TEL AVIV UNIVERSITY

Information Security – Theory vs. Reality

0368-4474, Winter 2015-2016

Lecture 11: Fully homomorphic encryption

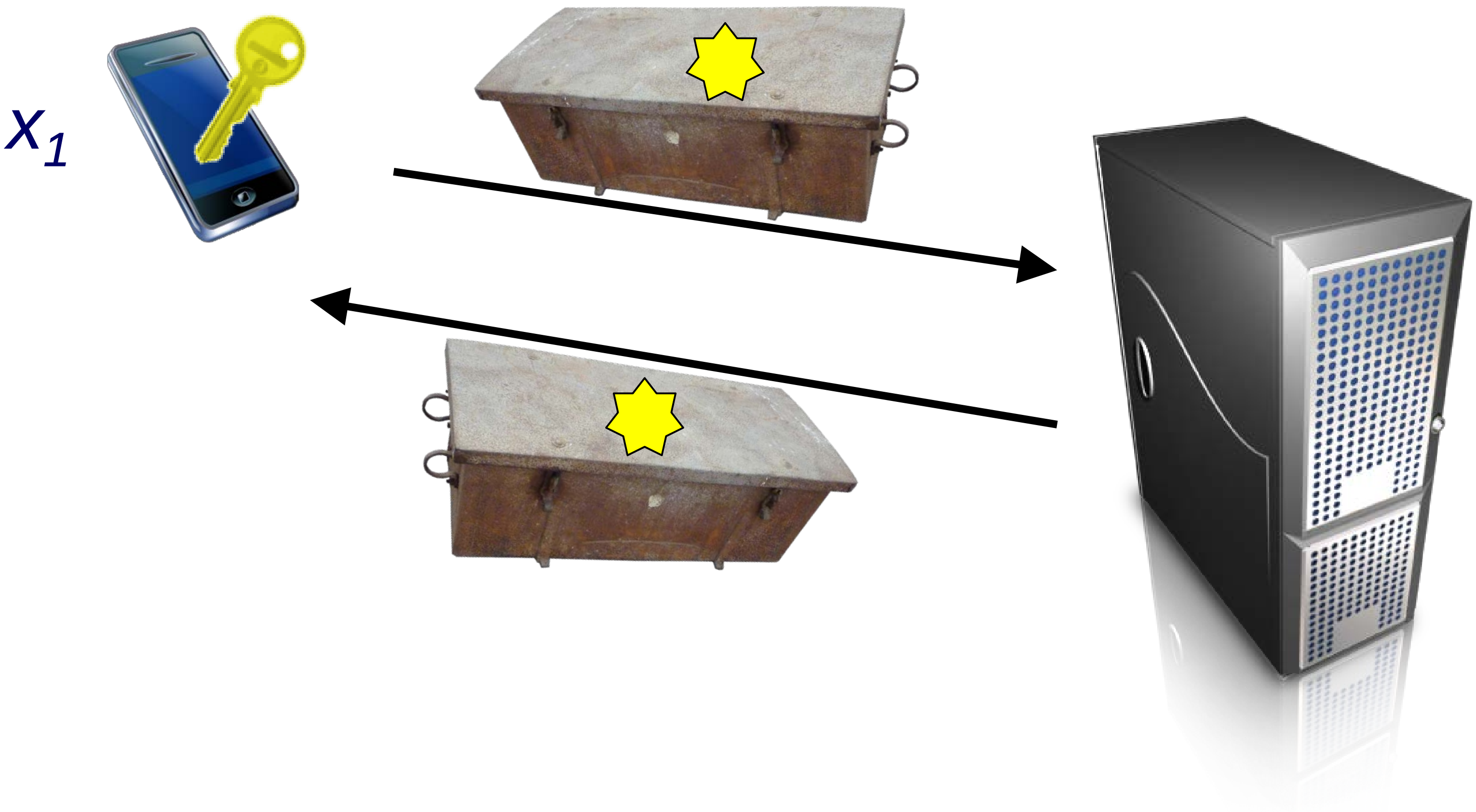
Lecturer:

