Regensburger
DISKUSSIONSBEITRÄGE
zur Wirtschaftswissenschaft

University of Regensburg Working Papers in Business,
Economics and Management Information Systems

Laufzeitanalyse dreier Versionen eines Mehrparteien-Multiplikationsprotokolls

Jürgen Wenzl*

April 09, 2010

Nr. 440

JEL Classification: *Y80*

Key Words: *polynomial secret sharing, secure multiparty computation, distributed multiplication protocol*

* Jürgen Wenzl (B.Sc. in Management Information Systems, University of Regensburg) is a software developer at TMMO GmbH, Vilsgasse 25, D-93183 Kallmünz, Germany
E-mail: juergen.wenzl@gmx.de

Inhaltsverzeichnis

1 Einführung

Die vorliegende Arbeit beschäftigt sich mit der Laufzeitanalyse dreier Versionen eines für die Sicherheitstechnik relevanten Protokolls, dem MUL-Protokoll (Multiplikation in verteilter Berechnung).
Bei den drei Versionen handelt es sich um das Protokoll nach Gennaro, Rabin und Rabin und zwei schrittweisen Verbesserungen durch Prof. Lory, die natürlich zu denselben Ergebnissen führen, jedoch in geringerer Laufzeit.
Inhalt dieser Arbeit ist nach einer kurzen Beschreibung zunächst die Verifizierung der Korrektheit der beiden Verbesserungen und die anschließende Analyse der Laufzeiten der drei Versionen.
Hierzu wurde eine Applikation in der Programmiersprache C entwickelt, die die 3 Versionen des MUL-Protokolls nacheinander aufruft, wobei entschieden werden kann, ob die Verifizierung anhand vordefinierter Polynome oder die Laufzeitanalyse und welche(r) Schritt(e) für vordefinierte *n* und *t* durchgeführt werden soll(en).

Als Entwicklungsumgebung wurde Dev-C++ (Bloodshed) mit MinGW (**Min**imalist **G**NU for **W**indows) und der Bibliothek GMP (**G**NU **M**ultiple **P**recision Arithmetic Library) verwendet.

Die Quellcodes der fertigen Applikation können per E-Mail beim Autor erbeten werden.

2 GMP-Bibliothek

2.1 Was ist GMP?

GMP (**G**NU **M**ultiple **P**recision Arithmetic Library) ist das Produkt eines Open-Source-Projektes und somit frei verfügbar.
Es handelt sich hierbei um eine Bibliothek für beliebig genaue Präzisionsarithmetik, die für die Programmiersprachen C und C++ verwendet werden kann. Sie operiert auf vorzeichenbehafteten ganzen Zahlen (signed Integer), rationalen Zahlen und Fließkommazahlen (float). Der Vorteil von GMP liegt vor allem darin, dass die Genauigkeit, d.h. der Wertebereich der verwendeten Dateistrukturen und der mit ihr verwendeten Arithmetik allein durch den vom ausführenden Rechner zur Verfügung stehenden Speicher begrenzt wird.

2.2 Wozu wird die GMP-Bibliothek benötigt?

Da der Wertebereich für signed Integer in C im Intervall $\left[-2^{32}; 2^{32}\right]$ liegt, ist es nicht möglich die Anforderungen hinsichtlich der Bitgröße aller verwendeten Parameter des hier behandelten Protokolls zu gewährleisten. Ebenfalls könnten die in der Praxis verwendeten Wertebereiche von $\left[-2^{512;}\, 2^{512}\right]$ oder höher nicht erreicht werden. Durch das Einbinden der GMP-Bibliothek können nun die in ihr beinhalteten „High-level signed integer arithmetic functions (mpz)“ genutzt und somit die erforderlichen Werte erreicht werden. Dabei handelt es sich um spezielle Funktionen, die mit dem Datentyp mpz arbeiten, der einen beliebig großen String speichert, in den die ganzen Zahlen geschrieben werden.
Dies macht es möglich, sowohl immer noch ganze Zahlen als Grundlage unserer Berechnungen verwenden zu können, als auch in fast beliebig hohem Bitbereich zu operieren.

