*Secure computing*, seventh lecture

# Secure computing: other directions and conclusions

Xavier Leroy

2025-12-18

Collège de France, chair of Software sciences
`xavier.leroy@college-de-france.fr`

# Secure memory, 1:
# Oblivious RAM

# Secure computing with random memory accesses

The approaches seen so far (homomorphic encryption, multi-party computation) represent computations by combinatorial circuits (Boolean or arithmetic).

**Theorem**

*Any Turing machine of size $n$ that always terminate in time* $\mathrm{poly}(n)$ *can be implemented by a circuit of size* $\mathrm{poly}(n)$.

However, many computations are easier to express and faster to execute if we can use a random access memory:

- ROM (read-only): tables of constants, data base
- RAM (read-write): to store intermediate results.

# Secure computing with random memory accesses

We can perform random access to an address $x \in \{0, \ldots, n-1\}$ using circuits:

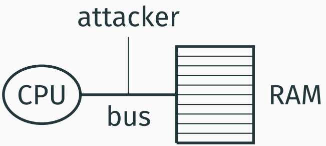
- Boolean circuits: $n$ multiplexers arranged as a binary tree.
- Arithmetic circuit: interpolation polynomial

$$P(x) = \sum_{i=0}^{n-1} a_i \lambda_i(x) \quad \text{where} \quad \lambda_i(x) \prod_{j=0, j \neq i}^{n-1} \frac{X - i}{j - i}$$

Any access “touches” the $n$ memory cells.

**Folklore:** a private access to a database must access all entries of the database; otherwise, it reveals information on the access.

# Oblivious RAM (ORAM)



Example of use: a secure enclave that uses a standard RAM to store data.

RAM accesses (addresses and data) are visible from the attacker, yet must not reveal anything about the computation performed in the enclave (passive security).

- Data must be encrypted.
- Addresses must be garbled.
- Two accesses to the same logical address must use different garbled addresses.

# Square-root ORAM

(O. Goldreich, R. Ostrovsky, *Software protection and simulation on Oblivious RAM*, JACM, 1996.)

An ORAM of size $N$ comprises:

- A cache of size $n$ (ideally, $n \approx \sqrt{N}$, hence the name)
- A RAM of size $N + n$ ($N$ actual data + $n$ decoys)
- A random permutation of addresses:
  $\pi : \{0, \ldots, N + n - 1\} \to \{0, \ldots, N + n - 1\}$
- The cell $\pi(i)$ of the RAM contains an encryption of $(i, v_i)$ for $i \in \{0, \ldots, N - 1\}$.
- The cells $\pi(N), \ldots, \pi(N + n - 1)$ are decoys.

# Reading from the ORAM

If the adress $i$ is not in the cache:

- Read $e$ from address $\pi(i)$ in the RAM.
- Decrypt $(i, x) \leftarrow \mathcal{D}(e)$.
- Add $i \mapsto x$ to the cache.
- Return $x$.

If the address $i$ is in the cache, with value $x$:

- Read from address $\pi(N + k)$ in the RAM.
- Return $x$.

In both cases:

- Increment $k$.
- If $k \geq n$, draw and apply a new random permutation $\pi'$.

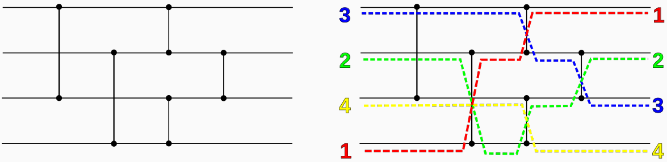
# Applying a permutation obliviously

The RAM contains encrypted pairs $\mathcal{E}(i, v_i)$. We want to order them according to the permutation $\pi$:

$$RAM[\pi(i)] = \mathcal{E}(i, v_i)$$

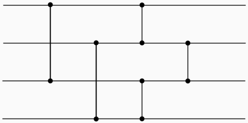
This amounts to sorting the RAM by increasing values of $\pi(i)$.

We use a sorting network, so that RAM accesses follow a pattern that is independent from the data to be sorted.



(Oskar Sigvardsson, CC BY)

# Applying a permutation obliviously



For each pair of points $(u, v), u < v$ in the sorting network:

- Read and decrypt

  $(i, v_i) \leftarrow \mathcal{D}(RAM[u]) \qquad (j, v_j) \leftarrow \mathcal{D}(RAM[v])$

- If $\pi(i) < \pi(j)$: re-encrypt and re-write in the same order

  $RAM[u] \leftarrow \mathcal{E}(i, v_i) \qquad RAM[v] \leftarrow \mathcal{E}(j, v_j)$

- If $\pi(i) > \pi(j)$: swap, re-encrypt and re-write

  $RAM[u] \leftarrow \mathcal{E}(j, v_j) \qquad RAM[v] \leftarrow \mathcal{E}(i, v_i)$

Same RAM access patterns in all cases.

# Implicit representation of a random permutation

The client doesn't have enough memory to store $\pi$ *in extenso*.

