Le calcul sécurisé, septième cours

Sécuriser le calcul : autres directions et conclusions

Xavier Leroy

2025-12-18

Collège de France, chaire de sciences du logiciel
xavier.leroy@college-de-france.fr

Mémoire sécurisée, 1 :
RAM inconsciente (*Oblivious RAM*)

Calcul sécurisé avec accès direct à une mémoire

Les approches vues jusqu'ici (chiffrement homomorphe et calcul multipartite) représentent les calculs par des circuits combinatoires booléens ou arithmétiques.

Théorème

Toute machine de Turing de taille n qui termine en temps $\mathrm{poly}(n)$ *peut être réalisée par un circuit de taille* $\mathrm{poly}(n)$.

Cependant, beaucoup de calculs s'expriment plus facilement et s'exécutent plus rapidement en utilisant une mémoire à accès direct (*random-access memory*) :

- ROM (lecture seule) : tables de constantes, base de données
- RAM (lecture/écriture) : stockage de résultats.

Calcul sécurisé avec accès direct à une mémoire

On peut réaliser l'accès direct à un index $x \in \{0, \ldots, n-1\}$ avec des circuits :

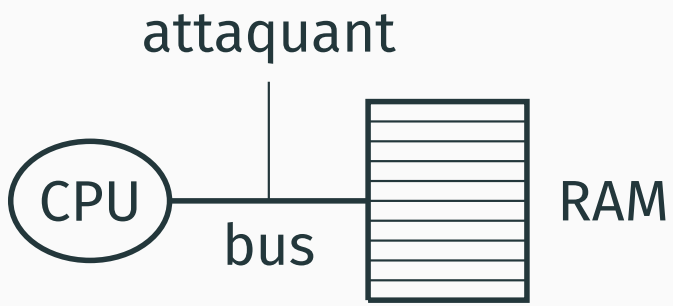
- Circuit booléen : n multiplexeurs organisés en arbre binaire.
- Circuit arithmétique : polynôme d'interpolation

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

L'accès ≪touche≫ les n entrées de la mémoire.

Folklore : un accès privé à une base de données doit accéder à toutes les entrées de la base ; sinon, il révèle des informations sur l'accès.

RAM inconsciente (ORAM, *Oblivious RAM*)



Exemple d'utilisation : une enclave sécurisée qui utilise une RAM ordinaire pour stocker des données.

Les accès à la RAM (adresses et données écrites ou lues) sont visibles de l'attaquant, mais ne doivent rien révéler sur le calcul fait dans l'enclave (sécurité passive).

- Les données doivent être chiffrées.
- Les adresses doivent être brouillées.
- Deux accès à la même adresse logique doivent être brouillés différemment.

Square-root ORAM

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

Une ORAM de taille N :

- Un cache de taille n (idéalement, $n \approx \sqrt{N}$, d'où le nom)
- Une RAM de taille $N + n$ (N vraies données + n leurres)
- Une permutation aléatoire des adresses :
  $\pi : \{0, \ldots, N + n - 1\} \to \{0, \ldots, N + n - 1\}$
- La case $\pi(i)$ de la RAM contient un chiffré de (i, v_i) pour $i \in \{0, \ldots, N - 1\}$.
- Les cases $\pi(N), \ldots, \pi(N + n - 1)$ sont des leurres.

Lecture d'une entrée

Si l'adresse i n'est pas dans le cache :

- Lire e à l'adresse $\pi(i)$ dans la RAM externe.
- Déchiffrer $(i, x) \leftarrow \mathcal{D}(e)$.
- Ajouter $i \mapsto x$ dans le cache.
- Renvoyer x.

Si l'adresse i est dans le cache, avec la valeur x :

- Lire à l'adresse $\pi(N + k)$ dans la RAM externe.
- Renvoyer x.

Dans les deux cas :

- Incrémenter k.
- Si $k \geq n$, tirer et appliquer une nouvelle permutation π'.

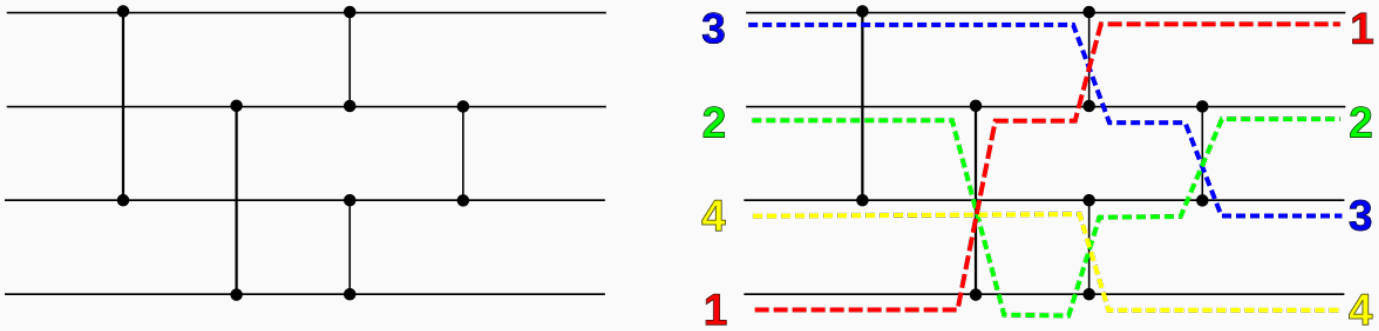
Appliquer une permutation de manière inconsciente

