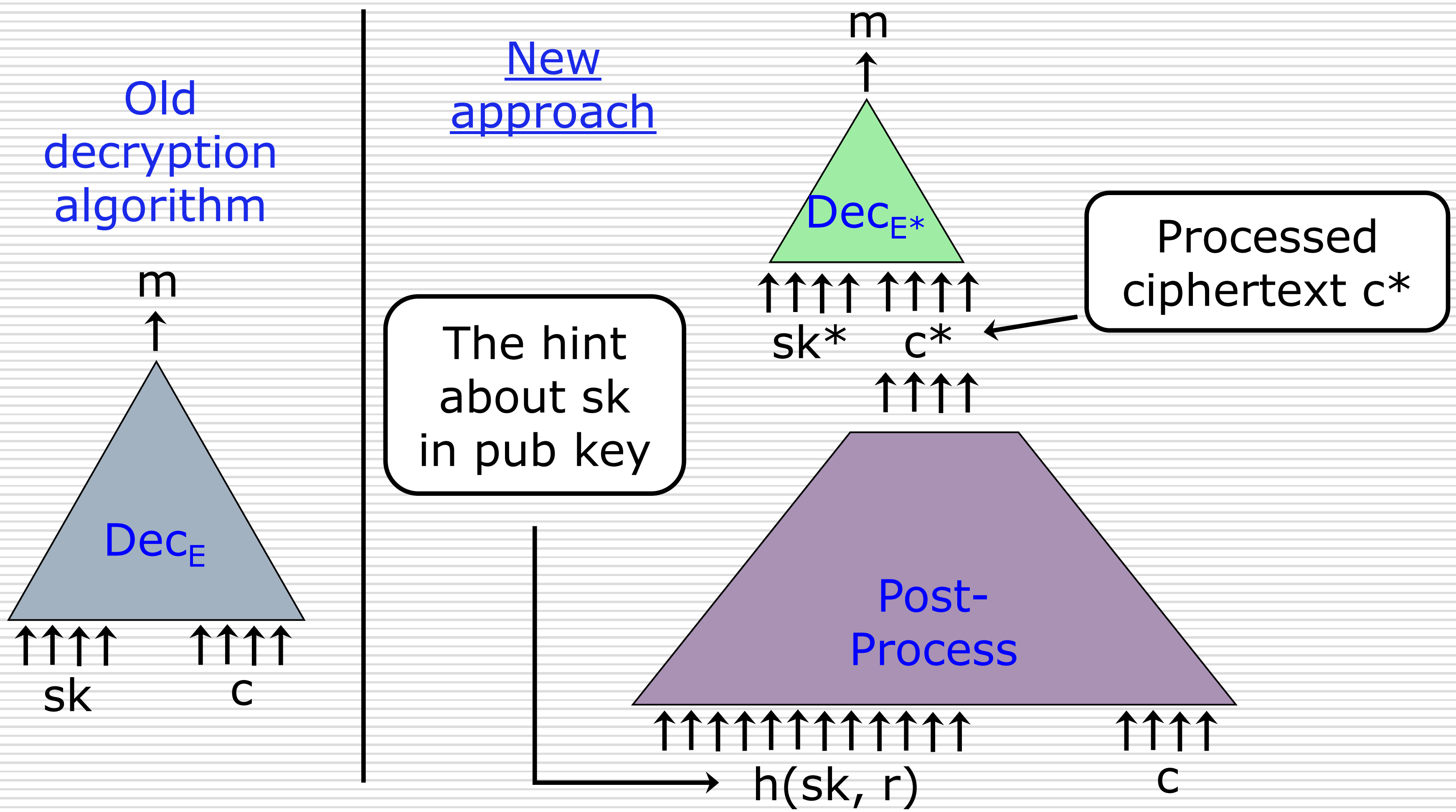
- Crazy idea: Put hint about sk in E* public key! Hint lets anyone post-process the ciphertext, leaving less work for Dec_{E*} to do.
- This idea is used in server-aided cryptography.

How do we “simplify” decryption?