Instead, the client generates $\pi$ from a pseudo-random function $h$:

$$\pi(i) = \text{number of } j \in \{0, \ldots, N + n - 1\} \text{ such that } h(j) < h(i)$$

The RAM contains pairs $\mathcal{E}(i, v_i)$ sorted by increasing $h(i)$.

To access $RAM[\pi(i)]$ we use binary search:

```
$(j, v_j) \leftarrow \mathcal{D}(RAM[p])$
if $j = i$, found
if $h(i) < h(j)$, move $p$ to the left
if $h(i) > h(j)$, move $p$ to the right
```

The RAM access pattern reveals nothing more than the value of $\pi(i)$.

# Summary: an oblivious ROM

Initially: the data $v_0, \ldots, v_{N-1}$ are stored in $RAM[0], \ldots, RAM[N-1]$.

Preparation: $RAM[i] \leftarrow \mathcal{E}(i, RAM[i])$ for $i = 0, \ldots, N+n-1$.

Repeat until the client stops:

- Draw a pseudo-random function $h$ (typically, a block cipher such as AES with a random key).
- Sort the RAM by increasing $h(i)$ using a sorting network.
- Read $n$ values.

Amortized time for an access: $\mathcal{O}(\log N + N \log^2 N/n)$
i.e. $\mathcal{O}(\sqrt{N} \log^2 N)$ if $n \approx \sqrt{N}$.

# Extension: an oblivious RAM

All accesses are Read-Modify-Write:

$$x \leftarrow ORAM[i]; \quad y \leftarrow f(x); \quad ORAM[i] \leftarrow y$$

to unify reads ($f(x) = x$) and writes ($f(x) = v$).

Every access reads from the RAM and keeps the new value $y$ in the cache.

After $n$ accesses, before re-permuting the RAM, we flush the cache by performing writes in the RAM at the same addresses and in the same order as the previous reads.

# Improving performance

Goldreich and Ostrovsky propose to use a hierarchy of caches of sizes $2, 4, 8, \ldots, 2^p$, stored in RAM, re-permuted with a frequency inversely proportional to their sizes.
$\Rightarrow$ Access in amortized time $\mathcal{O}(\log^3 N)$. Total space $\mathcal{O}(N \log N)$.

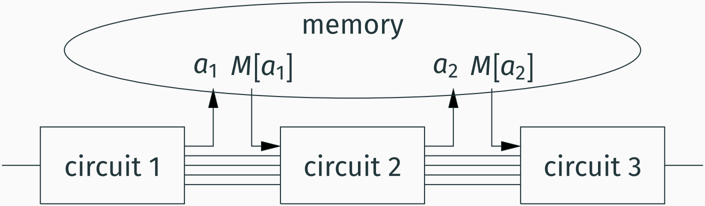
Other approach: use binary trees instead of hash tables.

Best implementation known: amortized time $\mathcal{O}(\log N)$, about 26 RAM accesses per ORAM access for $N = 1\,\mathrm{To}$.

(E. Stefanov, E. Shi, D. Song, *Towards practical Oblivious RAM*, NDSS 2012.)

# Secure memory, 2: homomorphic encrypted table lookup

# Circuits that can access a read-only table



An intermediate model of computation between circuits and the RAM model.

Appropriate for the computation of simple statistics on large data.

Can we perform this computation homomorphically, using encrypted addresses and encrypted data?

# Private Information Retrieval (PIR)

Can we execute a request on a database *DB* of size *n* without the DB server learning anything about the request and its result?

**Folklore:** all database entries must be accessed. Otherwise, the server would learn that the request doesn't depend on the non-accessed entries.

$\Rightarrow$ Each request takes time $\mathcal{O}(n)$ or more.

**The Doubly-Efficient PIR approach:** (Lin, Mook, Wichs, 2023)

- Preprocess *DB* into a slightly larger base $DB'$, in time superlinear in *n*.
- Execute each private request on $DB'$, in time sublinear in *n*.
- The access patterns in $DB'$ reveal no access pattern in *DB*.

# Preprocessing the evaluation of a polynomial

Let $P \in \mathbb{Z}_q[X]$ be a polynomial of degree $d$.

We want to compute $P(x)$ for multiple values of $x$.

**Without preprocessing:**

- Evaluate $P$ directly for each $x$, in time $\mathcal{O}(d)$.

**Naive tabulation:**

- Tabulate the $q$ values of $P$ : $\quad t[i] = P(i)$ for $i = 0, \ldots, q-1$ (time and space $\mathcal{O}(q)$)
- For each $x$, return $t[x]$ in time $\mathcal{O}(1)$.

# Preprocessing the evaluation of a polynomial

## "Chinese" tabulation:

Let's view $P$ as a polynomial $\tilde{P}$ with coefficients in $\mathbb{Z}$.
$P(x) = \tilde{P}(x) \bmod q$ for all $x \in \{0, \ldots, q-1\}$.

We have $0 \leq \tilde{P}(x) < M = d(q-1)^d$ for all $x \in \{0, \ldots, q-1\}$.