Eran Tromer

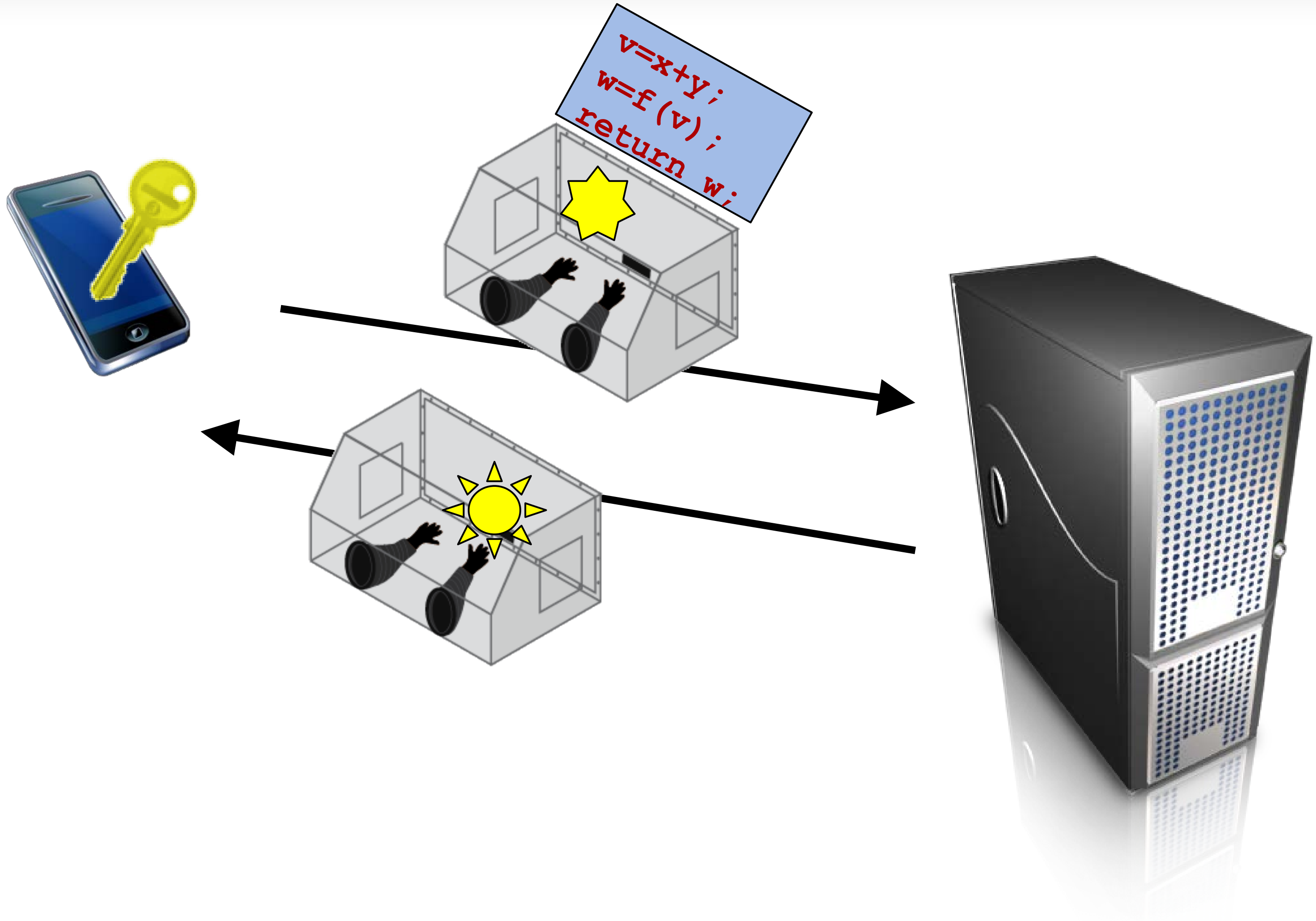
Including presentation material by Vinod Vaikuntanathan, MIT

Fully Homomorphic Encryption

Confidentiality of static data: plain encryption



Confidentiality of data inside computation: Fully Homomorphic Encryption



Fully Homomorphic Encryption

- Goal: delegate computation on data without revealing it
- A <u>confidentiality</u> goal

Example 1: Private search

Delegate processing of data

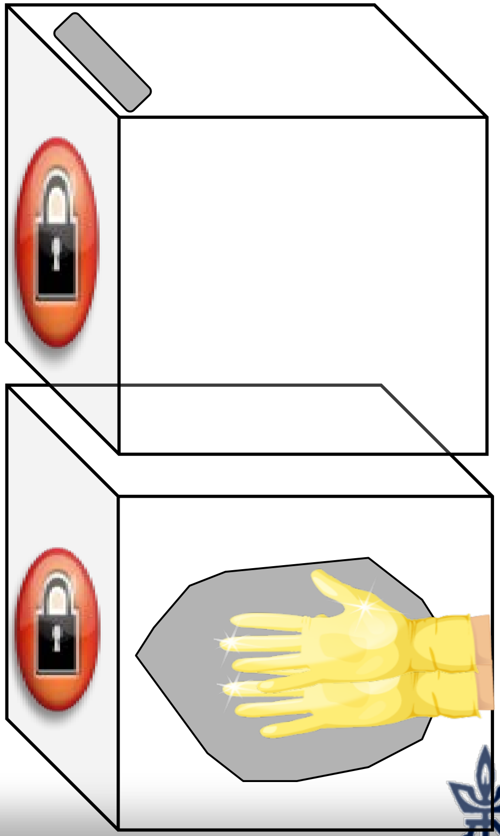
without revealing it

- **You:** Encrypt the query, send to Google

  (Google does not know the key, cannot “see” the query)

- **Google:** Encrypted query → Encrypted results

  (You decrypt and recover the search results)



Example 2: Private Cloud Computing

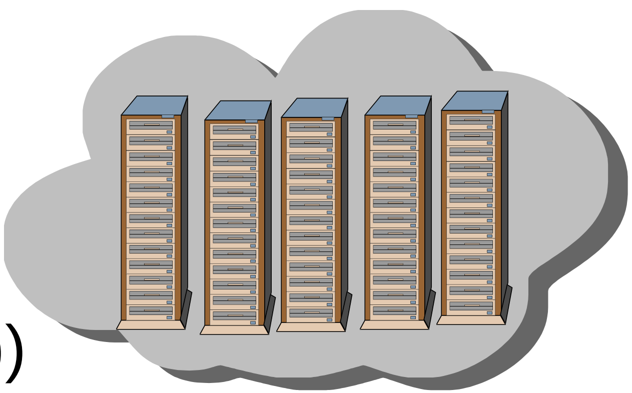
Delegate processing of data

without revealing it

(Input: x)