La RAM contient des couples chiffrés $\mathcal{E}(i, v_i)$. On veut les ordonner suivant la permutation π :

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

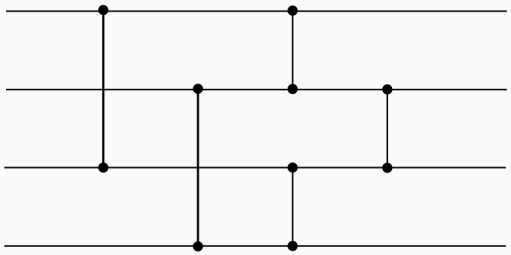
Cela revient à trier la RAM par $\pi(i)$ croissant.

On utilise un réseau de tri, de sorte que les accès dans la RAM suivent un motif fixé à l'avance, indépendant des données.



(Oskar Sigvardsson, CC BY)

Appliquer une permutation de manière inconsciente



Pour chaque paire de points $(u, v), u < v$ dans le réseau de tri :

- Lire et déchiffrer
  $$(i, v_i) \leftarrow \mathcal{D}(RAM[u]) \qquad (j, v_j) \leftarrow \mathcal{D}(RAM[v])$$
- Si $\pi(i) < \pi(j)$: rechiffrer et réécrire dans le même ordre
  $$RAM[u] \leftarrow \mathcal{E}(i, v_i) \qquad RAM[v] \leftarrow \mathcal{E}(j, v_j)$$
- Si $\pi(i) > \pi(j)$: échanger et réécrire
  $$RAM[u] \leftarrow \mathcal{E}(j, v_j) \qquad RAM[v] \leftarrow \mathcal{E}(i, v_i)$$

Mêmes motifs d'accès à la RAM dans tous les cas.

Représenter implicitement une permutation aléatoire

Le client n'a pas assez de mémoire pour stocker π *in extenso*.

Il engendre π à partir d'une fonction pseudo-aléatoire h :

$$\pi(i) = \text{nombre de } j \in \{0, N+n-1\} \text{ tels que } h(j) < h(i)$$

La RAM contient des couples $\mathcal{E}(i, v_i)$ triés par $h(i)$ croissant.

Pour accéder à $RAM[\pi(i)]$ on fait une recherche dichotomique :

$(j, v_j) \leftarrow \mathcal{D}(RAM[p])$
si $j = i$, on a trouvé
si $h(i) < h(j)$, déplacer p vers la gauche
si $h(i) > h(j)$, déplacer p vers la droite

Le motif des accès à la RAM ne révèle rien d'autre que $\pi(i)$.

Résumé : une ROM inconsciente

Initialement : les données $v_0, \ldots, v_{N-1}$ sont stockées dans $RAM[0], \ldots, RAM[N-1]$.

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

Répéter jusqu'à arrêt du client :

- Tirer une fonction pseudo-aléatoire h
  (typiquement un chiffrement par bloc avec une clé aléatoire).
- Trier la RAM par $h(i)$ croissants avec un réseau de tri.
- Lire n valeurs.

Temps amorti d'une lecture : $\mathcal{O}(\log N + N \log^2 N/n)$
c.à.d. $\mathcal{O}(\sqrt{N} \log^2 N)$ si $n \approx \sqrt{N}$.

Extension : une RAM inconsciente

Tous les accès sont du type lecture-modification-écriture (RMW) :

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

pour ne pas distinguer lectures ($f(x) = x$) et écritures ($f(x) = v$).

Chaque accès fait une lecture en RAM et conserve la nouvelle valeur y dans le cache.

Après n accès, avant de re-permuter la RAM, on vide le cache en effectuant des écritures en RAM aux mêmes adresses et dans dans le même ordre que les lectures.

Amélioration des performances

Goldreich et Ostrovsky proposent d'utiliser une hiérarchie de caches de tailles $2, 4, 8, \ldots, 2^p$, stockés en RAM, re-permutés avec une fréquence inversement proportionnelle à leur taille.
$\Rightarrow$ Accès en temps amorti $\mathcal{O}(\log^3 N)$. Espace total $\mathcal{O}(N \log N)$.

