(Teil)Homomorphe Verschlüsselung

S.Seidl, M.Nening, T.Niederleuthner

Inhalt

Inhalt

Motivation & Ziele

- Primär sicheres **sharing** von Daten
- Berechnungen $\rightarrow$ Entschlüsselung $\rightarrow$ angreifbar
- **Lösung:** Operationen auf verschlüsselten Daten ermöglichen

- **Vorteil:** Berechnungen auslagern *(3.Partei)*

Homomorphismus

- **Informell:** Abbildung von einer Menge in eine zweite, sodass Relation zwischen den Elementen erhalten bleibt
- Problem in algebraischem System

Geschichte - Idee

- Idee schon lange bekannt
- Vorgeschlagen von Rivset, Adleman und Dertouzos *(1978)*
- 2 Agenda
  - sicheres Schema
  - interessante Operationen
- es folgten einige teilhomomorphe Schemen *(+,*)*
- beides schwer
- man hielt vollhomomorphes Schema für möglich

Geschichte - Idee

- ab 1991 wieder mehr Fokus auf vollhomomorphes Schema
- **doppelt homomorph:**
  - "erster Schritt"= "letzter Schritt"
  - bitwise - 1+A*B (NAND)
- Craig Gentry: erstes Vollhomomorphes Schema *2009*
- **Aber:** extrem hoher Rechenaufwand

"Eine dieser Boxen mit Handschuhen, die bei der Arbeit mit toxischen Chemikalien verwendet werden...Alle Manipulationen geschehen in der Box und die Chemikalien werden nie der Außenwelt ausgesetzt"

Inhalt

Gruppenhomomorphismus

Definition (Gruppenhomomorphismus (G,$*_G$), (H,$*_H$))

Eine Funktion $f : G \to H$ ist ein Gruppenhomomorphismus, wenn die Gruppenoperation wie folgt erhalten bleibt:

$$f(g_1 *_G g_2) = f(g_1) *_H f(g_2)$$

Sein e_G, e_H die neutralen Elemente:

$$f(e_G) = e_H$$

Auch die inverse Abbildung muss erhalten bleiben:

$$f(g^{-1}) = f(g)^{-1}$$

Inhalt

Teilhomomorphe Verschlüsselung

Teilhomomorphe Verschlüsselungen unterstützen entweder Addition oder Multiplikation auf Ciphertexte.

Additive Verfahren: $\exists \psi : m(x_1)\, \psi\, m(x_2) = m(x_1 + x_2)$

- Paillier

Multiplikative Verfahren: $\exists \psi : m(x_1)\, \psi\, m(x_2) = m(x_1 * x_2)$

- RSA
- ElGamal

Vollhomomorphe Verschlüsselung

- Eine Verschlüsselung, die beliebige Operationen auf den Ciphertext erlaubt, nennt man Vollhomomorphe Verschlüsselung.

- Erstes System von Craig Gentry 2009
- Brakerski-Gentry-Vaikuntanathan cryptosystem BGV 2011-2012

Probleme

- sehr aufwendig und kompliziert
- wesentlich höhere Rechenleistung notwendig
  Erstes System benötigte 30 Minuten für eine Bitoperation.
- Viele Systeme haben eine beschränkte Anzahl an Operationen, die ausgeführt werden können, oder sind durch Konstruktion beschränkt.

Inhalt

Cloud Computing - Gründe

- Outsourcing
- Lastspitzen abdecken
- Flexibilität
- Kosten

Aber: Ob es wirklich einsetzbar ist, steht und fällt mit der Vertrauenswürdigkeit des Cloud-Anbieters!

Probleme

- Vertrauenswürdigkeit des Anbieters
- Rechtssprechung des Serverstandorts
- Angriffe gegen den Anbieter
- (evtl Outsourcing des Anbieters)

Lösung: zu keiner Zeit unverschlüsselte Daten beim Cloud-Anbieter

$\Rightarrow$ (idealerweise voll-)homomorphe Verschlüsselung

Wahlen - Anforderungen

- einfache Durchführung für den Wähler
- jeder Wähler muss eindeutig identifizierbar sein, um Wahlberechtigung zu verifizieren
- jeder Wähler darf nur eine Stimme abgeben
- sichere und anonyme Stimmabgabe
- korrekte Auszählung

Vorgehen

- Wähler verschlüsselt seine Stimme
- Stimme wird verschlüsselt übertragen
- durch (additiv) homomorphe Verschlüsselung wird das Wahlergebnis berechnet

$\rightarrow$ mehr dazu später

Inhalt

non-padded RSA

- Verschlüsselung mit ε :

$$\varepsilon(m_1) = m_1^e \pmod{n}$$
$$\varepsilon(m_2) = m_2^e \pmod{n}$$

- **multiplikative** homomorphe Eigenschaft:

$$\begin{aligned} \varepsilon(m_1 * m_2) &= (m_1 * m_2)^e && \pmod{n} \\ &= m_1^e * m_2^e && \pmod{n} \\ &= \varepsilon(m_1) * \varepsilon(m_2) && \pmod{n} \end{aligned}$$

Unterschied padded - non-padded RSA

- Padding wird dazu eingesetzt, um das Ergebnis zu randomisieren.
- Damit wird erreicht, dass bei zweimaligem Verschlüsseln derselben Nachricht zwei unterschiedliche Ciphertexte erzeugt werden.
- Padding zerstört die homomorphe Eigenschaft von RSA.