Let $p_1, \ldots, p_n$ be small prime numbers such that

$$M \leq p_1 \cdots p_n$$

(The first $16 \log M$ primes always work.)

For each $p_i$, we tabulate $\tilde{P} \bmod p_i$:

$$t_i[j] = \tilde{P}(j) \bmod p_i \quad \text{for } j = 0, \ldots, p_i - 1$$

# Preprocessing the evaluation of a polynomial

$$t_i[\,j\,] = \tilde{P}(j) \bmod p_i \quad \text{pour } j = 0, \ldots, p_i - 1$$

**“Chinese” evaluation:**

To evaluate $P(x) = \tilde{P}(x) \bmod q$, we look up the $n$ tables $t_i$:

$$y_i = t_i[\,x \bmod p_i\,] = \tilde{P}(x \bmod p_i) \bmod p_i = \tilde{P}(x) \bmod p_i$$

Using the Chinese remainder theorem, we compute

$$y = \tilde{P}(x) \bmod p_1 \cdots p_n = \tilde{P}(x) \quad \text{since } \tilde{P}(x) < M \leq p_1 \cdots p_n$$

Hence, $P(x) = y \bmod q$.

# Preprocessing the evaluation of a polynomial

$$t_i[j] = \tilde{P}(j) \bmod p_i \quad \text{pour } j = 0, \ldots, p_i - 1$$

**“Chinese” evaluation:**

To evaluate $P(x) = \tilde{P}(x) \bmod q$, we look up the $n$ tables $t_i$:

$$y_i = t_i[x \bmod p_i] = \tilde{P}(x \bmod p_i) \bmod p_i = \tilde{P}(x) \bmod p_i$$

Using the Chinese remainder theorem, we compute

$$y = \tilde{P}(x) \bmod p_1 \cdots p_n = \tilde{P}(x) \quad \text{since } \tilde{P}(x) < M \leq p_1 \cdots p_n$$

Hence, $P(x) = y \bmod q$.

**Space optimisation:** if some tables $t_i$ are too large, we can recurse to represent $P \bmod t_i$ by several smaller tables.

# Example

Polynomial of degree $d = 20$ modulo $2^{64}$. Two-level evaluation:

- 37 tables giving the values of $P$ modulo 2, 3, 5, . . . , 157.
- Using the Chinese remainder theorem, we obtain the exact values of $P(i)$ for all $i \leq 1028$.
- Thus, we know $P \bmod p$ for $p = 163, 167, \ldots, 937$.
- Using the Chinese remainder theorem again, we obtain $P \bmod 2^{64}$.

The calculation remains more costly than direct evaluation of $P$ (37 memory reads instead of 21).

But this is no longer the case for a multivariate polynomial!

Example: polynomial with 4 variables of degree 20 modulo $2^{64}$: 126 tables *vs* $21^4 = 194481$ coefficients.

# Preprocessing the evaluation of a polynomial

(K. S. Kedlaya, C. Umans: *Fast Polynomial Factorization and Modular Composition*, CCDP 2008, SIAM J. Comput. 2011)

This approach extends to polnomials with $m$ variables of degree $d$ in each variable, and to rings $R$ other than $\mathbb{Z}_q$.

Precomputation in time and space (with 2 levels of tables)

$$d^m \cdot \mathrm{poly}(m, d, \log|R|) \cdot \mathcal{O}(m \cdot (\log m + \log d + \log\log|R|))^m$$

Evaluation in time $\mathrm{poly}(d, m, \log|R|)$ .

Precomputation is quasilinear in $N = (d+1)^m$ (the number of coefficients of $P$), and evaluation is polylogarithmic in $N$.

# Application to Private Information Retrieval

(W-K. Lin, E. Mook, D. Wichs, *Doubly Efficient Private Information Retrieval and Fully Homomorphic RAM Computation from Ring LWE*, STOC 2023.)

Assume that the database DB is an $m$-dimension array containing $d^m$ entries.

We can apply the Kedlaya-Umans algorithm to the interpolation polynomial $P$ corresponding to DB:

$$P(i_1, i_2, \dots, i_n) = DB[\,i_1\,][\,i_2\,]\dots[\,i_n\,]$$

To make requests private, Lin, Mook and Wichs propose to apply Kedlaya-Umans to an encrypted version $P'$ of $P$:

$$P'(\mathcal{E}(i_1), \dots, \mathcal{E}(i_n)) = \mathcal{E}(DB[\,i_1\,][\,i_2\,]\dots[\,i_n\,])$$

using an algebraic somewhat homomorphic encryption.