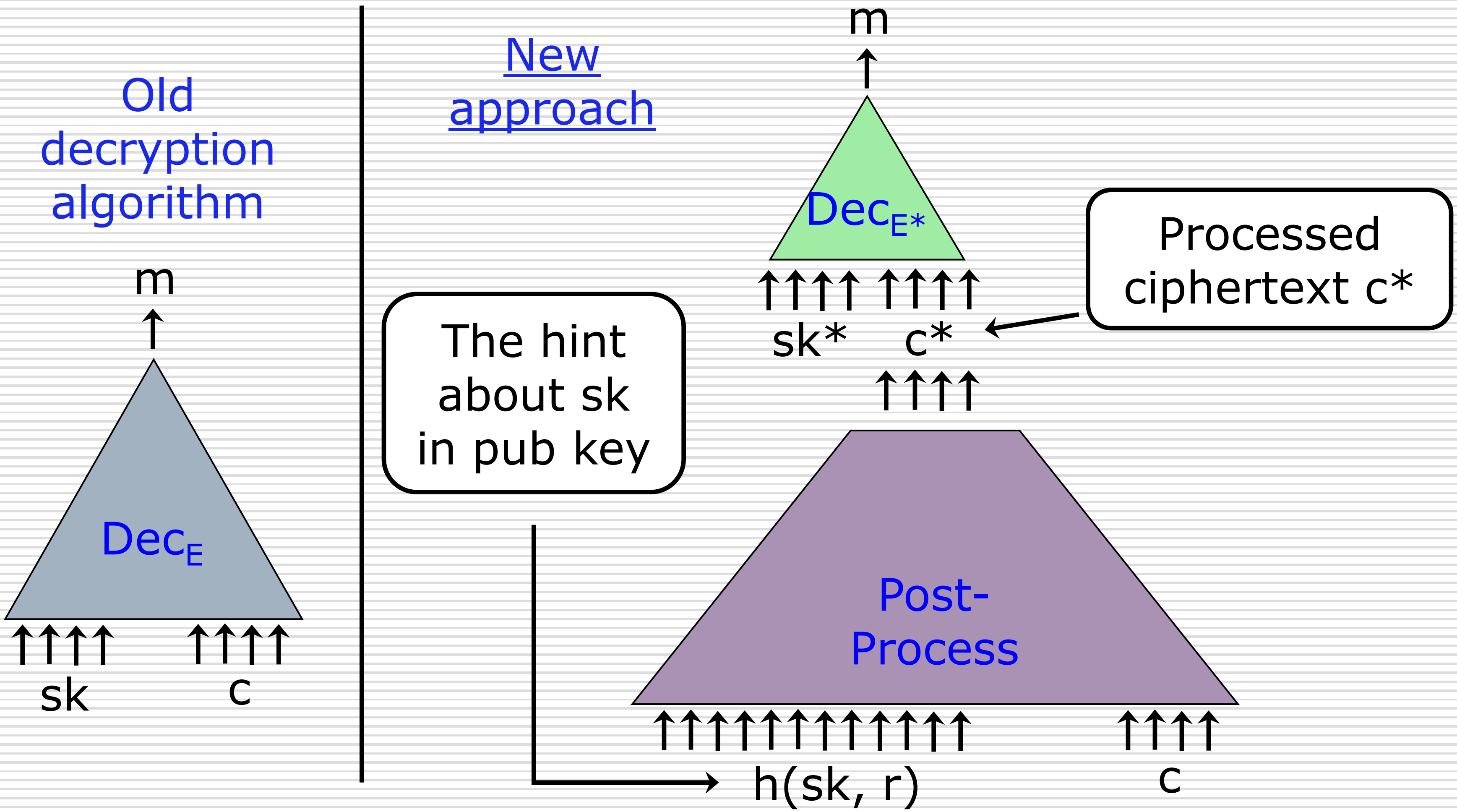
Hint in pub key lets anyone post-process the ciphertext, leaving less work for Dec$_{E*}$ to do.

How do we "simplify" decryption?



(Post-Process, Dec_{E^*}) should work on any c that Dec_E works on

How do we “simplify” decryption?



E* is semantically secure if E is, if h(sk,r) is computationally indistinguishable from h(0,r’) given sk, but not sk*.

Concretely, what is hint about p?

