CEA LIST

Introduction au cryptocalcul à base de systèmes homomorphes

Renaud Sirdey
(renaud.sirdey@cea.fr)
Travail en collaboration avec Simon Fau, Guy Gogniat et Caroline Fontaine

Mai 2012

digiteo

2

Agenda

- Chiffrement homomorphe ?
- Bref état de l'art.
- Le système Brakerski-Gentry-Vaikuntanathan (BGV-12).
- Que peut-on faire avec une telle primitive ?
- Premiers retours d'expériences.

3

Cryptosystèmes homomorphes ?

- **Informellement : un cryptosystème homomorphe est un cryptosystème qui permet le calcul, en plus des opérations de chiffrement et de déchiffrement.**
  - Le cryptocalculateur peut injecter des données dans le calcul.
    - En les chiffrant avec la clef publique ou en utilisant des préchiffrés de 0 et de 1 (appliquable au cas symétrique).
  - Le cryptocalculateur n'a pas accès aux résultats de calcul dans le domaine chiffré.
    - E.g. le cryptocalculateur ne peut évaluer une condition qui dépend de données du domaine chiffré.
- **Rêvons un peu :**
  - Calculs déportés (protection des données et des algorithmes), requêtes masquées sur BD (publiques ou pas), inspection de paquets à critères masqués, etc.

cea list

4

Cryptosystèmes homomorphes ?

- **Plus formellement,**
  - Par nécessité, il s'agit de cryptosystèmes probabilistes.
    - Traditionnellement asymétrique et opérant au niveau bit.
  - Spécification :
    - $enc_{pk} : \mathbb{Z}_2 \to \Omega$.
    - $dec_{sk} : \Omega \to \mathbb{Z}_2$.
    - $add_{pk} : \Omega \times \Omega \to \Omega$.
    - $mul_{pk} : \Omega \times \Omega \to \Omega$.

    où Ω est un ensemble de grande taille e.g. $\mathbb{Z}_q{}^n$.
  - Avec, pour tout $m_1 \in \mathbb{Z}_2$ et tout $m_2 \in \mathbb{Z}_2$,
    - $dec_{sk}(add_{pk}(enc_{pk}(m_1),encpk(m_2))=m_1 \oplus m_2$ (ou-exclusif).
    - $dec_{sk}(mul_{pk}(enc_{pk}(m_1),encpk(m_2))=m_1 \otimes m_2$ (et logique).
  - *Mais*, être capable de déchiffrer après une opération, n'implique pas la capacité de déchiffrer après un nombre arbitraire d'opérations, donc ce n'est pas suffisant…

cea list

5

Cryptosystèmes homomorphes ?

- **Ce que l'on veut vraiment :**
  - Pour tout polynôme $p_{\oplus;\otimes} : \mathbb{Z}_2^n \rightarrow \mathbb{Z}_2$ et tout $m_1 \in \mathbb{Z}_2$, …, $m_n \in \mathbb{Z}_2$ on doit avoir,
    - $p_{\oplus;\otimes}(m_1,\ldots,m_n)=dec_{sk}(p_{add;mul}(enc_{pk}(m_1),\ldots,enc_{pk}(m_n)))$.
- **On peut alors évaluer n'importe quel circuit booléen.**
  - I.e., n'importe quel programme à structure de contrôle statique.
    - I.e. pas de if-then-else, pas de boucle à critère d'arrêt dépendante des données chiffrées.
- **La question est donc de savoir s'il existe des cryptosystèmes homomorphes sûrs et efficaces.**
  - Sécurité : réduction à un problème de référence et choix de paramètres, $\theta(\lambda)$, relativement aux meilleures attaques connues.
  - Efficacité : overhead polynomial, $p(\lambda)$, par opération.
    - Efficacité (pratique) : polynôme de petit degré, voire polylog.