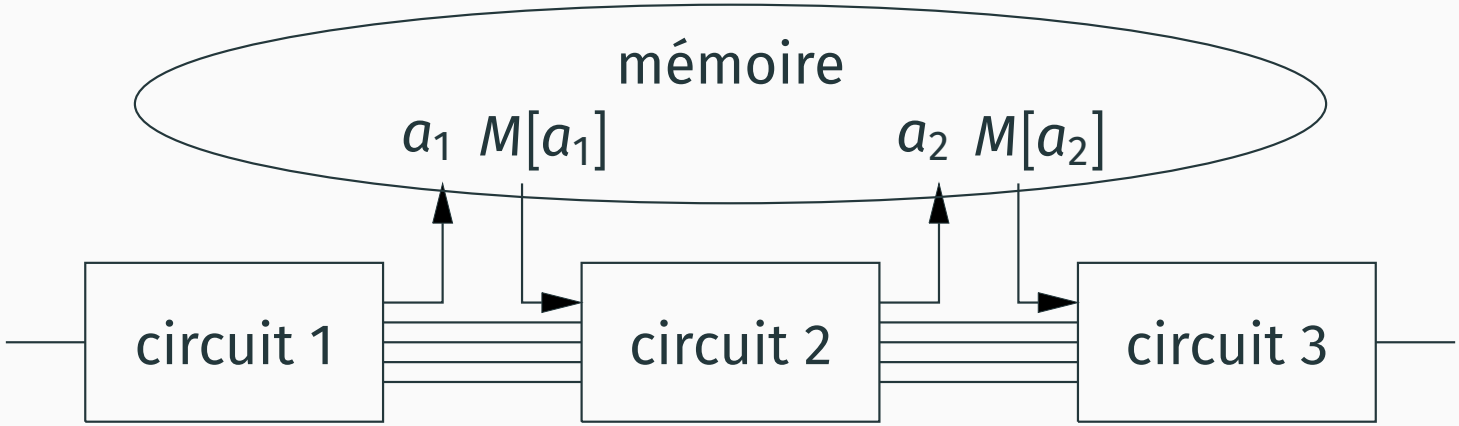
Autres approches : organiser les données en arbres binaires au lieu de tables de hachages.

Meilleur résultat connu : temps amorti $\mathcal{O}(\log N)$, environ 26 accès RAM par accès ORAM pour $N = 1\,\text{To}$.

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

Mémoire sécurisée, 2 : accès chiffré homomorphe à une table

Circuits ayant accès à une table en lecture seule



Un modèle de calcul intermédiaire entre circuits et modèle RAM.

Approprié pour le calcul de statistiques simples sur un grand volume de données.

Peut-on effectuer ce calcul de manière homomorphe, avec des adresses et des données chiffrées ?

Le problème PIR (*Private Information Retrieval*)

Exécuter une requête sur une base de données DB de taille n sans que le serveur n'apprenne rien sur la requête et sur son résultat.

Folklore : toutes les entrées de la base doivent être consultées. Sinon le serveur apprendrait que la requête ne dépend pas des entrées non consultées.

$\Rightarrow$ Chaque requête est en temps $\mathcal{O}(n)$ au moins.

L'approche DEPIR : (*doubly-efficient PIR*) (Lin, Mook, Wichs, 2023)

- Préprocesser DB en une base DB' légèrement plus grande, en temps superlinéaire en n.
- Exécuter chaque requête privée sur DB' en temps sous-linéaire en n, voire $\text{poly}(\log n)$.
- Les motifs d'accès dans DB' ne révèlent pas de motif d'accès dans DB.

Préprocesser l'évaluation d'un polynôme

Soit $P \in \mathbb{Z}_q[X]$ un polynôme de degré d.

On veut calculer $P(x)$ pour plusieurs valeurs de x.

Sans précalcul :

- Évaluer directement P pour chaque x, en temps $\mathcal{O}(d)$.

Tabulation naïve :

- Tabuler les q valeurs de P : $t[i] = P(i)$ pour $i = 0, \ldots, q-1$ (temps et espace $\mathcal{O}(q)$)
- Pour chaque x, renvoyer $t[x]$ en temps $\mathcal{O}(1)$.

Préprocesser l'évaluation d'un polynôme

Tabulation «chinoise» :

On voit P comme un polynôme $\tilde{P}$ à coefficients dans $\mathbb{Z}$.
$P(x) = \tilde{P}(x) \bmod q$ pour tout $x \in \{0, \ldots, q-1\}$.

On a $0 \leq \tilde{P}(x) < M = d(q-1)^d$ pour tout $x \in \{0, \ldots, q-1\}$.

Soient $p_1, \ldots, p_n$ des petits nombres premiers tels que

$$M \leq p_1 \cdots p_n$$

(Les $16 \log M$ premiers premiers conviennent.)

Pour chaque p_i, on tabule $\tilde{P} \bmod p_i$:

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

Préprocesser l'évaluation d'un polynôme

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

Évaluation ≪chinoise≫ :

Pour évaluer $P(x) = \tilde{P}(x) \bmod q$, on consulte les n tables t_i :

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

On utilise le théorème des restes chinois pour calculer

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

On a donc $P(x) = y \bmod q$.

Préprocesser l'évaluation d'un polynôme

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

Évaluation ≪chinoise≫ :

Pour évaluer $P(x) = \tilde{P}(x) \bmod q$, on consulte les n tables t_i :

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

On utilise le théorème des restes chinois pour calculer

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

On a donc $P(x) = y \bmod q$.

Optimisation en espace : si certaines tables t_i sont trop grandes, on peut récurser pour représenter $P \bmod t_i$ par plusieurs petites tables.

Exemple

Polynôme de degré $d = 20$ modulo 2^{64}. Évaluation en 2 niveaux :

- 37 tables qui donnent les valeurs de P modulo 2, 3, 5, …, 157.
- En utilisant le théorème des restes chinois, on obtient les valeurs exactes de $P(i)$ pour $i \leq 1028$.
- On connaît donc $P \bmod p$ pour $p = 163, 167, \ldots, 937$.
- En utilisant à nouveau le théorème des restes chinois, on connaît donc $P \bmod 2^{64}$.