Encrypt x

(Enc(x), P) → Enc(P(x))



(Program: P)

Fully Homomorphic Encryption

Encrypted x, Program P → Encrypted P(x)

Definition: $\big($KeyGen, Enc, Dec, **Eval**$\big)$

(as in regular public/private-key encryption)

- ❑ **Correctness of Eval:** For every input x, program P
  – If **c = Enc(PK, x)** and **c′ =** $\boxed{\textbf{Eval (PK, c, P),}}$
  then **Dec (SK, c') = P(x).**

- ❑ **Compactness:** Length of c′ independent of size of P

- ❑ **Security:** semantic security / indistinguishability [GM82]

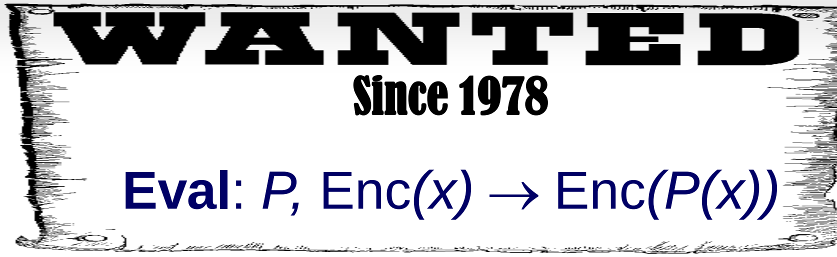
History of Fully Homomorphic Encryption

– **First Defined:**

“Privacy homomorphism”

[Rivest Adleman Dertouzos 78]

motivation: searching encrypted data



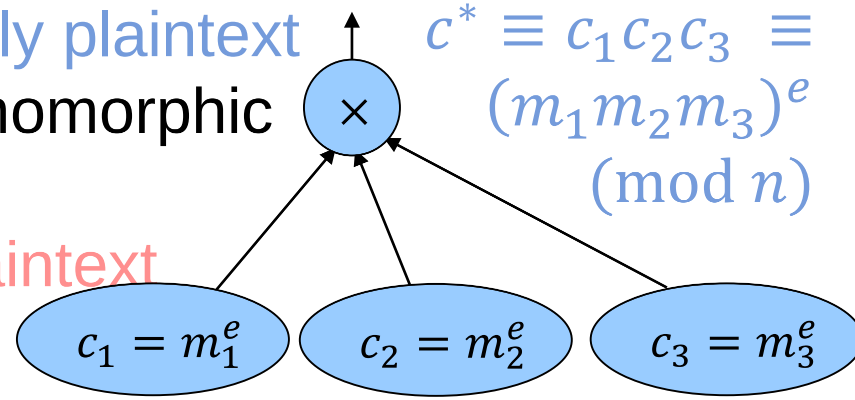
- **Limited homomorphism:**
  - RSA & El Gamal: multiplicatively homomorphic

    multiply ciphertexts ↦ multiply plaintext
  - GM & Paillier: additively homomorphic

    plaintext in exponent

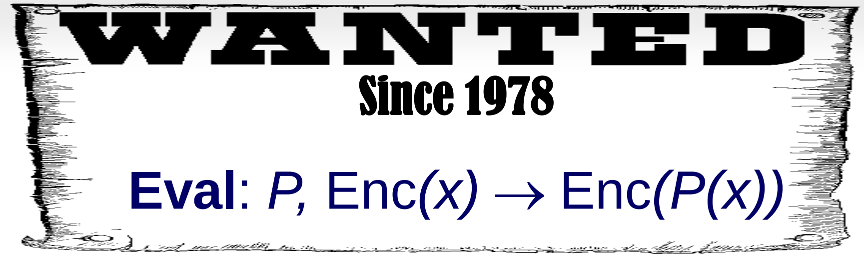
    multiply ciphertext ↦ add plaintext
  - Quadratic formulas

    [BGN 05] [GHV 10]



- **Non-compact homomorphic encryption:**
  - Based on Yao garbled circuits
  - [SYY 99] [MGH 08]: c* grows exp with degree/depth
  - [IP 07] branching programs

Fully Homomorphic Encryption



Big Breakthrough: [Gentry09]

First Construction of Fully Homomorphic Encryption

using algebraic number theory & "ideal lattices"

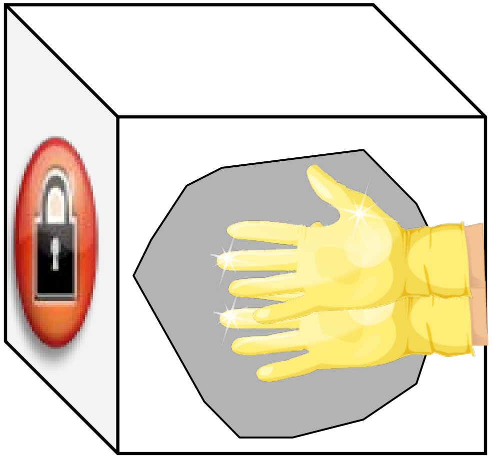
- ►Full-semester course
- ►Today: an alternative construction [DGHV 10]
  - – using just integer addition and multiplication
  - – easier to understand, implement and improve