6

Historique de la question

- **1978 : Rivest et al. pose la question de l'existence de tels systèmes.**
  - Partant du constat de l'homomorphisme du RSA pour la multiplication.
- **1978-2009 (cf. Fontaine et Galand 2007).**
  - Nombreuses tentatives infructueuses.
  - Nombreuses réponses partielles.
    - E.g. systèmes homomorphe pour l'addition ou la multiplication (RSA, GM, Paillier,…).
      - Σ Suffisant par ex. pour tous les opérateurs linéaires en traitement du signal (convolution, DFT, DWT, etc.).
- **2009 : breakthrough théorique (Gentry, STOC'09).**
  - Première construction d'un système à sécurité sémantique et à overhead polynomial (donc théoriquement efficace) fondé sur la théorie des réseaux.
  - Système totalement inefficace en pratique car les polynômes sont de degré trop important… Mais la boîte de Pandore est ouverte.
- **2009-to present (début 2012) :**
  - Nouvelles constructions à un rythme de 2 à 3 par an.
    - Fondements mathématiques de plus en plus simples, diminution de l'overhead théorique.
  - État de l'art : Brakerski (2011), système sans bootstrapping, Gentry (2011), système à overhead polylogarithmique.

7

Exemple illustratif

- **van Dijk et al. (2010), version symétrique :**
  - Clef :
    - Un entier impair $p \in [2^{\eta-1}, 2^{\eta}[$, aléatoirement choisi.
  - Chiffrement de $m \in \{0,1\}$:
    - Choisir aléatoirement q et r $(2r<p/2)$ et poser $c: = qp+2r+m$.
  - Déchiffrement :
    - $m := (c \bmod p) \bmod 2$.
- **Sécurité de la construction :**
  - Problème de référence (approximate GCD) :
    - Étant donné un ensemble d'entier $x_1,\dots,x_n$ choisis aléatoirement proche de multiples d'un entier p, trouver p.
  - Fait : si le problème ci-dessus est difficile, alors cette construction possède la propriété de sécurité sémantique.

8

Exemple illustratif (cont'd)

- **Propriétés homomorphiques :**
  - $m_1+m_2=D(E(m_1)+E(m_2))$.
    - $(((q_1+q_2)p+2(r_1+r_2)+m_1+m_2) \bmod p) \bmod 2=m1\oplus m2$.
  - $m_1m_2=D(E(m_1)E(m_2))$.
    - $(((q_1+2r_1+m_1)(q_2+2r_2+m_2)) \bmod p) \bmod 2 = ((q'p+2r'+m_1m_2) \bmod p) \bmod 2=m_1m_2$.
- ***Problème :*** **ces propriétés ne sont stables que jusqu'à un certain point*.***
  - Si r_1 et r_2 sont respectivement sur N_1 et N_2 bits alors r_1+r_2 (resp. r') est sur $\max(N_1,N_2)+1$ (resp N_1+N_2) bits.
  - Tant que le bruit, r, en sortie d'une opération est <p on peut en déchiffrer le résultat sinon on est en overflow.
    - C'est là que les « ennuis commencent » et qu'il faut avoir recours à des techniques algorithmiques avancées et coûteuses, telles le bootstrapping.
    - Les systèmes les plus récents permettent de maîtriser ce bruit de manière beaucoup plus efficace.

9

Deux principales approches de construction

- **Bootstraping.**
  - Intuitivement : utiliser un système homomorphe instable pour évaluer homomorphiquement
    $c=c_1+$ (ou x) c_2; $c'=enc_{pk}(dec_{sk}(c))$
    et donc débruiter au gré du calcul.
  - Problèmes : techniques algorithmiques complexes et overhead par opération important.
    - Sans compter l'hypothèse de circularité : chiffrer la clef secrète avec la clef publiques ne doit pas poser de problème.
  - À l'instant t, on ne connait pas de système à base de bootstraping dont l'efficacité pratique ne soit pas prohibitive.
    - Exception faite de CHIMERIC ? (cf. Gentry & Halevi, 11).
- **Systèmes banqués.**
  - Pas de bootstraping.
  - Les calculs se font sur un « banc » de cryptosystèmes.
    - Ces systèmes gèrent le bruit de manière bcp plus efficace mais tous les problèmes ne sont pas résolus pour autant (cf. ci-après).

