Big-Integer-Arithmetik in C#

und ihre Anwendung auf digitale Signaturen

Axel Heer

Institut für Diskrete Mathematik und Geometrie
Technische Universität Wien

21. März 2011

Agenda

1. Big-Integer-Arithmetik
   - „Große“ Zahlen
   - „Noch größere“ Zahlen
   - Performance
2. Digitale Signaturen
   - Primzahlen
   - Asymmetrische Kryptosysteme
   - Hashfunktionen
3. Praktische Anwendungen
   - „Klassische“ Unterschrift
   - Kopierschutzmechanismen

Agenda

1. Big-Integer-Arithmetik
   - „Große“ Zahlen
   - „Noch größere“ Zahlen
   - Performance
2. Digitale Signaturen
   - Primzahlen
   - Asymmetrische Kryptosysteme
   - Hashfunktionen
3. Praktische Anwendungen
   - „Klassische“ Unterschrift
   - Kopierschutzmechanismen

Darstellung großer Zahlen

Über den Aufbau der .NET Datenstruktur *Heer.Axel.Thesis.IntBig*

- Wahl einer geeigneten Basis b
  - $a = a_n b^n + \cdots + a_1 b^1 + a_0 b^0$
  - Für binäre Operationen Zweierpotenz optimal
  - Nicht zu groß, um *Overflows* zu vermeiden
- Implementierung aller „Standardoperationen“
  - Schulmethoden für nicht allzu große Werte
  - Divisionsalgorithmus am komplexesten
  - Effizienz extrem wichtig (!)
- Maximale Ähnlichkeit mit gewöhnlichen Integers
  - Klasse ist *immutable*
  - Überladung von Operatoren
  - Ähnlicher Contract wie *System.Int32*

Große Zahlen im Code

Ein syntaktischer Vergleich zwischen *int* und *IntBig*

int Pow(int a, int b)

```
int v = 1;

while (b != 0) {
   if (b % 2 == 1) {
      v = a * v;
   }
   a = a * a;
   b = b / 2;
}

return v;
```

IntBig Pow(IntBig a, IntBig b)

```
IntBig v = 1;

while (b != 0) {
   if (b.IsOdd) {
      v = a * v;
   }
   a = a * a;
   b = b >> 1;
}

return v;
```

Beispiel: Multiplizieren

Ein einfacher Algorithmus für die Multiplikation großer Zahlen

IntBig Multiply(IntBig left, IntBig right)

```
for (int i = 0; i < right.Length; i++) {
  ulong carry = 0UL;
  for (int j = 0; j < left.Length; j++) {
    ulong digits = bits[i + j] + carry
      + (ulong)left[j] * (ulong)right[i];
    bits[i + j] = (uint)digits;
    carry = digits >> 32;
  }
  bits[i + left.Length] = (uint)carry;
}
```

Beispiel: Quadrieren

Die für zwei gleiche Zahlen optimierte Multiplikationsschleife

IntBig Square(IntBig value)

```
ulong carry = 0UL;
for (int j = 0; j < i; j++) {
  ulong d1 = bits[i + j] + carry;
  ulong d2 = (ulong)value[j] * value[i];
  bits[i + j] = (uint)(d1 + (d2 << 1));
  carry = (d2 + (d1 >> 1)) >> 31;
}
ulong d = (ulong)value[i] * value[i] + carry;
bits[i * 2] = (uint)d;
bits[i * 2 + 1] = (uint)(d >> 32);
```

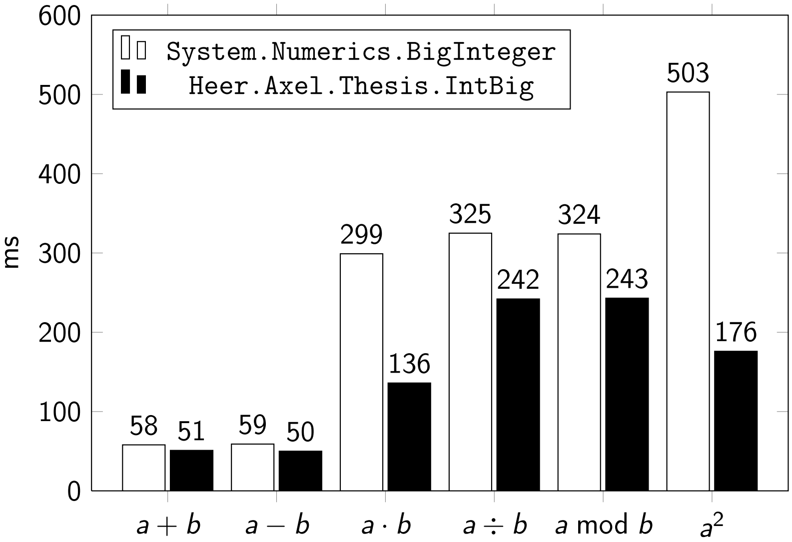
Optimierung der Rechenoperationen

Methoden zur Beschleunigung von *IntBig* in Hinblick auf digitale Signaturen

- Operationen mit kleinem zweiten Operanden
  - Aufwendiges Abschätzen / Allokieren nicht notwendig
  - Schnelles Dividieren durch kleine Primzahlen
  - Wichtig für *Primzahltests*
- Schnelles Potenzieren
  - Square-and-multiply in 8-Bit-Schritten
  - Effiziente Reduktion mittels Montgomery Zahlen
  - Effiziente Reduktion mittels Barrett Methode
- Schnelles Multiplizieren
  - *Divide-and-conquer* mittels Karatsuba-Algorithmus
  - Verallgemeinerung mittels Toom-Cook-Algorithmus
  - Aufteilung auf mehrere CPUs