Constructing

fully-homomoprhic encryption

assuming

hardness of approximate GCD



A Roadmap

1. Secret-key "Somewhat" Homomorphic Encryption

(under the approximate GCD assumption)

(a simple transformation)

2. Public-key "Somewhat" Homomorphic Encryption

(under the approximate GCD assumption)

(borrows from Gentry's techniques)

3. Public-key FULLY Homomorphic Encryption

(under approx GCD + sparse subset sum)

Secret-key Homomorphic Encryption

❶ Secret key: a large **odd** number **p** (sec. param = n)

❷ To Encrypt a bit **b**:

– pick a random "large" multiple of p, say **q·p** (q ~ n^5 bits)

– pick a random "small" even number **2·r** (r ~ n bits)

– Ciphertext **c = q·p+2·r+b**

"noise"

❸ To Decrypt a ciphertext **c**:

– c (mod p) = **2·r+b (mod p)**

– read off the least significant bit

Secret-key Homomorphic Encryption

❹ How to Add and Multiply Encrypted Bits:

– Add/Mult two near-multiples of p gives a near-multiple of p.

– $\mathbf{c_1} = q_1 \cdot p + (2 \cdot r_1 + b_1)$, $\mathbf{c_2} = q_2 \cdot p + (2 \cdot r_2 + b_2)$

– $$\mathbf{c_1+c_2} = \mathbf{p\cdot}(q_1 + q_2) + \underbrace{\mathbf{2\cdot}(r_1+r_2) + (b_1+b_2)}_{\text{LSB} = b_1 \text{ XOR } b_2} \ll \mathbf{p}$$

– $$\mathbf{c_1c_2} = \mathbf{p\cdot}(c_2 \cdot q_1 + c_1 \cdot q_2 - q_1 \cdot q_2) + \underbrace{\mathbf{2\cdot}(r_1r_2+r_1b_2+r_2b_1) + b_1b_2}_{\text{LSB} = b_1 \text{ AND } b_2} \ll \mathbf{p}$$

Problems

❶ **Ciphertext grows** with each operation

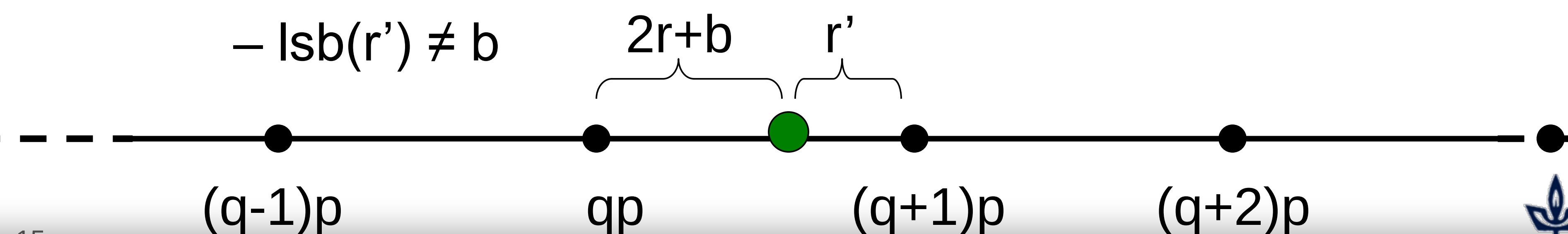
❖ Useless for many applications (cloud computing, searching encrypted e-mail)

❷ **Noise grows** with each operation

– Consider **c = qp+2r+b** $\leftarrow$ Enc(b)

– c (mod p) = r' ≠ 2r+b

– lsb(r') ≠ b



Problems

➊ **Ciphertext grows** with each operation

- ❖ Useless for many applications (cloud computing, searching encrypted e-mail)

➋ **Noise grows** with each operation

- ❖ Can perform “limited” number of hom. operations
- ❖ What we have: **“Somewhat Homomorphic”** Encryption

Public-key Homomorphic Encryption

❶ Secret key: an n^2-bit odd number **p**

Public key: $[q_0p+2r_0, q_1p+2r_1, \ldots, q_tp+2r_t] \stackrel{\Delta}{=} (x_0, x_1, \ldots, x_t)$

– t+1 encryptions of 0

– Wlog, assume that x_0 is the largest of them

❸ To Decrypt a ciphertext **c**:

– c (mod p) = **2·r+b (mod p) = 2·r+b**

– read off the least significant bit

❹ Eval (as before)

Public-key Homomorphic Encryption

❶ Secret key: an n^2-bit odd number **p**

Public key: $[q_0p+2r_0, q_1p+2r_1, \ldots, q_tp+2r_t] \stackrel{\Delta}{=} (x_0, x_1, \ldots, x_t)$

❷ To Encrypt a bit **b**: pick random subset $S \subseteq [1\ldots t]$

$$c = \sum_{i \in S} x_i + 2r + b \pmod{x_0}$$

❸ To Decrypt a ciphertext **c**:

– c (mod p) = **2·r+b (mod p) = 2·r+b**

– read off the least significant bit

❹ Eval (as before)