Sicherheit von non-padded RSA (1)

Deterministisch

- 2 Gleiche Plaintexte ergeben 2 gleiche Ciphertexte.
- bei kurzem Plaintext oder wenn die möglichen Plaintexte bekannt sind, ist es möglich, alle unterschiedlichen Ciphertexte zu berechnen und diese mit dem abgefangenen Ciphertext zu vergleichen.

Sicherheit von non-padded RSA (2)

Verändern der Nachricht

- Angreifer fängt Ciphertext c ab und berechnet $c' = c * 2^e \mod n$
- Entschlüsseln: $2m = c'^d \mod n$
  Somit wurde die Nachricht im verschlüsselten Zustand verändert

RSA Beispiel(1)

- $p = 5; q = 11$
- $N = p * q = 55$
- $\varphi(N) = (p-1) * (q-1) = 4 * 10 = 40$
- $e = 7$
- $7 * d + 40k = 1 = ggT(7, 40) \rightarrow d = 23$

RSA Beispiel(2)

Verschlüsseln von $m_1 = 4$ und $m_2 = 6$

- $c_1 \equiv 4^7 \mod 55 \rightarrow c_1 = 49$
- $c_2 \equiv 6^7 \mod 55 \rightarrow c_2 = 41$

Berechnen von $c_1 * c_2$

- $\varepsilon(m_1) * \varepsilon(m_2) = c_1 * c_2 \mod N = 49 * 41 \mod N = 29$

Entschlüsseln

- $m_3 = c_3^d \mod N = 29^{23} \mod N = 24$

Paillier - Eigenschaften

- 1999 entwickelt
- probabilistisch
- additiv homomorph
- sicher, da nicht effizient berechenbar, ob:

$$\exists y \; z \equiv_{n^2} y^n$$

Restklassen

- Restklassenring $(\mathbb{Z}/n\mathbb{Z})$
- Restklasse $[x]_n$ enthält die Zahlen, die $\quad \bmod n$ x ergeben
- prime Restklasse: wenn für $[x]_n$ gilt, dass $ggT(x, n) = 1$
- prime Restklassengruppe $(\mathbb{Z}/n\mathbb{Z})^*$
  Gruppe, zusammengesetzt aus primen Restklassen

Schlüssel generieren

- wähle 2 zufällige Primzahlen p und q
- setze $n = p \cdot q$
- wähle g zufällig, wobei $g \in (\mathbb{Z}/n^2\mathbb{Z})^*$
- berechne $\lambda = kgV(p-1, q-1)$

Der Public-Key ist Tupel (n, g), der Private-Key λ.

Verschlüsseln

- Sei m die Nachricht in Plaintext, c der Ciphertext
- wähle zufälligen Wert $r \in (\mathbb{Z}/n\mathbb{Z})^*$

$$c = (g^m \cdot r^n) \mod n^2$$

Entschlüsseln

Definiere Funktion L folgendermaßen:

$$L(x) := \frac{x-1}{n}$$

Dann ist die entschlüsselte Nachricht m:

$$m = L(c^{\lambda} \mod n^2) * \lambda^{-1} \mod n$$

Homomorphismus

Addition 2er Nachrichten $m_1 + m_2$:

$$
\begin{aligned}
\varepsilon(m_1) \cdot \varepsilon(m_2) &= && \mod n^2 \\
&= g^{m_1} \cdot r_1^n \cdot g^{m_2} \cdot r_2^n && \mod n^2 \\
&= g^{m_1} \cdot g^{m_2} \cdot r_1^n \cdot r_2^n && \mod n^2 \\
&= g^{m_1+m_2} \cdot (r_1 \cdot r_2)^n && \mod n^2 \\
&= \varepsilon(m_1 + m_2) && \mod n^2
\end{aligned}
$$

Homomorphismus

Addition Nachricht m + Plaintext k

$$\begin{aligned}
\varepsilon(m) \cdot g^k &= && \mod n^2 \\
&= g^m \cdot r^n \cdot g^k && \mod n^2 \\
&= g^{(m+k)} \cdot r^n && \mod n^2 \\
&= \varepsilon(m+k) && \mod n^2
\end{aligned}$$

Homomorphismus

Multiplikation Nachricht m · Plaintext k

$$
\begin{aligned}
\varepsilon(m)^k &= && \bmod n^2 \\
&= (g^m \cdot r^n)^k && \bmod n^2 \\
&= (g^m)^k \cdot (r^n)^k && \bmod n^2 \\
&= g^{km} \cdot (r^k)^n && \bmod n^2 \\
&= \varepsilon(k \cdot m) && \bmod n^2
\end{aligned}
$$