2.3 Wie benutzt man die GMP-Bibliothek?

Sie können entweder die Quellcode-Dateien der Bibliothek kompilieren und die resultierenden Dateien einbinden oder direkt die vorkompilierten verwenden.
Sie müssen die Datei „libgmp.a“ (C) bzw. „libgmpxx.a“ (C++) in das Bibliotheksverzeichnis und die Header-Datei „gmp.h“ (C) bzw. „gmpxx.h“ (C++) in das Include-Verzeichnis des Compilers kopieren.

Desweiteren muss dem Linker ein Verweis auf „libgmp.a“ (C) bzw. „libgmpxx.a“ (C++) hinzugefügt werden.
Durch die entsprechenden include-Befehle im Quellcode kann nun auf die Funktionen der GMP-Bibliothek zugegriffen werden.

3 Multiplikation in verteilter Berechnung (MUL)

Ausgehend von zwei polynomial verteilten Geheimnissen α und β generiert das MUL-Protokoll polynomial verteilte Shares des Produkts $\alpha\beta$ in verteilter Berechnung.

3.1 Grundlagen des Secret Sharing

Die im Folgenden verwendeten Secret Sharing Methoden werden im Körper $\mathbb{Z}_q$ (q ist eine Primzahl) durchgeführt.

3.1.1 Polynomiales Secret Sharing

Um einen geheimen Wert α über ein Polynom unter *n* Spielern zu verteilen, wählt man für das Polynom zunächst *t* (*t*<*n*) zufällige Koeffizienten $k_1, \ldots, k_t$ aus den Integern oder $\mathbb{Z}_q$ aus. *t* steht hier für den Grad des Polynoms. α ist der zu verteilende Wert, er wird als freier Koeffizient gesetzt. Das Share jedes Spielers besteht nun aus einem Paar $(x, f(x))$, wobei x den Spieler repräsentiert und $f(x)$ der zugehörige Funktionswert des konstruierten Verteilungspolynoms ist: $f(x) = \alpha + \sum_{i=1}^{t} k_i * x^i \bmod q$. Die Verteilung der Shares erfolgt sukzessive, indem der erste Spieler das Paar $(1, f(1))$, der zweite Spieler das Paar $(2, f(2))$ und in analoger Vorgehensweise die anderen Spieler ihre Shares in Form dieser Paare erhalten.

Irgendwelche $t+1$ Spieler $j_1, j_2, \ldots j_{t+1}$ können den geheimen Wert berechnen durch Auswertung der Lagrange-Interpolationsformel:

$$f(x) = \sum_{i=1}^{t+1} f(j_i) \prod_{k=1, k\neq i}^{t+1} \frac{x - j_k}{j_i - j_k} \bmod q$$

an der Stelle $x = 0$.

Nun kann der verteilte Wert α über diese Formel rekonstruiert werden. Voraussetzung hierfür ist, dass das Verteilerpolynom den Grad t hat und so eine beliebige Menge von mindestens (*t+1*) Spielern den Wert α berechnen kann.

Somit ist es möglich, durch polynomiale Verteilung eines Geheimnisses, über den Grad des Polynoms zu bestimmen, wie viele Personen in einer Gruppe mindestens zusammenwirken müssen, um den verteilten Wert α berechnen zu können.

3.2 Protokoll nach Gennaro, Rabin und Rabin

Das Protokoll nach Gennaro, Rabin und Rabin nimmt an, dass 2 Geheimnisse α und β über die Polynome $f_\alpha(x)$ und $f_\beta(x)$ verteilt sind und die Spieler wollen Shares des Produkts $\alpha\beta$ berechnen. Beide Polynome sind höchstens vom Grad t. Dabei stellen $f_\alpha(i)$ und $f_\beta(i)$ die Shares des Spielers i dar. Das Produkt dieser beiden Polynome ist

$$f_\alpha(x) f_\beta(x) = \gamma_{2t} x^{2t} + \ldots + \gamma_1 x + \alpha\beta \equiv f_{\alpha\beta}(x).$$

Infolge der Lagrange Interpolations-Formel gilt

$$\alpha\beta = \lambda_1 f_{\alpha\beta}(1) + \ldots + \lambda_{2t+1} f_{\alpha\beta}(2t+1) \qquad (1)$$

mit den bekannten nicht-null Konstanten

$$\lambda_i = \prod_{\substack{1 \le k \le 2t+1, \\ k \ne i}} \frac{k}{k-i} \bmod q. \qquad (2)$$

Seien $h_1(x), \ldots, h_{2t+1}(x)$ Polynome mit maximalem Grad t, so dass $h_i(0) = f_{\alpha\beta}(i)$ für $1 \le i \le 2t+1$ erfüllt ist.