Public-key Homomorphic Encryption

❶ Secret key: an n^2-bit odd number **p**

Public key: $[q_0p+2r_0, q_1p+2r_1, \ldots, q_tp+2r_t] \overset{\Delta}{=} (x_0, x_1, \ldots, x_t)$

❷ To Encrypt a bit **b**: pick random subset $S \subseteq [1\ldots t]$

$$c = \sum_{i\in S} x_i + 2r + b \pmod{x_0}$$

$$c = p\Big[\sum_{i\in S} q_i\Big] + 2\Big[r+\sum_{i\in S} r_i\Big] + b \pmod{x_0}$$

$- kx_0$ (for a small k)

$$= p\Big[\sum_{i\in S} q_i - kq_0\Big] + 2\Big[r+\sum_{i\in S} r_i - kr_0\Big] + b$$

(mult. of p) + ("small" even noise) + b

Ciphertext Size Reduction

❶ Secret key: an n^2-bit odd number **p**

Public key: $[\mathbf{q_0p+2r_0, q_1p+2r_1, \ldots, q_tp+2r_t}] \stackrel{\Delta}{=} (x_0, x_1, \ldots, x_t)$

❷ To Encrypt a bit **b**: pick random subset $S \subseteq [1\ldots t]$

$$c = \sum_{i \in S} x_i + 2r + b \pmod{x_0}$$

❸ To Decrypt a ciphertext **c**:

– c (mod p) = **2·r+b (mod p) = 2·r+b**

– read off the least significant bit

❹ Eval: **Reduce mod x_0 after each operation**

(*) additional tricks for mult

Ciphertext Size Reduction

❶ Secret key: an n^2-bit odd number **p**

Public key: $[\mathbf{q_0p+2r_0,q_1p+2r_1,\ldots,q_tp+2r_t}] \stackrel{\Delta}{=} (x_0,x_1,\ldots,x_t)$

– Resulting ciphertext $< x_0$

– Underlying bit is the same (since x_0 has even noise)

– Noise does not increase by much(*)

– read o… least significant bit

❹ Eval: **Reduce mod x_0 after each operation**

(*) additional tricks for mult

A Roadmap

☑ **Secret-key “Somewhat”** Homomorphic Encryption

↓

☑ **Public-key “Somewhat”** Homomorphic Encryption

↓

3. Public-key FULLY Homomorphic Encryption

How “Somewhat” Homomorphic is this?

Can evaluate (multi-variate) polynomials with m terms, and maximum degree d if d << n.

$$m \cdot 2^{nd} < p/2 = 2^{n^2}/2 \quad \text{or} \quad d \sim n$$

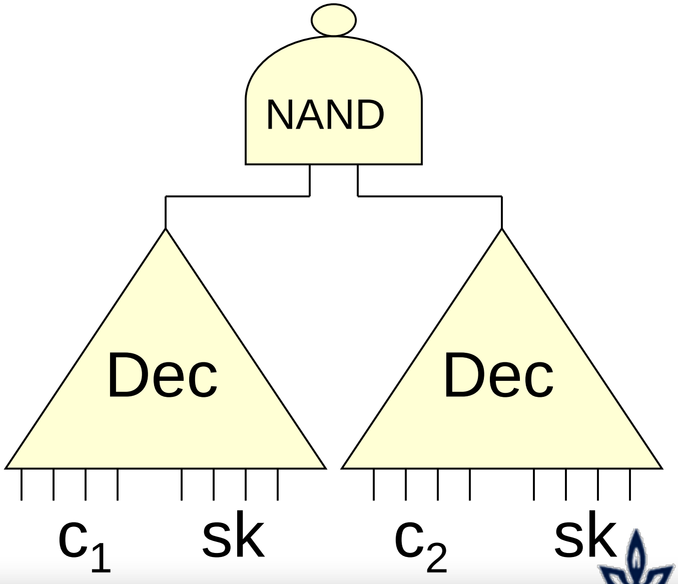
$$f(x_1, \ldots, x_t) = \underbrace{x_1 \cdot x_2 \cdot x_d + \ldots + x_2 \cdot x_5 \cdot x_{d-2}}_{\text{m terms}}$$

Say, noise in Enc(x_i) < 2^n

Final Noise ~ $(2^n)^d + \ldots + (2^n)^d = m \bullet (2^n)^d$

Bootstrapping: from "somewhat HE" to "fully HE"