10

Rappels sur les circuits booléens

- **Définition : un graphe orienté G=(V,A) dont les sommets sont des entrées, des sorties ou des opérateurs (XOR, AND) et les arcs des « transferts » de données.**
- **Caractéristiques importantes :**
  - Profondeur : nombre d'arcs du plus long chemin séparant un sommet d'entrée (sans prédécesseurs) d'un sommet de sortie (sans successeurs).
  - Profondeur multiplicative : plus grand nombre d'opérateurs AND intervenant dans le calcul d'une sortie.
  - Ordre topologique : $f : V \rightarrow \{1,\dots,|V|\}$ t. q. pour tout arc (v,w), $f(v)<f(w)$. Défini un ordre dans lequel on peut évaluer les opérateurs.
  - Classe topologique : les opérateurs de même profondeur, peuvent être réalisés en parallèle.

BGV-12

- **Cf. Z. Brakerski, G. Gentry, V. Vaikuntanathan, « (Leveled) fully homomorphic encryption without bootstraping », ITCS'12, pp. 309-325.**
- **Premier candidat à overhead non prohibitif ($O(\lambda L^3)$) par opération).**
- **Cryptosystème homomorphe « banqué ».**
  - On travaille sur une séquence de cryptosystèmes (des grands vers les petits modules).
    - Les additions (XOR) sont réalisables au sein d'un même niveau.
    - Les multiplication (AND) font changer de niveau.
  - Le paramètre clef est donc la profondeur multiplicative.

Principe des systèmes banqués

- **Addition :**
  - Soit deux cryptobits c_1 et c_2 de niveau n_1 et n_2.
  - Step 1 : mise à niveau des cryptobits à $\max(n_1,n_2)$.
  - Step 2 : opérateur d'addition.
- **Multiplication :**
  - Soit deux cryptobit c_1 et c_2 de niveau n_1 et n_2.
  - Step 1 : mise à niveau des cryptobits à $\max(n_1,n_2)$.
  - Step 2 : opérateur de multiplication.
    - Donne une entité que l'on peut mettre à niveau mais qui n'est pas déchiffrable en tant que tel (cf. plus loin).
  - Step 3 : mise au niveau $1+\max(n_1,n_2)$.
    - Et on obtient bien un cryptobit déchiffrable à ce niveau.

13

BGV-12 : chiffrement/déchiffrement (variante vectorielle)

- **Paramètres :**
  - μ, n, N, entiers.
  - q un module sur μ bits.
  - χ : une loi de probabilité sur $\mathbb{Z}_q$.
- **Clef secrète :**
  - Poser $s'=(s'_0 \ldots s'_{n-1})$ où les s'_i sont tirés selon χ.
  - $sk=s=(1\ s'_0 \ldots s'_{n-1})$.
- **Clef publique :**
  - Tirer uniformément une matrice Nxn, A', dans $\mathbb{Z}_q$.
  - Tirer un vecteur de dimension N, e, selon χ.
  - Poser $pk=A=(A's'+2e \mid -A')$.
- **Chiffrement ($m \in \mathbb{Z}_2$) :**
  - Poser $m'=(m\ 0 \ldots 0) \in \mathbb{Z}_2^{n+1}$, tirer r uniformément dans $\mathbb{Z}_2^N$.
  - $c=m'+A^T r \in \mathbb{Z}_q^{n+1}$.
- **Déchiffrement ($c \in \mathbb{Z}_q^{n+1}$) :**
  - $m=[[c^T s]_q]_2$.