# Algebraic somewhat homomorphic encryption (ASHE)

A cipher $\mathcal{E} : \mathbb{Z}_q \to R$, where $R$ is a ring that can be handled by Kedlaya-Umans. Homomorphic addition is ring addition, and likewise for multiplication.

$$\mathcal{E}(x + y) = \mathcal{E}(x) + \mathcal{E}(y)$$
$$\mathcal{E}(x \cdot y) = \mathcal{E}(x) \cdot \mathcal{E}(y)$$

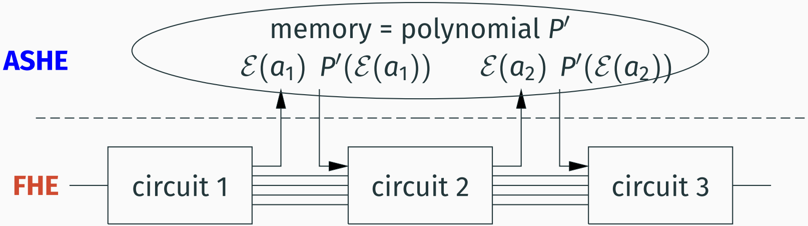
Somewhat homomorphic: all we need is to evaluate monomials $c \cdot X_1^{e_1} \cdots X_m^{e_m}$ of degree $\leq d$.

Example: the cipher by van Dijk *et al* 2010 (lecture # 2):

$$\mathcal{E}_p(b) = pn + qr + b \qquad \begin{array}{l} n \text{ nonnegative random integer} \gg p \\ r \text{ random integer, } |r| < p/4q \end{array}$$

Better ASHE ciphers can be constructed from the RLWE problem.

# Application to homomorphic computation



The circuits are evaluated by fully homomorphic encryption (FHE) as in lecture #3.

Conversions FHE → ASHE for the memory addresses and ASHE → FHE on the values read.

Extension to a RAM (read-write access): see Lin, Mook, Wichs.

# Indistinguisable Obfuscation

# Software obfuscation

Make executable or source code impossible to understand: resist disassembly, code review, static analysis, etc.

Uses:

- Prevent reverse engineering
  (proprietary algorithms, deactivated features, ...)
- Prevent circumventing software protections
  (anti-copy protections, digital rights management, ...)
- For fun!
  (Easter eggs, the IOCCC competition, ...)

# Examples of obfuscated codes

```c
#include <stdio.h>
int l;int main(int o,char **O,
int I){char c,*D=O[1];if(o>0){
for(l=0;D[l               ];D[l
++]-=10){D    [l++]-=120;D[l]-=
110;while    (!main(0,O,l))D[l]
+=   20;    putchar((D[l]+1032)
/20   )    ;}putchar(10);}else{
c=o+      (D[I]+82)%10-(I>l/2)*
(D[I-l+I]+72)/10-9;D[I]+=I<0?0
:!(o=main(c/10,O,I-1))*((c+999
)%10-(D[I]+92)%10);}return o;}
```

*(Raymond Cheong, IOCCC 2001)*

```javascript
if(!qa)return a;try{var b=ka();a=qa.createPolicy(
"goog#html",{createHTML:b,createScript:b,
createScriptURL:b})}catch(c){}return a},
ta=function(){sa===void 0&&(sa=ra());return sa},
va=function(a){var b=ta();a=b?b.createScriptURL(a)
:a;return new ua(a)},ya=function(a){
if(a instanceof ua)return a.g;throw Error("p");},
za=function(a,b,c,d,e,f,g){var h="";a&&(h+=a+":");
c&&(h+="//",b&&(h+=b+"@"),h+=c,d&&(h+=":"+d));
e&&(h+=e);f&&(h+="?"+f);g&&(h+="#"+g);return h},
Aa=function(a,b){if(a){a=a.split("&");for(var c=
0;c<a.length;c++){var d=a[c].indexOf("="),
e=null;if(d>=0){var f=a[c].substring(0,d);
e=a[c].substring(d+1)}else f=a[c];b(f,e?
decodeURIComponent(e.replace(/\+/g," ")):"")}}},
Ba=function(a,b,c){if(Array.isArray(b))
```

*(gmail.com)*

# Some obfuscation techniques

Renaming functions and variables.

Insertion of redundant computations:

```
    return x + 3;
→  if ((n + 1) * n & 1) return n << 2; else return n + 3;
```

Encoding the control flow as tables or automata.

```
    while (1) switch(pc) {
        case 1: ...; pc = 12; break;
        ...
```

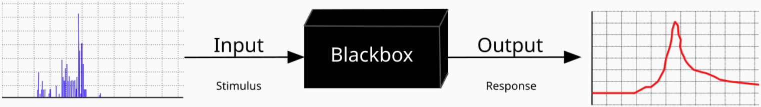
Changing the representation of data.

Masking or encryption of the code. Self-modifying code.

(See Goldberg and Nagra, *Surreptitious Software*, 2010.)

# Cryptographic-quality obfuscation

Transform a source code containing secrets into a “black box” executable program.



The only information that can be extracted from the black-box executable are observations of its executions.

# Standard compilation is not obfuscation

Consider the following C program compiled to `prog.exe`:

```c
#include <stdio.h>
int main(int argc, char ** argv)
{
  if (argc < 0)
    printf("The secret is: AQFADHFJ\n");
  else
    printf("Hello, world!\n");
}
```

Running `prog.exe` will never print the secret.
(POSIX operating systems guarantee that `argc >= 0`).

Yet. the secret is contained in the file `prog.exe` and can be extracted by `strings prog.exe | grep secret`

# Cryptographic-quality obfuscation

One of the first directions proposed to implement public-key encryption!