- E*'s pub key includes real numbers
  - $r_1, r_2, \ldots, r_n \in [0,2]$
  - $\exists$ sparse subset S for which $\Sigma_{i \in S}\, r_i = 1/p$
- Security: Sparse Subset Sum Prob (SSSP)
  - Given integers $x_1, \ldots, x_n$ with a subset S with $\Sigma_{i \in S}\, x_i = 0$, output S.
    - Studied w.r.t. server-aided cryptosystems
    - Potentially hard when $n > \log \max\{|x_i|\}$.
      - Then, there are exponentially many subsets T (not necessarily sparse) such that $\Sigma_{i \in S}\, x_i = 0$
    - Params: $n \sim \lambda^5$ and $|S| \sim \lambda$.
  - Reduction:
    - If SSSP is hard, our hint is indist. from h(0,r)

How E* works…

- Post-processing: output $\psi_i = c \times r_i$
  - Together with c itself
  - The ψ_i have about log n bits of precision
- New secret key is bit-vector $s_1, \ldots, s_n$
  - $s_i = 1$ if $i \in S$, $s_i = 0$ otherwise
- $Dec_{E*}(s,c) =$ LSB(c) XOR LSB($[\Sigma_i s_i \psi_i]$)
- E* can handle any function E can:
  - $c/p = c \Sigma_i s_i r_i = \Sigma_i s_i \psi_i$, up to precision
  - Precision errors do not changing the rounding
    - Precision errors from ψ_i imprecision $< 1/8$
    - c/p is with 1/4 of an integer

Are we bootstrappable yet?

- Dec_{E*}(s,c)= LSB(c) XOR LSB([Σ_i $s_i\psi_i$])
- Notice: s has low Hamming weight – namely |S|
- We can compute LSB([Σ_i $s_i\psi_i$]) as a low-degree poly (about |S|).
- To bootstrap:
  - Just make |S| smaller than the degree (about λ) that our scheme E* can handle!

Yay! We have a FHE scheme!

Great. But is it secure?

Known Attacks…

Two Problems We Hope Are Hard

- Approximate GCD (approx-gcd) Problem:
  - Given many $x_i = s_i + q_i p$, output p
  - Example params: $s_i \sim 2^{\lambda}$, $p \sim 2^{\lambda^2}$, $q_i \sim 2^{\lambda^5}$, where λ is security parameter
- Sparse Subset Sum Problem (SSSP)
  - Given integers $x_1, \ldots, x_n$ with a subset S with $\Sigma_{i \in S}\, x_i = 0$, output S.
  - Example params: $n \sim \lambda^5$ and $|S| \sim \lambda$.
  - (Studied by Phong and others in connection with server-aided cryptosystems.)

Hardness of Approximate-GCD

- Several lattice-based approaches for solving approximate-GCD
  - Related to Simultaneous Diophantine Approximation (SDA)
  - Studied in [Hawgrave-Graham01]
    - We considered some extensions of his attacks
- All run out of steam when $|q_i|>|p|^2$, where $|p|$ is number of bits of p
  - In our case $|p| \sim \lambda^2$, $|q_i| \sim \lambda^5 \gg |p|^2$

Relation to SDA

- $x_i = q_i p + r_i$ ($r_i \ll p \ll q_i$), $i = 0,1,2,\ldots$
  - $y_i = x_i/x_0 = (q_i+s_i)/q_0$, $s_i \sim r_i/p \ll 1$
  - $y_1, y_2, \ldots$ is an instance of SDA
    - q_0 is a denominator that approximates 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

$$L = \begin{pmatrix} R & x_1 & x_2 & \ldots & x_t \\ & -x_0 & & & \\ & & -x_0 & & \\ & & & \ldots & \\ & & & & -x_0 \end{pmatrix}$$

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 $\sim \det(L)^{1/t+1}$

Minkowski bound

  - This only holds for $t > |q_0|/|p|$
  - The dimension of the lattice is $t+1$
  - Quality of lattice-reduction deteriorates exponentially with t
  - When $|q_0| > (|p|)^2$ (so $t>|p|$), LLL-type reduction isn't good enough anymore

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.
    - $2^{|q_0|/|p|^2} = 2^\lambda$ time to get $2^{|p|} = p$ approx.

Minkowski bound

- Bottom line: no known eff. attack on approx-gcd

Lattice-based scheme seems "more secure"

- The security of the somewhat homomorphic scheme (quantumly) can be based on the *worst-case* hardness of SIVP over ideal lattices. (Crypto '10)
- This worst-case / average-case reduction is quite different from the reduction for ring-LWE [LPR EC'10]

A working implementation!!!

… and its surprisingly not-entirely-miserable performance

Performance

- Well, a little slow…
  - In E, a ciphertext is c_i is about λ^5 bits.
  - Dec_{E*} works in time quasi-linear in λ^5.
  - Applying $Eval_{E*}$ to Dec_{E*} takes quasi-λ^{10}.
    - To bootstrap E* to $E*^{FHE}$, and to compute $Eval_{E*FHE}(pk, f, c_1, \ldots, c_t)$, we apply $Eval_{E*}$ to Dec_{E*} once for each Add and Mult gate of f.
    - Total time: quasi- $\lambda^{10} \cdot S_f$, where S_f is the circuit complexity of f.