cea list

14

BGV-12 : définition du banc

- **Paramètres :**
  - μ, n, N, entiers.
  - On construit un banc de L+1 systèmes tels que précédent.
    - Modules de q_L (sur (L+1)μ bits) à q_0 (sur μ bits).
    - La loi de probabilité χ ne varie pas forcément.
- **Changement de clef (switchKey) :**
  - Pour $s_1 \in Z_q^{n1}$ et $s_2 \in Z_q^{n2}$, on construit un opérateur de changement de clef t. q. à partir d'un chiffré c_1 (déchiffrable avec s_1 et q) on obtient un chiffré c_2 (déchiffrable avec s_2 et q).
    - En pratique, on calcule une matrice B ($n_1$log(q)xn_2) t. q. BitDecomp$(c_1)^T B=c_2$.
- **Changement de module (rescale) :**
  - Permet de passer d'un chiffré c_2 déchiffrable avec s_2 et q_1 à un chiffré déchiffrable avec s_2 et $q_2<q_1$.
- **Dès lors on peut convertir des chiffrés de niveau i en chiffrés de niveau i-1 (refresh).**
  - Auquel cas il faut « réinterpréter » un chiffré de niveau i comme étant chiffré avec la clef $s_i \otimes s_i$.

cea list

15

BGV-12 : porte XOR

- **Entrée : deux chiffrés de même niveau, i.**
  - Refresh permet de mettre à niveau l'un des deux chiffrés si nécessaire.
- **Sortie : un chiffrée de niveau, i.**
- **Opération : une simple sommation terme à terme modulo q_i.**
- **Refresh possible pour passer au niveau suivant mais non nécessaire.**
  - Auquel cas (déjà dit) il faut « réinterpréter » le résultat comme étant chiffré avec la clef $s_i \otimes s_i$.
  - En pratique, le bruit induit par des additions successives est faible donc on fait l'économie de cette mise à niveau.

16

BGV-12 : porte ET

- **Entrée : deux chiffrés c_1 et c_2 de même niveau, i.**
  - Refresh permet de mettre à niveau l'un des deux chiffrés si nécessaire.
- **Sortie : un chiffrée de niveau, i-1.**
- **Opération :**
  - Produit tensoriel des deux chiffrés.
    - « Replié » par rapport à la 1ère diagonale.
    - Donne un chiffré déchiffrable avec $s_i \otimes s_i$.
  - Refresh du niveau i vers le niveau i-1 pour récupérer un chiffré de niveau i-1.
- **Donc opération plus coûteuse, qui nécessite un appel systématique à refresh et qui fait augmenter le nombre de niveaux du système.**

Implémentations

- **De nombreuses implémentations non publiques en cours.**
  - Notamment Gentry et al. sur l'AES (cf. plus loin).
- **Une implémentation open source (par Perl et al.) du système de Smart & Vercauten (à base de bootstraping).**
  - www.hcrypt.com.
- **Implémentation CEA LIST (S. Fau, R. Sirdey).**
  - Variantes vectorielle et polynomiale de BGV-12.
  - Support d'exécution littérale d'algorithmes haut-niveau.
    - Avec parallélisme « interne » au cryptosystème.
  - Compilation en circuits booléens et exécution niveau circuit.
    - Avec parallélisme « interne » ou « externe » au cryptosystème selon capacités du calculateur cible.

Calculer dans le domaine des chiffrés

- **Il est facile de définir un type entier doté d'opérateurs réalisables hermétiquement dans le domaine chiffré :**
  - Additionneurs et multiplieurs classiques x-bits.
    - Avec optimisation pour les cas avec booléens.
  - Multiplieur par un entier public.
  - Négation et soustraction (par complément à 2).
  - Décalages vers la gauche (par injection de chiffrés de 0) et vers la droite (par recopie du « cbit » de signe).
  - Comparaisons <, >, etc.
- **Et le problème de la structure de contrôle statique ?**
  - On régularise le contrôle, par exemple en utilisant un opérateur d'affectation conditionnelle :
    - x=c?a:b revient à x=c*a XOR (!c)*b.
      - Σ Le calculateur peut savoir si une variable est booléenne (le résultat de l'évaluation d'un opérateur booléen l'est toujours), mais il ne peut pas avoir accès à sa valeur.