Beispiel Wahlvorgang - Vorbereitung

Wahlkommission kennt/setzt folgende Werte:

- Abstimmung Ja= 10, Nein= 01
- Wähler V_1, V_2, V_3
- p=17, q=19
- $g = 324$
- Public-Key $(323, 324)$
- Private-Key $(\lambda = 288)$
- $\lambda^{-1} = 203$

Wählersicht

Wähler	Gewähltes	r_i	c_i
V_1	10	3	33.092
V_2	01	8	57.734
V_3	10	2	84.617

"Gläserne Wahlurne"
33.092
57.734
84.617

Ergebnis

Es wird ausgezählt, also das Produkt der c_i gebildet.

$$\Rightarrow c = 29.927$$

Die Wahlkommission kennt den Private-Key, berechnet das Wahlergebnis:

$$m = L(29.927^{288} \mod 323^2) * 203 \mod 323 =$$

$$= \frac{75.582}{323} * 203 \mod 323 =$$

$$= 21$$

Inhalt

GMP

- GNU Multiple Precision Library
- Augenmerk auf Schnelligkeit
- `#include <gmp.h>`
- mit `libgmp` library linken (-lgmp)

Typen & Funktionen

Integers:	`mpz_t`	*(ca. 150 Funktionen)*
Rationals:	`mpq_t`	*(ca. 35 Funktionen)*
Floats:	`mpf_t`	*(ca. 70 Funktionen)*
Random state:	`gmp_randstate_t`	

- low-level Funktionen (von Obigen verwendet)
- GMP kümmert sich um Speicherverwaltung

Initialisieren & Zuweisen

Initialisieren und freigeben

- *void* **mpz_inti[s]** *(mpz_t x [...])*
- *void* **mpz_clear[s]** *(mpz_t x [...])*

- *void* **mpz_set_[ui/si/d/str...]** *(mpz_t rop,...)*
- *void* **mpz_init_set_[ui/si/d/str...]** *(mpz_t rop,...)*

Konvertieren

- *...* **mpz_get_[ui/si/d/str...]** *(const mpz_t op,...)*

Arithmetische Funktionen

- *void* **mpz_add[_ui]** *(mpz_t rop, const mpz_op,...)*
- *void* **mpz_sub[_ui]** *(mpz_t rop, const mpz_op,...)*
- *void* **mpz_mul[_ui/si]** *(mpz_t rop, const mpz_op,...)*

- *...* **mpz_mod[_ui]** *(mpz_t rop, const mpz_op,...)*
- *void* **mpz_powm[_ui]** *mpz_t rop,..., mpz_t mod)*
- *int* **mpz_congruent_p** *(mpz_t rop, ...)*

Weitere Funktionen

- *int* **mpz_probab_prime_p** *(mpz_t n, int reps)*
- *void* **mpz_nextprime** *(mpz_t rop, const mpz_op)*
- *...* **mpz_gcd[_ui]** *(mpz_t rop, const mpz_op1,...)*
- *void* **mpz_invert** *(mpz_t rop, const mpz_op1,...)*

- *int* **mpz_comp_[d/si,ui]** *(const mpz_t op1, ...)*
- auch bitwise Funktionen

Weitere Funktionen

File I/O

- *size_t* **mpz_out_str** *(FILE*stream, int base,...)*
- *size_t* **mpz_inp_str** *(mpz_t rop, FILE*stream,...)*

Zufallszahlen

- *void* **mpz_urandomb** *(mpz_t rop,...)*
- *void* **gmp_randseed[_ui]** *(gmp_randstate_t,...)*

krypto.h

```
1  // ------------------------------ RSA --------------------------------
2  typedef struct {
3      mpz_t n;        //modulo
4      mpz_t e;        //exponent
5  } publicKeyRSA;
6
7  typedef struct {
8      mpz_t n;        //modulo
9      mpz_t d;        //inverse of exponent
10 } privateKeyRSA;
11
12 unsigned long long randomSeed();
13 void generateKeysRSA(publicKeyRSA* pk, privateKeyRSA* sk, long seed);
14 void storePublicKey(publicKeyRSA* pk, char* filename);
15 void readPublicKey(publicKeyRSA* pk, char* filename);
16
17 void encryptRSA(publicKeyRSA* pk, mpz_t op, mpz_t cipher);
18 void decryptRSA(privateKeyRSA* sk, mpz_t cipher, mpz_t op);
```

krypto.c

```
1  // ----------------------------- p, q -------------------------------
2      mpz_ui_pow_ui (rangeMin, 2, size-1);
3      mpz_ui_pow_ui (rangeMax, 2, size);
4
5      do {
6          do {
7              // find "good" p
8              mpz_urandomb(p, state, size);
9              mpz_nextprime (p, p);
10         } while (mpz_cmp(p, rangeMin) < 0 || mpz_cmp(p, rangeMax) > 0);
11
12         // find "good" q
13         mpz_nextprime (q, p);
14
15     } while (mpz_cmp(q, rangeMax) > 0);
```