Le calcul reste plus coûteux que l'évaluation directe de P (37 lectures mémoire au lieu de 21).

Mais ce n'est plus vrai pour un polynôme multivarié !

Exemple : polynôme à 4 variables de degré 20 modulo 2^{64} : 126 tables *vs* $21^4 = 194481$ coefficients.

Préprocesser l'évaluation d'un polynôme

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

Cette approche s'étend à des polynômes à m variables de degré d en chaque variable, et à des anneaux R autres que $\mathbb{Z}_q$.

Précalcul en temps et espace (avec 2 niveaux de tables)

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

Évaluation en temps $\mathrm{poly}(d, m, \log |R|)$.

Le précalcul est quasilinéaire en $N = (d+1)^m$ (le nombre de coefficients de P), et l'évaluation est polylogarithmique en N.

Application au PIR

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

On suppose que la base de données DB est un tableau à m dimensions contenant d^m entrées.

On peut appliquer le traitement de Kedlaya-Umans au polynôme d'interpolation P correspondant à DB :

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

Pour rendre les requêtes vraiment privées, Lin, Mook et Wichs proposent de pré-traiter une version chiffrée P' de P :

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

en utilisant un chiffrement faiblement homomorphe algébrique.

Chiffrement faiblement homomorphe algébrique (ASHE)

Un chiffrement $\mathcal{E} : \mathbb{Z}_q \to R$, où R est un anneau traitable par Kedlaya-Umans. L'addition homomorphe est l'addition de l'anneau, et de même pour la multiplication.

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

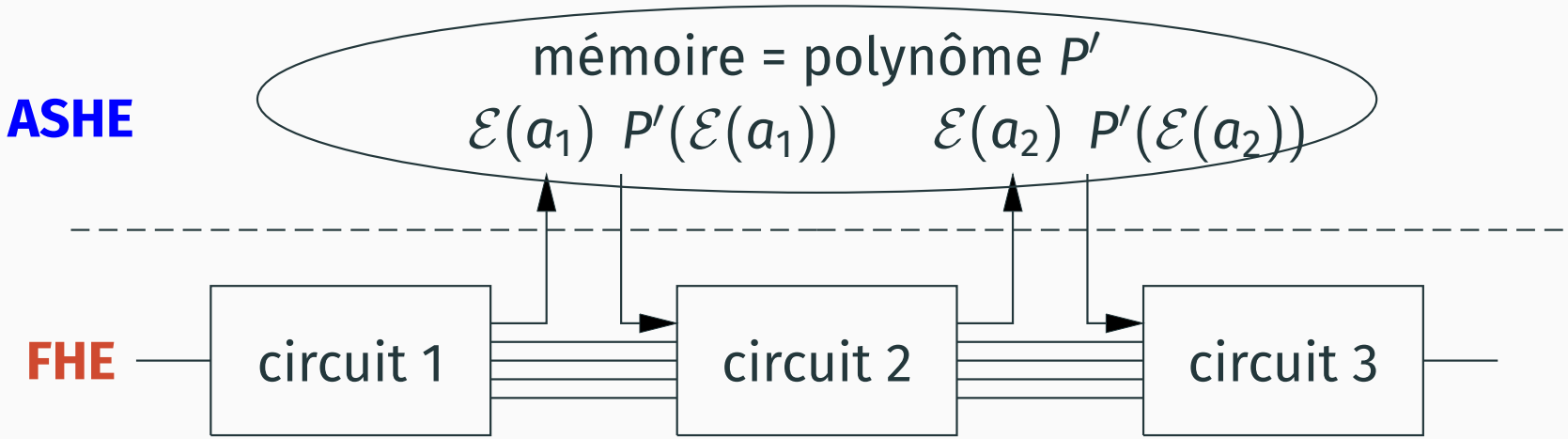
Faiblement homomorphe : il suffit de pouvoir évaluer des monômes $c \cdot X_1^{e_1} \cdots X_m^{e_m}$ de degré $\leq d$.

Exemple : le chiffrement de van Dijk *et al* 2010 (2e cours) :

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

De meilleures constructions sont possibles à base de RLWE.

Application au calcul homomorphe



Évaluation des circuits avec un chiffrement totalement homomorphe (FHE) comme vu au 3e cours.

Conversions FHE $\rightarrow$ ASHE sur les adresses et ASHE $\rightarrow$ FHE sur les résultats des lectures.

Ajout d'une mémoire RAM (avec écritures) : voir Lin, Mook, Wichs.

Obfuscation indistinguable

L’obfuscation de logiciels

Rendre un code exécutable ou source inintelligible :
résiste au désassemblage, à la revue de code, à l’analyse statique.

Utilisations :

- Empêcher la rétro-ingéniérie (*reverse engineering*)
  (algorithmes propriétaires, fonctionnalités désactivées, …)
- Empêcher le contournement de protections logicielles
  (contre la copie, *digital rights management*, …)
- Pour s’amuser !
  (œufs de Pâques, concours IOCCC, …)

Exemples de codes obfusqués

```
#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)

```
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)

Quelques techniques d'obfuscation

Renommages des fonctions et des variables.