Performance

- STOC09 lattice-based scheme performs better:
  - Originally, applying Eval to Dec took $\tilde{O}(\lambda^6)$ computation if you want 2^λ security against known attacks.
  - Stehle and Steinfeld recently got the complexity down to $\tilde{O}(\lambda^3)$!

So what. Regev said $O(\lambda^2)$ is horrible in practice…

Ongoing work with Shai Halevi

But we have an implementation!

- Somewhat similar to [Smart-Vercauteren PKC'10]. But maybe better. ☺
- Initially planned to use IBM's Blue-Gene, but ended up not needing it
  - Implementation using NTL/GMP
  - Timing on a "strong" 1-CPU machine

Xeon E5440 / 2.83 GHz (64-bit, quad-core) 24 GB memory

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

Underlying Somewhat HE

- PK is 2 integers, SK is one integer

Dimension	KeyGen	Enc (amortized)	Dec	Degree
512 200,000-bit integers	0.16 sec	4 millisec	4 millisec	~200
2048 800,000-bit integers	1.25 sec	60 millisec	23 millisec	~200
8192 3,200,000-bit integers	10 sec	0.7 sec	0.12 sec	~200
32728 13,000,000-bit integers	95 sec	5.3 sec	0.6 sec	~200

Fully Homomorphic Scheme

- Re-Crypt polynomial of degree 15

Dimension	KeyGen	PK size	Re-Crypt
512 200,000-bit integers	2.4 sec	17 MByte	6 sec
2048 800,000-bit integers	40 sec	70 MByte	31 sec
8192 3,200,000-bit integers	8 min	285 MByte	3 min
32728 13,000,000-bit integers	2 hours	2.3 GByte	30 min

Thank You! Questions?

Can Eval_E handle Dec_E?

- The boolean function $\text{Dec}_E(p,c)$ sets:

$$m = \text{LSB}(c) \ \text{XOR} \ \text{LSB}([c/p])$$

- Can E handle (i.e., Evaluate) Dec_E followed by Add_E or Mult_E?
  - If so, then E is bootstrappable, and we can use E to construct an FHE scheme E^{FHE}.
- Most complicated part:

$$f(c,p^{-1}) = \text{LSB}([c \times p^{-1}])$$

  - The numbers c and p^{-1} are in binary rep.

Multiplying Numbers

$f(c,p^{-1}) = LSB([c \times p^{-1}])$

- Let's multiply a and b, rep'd in binary:

$$(a_t, \ldots, a_0) \times (b_t, \ldots, b_0)$$

- It involves adding the t+1 numbers:

			a_0b_t	a_0b_{t-1}	…	a_0b_1	a_0b_0
		a_1b_t	a_1b_{t-1}	$a1b_{t-2}$	…	a_1b_1	0
	…	…	…	…	…	…	…
a_tb_t	…	a_tb_1	a_tb_0	0	…	0	0

Adding Two Numbers

$f(c,p^{-1}) = LSB([c \times p^{-1}])$

Carries:	$x_1y_1+x_1x_0y_0+y_1x_0y_0$	x_0y_0	
	x_2	x_1	x_0
	y_2	y_1	y_0
Sum:	$x_2+y_2+x_1y_1+x_1x_0y_0+y_1x_0y_0$	$x_1+y_1+x_0y_0$	x_0+y_0

- Adding two t-bit numbers:
  - Bit of the sum = up to t-degree poly of input bits

Adding Many Numbers

$f(c,p^{-1}) = LSB([c\times p^{-1}])$

- 3-for-2 trick:
  - 3 numbers → 2 numbers with same sum
  - Output bits are up to degree-2 in input bits

	x_2	x_1	x_0
	y_2	y_1	y_0
	z_2	z_1	z_0
	$x_2+y_2+z_2$	$x_1+y_1+z_1$	$x_0+y_0+z_0$
$x_2y_2+x_2z_2+y_2z_2$	$x_1y_1+x_1z_1+y_1z_1$	$x_0y_0+x_0z_0+y_0z_0$	

  - t numbers → 2 numbers with same sum
    - Output bits are degree $2^{\log_{3/2} t} = t^{\log_{3/2} 2} = t^{1.71}$

Back to Multiplying

$f(c,p^{-1}) = LSB([c \times p^{-1}])$

- Multiplying two t-bit numbers:
  - Add t t-bit numbers of degree 2
    - 3-for-2 trick $\rightarrow$ two t-bit numbers, deg. $2t^{1.71}$.
    - Adding final 2 numbers $\rightarrow$ deg. $t(2t^{1.71}) = 2t^{2.71}$.
- Consider $f(c,p^{-1}) = LSB([c \times p^{-1}])$
  - p^{-1} must have $\log c > \log p$ bits of precision to ensure the rounding is correct
  - So, f has degree at least $2(\log p)^{2.71}$.
- Can our scheme E handle a polynomial f of such high degree?
  - Unfortunately, no.