Definiere

$$H(x) \equiv \sum_{i=1}^{2t+1} \lambda_i h_i(x), \qquad (3)$$

dann ist diese Funktion ein Polynom von maximalem Grad t mit der Eigenschaft

$$H(0) = \lambda_1 f_{\alpha\beta}(1) + \ldots + \lambda_{2t+1} f_{\alpha\beta}(2t+1) = \alpha\beta.$$

Also, $H(j) = \sum_{i=1}^{2t+1} \lambda_i h_i(j)$. Wenn jeder der Spieler $P_i\ (1 \le i \le 2t+1)$ sein Share $f_{\alpha\beta}(i)$ mit den anderen Teilnehmern teilt, indem er ein Polynom $h_i(x)$ mit den oben definierten Eigenschaften benutzt, dann wird das Produkt $\alpha\beta$ über das Polynom $H(x)$ von maximalem Grad t verteilt.

Diese Ideen sind die Basis des folgenden Protokolls (alle Operationen in $\mathbb{Z}_q$):

Input des Spielers P_i: die Werte $f_\alpha(i)$ und $f_\beta(i)$.

1. Spieler $P_i\ (1 \le i \le 2t+1)$ berechnet $f_\alpha(i)\, f_\beta(i)$ und verteilt diesen Wert, indem er ein zufälliges Polynom $h_i(x)$ von maximalem Grad t wählt, so dass

   $h_i(0) = f_\alpha(i) f_\beta(i)$.

   Er gibt dem Spieler $P_j\ (1 \le j \le n)$ den Wert $h_i(j)$.

2. Jeder Spieler $P_j\ (1 \le j \le n)$ berechnet sein Share von $\alpha\beta$ über ein zufälliges Polynom H, d.h. den Wert $H(j)$, durch lokale Berechnung der Linearkombination

$$H(j) = \sum_{i=1}^{2t+1} \lambda_i h_i(j).$$

Abbildung 1: Das Multiplikations-Protokoll von Gennaro, Rabin und Rabin

Für die Untersuchung der Zeitkomplexitäten werden zwei Grundannahmen gemacht:

(a) Die Addition oder Subtraktion von zwei k-Bit-Ganzzahlen benötigt $\rho_{add} k$ Bit-Operationen.

(b) Die Multiplikation einer k-Bit-Ganzzahl mit einer l-Bit-Ganzzahl benötigt $\rho_{mult} kl$ Bit-Operationen. Dies ist eine vernünftige Schätzung für nicht zu große Werte.

Die konkreten Werte für ρ_{add} und ρ_{mult} sind maschinenabhängig.

Es wird angenommen, dass $n \ge 2t+1$ ist.

Schritt 1 dieses Protokolls benötigt n Auswertungen des Polynoms $h_i(x)$ vom Grad t. Falls hierfür das Horner-Schema benutzt wird, benötigt eine Auswertung t Multiplikationen einer k-Bit-Ganzzahl (Integer) mit einer Ganzzahl von höchstens $\log_2 n$ Bits und t Additionen von zwei k-Bit-Ganzzahlen.

In Schritt 2 des Protokolls berechnet jeder Spieler $2t+1$ Multiplikationen und $2t$ Additionen von zwei k-Bit Zahlen. Somit ergeben sich in Anbetracht der eben gemachten Annahmen insgesamt

$$\rho_{mult}\left[ntk\lceil\log_2 n\rceil + (2t+1)k^2\right] + \rho_{add}(ntk + 2tk), \qquad (4)$$

d.h. $O\left(n^2 k \log n + nk^2\right)$ Bit-Operationen je Spieler.