> *A more practical approach to finding a pair of easily computed inverse algorithms E and D such that D is hard to infer from E makes use of the difficulty of analyzing programs in low level languages. [. . . ] Essentially what is required is a one-way compiler: one which takes an easily understood program written in a high-level language and translates it into an incomprehensible program in some machine language.*

(W. Diffie, M. Hellman, *New Directions in Cryptography*, 1976.)

# Virtual black box (VBB) obfuscation

(B. Barak et al, *On the (im)possibility of obfuscating programs*, CRYPTO 2001, JACM 2012.)

**Obf** : prog → prog is a VBB obfuscator if:

1. It preserves functionality:

$$[\![\mathbf{Obf}(P)]\!](x) = [\![P]\!](x) \qquad \text{for all inputs } x$$

2. The only observable properties of **Obf**$(P)$ are those that can be inferred from its I/O behavior.

For any p. p. time attacker $\mathcal{A}$ : program → $\{0, 1\}$, there exists a p. p. time simulator $\mathcal{S}$ : I/O traces → $\{0, 1\}$ that “performs as well as” $\mathcal{A}$ despite having only access to the I/O behavior of $P$.

$$\left| \Pr\left[\mathcal{A}(\mathbf{Obf}(P)) = 1\right] - \Pr\left[\mathcal{S}([\![P]\!]) = 1\right] \right| \text{ is negligible}$$

# Obfuscation: the ultimate cryptographic primitive?

Most known or proposed cryptographic protocols could easily be implemented if we had a VBB obfuscator.

**Public-key encryption:**

Secret key: $sk \in \{0, 1\}^n$ random.

Public key: $pk = \mathcal{H}(sk)$. ($\mathcal{H}$: pseudo-random function.)

The encryption of message $m$ is the obfuscated function

$$\mathcal{E}_{pk}(m) = \mathbf{Obf}(\lambda s.\ \mathtt{if}\ \mathcal{H}(s) = pk\ \mathtt{then}\ m\ \mathtt{else}\ \bot)$$

Decryption is function application: $\mathcal{D}_{sk}(c) = c(sk)$.

# Obfuscation: the ultimate cryptographic primitive?

## Homomorphic encryption:

If $F$ is a function and $k$ a secret key, the function homomorphic to $F$ that computes over data encrypted with $k$ is

$$\hat{F} = \mathbf{Obf}(\lambda c.\ \mathcal{E}_k(F(\mathcal{D}_k(c))))$$

## Functional encryption:

From the private key $sk$, we can derive evaluation keys $sk_F$ for several functions $F$.

$sk_F$ allows us to compute $F$ over an encrypted input, producing cleartext output, without revealing anything else about the input.

$$sk_F = \mathbf{Obf}(\lambda x.\ F(\mathcal{D}_{sk}(x)))$$

# Obfuscation: the ultimate cryptographic primitive?

**Witness encryption:**

The message is encrypted with respect to a proposition $P$, and can be decrypted only if we provide a proof $\pi$ of $P$.

$$\mathcal{E}_P(m) = \mathbf{Obf}(\lambda\pi.\ \mathtt{if}\ \pi \models P\ \mathtt{then}\ m\ \mathtt{else}\ \bot)$$

$$\mathcal{D}_\pi(c) = c(\pi)$$

# Impossibility of VBB obfuscation

There exists programs that no obfuscator can turn into a black box.

Example (D. Wichs) Let $(sk, pk)$ be a key pair for a fully homomorphic cipher, and random $\alpha, \beta, \gamma$.

$$P = \lambda x. \begin{cases} \mathcal{E}_{pk}(\alpha) & \text{if } x = 0 \\ \beta & \text{if } x = \alpha \\ \gamma & \text{if } \mathcal{D}_{sk}(x) = \beta \\ \perp & \text{otherwise} \end{cases}$$

By homomorphic evaluation of $\mathbf{Obf}(P)$, the attacker obtains $\mathcal{E}_{pk}(\beta)$:

$$\widehat{\mathbf{Obf}(P)}(\mathbf{Obf}(P)(0)) = \widehat{\mathbf{Obf}(P)}(\mathcal{E}_{pk}(\alpha)) = \mathcal{E}_{pk}(\mathbf{Obf}(P)(\alpha)) = \mathcal{E}_{pk}(P(\alpha)) = \mathcal{E}_{pk}(\beta)$$

and recovers $\gamma = \mathbf{Obf}(P)(\mathcal{E}_{pk}(\beta))$.

The simulator $\mathcal{S}$ that has only access to $[\![P]\!]$ has negligible probability to recover $\beta$ and $\gamma$.