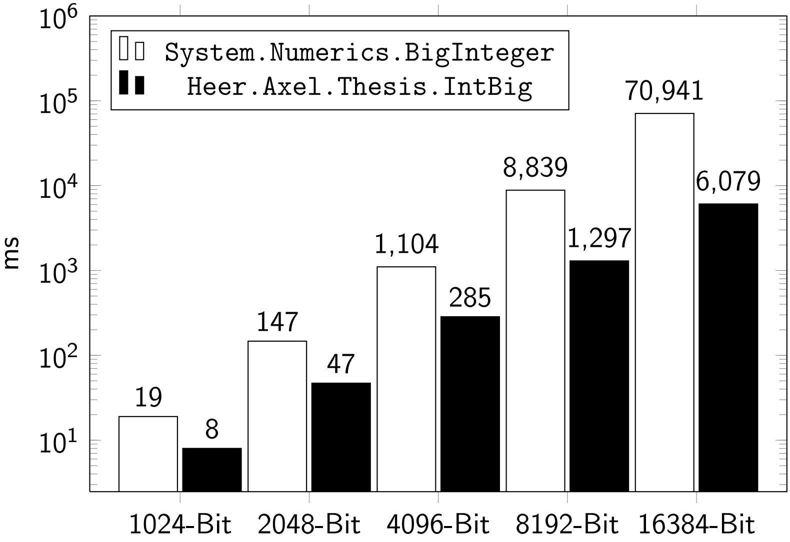
IntBig vs. *BigInteger* ($256 \leq$ Bit-Anzahl ≤ 1024)

Die Performance von *IntBig* im Vergleich zur Big-Integer-Arithmetik von Microsoft



IntBig vs. *BigInteger* (a^b mod n)

Die Performance von *IntBig* im Vergleich zur Big-Integer-Arithmetik von Microsoft



Agenda

- „Einwegfunktionen“ für asymmetrische Kryptosysteme
  - Komplexität des diskreten Logarithmus
  - Komplexität der Primfaktorzerlegung
  - Effiziente Algorithmen nicht bekannt
- Probabilistische Primzahltests
  - Deterministische Verfahren zu aufwendig
  - Aussieben mittels einfacher Teilbarkeitstests
  - Gegebenenfalls anschließender *Rabin-Miller-Test*
- „Sichere“ Zufallszahlen
  - Deterministische Zufallszahlengeneratoren ungeeignet
  - Physikalische Messdaten / Hardwareereignisse
  - Von „modernem“ Betriebssystem bereitgestellt

Die digitale Signatur

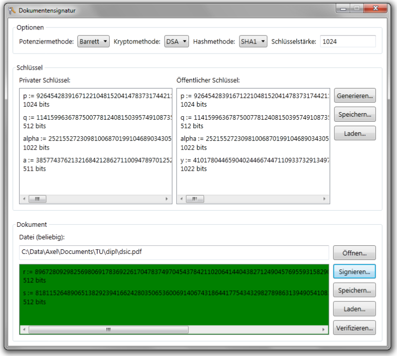
Mit Hilfe von Hashfunktionen zum gewünschten Resultat

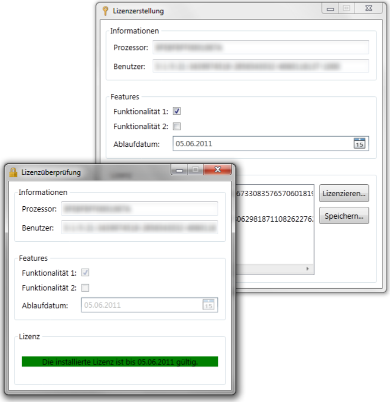
- Hashfunktionen
  - Transformiert Nachricht *m* auf fixe (kleine) Größe
  - Muss „kryptologisch sicher“ und standardisiert sein
  - Implementierung schwierig / Effizienz (Speicher!) wichtig
- Erzeugung einer Signatur
  - Schlüsselpaar generieren / öffentlichen Teil bekanntgeben
  - Kurzfassung der Nachricht mittels Hashfunktion erstellen
  - Privaten Schlüssel für „Verschlüsselung“ verwenden
- Verifikation einer Signatur
  - Öffentlichen Schlüssel für „Entschlüsselung“ verwenden
  - Kurzfassung der Nachricht mittels Hashfunktion neu erstellen
  - Ergebnis mit „entschlüsselter“ Signatur vergleichen

Agenda

Praktische Anwendungen

Als Beispiel die „klassische“ Unterschrift sowie Kopierschutzmechanismen





Agenda

1. Big-Integer-Arithmetik
   - „Große“ Zahlen
   - „Noch größere“ Zahlen
   - Performance
2. Digitale Signaturen
   - Primzahlen
   - Asymmetrische Kryptosysteme
   - Hashfunktionen
3. Praktische Anwendungen
   - „Klassische“ Unterschrift
   - Kopierschutzmechanismen

Zusammenfassung

- Microsoft sollte Mathematiker anstellen ;-)
- Performance der Multiplikation ausschlaggebend
- Digitale Signaturen sind mehr als nur eine Unterschrift

- Ausblick
  - Schönhage-Strassen-Algorithmus für noch viel größere Zahlen
  - Noch bessere Performance dank neuer GPGPU Technologien?