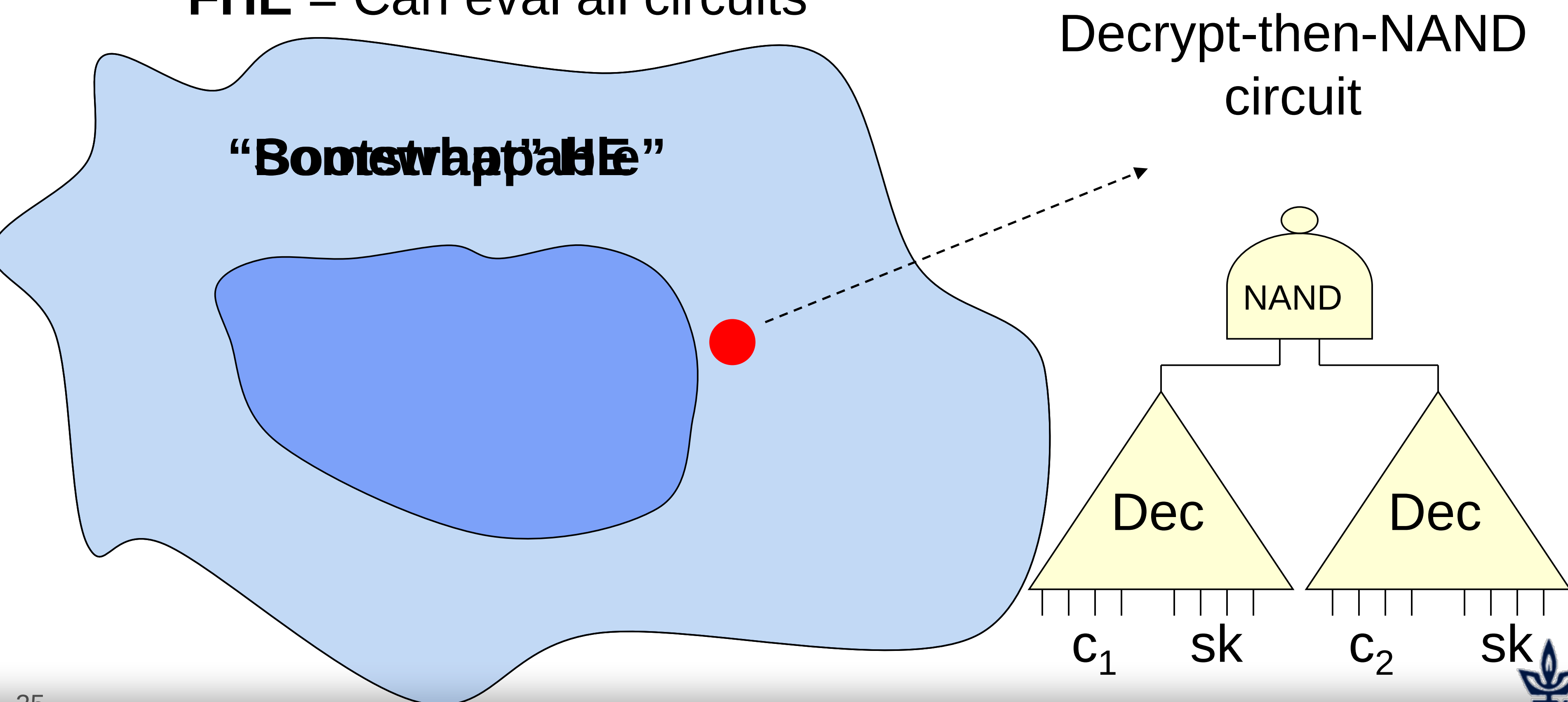
Decrypt-then-NAND circuit



Bootstrapping: from “somewhat HE” to “fully HE”

Theorem [Gentry’09]: Convert “bootstrappable” → FHE.

FHE = Can eval all circuits



Is our Scheme “Bootstrappable”?

What functions can the scheme evaluate?

(polynomials of degree < n)

$$\overset{(?)}{\supseteq}$$

Complexity of the Decrypt-then-NAND circuit

(degree ~ $n^{1.73}$ polynomial)

Can be made bootstrappable by “preprocessing” some of the decryption outside the decryption circuit (Following [Gentry 09])

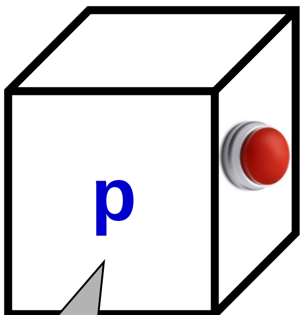
Caveat: Assume Hardness of “Sparse Subset Sum”

Security

(of the "somewhat" homomorphic scheme)

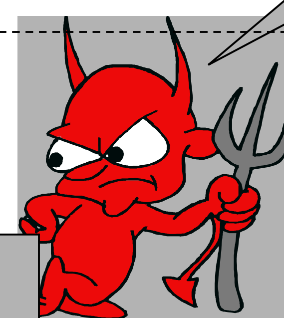
The Approximate GCD Assumption

Parameters of the Problem: Three numbers P,Q and R



$(q_1p+r_1,\ldots, q_tp+r_t)$

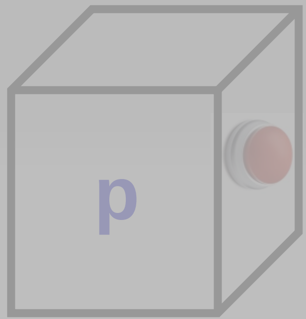
$q_1 \leftarrow [0\ldots Q]$
$r_1 \leftarrow [-R\ldots R]$



p?