3.3 1. Beschleunigung des Protokolls von Lory [1]

In dem Multiplikationsprotokoll von Gennaro, Rabin und Rabin wählt jeder Spieler $P_i\,(1 \le i \le 2t+1)$ zufällig die Koeffizienten $a_j\,(1 \le j \le t)$ eines Polynoms von einem maximalen Grad t

$$h_i(x) = a_t x^t + a_{t-1} x^{t-1} + \ldots + a_1 x + a_0$$

mit $a_0 = f_\alpha(i) f_\beta(i)$. Dann muss der Spieler sein Polynom an n unterschiedlichen Punkten auswerten. Stattdessen wählt in dem neuen Protokoll jeder Spieler zufällig t Stützstellen f_j für die t Abszissen $x_j = j\,(1 \le j \le t)$. Zusammen mit der Bedingung

$$h_i(0) = f_\alpha(i) f_\beta(i)$$

definiert dies implizit das eindeutige Interpolationspolynom $h_i(x)$ vom maximalen Grad t. Nun muss der Spieler P_i dieses Polynom für $x_j = j\,(t+1 \le j \le n)$ auswerten. Schritt 1 in Abbildung 2 führt diese Berechnungen sehr effizient aus, indem das Newtonsche Schema der Dividierten Differenzen benutzt wird. Aus Gründen der Lesbarkeit wird der Index i weitgehend weggelassen.

Schritt 2 aus Abb.1 wird unverändert beibehalten.

Input des Spielers P_i : die Werte $f_\alpha(i)$ und $f_\beta(i)$.

1. Spieler $P_i\ (1 \le i \le 2t+1)$ berechnet $f_\alpha(i)\, f_\beta(i)$ und verteilt diesen Wert, indem er t zufällige Stützstellen $f_1, f_2, \ldots, f_t$ wählt und folgende Teilschritte ausführt:

   (a) $g_0 := f_\alpha(i) f_\beta(i)$.

   (b) for $j = 1, 2, \ldots, t$:

   $g_j := f_j$

   for $k = j-1, j-2, \ldots, 0$:

   $g_k := g_{k+1} - g_k$

   (c) for $j = t+1, t+2, \ldots, n$:

   for $k = 0, 1, \ldots, t-1$:

   $g_{k+1} := g_{k+1} + g_k$

   $f_j := g_t$

   Er gibt dem Spieler $P_j\ (1 \le j \le n)$ den Wert

   $$h_i(j) := f_j .$$

2. Jeder Spieler $P_j\ (1 \le j \le n)$ berechnet sein Share von $\alpha\beta$ über ein zufälliges Polynom H, d.h. den Wert $H(j)$, durch lokale Berechnung der Linearkombination

   $$H(j) = \sum_{i=1}^{2t+1} \lambda_i h_i(j) .$$

Abbildung 2: 1. Beschleunigung des Multiplikations-Protokolls

Dieses Variante des Protokolls benötigt im 1. Schritt $t(t+1)/2 + (n-t)\,t$ Additionen von zwei k-Bit Zahlen und im 2. Schritt $(2t+1)$ Multiplikationen und $2t$ Additionen (wieder von zwei k-Bit Zahlen). Somit ergeben sich

$$\rho_{mult}(2t+1)k^2 + \rho_{add}\left[\left(n - \frac{t}{2}\right)t + \frac{5}{2}t\right]k$$

d.h. $O\left(n^2 k + n k^2\right)$ Bit-Operationen je Spieler.

3.4 2. Beschleunigung des Protokolls von Lory [2]

Schritt 1 wird aus Teilabschnitt 3.3 übernommen.

Schritt 2 in Abbildung 1 und Abbildung 2 berechnet

$$H(j) = \sum_{i=1}^{2t+1} \lambda_i h_i(j), \qquad (5)$$

wobei $\lambda_1, \lambda_2, \ldots, \lambda_{2t+1}$ die Langrange Interpolationskoeffizienten (2) aus Gleichung (1) sind. Folglich berechnet die rechte Seite der Gleichung (5) die Extrapolation der Stützpunkte $(i, h_i(j)), i = 1, 2, \ldots, 2t+1$ zur Abszisse 0.