krypto.c

```
1  // ---------------------------- n, phi(n) ----------------------------
2      mpz_mul(n, p, q);                          //n = p * q
3      mpz_sub_ui(t1, p, 1);                      //p - 1
4      mpz_sub_ui(t2, q, 1);                      //q - 1
5      mpz_mul(phin, t1, t2);                     //phi(n) = (p - 1)(q - 1)
6
7      mpz_urandomb(e, state, 128);
8      mpz_gcd(t1, phin, e);
9
10     while(mpz_cmp_ui(t1, 1) != 0) {
11         mpz_urandomb(e, state, 128);
12         mpz_gcd(t1, phin, e);
13     };
14
15     mpz_invert(d, e, phin);
16
17     mpz_set(pk -> n, n);
18     mpz_set(pk -> e, e);
19
20     mpz_set(sk -> n, n);
21     mpz_set(sk -> d, d);
22
```

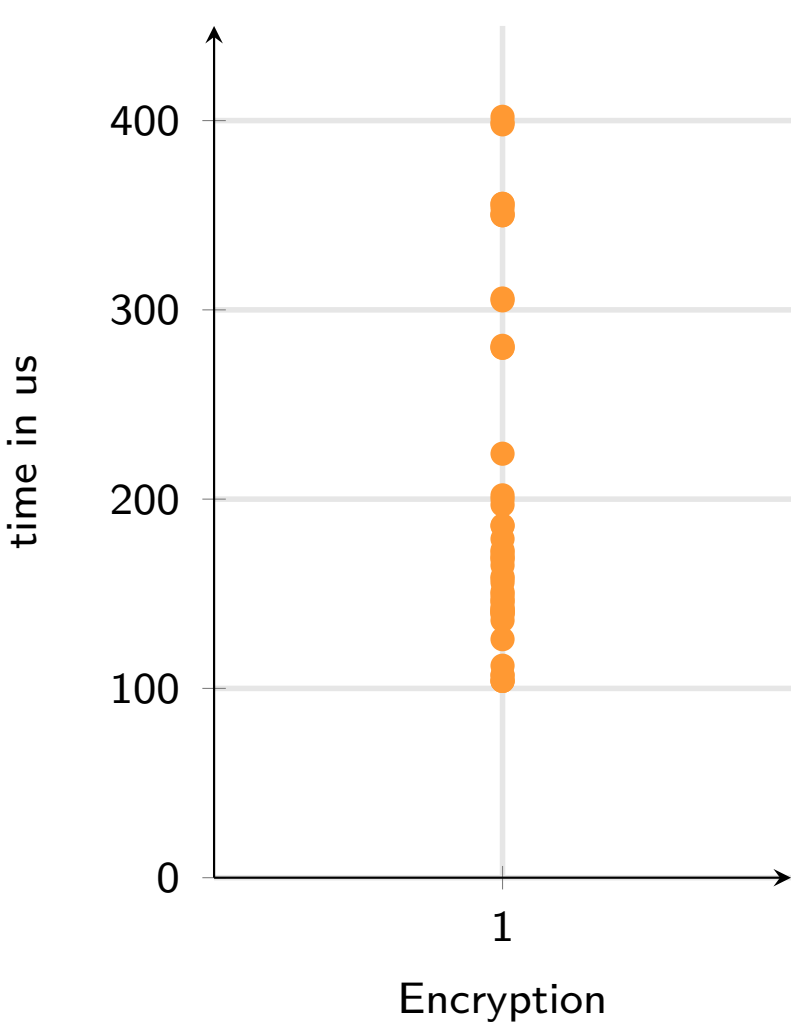
krypto.c

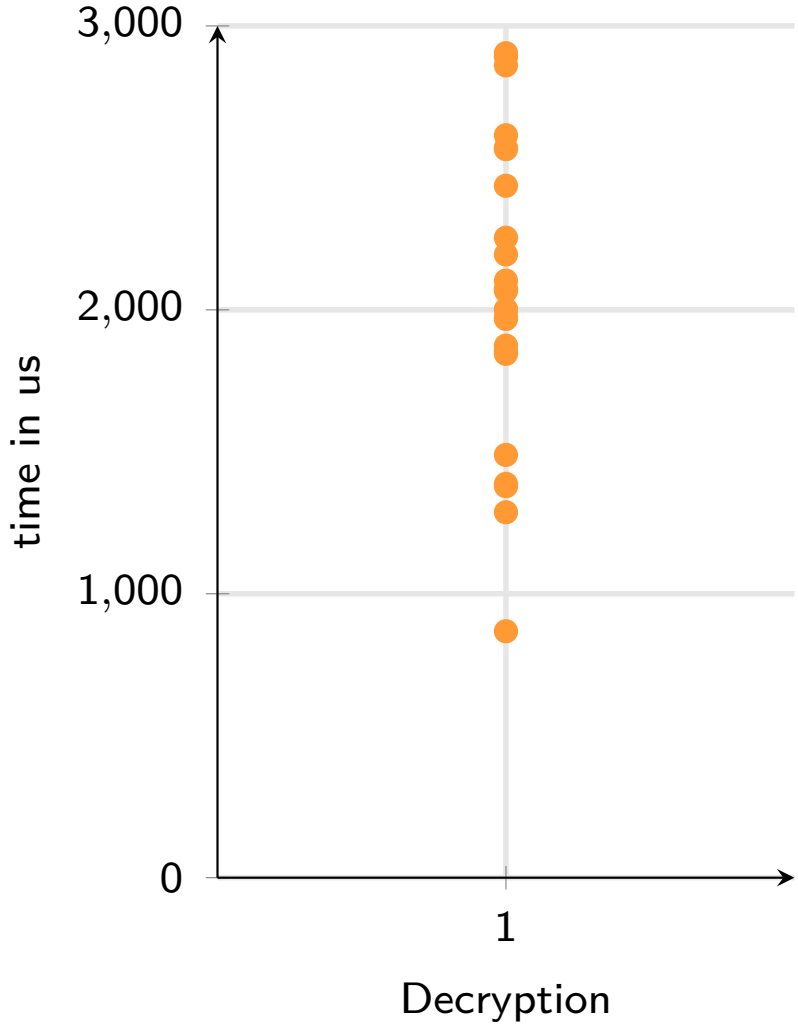
```
1   void encryptRSA(publicKeyRSA* pk, mpz_t op, mpz_t cipher){
2
3      mpz_powm(cipher, op, pk -> e, pk -> n);
4   }
5
6   void decryptRSA(privateKeyRSA* sk, mpz_t cipher, mpz_t op) {
7
8      mpz_powm(op, cipher, sk -> d, sk -> n);
9   }
10