Exemple (didactique) : tri à bulle

- **Principe :**
  - Un code unique paramétré par un type entier permettant :
    - L'exécution en clair (bit à bit ou pas) pour la mise au point.
    - L'exécution symbolique pour la caractérisation.
      - Σ Profondeur multiplicative, etc.
    - L'exécution littérale dans le domaine chiffré.
    - La génération de données de compilation.
      - Σ Graphe du circuit.
- **Exécution littérale :**
  - Type entier paramétrisé par un type bit (clair ou chiffré) supportant les opérateurs :
    - +, - (unaire et binaire), *, <<, >>, <, >, !, etc.
      - Σ Tous étant réalisables hermétiquement dans le domaine chiffré.

```
template<typename integer>
void bsort(integer * const arr,
  const int n)
{
 assert(n>0);

 for(int i=0;i<n-1;i++)
 {
  for(int j=1;j<n-i;j++)
  {
   integer swap=arr[j-1]>arr[j];
   integer t=select(swap,arr[j-1],arr[j]);
   arr[j-1]=select(swap,arr[j],arr[j-1]);
   arr[j]=t;
  }
 }
}
```

Où la fonction select régularise le contrôle dépendant des données (select(c,a,b)≡c?a:b).

Tableaux : déréférencement et affectation

- **Indices en clair.**
  - Trivial.
- **Indices chiffrés :**
  - Déréférencement : $t[i] \equiv \sum_{j=1}^{n} \chi(i,j) \times t[j]$

    avec χ(i,j)=1 si i=j, 0 sinon.
    - I.e. c'est juste l'opérateur ==.
  - Affectation (t[i]:=v) :

    $$t[j] := \chi(i,j) \times v \oplus (1-\chi(i,j)) \times t[j], \forall j$$

    pour j=1 à n.
  - Donc le déréférencement et l'affectation sont en O(n) (sic !).
    - Mais c'est le prix à payer pour ne rien révéler sur les indices.

21

Un peu de génie logiciel…

- **Support de calcul bit à bit.**
  - Schématiquement une classe C++

    template<typename bit,int size> class Integer,

    dotée des opérateurs usuels +, -, *, <<, >>, <, >, etc., sert de base à l'expression haut niveau (mais paramétrique) des algorithmes.
    - template<typename integer> void bsort(integer * arr,int n)
  - En phase d'analyse et de mise au point l'algorithme est alors instancié sur des « bits clairs ».
    - E.g., bsort<Integer<ClearBit,4> >(a,n);
  - En phase de déploiement, sur des « crypto bits ».
    - E.g., bsort<Integer<CryptoBit,4> >(a,n);
- **Indépendant du cryptosystème sous-jacent.**

22

… Et de parallélisme

- **switchKey est le point chaud :**
  - Techniques de caching pour éviter de répéter les mises à niveau déjà faites.
    - I.e., chaque cbit se rappelle de ses mises à niveau dans une structure de données associative.
  - Parallel for (OpenMP) dans l'outer loop du produit matriciel.
    - Meilleure stratégie pour les petits multicoeurs.
  - Et la possibilité d'utiliser une seule clef secrète !
    - Cf. Gentry et al., 12 (eprint.iacr.org/2012/099).
    - Non encore supporté par nos implémentations.
- **Sur des calculateurs parallèles plus massifs :**
  - Chercher le parallélisme au niveau des classes d'équivalence du circuit booléen.
    - Externe au cryptosystème donc.
  - Bien entendu on peut l'associer avec du parallélisme interne.

Caractérisation des algorithmes

- **Pour les programmes à structure de contrôle statique, analyse statique et dynamique coïncident.**
  - Conséquence : pour caractériser un programme il suffit de l'exécuter avec un support de calcul bit à bit sur n'importe quelles entrées.
  - Nécessité pour les systèmes banqués :
    - Profondeur multiplicative pour le dimensionnement du cryptosystème.
  - Compilation :
    - Construction du circuit booléen (pas forcément nécessaire mais c'est une bonne abstraction pour faire de la compilation).
    - Problème à adresser : organiser les calculs de manière à minimiser le nombre de changement de clefs (i.e., de manière à maximiser l'efficacité du cache de profondeur).