$f(c,p^{-1}) = LSB([c \times p^{-1}])$

Why Isn't E Bootstrappable?

- Recall: E can <u>handle</u> 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_E
- If f has degree > log p, then $|f(x_1, \ldots, x_t)|$ could definitely be bigger than p
  - E is (apparently) not bootstrappable…

A Different Way to Add Numbers

- $\mathrm{Dec}_{E*}(s,c) = \mathrm{LSB}(c)$ XOR $\mathrm{LSB}([\Sigma_i\ s_i \psi_i])$

A Different Way to Add Numbers

- $Dec_{E*}(s,c) = LSB(c)$ XOR $LSB([\Sigma_i\ s_i\psi_i])$

$a_{1,0}$	$a_{1,-1}$	…	$a_{1,-\log n}$
$a_{2,0}$	$a_{2,-1}$	…	$a_{2,-\log n}$
$a_{3,0}$	$a_{3,-1}$	…	$a_{3,-\log n}$
$a_{4,0}$	$a_{4,-1}$	…	$a_{4,-\log n}$
$a_{5,0}$	$a_{5,-1}$	…	$a_{5,-\log n}$
…	…	…	…
$a_{n,0}$	$a_{n,-1}$	…	$a_{n,-\log n}$

A Different Way to Add Numbers

- $Dec_{E*}(s,c)$= LSB(c) XOR LSB($[\Sigma_i s_i\psi_i]$)

Let b_0 be the binary rep of Hamming weight

$a_{1,0}$	$a_{1,-1}$	…	$a_{1,-\log n}$
$a_{2,0}$	$a_{2,-1}$	…	$a_{2,-\log n}$
$a_{3,0}$	$a_{3,-1}$	…	$a_{3,-\log n}$
$a_{4,0}$	$a_{4,-1}$	…	$a_{4,-\log n}$
$a_{5,0}$	$a_{5,-1}$	…	$a_{5,-\log n}$
…	…	…	…
$a_{n,0}$	$a_{n,-1}$	…	$a_{n,-\log n}$

$b_{0,\log n}$	…	$b_{0,1}$	$b_{0,0}$			

A Different Way to Add Numbers

- $Dec_{E*}(s,c) = LSB(c)$ XOR $LSB([\Sigma_i s_i \psi_i])$

Let b_{-1} be the binary rep of Hamming weight

$a_{1,0}$	$a_{1,-1}$	…	$a_{1,-\log n}$
$a_{2,0}$	$a_{2,-1}$	…	$a_{2,-\log n}$
$a_{3,0}$	$a_{3,-1}$	…	$a_{3,-\log n}$
$a_{4,0}$	$a_{4,-1}$	…	$a_{4,-\log n}$
$a_{5,0}$	$a_{5,-1}$	…	$a_{5,-\log n}$
…	…	…	…
$a_{n,0}$	$a_{n,-1}$	…	$a_{n,-\log n}$

$b_{0,\log n}$	…	$b_{0,1}$	$b_{0,0}$			
	$b_{-1,\log n}$	…	$b_{-1,1}$	$b_{-1,0}$		

A Different Way to Add Numbers

- $Dec_{E*}(s,c) = LSB(c)$ XOR $LSB([\Sigma_i\, s_i\psi_i])$

Let $b_{-\log n}$ be the binary rep of Hamming weight

$a_{1,0}$	$a_{1,-1}$	…	$a_{1,-\log n}$
$a_{2,0}$	$a_{2,-1}$	…	$a_{2,-\log n}$
$a_{3,0}$	$a_{3,-1}$	…	$a_{3,-\log n}$
$a_{4,0}$	$a_{4,-1}$	…	$a_{4,-\log n}$
$a_{5,0}$	$a_{5,-1}$	…	$a_{5,-\log n}$
…	…	…	…
$a_{n,0}$	$a_{n,-1}$	…	$a_{n,-\log n}$

$b_{0,\log n}$	…	$b_{0,1}$	$b_{0,0}$			
	$b_{-1,\log n}$	…	$b_{-1,1}$	$b_{-1,0}$		
		…	…	…	…	
			$b_{-\log n,\log n}$	…	$b_{-\log n,1}$	$b_{-\log n,0}$

A Different Way to Add Numbers

- $Dec_{E*}(s,c) = LSB(c)$ XOR $LSB([\Sigma_i\, s_i\psi_i])$

Only log n numbers with log n bits of precision. Easy to handle.