11  unsigned long long randomSeed() {
12    unsigned long long random;
13    FILE* file;
14
15    file = fopen("/dev/urandom", "r");
16    fread(&random, sizeof(random), 1, file);
17    fclose(file);
18
19    return random;
20  }
```

smartCalc.c

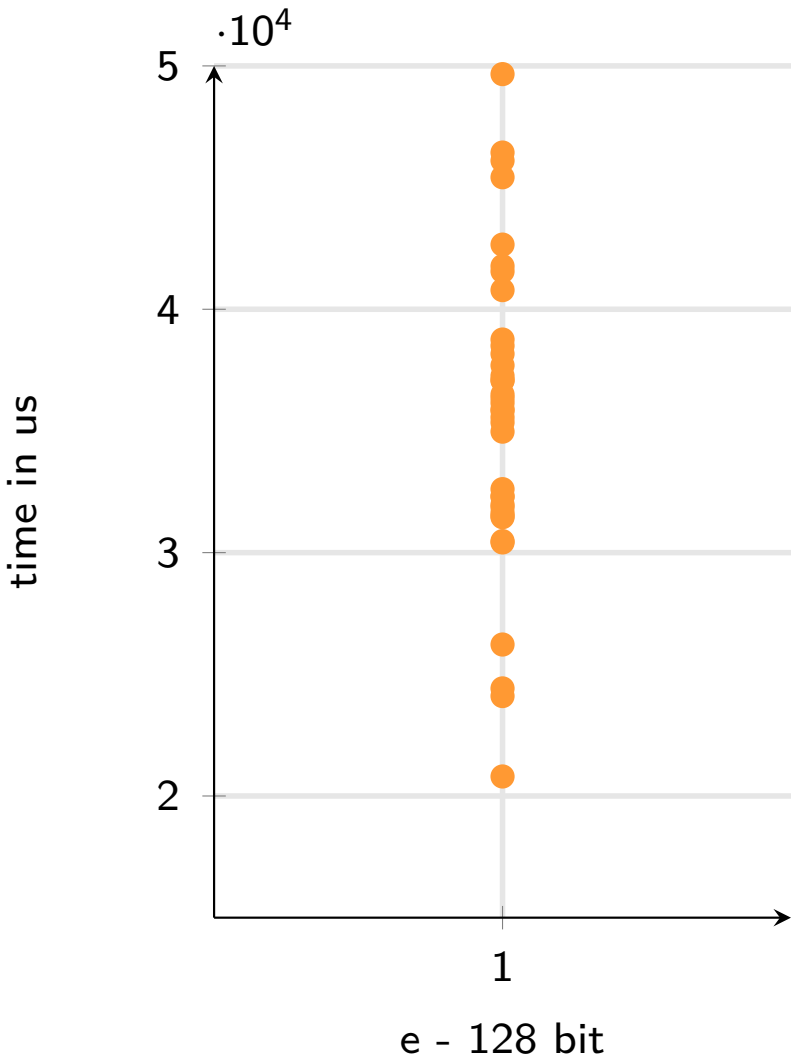
```
1   printf("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
2   printf("            Smart Caluclator\n");
3   printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n\n");
4   receiveString(buffer, address, &mode);
5   printf("Operator: %s\n\n", buffer);
6
7   if (strcmp(buffer, "*") == 0) {
8       receiveString(buffer, address, &mode);
9
10      while (strcmp(buffer, "=") != 0) {
11
12          mpz_set_str (cipher, buffer, BASE);
13          mpz_mul(result, result, cipher);
14          if (mode == 'r')
15              mpz_mod (result, result, pk -> n);
16       ...
17          receiveString(buffer, address, &mode);
18      }
19  }
20  mpz_get_str(buffer, BASE, result);
21  transmitString(buffer, address, mode);
```