Insertion de calculs redondants :

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

Codage du flux de contrôle sous forme de tables ou d'automates.

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

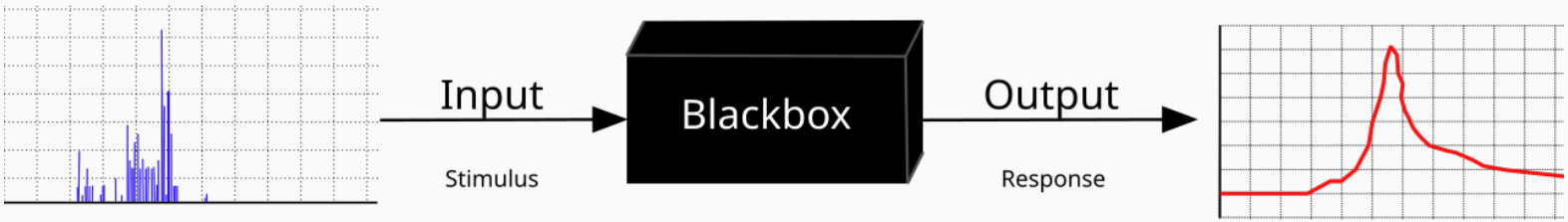
Changement de représentation des données.

Masquage ou chiffrement du code ; code auto-modifiant.

(Voir le séminaire de S. Blazy du 14/04/2022 et le livre de Goldberg et Nagra, *Surreptitious Software*, 2010.)

Obfuscation de qualité cryptographique

Transformer un code source contenant des secrets en un programme exécutable «boîte noire».



Les seules informations que l'on peut extraire du programme sont les observations de ses exécutions.

Obfuscation de qualité cryptographique

Une des premières pistes envisagées pour réaliser le chiffrement à clé publique !

> *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.)

Obfuscation ≪boîte noire virtuelle≫ (VBB, *virtual black box*)

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

Obf : prog $\to$ prog est un obfuscateur VBB si :

1. Il préserve la fonctionnalité :

$$[\![\mathbf{Obf}(P)]\!](x) = [\![P]\!](x) \qquad \text{pour toute entrée } x$$

2. Les seules propriétés que l'on peut établir de $\mathbf{Obf}(P)$ sont celles que l'on peut inférer de son comportement d'E/S.

Pour tout attaquant $\mathcal{A}$: programme $\to \{0,1\}$ en temps p.p., il existe un simulateur $\mathcal{S}$ en temps p.p. qui ≪fait aussi bien≫ en ayant seulement accès au comportement d'E/S de P :

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

L'obfuscation comme primitive cryptographique ultime

La plupart des primitives cryptographiques connues ou envisagées se réalisent facilement si on dispose d'un obfuscateur VBB.

Chiffrement à clé publique :

Clé secrète : $sk \in \{0,1\}^n$ aléatoire.

Clé publique : $pk = \mathcal{H}(sk)$. ($\mathcal{H}$: fonction pseudo aléatoire.)

Le chiffré du message m est la fonction obfusquée

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

Le déchiffrement est l'application $\mathcal{D}_{sk}(c) = c(sk)$.

L'obfuscation comme primitive cryptographique ultime

Chiffrement homomorphe :

Si F est une fonction et k une clé secrète, la version homomorphe de F qui calcule sur des données chiffrées avec k est

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

Chiffrement fonctionnel : (*functional encryption*)

À partir de la clé privée sk, on peut dériver des clés sk_F pour diverses fonctions F.

sk_F permet de calculer F sur une entrée chiffrée, sans pouvoir faire quoi que ce soit d'autre avec cette entrée.

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

L'obfuscation comme primitive cryptographique ultime

Chiffrement avec témoin (*witness encryption*)

Le message est chiffré par rapport à une proposition P, et ne peut être déchiffré que si on fournit une preuve π de P.

$$\mathcal{E}_P(m) = \textbf{Obf}(\lambda\pi.\ \texttt{if}\ \pi \models P\ \texttt{then}\ m\ \texttt{else}\ \bot)$$
$$\mathcal{D}_\pi(c) = c(\pi)$$

Impossibilité de l'obfuscation VBB

Il existe des programmes qu'aucun obfuscateur ne peut rendre «boîte noire».

Exemple (D. Wichs) Soit (sk, pk) une clé pour un chiffrement totalement homomorphe, et α, β, γ aléatoires.

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

Par évaluation homomorphe de $\mathbf{Obf}(P)$, l'attaquant obtient $\mathcal{E}_{pk}(\beta)$:

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

et en déduit $\gamma = \mathbf{Obf}(P)(\mathcal{E}_{pk}(\beta))$.

Le simulateur $\mathcal{S}$ qui observe uniquement $[\![P]\!]$ a une probabilité négligeable d'apprendre β et γ.