# Indistinguishability obfuscation (IO)

(B. Barak *et al*, *op. cit.*)

**iO** : prog $\rightarrow$ prog is an indistinguishable obfuscator if:

1. It preserves functionality:

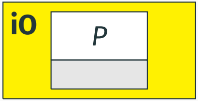
$$[\![\mathbf{iO}(P)]\!](x) = [\![P]\!](x) \qquad \text{for all inputs } x$$

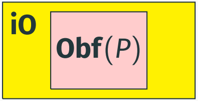
2. The obfuscations of two programs $P_1, P_2$ with the same size and the same I/O behavior ($[\![P_1]\!] = [\![P_2]\!]$) cannot be distinguished in polynomial probabilistic time.

Given $\{\mathbf{iO}(P_1), \mathbf{iO}(P_2)\}$, a p. p. time attacker has probability $1/2 + \varepsilon$ to determine which of the two obfuscated programs correspond to $P_1$.

# IO: the best possible obfuscation?

(S. Goldwasser, G. N. Rothblum, *On Best-Possible Obfuscation*, TCC 2007).





Let **Obf** be a functionality-preserving obfuscator.

Given a program $P$, we add padding to $P$ and **Obf**($P$) so that they have the same size:

$$P_1 = \mathrm{pad}(P) \quad P_2 = \mathrm{pad}(\mathbf{Obf}(P)) \quad |P_1| = |P_2|$$

$$[\![P_1]\!] = [\![P]\!] = [\![\mathbf{Obf}(P)]\!] = [\![P_2]\!]$$

Then, **iO**($P_1$) and **iO**($P_2$) are indistinguishable.

Hence, nothing is gained by applying **Obf** before **iO**.
Applying **iO** is enough.

# IO: the ultimate cryptographic primitive?

The indistinguishability property suffices to prove the security of IO-based implementations of many cryptographic primitives:

- public-key encryption
- witness encryption
- deniable encryption
- homomorphic encryption
- functional encryption
- identity-based encryption
- attribute-based encryption
- short signatures
- etc.

(A. Sahai, B. Waters, *How to use indistinguishability obfuscation: deniable encryption and more*, STOC 2014.)

# Security of IO-based public-key encryption

$\mathcal{H} : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ pseudo-random function.

$$\mathcal{E}_{pk}(m) = \mathbf{iO}(\lambda s.\ \texttt{if}\ \mathcal{H}(s) = pk\ \texttt{then}\ m\ \texttt{else}\ \bot)$$

Game #1: IND-CPA

C: draws $sk \in \{0,1\}^\lambda$, takes $pk = \mathcal{H}(sk)$, sends $pk$ to A

A: chooses $m_0, m_1$ and sends them to C.

C: chooses $b \in \{0,1\}$, sends $\mathcal{E}_{pk}(m_b)$ à A.

A: guesses the value of $b$.

# Security of IO-based public-key encryption

$\mathcal{H} : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ pseudo-random function.

$$\mathcal{E}_{pk}(m) = \mathbf{iO}(\lambda s.\ \texttt{if}\ \mathcal{H}(s) = pk\ \texttt{then}\ m\ \texttt{else}\ \bot)$$

Game #2:

C: draws $r \in \{0,1\}^{2\lambda}$, sends $r$ to A

A: chooses $m_0, m_1$ and sends them to C.

C: chooses $b \in \{0,1\}$, sends $\mathcal{E}_r(m_b)$ to A.

A: guesses the value of $b$.

The attacker cannot distinguish game #2 from game #1, since the results of $\mathcal{H}$ cannot be distinguished from random numbers.
(Assuming the random oracle hypothesis.)
The attacker advantage is the same in games #1 and #2.

# Security of IO-based public-key encryption

$\mathcal{H} : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ pseudo-random function.

$$\mathcal{E}_{pk}(m) = \mathbf{iO}(\lambda s.\ \texttt{if}\ \mathcal{H}(s) = pk\ \texttt{then}\ m\ \texttt{else}\ \bot)$$

Game #3:

C: draws $r \in \{0,1\}^{2\lambda}$, sends $r$ to A

A: chooses $m_0, m_1$ and sends them to C.

C: chooses $b \in \{0,1\}$, sends $\mathbf{iO}(\lambda s.\ \bot)$ to A.

A: guesses the value of $b$.

$r$ is not in the image of $\mathcal{H}$ with probability $1 - 2^{-\lambda}$, and in this case,

$$[\![\lambda s.\ \texttt{if}\ \mathcal{H}(s) = r\ \texttt{then}\ m\ \texttt{else}\ \bot]\!] = [\![\lambda s.\ \bot]\!]$$

and the obfuscations of these functions cannot be distinguished by the attacker. Same advantage in games #2 and #3.

# Security of IO-based public-key encryption

$\mathcal{H} : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ pseudo-random function.

$$\mathcal{E}_{pk}(m) = \mathbf{iO}(\lambda s.\ \texttt{if}\ \mathcal{H}(s) = pk\ \texttt{then}\ m\ \texttt{else}\ \bot)$$

Game #3:

C: draws $r \in \{0,1\}^{2\lambda}$, sends $r$ to A

A: chooses $m_0, m_1$ and sends them to C.

C: chooses $b \in \{0,1\}$, sends $\mathbf{iO}(\lambda s.\ \bot)$ to A.

A: guesses the value of $b$.