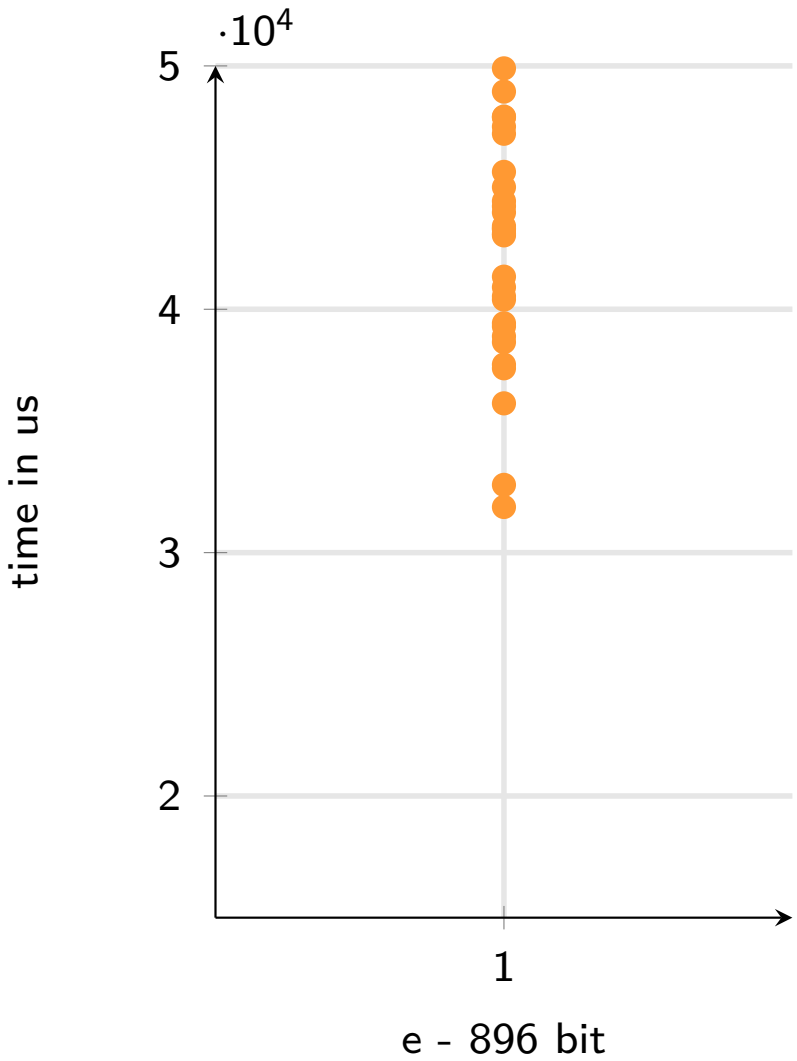
Zeitmessungen Encryption/Decryption



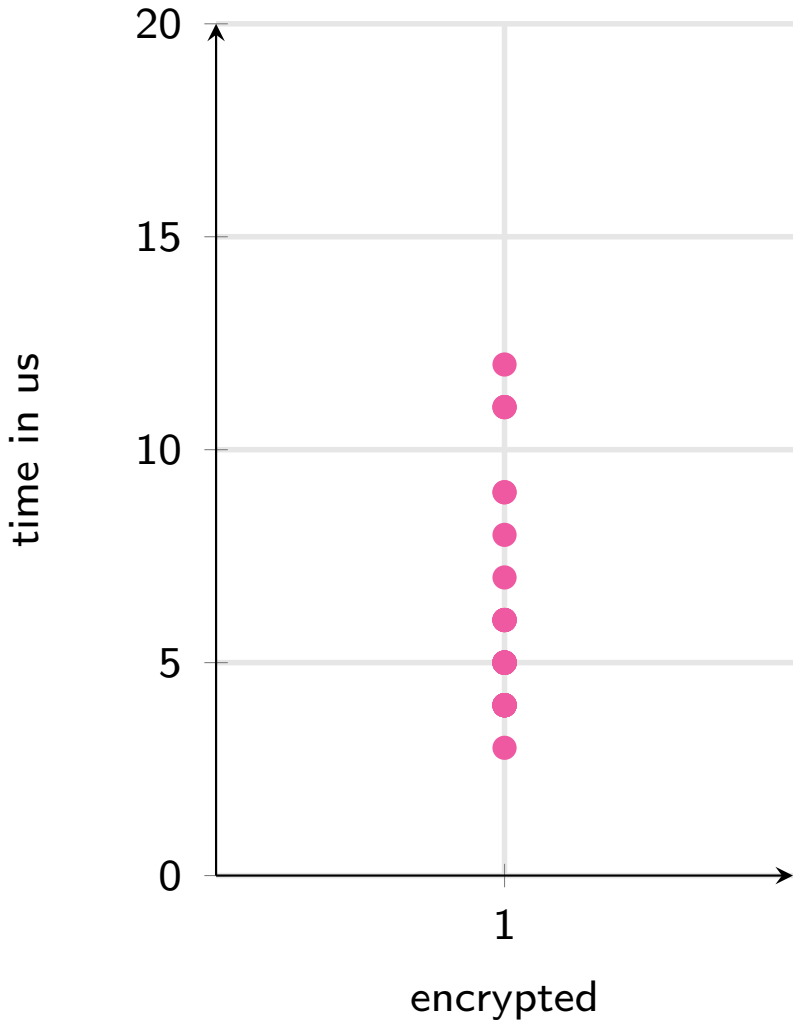


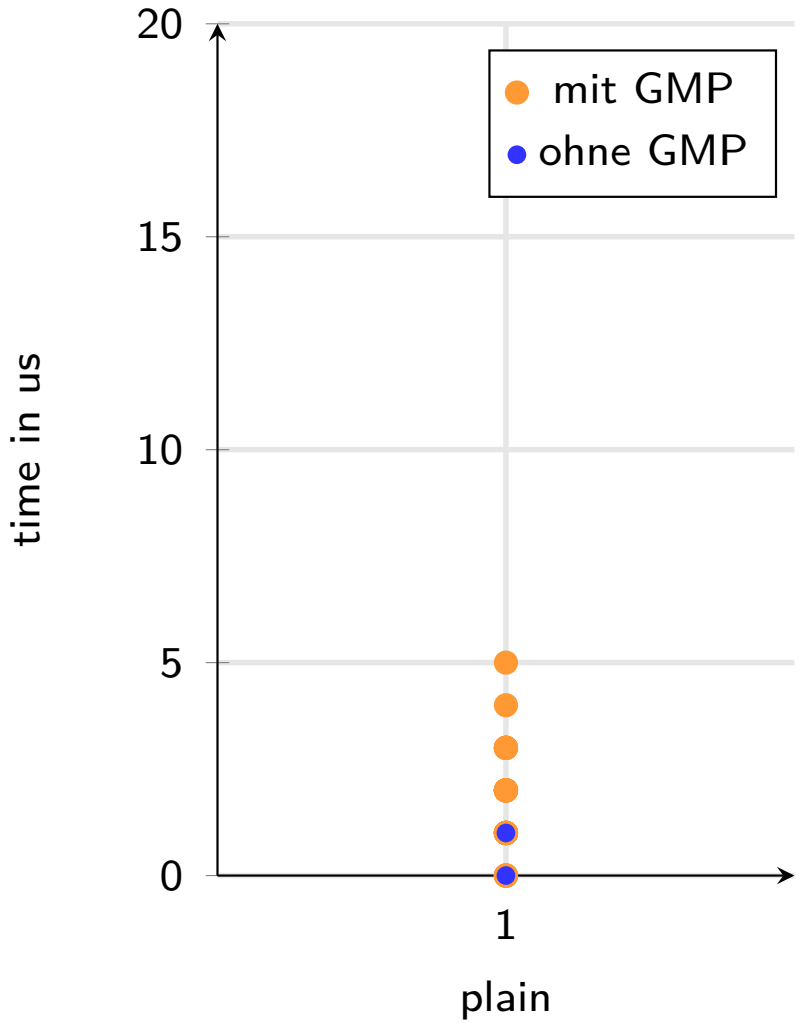
Zeitmessungen KeyGen



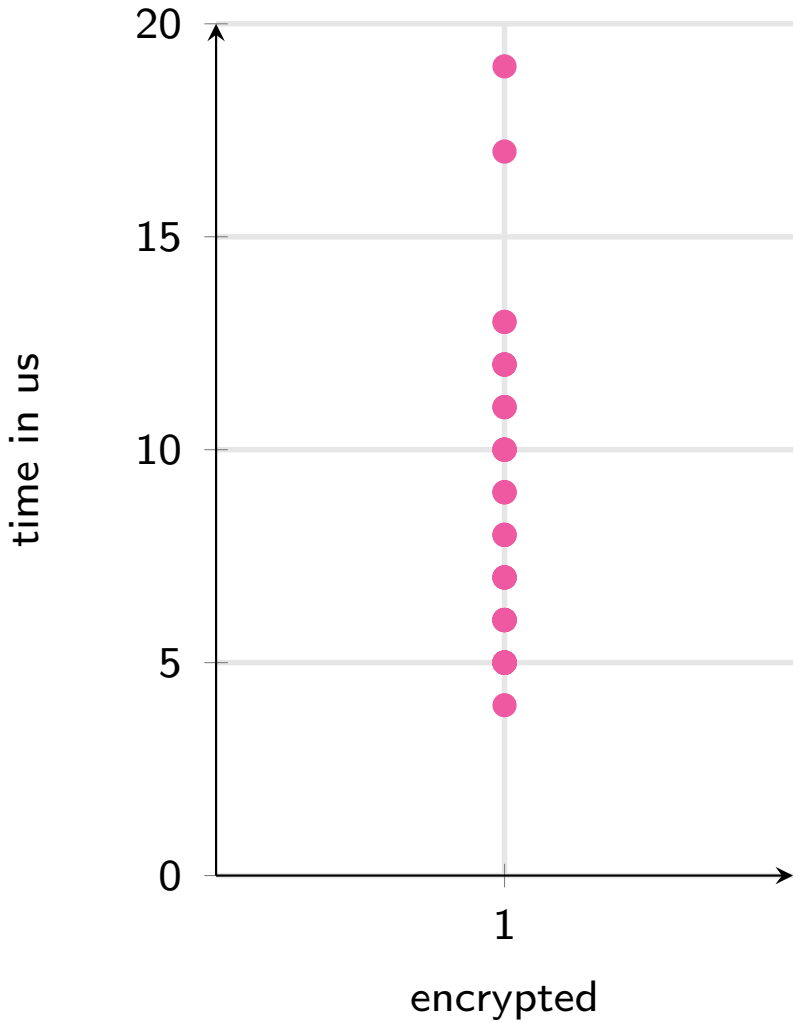