Obfuscation indistinguable (IO, *indistinguishability obfuscation*)

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

iO : prog $\to$ prog est un obfuscateur indistinguable si :

1. Il préserve la fonctionnalité :

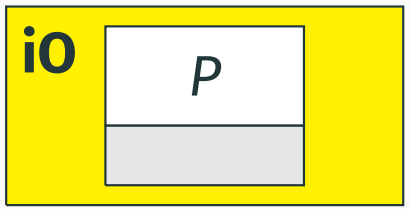
$$[\![\mathbf{iO}(P)]\!](x) = [\![P]\!](x) \qquad \text{pour toute entrée } x$$

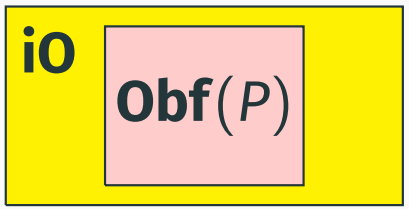
2. Les obfuscations de deux programmes P_1, P_2 de même taille et de même comportement d'entrée-sortie ($[\![P_1]\!] = [\![P_2]\!]$) sont impossibles à distinguer en temps polynomial probabiliste.

Étant donné $\{\mathbf{iO}(P_1), \mathbf{iO}(P_2)\}$, un attaquant en temps p.p. a une probabilité $1/2 + \varepsilon$ de déterminer lequel des deux programmes obfusqués correspond à P_1.

IO : la meilleure obfuscation que l'on peut atteindre ?

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





Soit **Obf** un autre obfuscateur qui préserve la fonctionnalité.

Pour un programme P donné, on ajoute du «*padding*» à P et $\mathbf{Obf}(P)$ pour qu'ils aient la même taille :

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

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

Alors, $\mathbf{iO}(P_1)$ et $\mathbf{iO}(P_2)$ sont indistinguables.

Cela n'a donc servi à rien d'appliquer **Obf** avant **iO**.
Appliquer **iO** suffit.

IO : la primitive cryptographique ultime ?

La propriété d'indistinguabilité suffit à montrer la sécurité d'implémentations à base d'IO de nombreuses primitives cryptographiques :

- chiffrement à clé publique
- chiffrement à témoin
- chiffrement déniable
- chiffrement homomorphe
- chiffrement fonctionnel
- chiffrement basé sur l'identité
- chiffrement basé sur des attributs
- signatures courtes
- etc.

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

Sécurité du chiffrement à clé publique à base d'IO

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

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

Jeu n° 1 : IND-CPA

C : tire $sk \in \{0,1\}^\lambda$, prend $pk = \mathcal{H}(sk)$, envoie pk à A

A : choisit m_0, m_1 et les envoie à C.

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

A : retrouve la valeur de b.

Sécurité du chiffrement à clé publique à base d'IO

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

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

Jeu n° 2 :

C : tire $r \in \{0,1\}^{2\lambda}$, envoie r à A

A : choisit m_0, m_1 et les envoie à C.

C : choisit $b \in \{0,1\}$, envoie $\mathcal{E}_r(m_b)$ à A.

A : retrouve la valeur de b.

L'attaquant A ne peut pas distinguer ce jeu du jeu n° 1, car les résultats de $\mathcal{H}$ sont indistinguables de nombres tirés au hasard. L'avantage de l'attaquant est le même dans les jeux 1 et 2.

Sécurité du chiffrement à clé publique à base d'IO

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

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

Jeu n° 3 :

C : tire $r \in \{0,1\}^{2\lambda}$, envoie r à A

A : choisit m_0, m_1 et les envoie à C.

C : choisit $b \in \{0,1\}$, envoie $\mathbf{iO}(\lambda s.\ \bot)$ à A.

A : retrouve la valeur de b.

r n'est pas dans l'image de $\mathcal{H}$ avec probabilité $1 - 2^{-\lambda}$. Donc,

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

et les obfuscations de ces deux fonctions sont indistinguables par l'attaquant. Donc pour lui c'est le même jeu que n° 2.

Sécurité du chiffrement à clé publique à base d'IO

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

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

Jeu n° 3 :

C : tire $r \in \{0,1\}^{2\lambda}$, envoie r à A

A : choisit m_0, m_1 et les envoie à C.

C : choisit $b \in \{0,1\}$, envoie $\mathbf{iO}(\lambda s.\ \bot)$ à A.

A : retrouve la valeur de b.

L'avantage de l'attaquant est nul dans le jeu n° 3. C'est le même dans les jeux n° 1 et n° 2. Donc ce chiffrement est IND-CPA.

Réaliser IO pour des circuits

Sans contraintes de taille et de temps :

Il suffit de représenter le circuit par sa table de vérité.

Même comportement d'entrée-sortie $\Rightarrow$ même table de vérité.

Réaliser IO pour des circuits

Sans contraintes de taille et de temps :

Il suffit de représenter le circuit par sa table de vérité.

Même comportement d'entrée-sortie $\Rightarrow$ même table de vérité.

Sans contraintes de temps :

On énumère tous les circuits dans un ordre fixé et on prend le premier circuit équivalent au circuit C donné.

Cela donne une forme canonique pour chaque circuit.

Réaliser IO pour des circuits

Sans contraintes de taille et de temps :

Il suffit de représenter le circuit par sa table de vérité.

Même comportement d'entrée-sortie $\Rightarrow$ même table de vérité.

Sans contraintes de temps :

On énumère tous les circuits dans un ordre fixé et on prend le premier circuit équivalent au circuit C donné.