The attacker advantage is null in game #3. It's the same advantage in games #1 and #2. Hence, this cipher is IND-CPA.

# Implementing IO for circuits

**Without size and time constraints:**

Just represent the circuit by its truth table.

Same I/O behavior $\Rightarrow$ same truth table.

# Implementing IO for circuits

**Without size and time constraints:**

Just represent the circuit by its truth table.

Same I/O behavior $\Rightarrow$ same truth table.

**Without time constraints:**

Enumerate all circuits in a fixed order. Take the first circuit equivalent to the given circuit.

This gives a canonical form for every circuit.

# Implementing IO for circuits

### Without size and time constraints:

Just represent the circuit by its truth table.

Same I/O behavior $\Rightarrow$ same truth table.

### Without time constraints:

Enumerate all circuits in a fixed order. Take the first circuit equivalent to the given circuit.

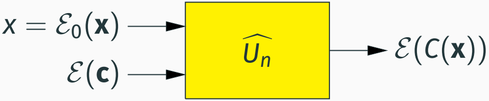
This gives a canonical form for every circuit.

### In probabilistic polynomial time:

No canonicalization possible (if $P \neq NP$).

Cryptography is required!

# Homomorphic evaluation of a universal circuit


$x = \mathcal{E}_0(\mathbf{x})$
$\mathcal{E}(\mathbf{c})$
$\widehat{U_n}$
$\mathcal{E}(C(\mathbf{x}))$

A circuit $C$ of size $n$ can be represented by a bit vector $\mathbf{c}$.

We can have it executed by a universal circuit $U_n$ ($\approx$ a FPGA):

$$U_n(\mathbf{c}, \mathbf{x}) = C(\mathbf{x}).$$

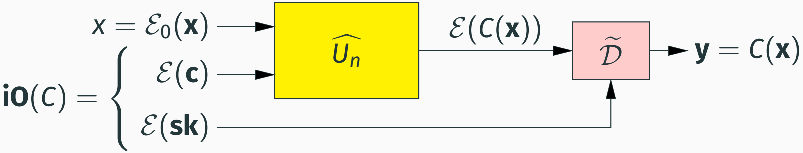
We define the obfuscation of $C$ as $\mathbf{iO}(C) = \mathcal{E}(\mathbf{c})$, where $\mathcal{E}$ is homomorphic encryption.

By homomorphic evaluation of $U_n$, we get

$$\mathbf{z} = \widehat{U_n}(\mathcal{E}(\mathbf{c}), \mathcal{E}_0(\mathbf{x})) = \mathcal{E}(U_n(\mathbf{c}, \mathbf{x})) = \mathcal{E}(C(\mathbf{x}))$$

Problem: $\mathbf{z}$ is encrypted! We want $\mathbf{y} = C(\mathbf{x}) = \mathcal{D}(\mathbf{z})$.

# The decryption circuit



$x = \mathcal{E}_0(\mathbf{x})$
$\mathbf{iO}(C) =$
$\mathcal{E}(\mathbf{c})$
$\mathcal{E}(\mathbf{sk})$
$\widehat{U_n}$
$\mathcal{E}(C(\mathbf{x}))$
$\widetilde{\mathcal{D}}$
$\mathbf{y} = C(\mathbf{x})$

To finish the computation, we must evaluate the decryption circuit $\widetilde{\mathcal{D}}$

- over encrypted inputs $\mathbf{z} = \mathcal{E}(C(\mathbf{x}))$ and $\mathcal{E}(\mathbf{sk})$ (= bootstrap)
- with cleartext output ($\neq$ bootstrap)

# The decryption circuit

For LWE-style encryption, decryption is simple:

$$\mathcal{D}_{\mathbf{sk}}((\mathbf{a}, b)) = \lfloor (b - \langle \mathbf{a}, \mathbf{sk} \rangle)/2^k \rceil$$

With a bit of work, it can be performed by a circuit of low multiplicative depth $d$ $\quad$ ($NC_0$ circuit).

The bits $y_i$ of the result $\mathbf{y} = C(\mathbf{x})$ are therefore multivariate polynomials of degree $d$:

$$y_i = P_i(\mathbf{z}, \mathbf{sk})$$

We need to evaluate these polynomials in an hybrid manner: the **sk** argument is encrypted, but the result is in the clear.

# Hybrid evaluation of a polynomial

Degree 2 polynomial: use a bilinear map (a pairing).

Example: evaluation of $xy + xz$.

$$\begin{tikzcd}
\mathcal{E}(x) \arrow[d, "\text{recoding}"'] & \mathcal{E}(y) \arrow[d] & \mathcal{E}(x) \arrow[d] & \mathcal{E}(z) \arrow[d] & \\
e(g^x, & g^y) \cdot & e(g^x, & g^z) \arrow[r, "\text{pairing}"] & e(g,g)^{xy+xz} = 1 \quad ?
\end{tikzcd}$$

This computation determines whether $xy + xz = 0$ or $xy + yz \neq 0$.

# Hybrid evaluation of a polynomial

Degree 2 polynomial: use a bilinear map (a pairing).