$a_{1,0}$	$a_{1,-1}$	…	$a_{1,-\log n}$
$a_{2,0}$	$a_{2,-1}$	…	$a_{2,-\log n}$
$a_{3,0}$	$a_{3,-1}$	…	$a_{3,-\log n}$
$a_{4,0}$	$a_{4,-1}$	…	$a_{4,-\log n}$
$a_{5,0}$	$a_{5,-1}$	…	$a_{5,-\log n}$
…	…	…	…
$a_{n,0}$	$a_{n,-1}$	…	$a_{n,-\log n}$

$b_{0,\log n}$	…	$b_{0,1}$	$b_{0,0}$			
	$b_{-1,\log n}$	…	$b_{-1,1}$	$b_{-1,0}$		
		…	…	…	…	
			$b_{-\log n,\log n}$	…	$b_{-\log n,1}$	$b_{-\log n,0}$

Computing Sparse Hamming Wgt.

- $\mathrm{Dec}_{E*}(s,c) = \mathrm{LSB}(c)$ XOR $\mathrm{LSB}([\Sigma_i\, s_i \psi_i])$

$a_{1,0}$	$a_{1,-1}$	…	$a_{1,-\log n}$
$a_{2,0}$	$a_{2,-1}$	…	$a_{2,-\log n}$
$a_{3,0}$	$a_{3,-1}$	…	$a_{3,-\log n}$
$a_{4,0}$	$a_{4,-1}$	…	$a_{4,-\log n}$
$a_{5,0}$	$a_{5,-1}$	…	$a_{5,-\log n}$
…	…	…	…
$a_{n,0}$	$a_{n,-1}$	…	$a_{n,-\log n}$

Computing Sparse Hamming Wgt.

- $\text{Dec}_{E*}(s,c) = \text{LSB}(c)$ XOR $\text{LSB}([\Sigma_i\, s_i \psi_i])$

$a_{1,0}$	$a_{1,-1}$	…	$a_{1,-\log n}$
0	0	…	0
0	0	…	0
$a_{4,0}$	$a_{4,-1}$	…	$a_{4,-\log n}$
0	0	…	0
…	…	…	…
$a_{n,0}$	$a_{n,-1}$	…	$a_{n,-\log n}$

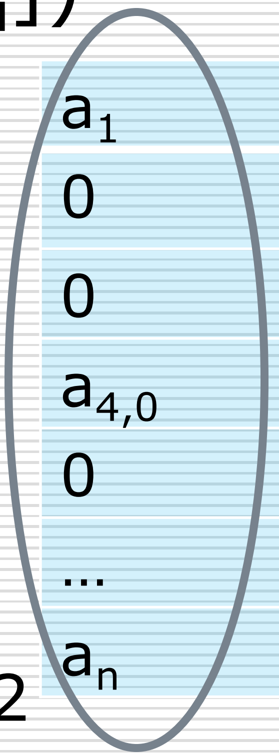
Computing Sparse Hamming Wgt.

- $\text{Dec}_{E*}(s,c) = \text{LSB}(c)$ XOR $\text{LSB}([\Sigma_i\, s_i \psi_i])$
- Binary rep of Hamming wgt of $\mathbf{x} = (x_1, \ldots, x_n)$ in $\{0,1\}^n$ given by:

$e_{2^\wedge[\log n]}(\mathbf{x}) \bmod 2, \ldots, e_2(\mathbf{x}) \bmod 2, e_1(\mathbf{x}) \bmod 2$
where e_k is the elem symm poly of deg k

- Since we know *a priori* that Hamming wgt is |S|, we only need

$e_{2^\wedge[\log |S|]}(\mathbf{x}) \bmod 2, \ldots, e_2(\mathbf{x}) \bmod 2, e_1(\mathbf{x}) \bmod 2$
up to deg < |S|



- Set $|S| < \lambda$, then E* is bootstrappable.