Anstelle die Lagrange Interpolationsformel zu benutzen, verwendet Schritt 2 in Abbildung 3 erneut das Newtonsche Schema der Dividierten Differenzen. Das Ergebnis ist die Eliminierung des $O(nk^2)$-Terms aus $O(n^2k + nk^2)$. Dies ist natürlich nur dann ein Gewinn, wenn $n < k$.

Input des Spielers P_i: die Werte $f_\alpha(i)$ und $f_\beta(i)$.

1. Spieler $P_i\ (1 \le i \le 2t+1)$ berechnet $f_\alpha(i)\, f_\beta(i)$ und verteilt diesen Wert, indem er t zufällige Stützstellen $f_1, f_2, \ldots, f_t$ wählt und folgende Teilschritte ausführt:

   (a) $g_0 := f_\alpha(i) f_\beta(i)$

   (b) for $j = 1, 2, \ldots, t$:

   $g_j := f_j$

   for $k = j-1, j-2, \ldots, 0$:

   $g_k := g_{k+1} - g_k$

   (c) for $j = t+1, t+2, \ldots, n$:

   for $k = 0, 1, \ldots, t-1$:

   $g_{k+1} := g_{k+1} + g_k$

   $f_j := g_t$

   Er gibt dem Spieler $P_j\ (1 \le j \le n)$ den Wert $h_i(j) := f_j$.

2. Jeder Spieler $P_j\ (1 \le j \le n)$ berechnet sein Share von $\alpha\beta$ über ein zufälliges Polynom H, d.h. den Wert $H(j)$, durch lokale Ausführung der folgenden Teilschritte:

   (a) for $i = 1, 2, \ldots, 2t+1$:

   $g_i := h_i(j)$

   for $k = i-1, i-2, \ldots 1$:

   $g_k := g_{k+1} - g_k$

   $X_i := g_1$

   (b) for $k = 2t, 2t-1, \ldots, 1$:

   $X_k := X_k - X_{k+1}$

   $H(j) := X_1$

Abbildung 3: 2. Beschleunigung des Multiplikations-Protokolls

4 Verifizierung der Korrektheit der Protokolle

Die Verifizierung der Korrektheit der Protokolle erfolgt anhand eines bewusst klein gewählten Beispiels, so dass die relevanten Werte händisch nachgeprüft werden können.

Wir wählen $\alpha = 37$ und $\beta = 14$; somit ergibt sich $\alpha\beta = 518$. Da wir uns in $\mathbb{Z}_q$ bewegen, bestimmen wir die nächst größere Primzahl: $q = 521$.

Nun bilden wir für die Shares von α und β die Polynome $f_\alpha(x)$ und $f_\beta(x)$ durch Wahl „zufälliger“ Koeffizienten:

$$f_\alpha(x) = x^3 + x^2 + x + 37 \quad \text{und} \quad f_\beta(x) = x^3 + 2x + 14$$

Der Grad der Polynome ist 3, somit $t = 3$.

Aufgrund der Bedingung $n \geq 2t + 1$ ergibt sich $n = 2*3+1 = 7$.

Damit ergeben sich für die beiden Polynome folgende Funktionswerte, die die Spieler als Shares erhalten:

Spieler P_i		1	2	3	4	5	6	7
$f_\alpha(i)$	37	40	51	76	121	192	295	436
$f_\beta(i)$	14	17	26	47	86	149	242	371

Die beiden Polynome $f_\alpha(x)$ und $f_\beta(x)$ wurden in der hierzu erstellten Applikation vordefiniert und implementiert. Diese werden beim Programmstart nach der Auswahl des Verifizierungsmodus zur Kontrolle ausgegeben.

Das Protokoll von Gennaro/Rabin/Rabin einerseits und die Protokolle von Lory andererseits liefern erwartungsgemäß unterschiedliche Werte für die $H(j)\,(j=1,\ldots,7)$:

H	Gennaro/Rabin/Rabin	Lory1	Lory1
1	439	249	249
2	170	337	337
3	410	377	377
4	295	485	485
5	3	256	256
6	233	327	327
7	121	293	293

Für Lory1 und Lory2 sind diese identisch, da mathematisch gleichwertige Operationen ausgeführt werden.