Example: evaluation of $xy + xz$.

$$\begin{tikzcd}
\mathcal{E}(x)\ \mathcal{E}(y)\ \mathcal{E}(x)\ \mathcal{E}(z) \arrow[d, "\text{recoding}"'] & \\
e(g^x, g^y) \cdot e(g^x, g^z) \arrow[r, "\text{pairing}"] & e(g,g)^{xy+xz} = 1 \quad ?
\end{tikzcd}$$

This computation determines whether $xy + xz = 0$ or $xy + yz \neq 0$.

Degree $d > 2$ polynomial: use a multilinear map with $d$ arguments?

$$e(g^{a_1}, \ldots, g^{a_d}) = e(g, \ldots, g)^{a_1 \cdots a_d}$$

# The ups and downs of multilinear maps

**2013** Garg, Gentry, Halevi, Raykova, Sahai, Waters: *Candidate indistinguishability obfuscation and functional encryption for all circuits*.

Main hypothesis: secure multilinear maps exist.

# The ups and downs of multilinear maps

**2013** Garg, Gentry, Halevi, Raykova, Sahai, Waters: *Candidate indistinguishability obfuscation and functional encryption for all circuits*.

Main hypothesis: secure multilinear maps exist.

**2014-2019** Barak, Boneh, Brakerski, Cheon, Coron, Fouque, Gentry, Halevi, Han, Hopkins, Hu, Jain, Jia, Komargodski, Kothari, Lee, Lepoint, Lin, Maji, Miles, Minaud, Raykva, Ryu, Sahai, Stehlé, Tibouchi, Vaikuntanathan, Wu, Zhandry, Zimmerman, and several others

Cryptanalysis of multilinear maps. They are insecure. It's hopeless!

# The ups and downs of multilinear maps

**2013** Garg, Gentry, Halevi, Raykova, Sahai, Waters: *Candidate indistinguishability obfuscation and functional encryption for all circuits*.

Main hypothesis: secure multilinear maps exist.

**2014-2019** Barak, Boneh, Brakerski, Cheon, Coron, Fouque, Gentry, Halevi, Han, Hopkins, Hu, Jain, Jia, Komargodski, Kothari, Lee, Lepoint, Lin, Maji, Miles, Minaud, Raykva, Ryu, Sahai, Stehlé, Tibouchi, Vaikuntanathan, Wu, Zhandry, Zimmerman, and several others

Cryptanalysis of multilinear maps. They are insecure. It's hopeless!

**2021** Jain, Lin, Sahai: *Indistinguishability obfuscation from well-founded assumptions*.

Doesn't use multilinear maps. Hypotheses: LPN, DLIN (security of bilinear maps), and existence of a PRG in $NC_0$.

# Summary on indistinguishable obfuscation

As interesting as ever as the ultimate cryptographic primitive: “crypto-complete”, “Obfustopia”, …

One construction that remains unbroken: Jain-Lin-Sahai 2021. Several other constructions rely on false or dubious hypotheses.

Still very far from a usable implementation.

Close connections with a problem closer to applications: functional encryption.

(→ David Pointcheval’s seminar)

# References

# References

On oblivious RAM and its applications for MPC:

- *A pragmatic introduction to secure multi-party computation*, David Evans, Vladimir Kolsnikov, Mike Rosulek, NOW Publishers, 2018. Sections 5.2 to 5.5.

On DEPIR and RAM-FHE:

- Wei-Kai Lin, Ethan Mook, Daniel Wichs, *Doubly Efficient Private Information Retrieval and Fully Homomorphic RAM Computation from Ring LWE*, STOC 2023. https://eprint.iacr.org/2022/1703

Overview articles on indistinguishable obfuscation:

- Boaz Barak, *Hopes, Fears and Software Obfuscation*, CACM, 2016. https://doi.org/10.1145/2757276
- Aayush Jain, Huijia Lin, Amit Sahai, *Indistinguishability Obfuscation from Well-Founded Assumptions*, CACM, 2024. https://doi.org/10.1145/3611095

# Conclusion

# Three sides of cryptography

## A perspective on information theory.

- One-time pad, Shannon's early work, …
- Information-theoretic security *vs.* computational security.

# Three sides of cryptography

A perspective on information theory.

- One-time pad, Shannon's early work, …
- Information-theoretic security *vs.* computational security.

A perspective on complexity theory.

- Average-case complexity.
- Different classes of computations depending on the existence (or not) of one-way functions, pseudo-random functions, pseudo-random generators, …

# Three sides of cryptography

A perspective on information theory.

- One-time pad, Shannon's early work, ...
- Information-theoretic security *vs.* computational security.

A perspective on complexity theory.

- Average-case complexity.
- Different classes of computations depending on the existence (or not) of one-way functions, pseudo-random functions, pseudo-random generators, ...

A new perspective on software and programming?

- What I tried to give in these lectures.
- Many circuits, not enough programming languages!
- An "extremist" approach to software security.

Computing over encrypted or private data:

a fruitful approach

that opens new approaches to computer security

# FIN