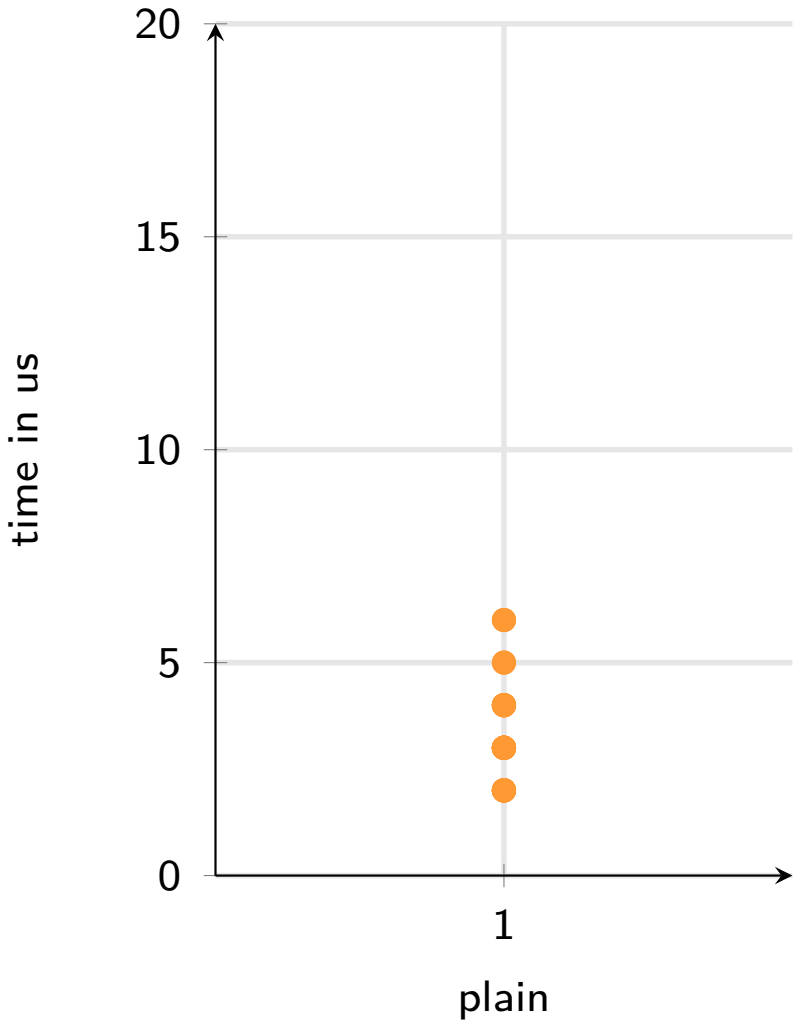
Zeitmessungen Berechnung (klein)





Zeitmessungen Berechnung (groß)





Quellen

- http://dmg.tuwien.ac.at/drmota/DA_Sigrun%20Goluch_FINAL.pdf
- http://www.liammorris.com/crypto2/Homomorphic%20Encryption%20Paper.pdf
- https://eprint.iacr.org/2016/430.pdf
- https://gmplib.org/manual/Integer-Functions.html#Integer-Functions

Vielen Dank für die Aufmerksamkeit!