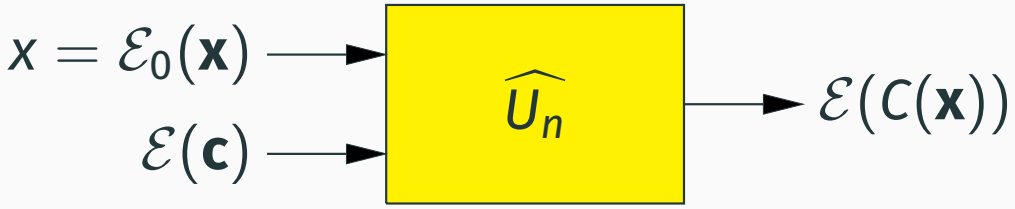
Cela donne une forme canonique pour chaque circuit.

En temps polynomial probabiliste :

Pas de mise en forme canonique possible (si $P \neq NP$).

Il faut faire de la cryptographie !

Évaluation homomorphe d'un circuit universel



On représente le circuit C de taille n par un vecteur de bits $\mathbf{c}$.

On peut le faire exécuter par un circuit universel U_n ($\approx$ FPGA) :

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

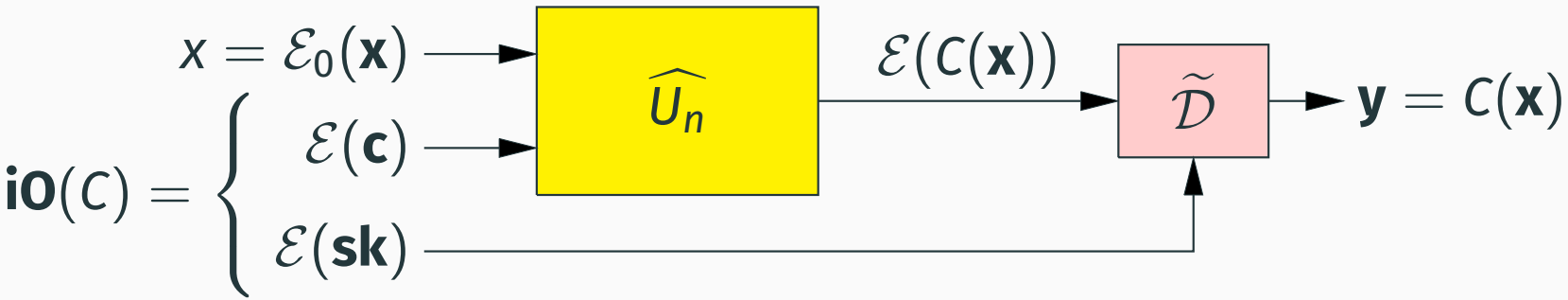
L'obfuscation de C est $\mathbf{iO}(C) = \mathcal{E}(\mathbf{c})$ (chiffrement homomorphe).

Par évaluation homomorphe de U_n, on obtient

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

Problème : $\mathbf{z}$ est chiffré ! Il nous faut $\mathbf{y} = C(\mathbf{x}) = \mathcal{D}(\mathbf{z})$.

Le circuit de déchiffrement



Pour finir le calcul, il faut évaluer le circuit de déchiffrement $\widetilde{\mathcal{D}}$

- sur des entrées chiffrées $\mathbf{z} = \mathcal{E}(C(\mathbf{x}))$ et $\mathcal{E}(\mathbf{sk})$ ($=$ bootstrap)
- avec une sortie en clair. ($\neq$ bootstrap)

Le circuit de déchiffrement

Pour un chiffrement de type LWE, le déchiffrement est simple :

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

En travaillant un peu, on peut le réaliser par un circuit NC_0 de faible profondeur multiplicative d.

Les bits y_i du résultat $\mathbf{y} = C(\mathbf{x})$ s'expriment donc comme des polynômes multivariés de degré d :

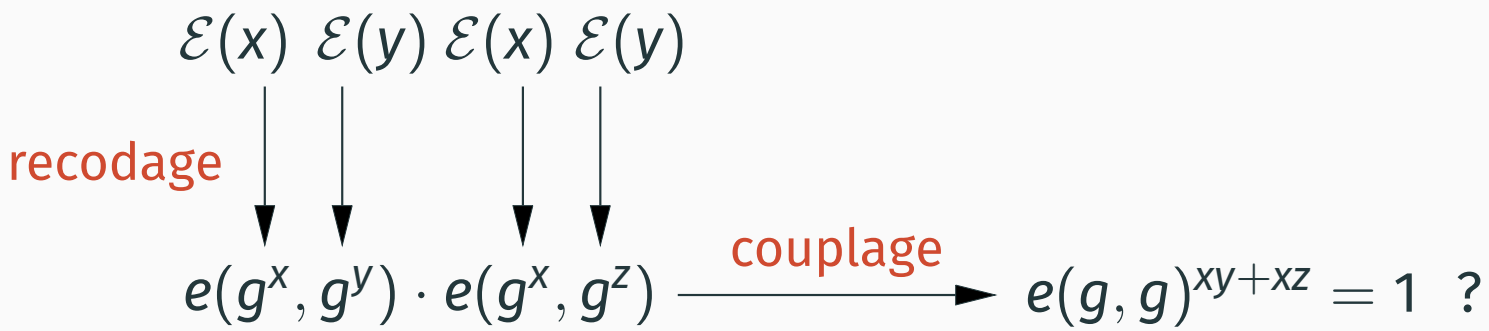
$$y_i = P_i(\mathbf{z}, \mathbf{sk})$$

Il faut les évaluer de manière mixte :
l'argument **sk** est chiffré mais le résultat est en clair.