Caractérisation de quelques algorithmes

	b^2-4ac (8 bits)	b^2-4ac (16 bits)	$\sum_{i=1}^{10} t[i]$ (8 bits)
# add	332	1188	207
# mul	302	1126	135
depth	43	83	24
× depth	16	32	8
Av. //	14.74	27.88	6.75

	$\sum_{i=1}^{10} t[i]$ (16 bits)	b. sort (10×4 bits)	b. sort (10×8 bits)
# add	423	1620	3240
# mul	279	1350	2790
depth	48	214	350
× depth	16	68	136
Av. //	14.62	13.88	17.23

	FFT (256×32 bits)
# add	7291592
# mul	5296128
depth	674
× depth	166
Av. //	18676.10

Quelques temps d'exécution

- **Variante vectorielle de BGV-12.**
  - Sur un Intel dual core à 2GHz, jeu de petits paramètres.
  - Cache de profondeur pour éviter les mises à niveau redondantes.
  - Parallel for (OpenMP) dans l'outer loop du produit matriciel de switch key (speedup ~ 40%).

	$b^2 - 4ac$ (8 bits)	$b^2 - 4ac$ (16 bits)	$\sum_{i=1}^{10} t[i]$ (8 bits)
Temps	0.406 s	4.124 s	0.125 s
Eff. cache	46%	40%	47%
Taille PK	1.1 Mo	7.8 Mo	196 ko

	$\sum_{i=1}^{10} t[i]$ (16 bits)	b. sort (10×4 bits)	b. sort (10×8 bits)
Temps	0.562 s	5.219 s	18.110 s
Eff. cache	47%	64%	64%
Taille PK	1.1 Mo	68.5 Mo	525 Mo

D'autres temps d'exécution (HCRYPT)

- **Implémentation par Brenner et al. du système de Smart & Vercauten.**
  - Sur un Intel dual core à 2 GHz (1 seul cœur mobilisé).
  - Paramètres par défaut de la librairie.
    - Cf. www.hcrypt.com.

	$b^2 - 4ac$ (8 bits)	$b^2 - 4ac$ (16 bits)	$\sum_{i=1}^{10} t[i]$ (8 bits)
Temps	58.9 s	3 m 39 s	27.2 s

	$\sum_{i=1}^{10} t[i]$ (16 bits)	b. sort (10×4 bits)	b. sort (10×8 bits)
Temps	55.4	5 m 5 s	9 m 41 s

  - Rappelons qu'en théorie, le système de Smart & Vercauten est (bcp) moins attractif que BGV-12.

27

Conclusions et perspectives

- **Avancées théoriques.**
  - À l'instant t de cet exposé, l'état de l'art est instable : stay tuned !
    - E.g. Z. Brakersky, « Fully homomorphic encryption without modulus switching from classical GapSVP » (eprint.iacr.org/2012/078).
- **Problèmes des paramètres.**
  - Les systèmes proposés repose sur des hypothèses standards.
    - E.g. xLWE pour BGV-12.
  - Par contre, la fixation concrètes des paramètres pour résister aux meilleures attaques connues est à approfondir.
    - Pour l'instant on se contente de petites valeurs.

28

Conclusions et perspectives (cont'd)

- **Le cryptocalcul à base de systèmes homomorphes devrait permettre d'exécuter à relativement court terme des algorithmes simples de petite profondeur multiplicative.**
  - Par ex. des règles de décision simples en publicité ciblée, des opérateur simple en traitement du signal (y compris non linéaire), etc.
- **Les algorithmes plus volumineux sont hors de portée.**
  - E.g. ~4 h pour effectuer une FFT 256x32 sur un 48 cœurs (avec parallélisme externe).
- **Un autre axe prioritaire concerne les algorithmes de chiffrement symétriques.**
  - Pour se débarrasser de l'overhead données du aux systèmes proba.
    - E.g., on chiffre en homomorphe une clef de 128 bits, on chiffre « classiquement » les données, on déchiffre en homomorphe.
  - Mais, 1 semaine pour une exécution d'AES (Gentry et al., 12, ibid.).
    - Avec un jeu de paramètres plus réaliste.
  - Problème des calculs récurrents avec les systèmes banqués.
    - Car leur profondeur multiplicative n'est pas (pratiquement) bornée.
- **Les travaux en compilation, en optimisation, en « homomorphic friendlyness » des algorithmes ne font que commencer…**