Da der Grad der Polynome 3 ist $(t=3)$, sind zur Berechnung des geheimen Wertes 4 Shares notwendig. Dazu werden neue Lagrange Interpolationskoeffizienten berechnet:

Lagrange Interpolationskoeffizienten für die Spieler 1, 2, 3, 4 (vgl. Gleichung (2)):

$$\lambda_1' = \frac{2\cdot 3\cdot 4}{1\cdot 2\cdot 3} = 4$$

$$\lambda_2' = \frac{1\cdot 3\cdot 4}{(-1)\cdot 1\cdot 2} = -6 = 515 \bmod 521$$

$$\lambda_3' = \frac{1\cdot 2\cdot 4}{(-2)\cdot(-1)\cdot 1} = 4$$

$$\lambda_4' = \frac{1\cdot 2\cdot 3}{(-3)\cdot(-2)\cdot(-1)} = -1 = 520 \bmod 521$$

Zur Verifizierung müssen die λ_j' mit den jeweiligen $H(j)$ multipliziert werden. Die Summe dieser Produkte muss das Produkt $\alpha\beta = 518$ ergeben.

In diesem Fall also für Gennaro/ Rabin/ Rabin:

$$(4\cdot 439 + 515\cdot 170 + 4\cdot 410 + 520\cdot 295) \bmod 521 = 518$$

Analog müssen die Lagrange Interpolationskoeffizienten für die Spieler 2, 3, 4, 5 / 3, 4, 5, 6 und 4, 5, 6, 7 berechnet werden.

Dadurch ergeben sich folgende Werte für Gennaro/ Rabin/ Rabin:

Lagrange Interpolationskoeffizient	1, 2, 3, 4	2, 3, 4, 5	3, 4, 5, 6	4, 5, 6, 7
λ_1'	4			
λ_2'	515	10		
λ_3'	4	501	20	
λ_4'	520	15	476	35
λ_5'		517	36	437
λ_6'			511	70
λ_7'				501

Für diese zeigt sich analog:

2, 3, 4, 5: $(10 \cdot 170 + 501 \cdot 410 + 15 \cdot 295 + 517 \cdot 3) \bmod 521 = 518$

3, 4, 5, 6: $(20 \cdot 410 + 476 \cdot 295 + 36 \cdot 3 + 511 \cdot 233) \bmod 521 = 518$

4, 5, 6, 7: $(35 \cdot 295 + 437 \cdot 3 + 70 \cdot 233 + 501 \cdot 121) \bmod 521 = 518$

Des Weiteren müssen diese Schritte für „Lory1“ und „Lory2“ wiederholt werden. Auch hier zeigt sich, dass sich jeweils 518 ergibt:

1, 2, 3, 4: $(4 \cdot 249 + 515 \cdot 337 + 4 \cdot 377 + 520 \cdot 485) \bmod 521 = 518$

2, 3, 4, 5: $(10 \cdot 337 + 501 \cdot 377 + 15 \cdot 485 + 517 \cdot 256) \bmod 521 = 518$

3, 4, 5, 6: $(20 \cdot 377 + 476 \cdot 485 + 36 \cdot 256 + 511 \cdot 327) \bmod 521 = 518$

4, 5, 6, 7: $(35 \cdot 485 + 437 \cdot 256 + 70 \cdot 327 + 501 \cdot 293) \bmod 521 = 518$

Damit ist die Korrektheit der Protokolle für das (kleine) Beispiel verifiziert.

5 Verifizierung der Beschleunigungen durch Zeitmessung

Die Verifizierung der Verbesserungen hinsichtlich der Komplexität und damit der Laufzeit erfolgt durch Zeitmessung, in dem die drei Versionen des MUL-Protokolls hintereinander aufgerufen und die jeweiligen Start- und Stopzeiten festgehalten werden und die Dauer ermittelt wird.