Assumption: no PPT adversary can guess the number p

odd p $\leftarrow [0\ldots P]$



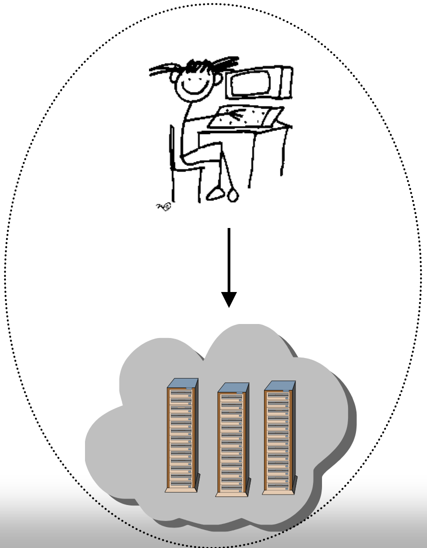
$$(q_1p+r_1, \ldots, q_tp+r_t) \longrightarrow$$

Assumption: no PPT adversary can **guess the number p**

(proof of security)

=

Semantic Security [GM'82]: no PPT adversary can **guess the bit b**



$$\mathrm{PK} = (q_0p+2r_0, \{q_ip+2r_i\})$$

$$\mathrm{Enc}(b) = (qp+2r+b)$$

Progress in FHE

► "Galactic" → "Efficient"

Asymptotically: nearly *linear-time** algorithms

Practically:

- Implementations, including bootstrapping and "packing"
  `github.com/shaih/HElib` `github.com/lducas/FHEW`
- a few milliseconds for Enc, Dec [LNV'11,Gentry Halevi Smart '11]
- a few minutes (amortized) for evaluating an AES block [GHS '12]
- single bootstrapping < 1 second [Ducas Micciancio '14]
- bootstrapping takes 5.5 minutes and allows a "payload" of depth 9 computation on $GF(2^{16})^{1024}$ vectors

► Strange assumptions → Mild assumptions

– **Best Known [BGV11]**: (leveled) FHE from worst-case hardness of $n^{O(\log n)}$-approx short vectors on lattices

*linear-time in the security parameter

Multi-key FHE

sk_1, pk_1

x_1

$c_1 = \mathrm{Enc}(pk_1, x_1)$

Function

f

sk_2, pk_2

$c_2 = \mathrm{Enc}(pk_2, x_2)$

x_2

Multi-key FHE

sk_1, pk_1

x_1

Dec

$y = \mathrm{Eval}(f, c_1, c_2)$

Function

f

sk_2, pk_2

x_2

Correctness:

$\mathrm{Dec}(sk_1, sk_2\ y) = f(x_1, x_2)$

Fully Homomorphic Encryption

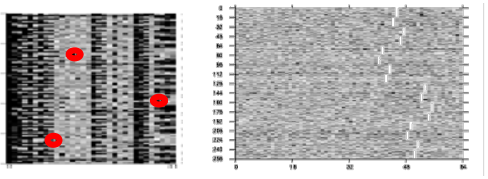
Whiteboard discussion:

- Properties
- Performance
- Contrast with obfuscation
- Usefulness

Protecting memory using Oblivious RAM

Motivation: memory/storage attacks

- Physical attacks
  - Memory/storage is on a physical separate device (DRAM chip, SD card, hard disk, …)
  - Communication between CPU and device is easy to tap
  - Memory/storage device may be under attack or stolen
    - Aggravated by data remanence problem
- Software side channels
  - Leakage of accesses memory addresses across software confinement boundaries (via data cache, instruction cache, page table, …)



- Network attacks
  - External storage (file server, Network Attached Storage, cloud service, …)
  - Remote server/appliance/provider may be compromised

Protecting against memory attack

- Computation model:
  - Random access memory
  - Small processor (logarithmic in memory size)
- Leakage/tampering model:
  - All memory accesses (both data and address) leak or are corrupted arbitrary (relaxation: by polytime adversary)
  - Processor assumed secure
- Goal: a compiler that converts any program into one that resists memory attacks
  - Functionality: input/output precisely preserved
  - Security: privacy against leakage [MR04] with suitable (restricted) circuit classes and admissible functions

Protecting memory content from leakage

- INEFFICIENT Encrypt the whole memory as a single message
- Encrypt every block separately
  - INSECURE – encrypt block data using AES
  - INSECURE – encrypt block number + data using AES
  - OK – encrypt block using semantically-secure (probabilistic encryption
- Keep the decryption key inside the secure processor

Protecting memory content from corruption

INSECURE

- Sign every block, keep the signing key inside the secure processor

INEFFICIENT

- Hash every block, keep digests inside the secure processor

OK

- Using Merkle trees
  - Maintain a Merkle hash tree over the memory
  - Merkle nodes stored in the unstrusted memory
  - Merkle root stored in secure processor
  - At every read, processor verifies Merkle path
  - At every write, update Merkle path

Oblivious RAM

[Goldreich Ostrovsky '96]…

Protecting against memory access leakage

Compile any program P and memory size n into a new program P', such that: (this definition follows [Chung Pass 2013])

For any P with memory size n, and input x:

- Correctness: $P'(x)=P(x)$ (up to some small failure probability)
- Efficiency:
  - P' on x runs $c(n)$ times longer than P on x, where $c(\cdot)$ is the computational overhead
  - P' uses memory of size $m(n) \cdot n$,where $m(\cdot)$ is the memory overhead
  - Extra registers in secure processor
- Obliviousness (security):
  For any P_1, P_2 with memory size n, and inputs x_1, x_2,
  such that the number of memory accesses done by P_1 on x_1 is the same as P_2 on x_2,
  the $\texttt{(address,val)}$ memory transcript of P_1' on x_1 is statistically close to that of P_2' on x_2.