Évaluation mixte d'un polynôme

Polynôme de degré 2 : utiliser une forme bilinéaire (un couplage).

Exemple : évaluation de $xy + xz$.

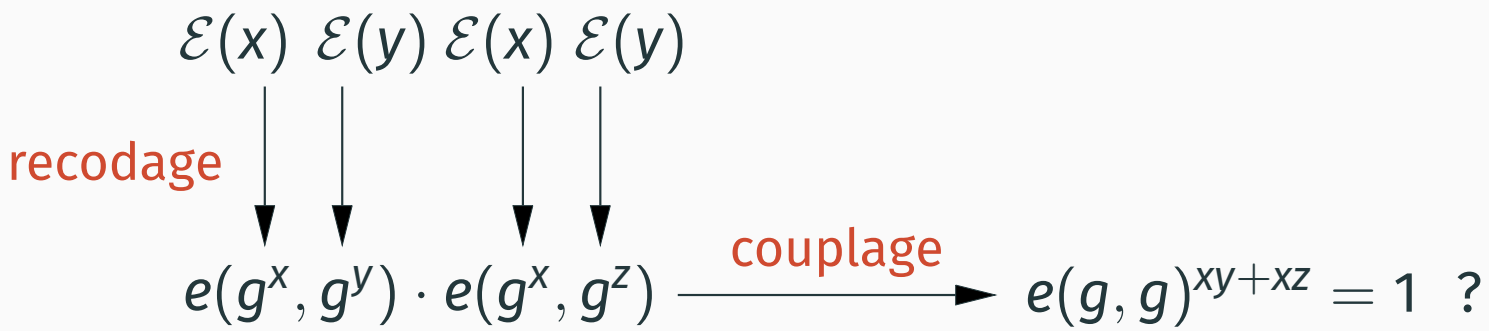


Ceci détermine si $xy + xz = 0$ ou $xy + yz \neq 0$.

Évaluation mixte d'un polynôme

Polynôme de degré 2 : utiliser une forme bilinéaire (un couplage).

Exemple : évaluation de $xy + xz$.



Ceci détermine si $xy + xz = 0$ ou $xy + yz \neq 0$.

Polynôme de degré $d > 2$: utiliser une forme multilinéaire à d arguments ?

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

Heurs et malheurs des formes multilinéaires (*multilinear maps*)

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

Hypothèse principale : sécurité des formes multilinéaires.

Heurs et malheurs des formes multilinéaires (*multilinear maps*)

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

Hypothèse principale : sécurité des formes multilinéaires.

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, *et al*

Cryptanalyse des formes multilinéaires. C'est sans espoir !

Heurs et malheurs des formes multilinéaires (*multilinear maps*)

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

Hypothèse principale : sécurité des formes multilinéaires.

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, *et al*

Cryptanalyse des formes multilinéaires. C'est sans espoir !

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

Pas de formes multilinéaires. Hypothèses : LPN, DLIN (sécurité des formes bilinéaires), existence d'un PRG dans NC_0.

Un point sur l'obfuscation indistinguable

Toujours aussi intéressante comme primitive cryptographique ultime : ≪*crypto-complete*≫, ≪*Obfustopia*≫, …

Une construction qui résiste encore : Jain-Lin-Sahai 2021.
Plusieurs autres s'appuient sur des hypothèses fausses ou douteuses.

Encore très loin d'une implémentation utilisable.

Liens étroits avec un problème plus proche des applications : le chiffrement fonctionnel.

($\rightarrow$ Séminaire de David Pointcheval)

Bibliographie

Bibliographie

Sur la RAM inconsciente et ses applications en MPC :

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

Sur DEPIR et 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

Articles de synthèse sur l'obfuscation indistinguable :

- 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

Trois moments de la cryptographie

Un regard sur la théorie de l'information.

- Masque jetable, travaux de Shannon, …
- Sécurité inconditionnelle *vs.* sécurité calculatoire.

Trois moments de la cryptographie

Un regard sur la théorie de l'information.

- Masque jetable, travaux de Shannon, …
- Sécurité inconditionnelle *vs.* sécurité calculatoire.

Un regard sur la théorie de la complexité.

- Complexité dans le cas moyen.
- Existence d'OWF, de PRF, de PRG, …

Trois moments de la cryptographie

Un regard sur la théorie de l'information.

- Masque jetable, travaux de Shannon, …
- Sécurité inconditionnelle *vs.* sécurité calculatoire.

Un regard sur la théorie de la complexité.

- Complexité dans le cas moyen.
- Existence d'OWF, de PRF, de PRG, …

Un regard nouveau sur le logiciel et la programmation ?

- Ce que j'ai voulu faire dans ce cours.
- Beaucoup de circuits, pas assez de langages !
- Une approche ≪maximaliste≫ de la sécurité du logiciel.

Calculer sur des données chiffrées ou privées :

une approche féconde et à même de renouveler

la sécurité informatique

FIN