Zur genaueren Analyse wird nicht nur die Laufzeit des jeweiligen kompletten Protokolls gemessen, sondern zwischen Schritt 1 und 2 unterschieden (vgl. die Abbildungen in Kapitel 3.2, 3.3 und 3.4).
Zur Erhöhung der Messgenauigkeit wurden je nachdem welches n gewählt wurde eine entsprechende Anzahl an Verlängerungslaufanweisungen durchgeführt, z.B. bei $n = 9$ 100.000 Wiederholungen.
Alle Berechnungen wurden auf einem Rechner mit Intel(R) Core(TM)2 Duo CPU T9400 @ 2,53 GHz und für $n = 2t + 1$ durchgeführt.

Im Folgenden die gemessenen Werte in Millisekunden [ms]:

- Vergleich zwischen Gennaro/Rabin/Rabin und Lory1 für **Schritt 1**:

$k = 1024$	Gennaro/Rabin/Rabin	Lory1
$n = 2^2 + 1$	0,025	0,010
$n = 2^3 + 1$	0,161	0,059
$n = 2^5 + 1$	8,800	2,781
$n = 2^7 + 1$	543,890	165,460
$n = 2^9 + 1$	34911,400	10390,200
$n = 2^{11} + 1$	2235898,000	667170,000

- Vergleich zwischen Gennaro/Rabin/Rabin und Lory2 für **Schritt 2**:

$k = 1024$	Gennaro/Rabin/Rabin	Lory2
$n = 2^2 + 1$	0,050	0,007
$n = 2^3 + 1$	0,170	0,032
$n = 2^5 + 1$	2,381	1,138
$n = 2^7 + 1$	40,780	61,870
$n = 2^9 + 1$	800,000	4121,800
$n = 2^{11} + 1$	17015,000	324804,000

Dennoch weisen identische Codesegmente, wo man ebenso identische Zeiten erwarten würde, durchaus unterschiedliche Zeiten auf, wenn auch im hundertstel Millisekunden-, aber auch bis in den Sekundenbereich.

Die Ursache hierfür ist zum Einen die Ressourcen-Auslastung des Rechners (CPU, Speicher, Festplattenzugriffe); auch wenn der Rechner augenscheinlich inaktiv ist, so sind ständig diverse Prozesse im Hintergrund aktiv, die mal mehr oder mal weniger Ressourcen benötigen und somit das Messergebnis beeinflussen. Zum Anderen muss ebenfalls bedacht werden, dass die Messgenauigkeit in C nicht 100% exakt ist, was die Abweichungen ebenfalls erklärt.

Literaturverzeichnis

[1] Peter Lory
Reducing the Complexity in the Distributed Multiplication Protocol of Two Polynomially Shared Values, 2007
In: Proceedings of the 21st International Conference on Advanced Networking and Applications (AINA 2007), Symposium on Security in Networks and Distributed Systems (SSNDS-07). IEEE Computer Society, Los Alamitos, Calif., S. 415-419. ISBN 0-7695-2847-3; 978-0-7695-2847-2.

[2] Peter Lory
Secure Distributed Multiplication of Two Polynomially Shared Values - Enhancing the Efficiency of the Protocol, 2009
In: Falk, Rainer und Goudalo, Wilson und Chen, Eric Y. und Savola , Reijo und Popescu, Manuela, (eds.) The Third International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2009). The Institute of Electrical and Electronics Engineers, Inc., S. 286-291. ISBN 978-0-7695-3668-2.

[3] Sandra Loosemore/ Richard M. Stallman/ Roland McGrath/ Andrew Oram/ Ulrich Drepper, Quelle: http://www.gnu.org/s/libc/manual, 11.05.2009
The GNU C Library Reference Manual 2007, Version 2.8, 2007

[4] The GMP developers, Quelle: http://gmplib.org, 06.05.2009
The GNU Multiple Precision Arithmetic Library, Edition 4.3.0, 2009

[5] http://bloodshed.net
The Dev-C++ Resource Site, 2009

[6] http://gmplib.org
The GNU MP Bignum Library, 2009

[7] http://suchideas.com/journal/2007/07/installing-gmp-on-windows
Installing GMP on Windows, 2007