“Simple ORAM” construction [Chung Pass ‘13]

Given a progam P and memory size n, output P':

P' proceeds like P, except:

- $read(r) \mapsto Oread(r)$
- write$(r, val) \mapsto \text{Owrite}(r, val)$
- Memory divided into blocks of size α.
- External memory holds a complete binary tree of depth $d = \log\left(\frac{n}{\alpha}\right)$
- Pos maps each memory blocks b to a leaf pos.

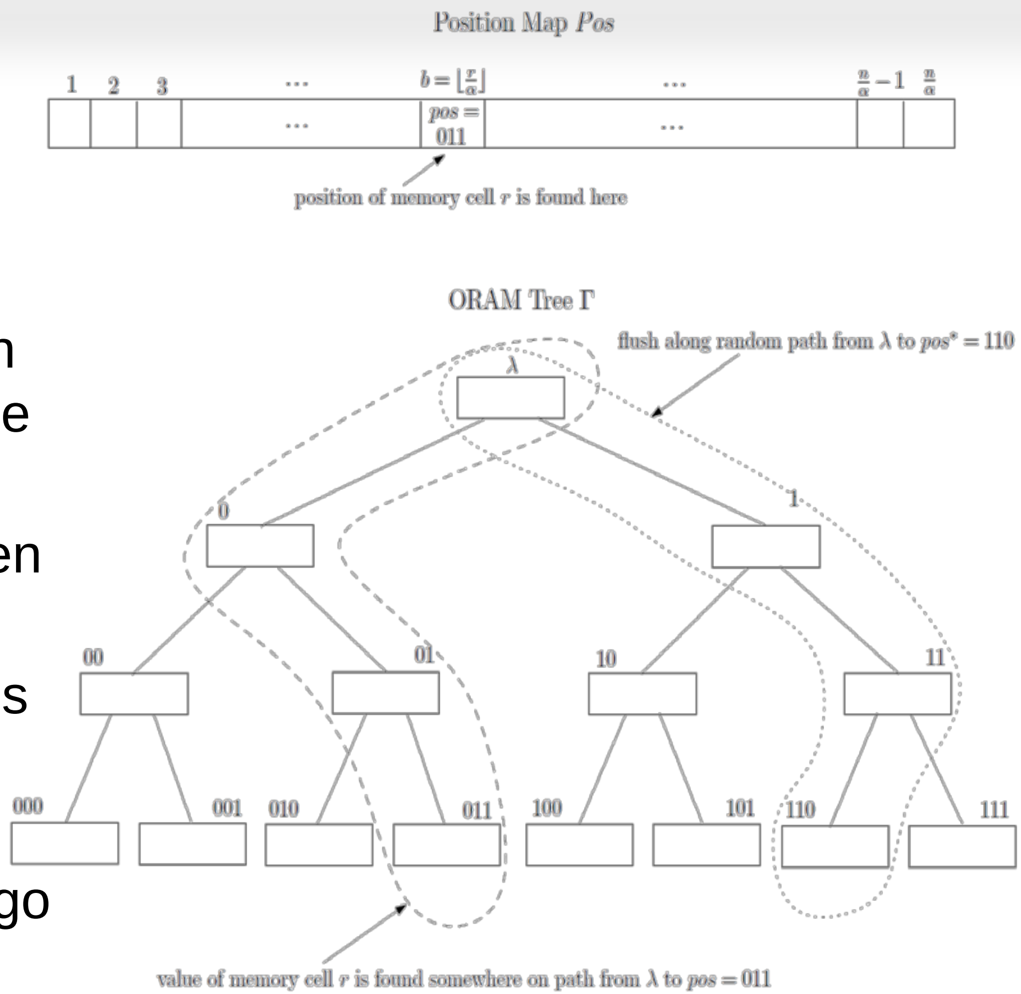
Invariant: the content of block b is stored somewhere along path to pos.

- Each node contains a bucket: at most K tuples $(b, pos, data)$ where b is a block index and v is the block’s data.
  ($K = \text{polylog}(n)$)
- All registers and memory are initialized to $\perp$.

Simple ORAM" construction: reading

$Oread(r)$:

- b is r's block
- $pos \leftarrow Pos[b]$
- Fetch r's block by traversing path from root to pos looking for a tuple (b, pos, v). (if not found, output ⊥)
- Update map $Pos[b] \leftarrow pos'$ chosen at random.
- Put back (b, pos', v) into the root's bucket. (if overflow, output ⊥)
- Flush tuples down a path to a random pos^*, as far as they can go while consistent with invariant. (if overflow, output ⊥)



Obliviousness: each $Oread$ operation traverses the tree along two paths that are chosen at random and independently of the history so far (doing a single read and single write at every node).

Simple “ORAM” construction: further details

- Writing:
  $Owrite(r, val)$:
  same as $Oread(r)$ except we put back the updated (b, pos', v').
- Storing the position map
  - Problem: the position map is too large.
  - Solution (“full-fledged construction”):
    recursively stored the position map in a smaller oblivious RAM (same K but smaller memory).
- <u>Correctness</u>:
  Obvious as long as overflows don’t happen. Easy probabilistic analysis shows that overflows happen with negligible probability (for suitable parameters α and K). See [Chung Pass ’13 – “A Simple ORAM”] for details.
- Overheads: all polylogarithmic. $O(1)$ registers suffice.

Other ORAMs

- Lower bound: $\log(n)$ computational overhead.
- There are several variants of such “path ORAM”, and others.
- Implemented in software, FPGA hardware.