Public-Key Building Blocks
Summer School on Cryptographic Hardware, Side-Channel and Fault Attacks
June 12-15, 2006

Louvain-la-Neuve, June 13, 2006

Christof Paar
Ruhr University Bochum, Germany
www.crypto.rub.de
Contents

1. Why do we need public-key cryptography?
2. Overview on public-key crypto schemes
3. Arithmetic
4. Open research problems
Contents

1. Why do we need public-key cryptography?
2. Overview on public-key crypto schemes
3. Arithmetic
4. Open research problems
IT Security vs. Cryptography

1. IT Security ≠ Cryptography
2. but: Cryptography is an important tool for achieving secure IT systems

Cryptography

- Symmetric Algorithms (2000 B.C. ... 1976)
- Public-key Algorithms (Diffie/Hellman/Merkle 1976)
The Cryptographic Toolkit

Cryptographic Algorithms

Symmetric
- Stream ciphers
- Block ciphers

Asymmetric
- Integer Factorization (RSA...)
- Discrete Logarithm (D-H, DSA,...)
- Elliptic Curves (ECDH, ECDSA,...)

Hash Fct.
- MD5
- SHA-1
- SHA-XY
- ?
What we can do with symmetric crypto (I): Confidentiality

Encryption ensures **confidentiality** of messages
Message Authentication Codes (MAC) detect malicious integrity violations.
What do we need public-key (or asymmetric) cryptography for?

Two main functions:
1. Key distribution over unsecure channel
2. Digital Signatures for non-repudiation
3. [Encryption]

Rem: symmetric ciphers are still needed because public-key algorithms are awfully slow.
(Note: purely practical/engineering reason)
Non-repudiation: Why we need it

Without non-repudiation:
1. Alice orders at favorite eCommerce vendor
2. stuff gets delivered
3. Alice doesn’t feel like buying: „I never ordered this“
4. vendor can not proof it (big monetary issue if vendor = BMW.com)
Non-repudiation with Digital Signature

with non-repudiation:
1. Alice orders at favorite eCommerce vendor
2. stuff gets delivered
3. Alice doesn't feel like buying: “I never ordered this”
4. vendor sues Alice: **proof** of order through Alice’s signature (only Alice knows $k_{\text{private}}$, not even the vendor!)

Non-repudiation is strong point of asymmetric cryptography
Contents

1. Why do we need public-key cryptography?
2. Overview on public-key crypto schemes
3. Arithmetic
4. Open research problems
The World of Public-key Algorithms

Much fewer schemes than in the symmetric case!

Public-key Schemes

Established Algorithms
1. Integer factorization family
2. Discrete log family
3. Elliptic curve family

Not-so established Alg.
- lattice-based (NTRU)
- high-field equations
- code-based (McEliece)
- …
Established public-key algorithms

The 3 families of algorithms of practical relevance:

**Integer Factorization**
- Ex: RSA, Rabin, ...
- Operands: 1024 – 4096 bits

**Discrete Logarithm**
- Ex: Diffie-Hellman, DSA, ...
- Operands: 1024 – 4096 bits

**Elliptic Curves (ECC)**
- Ex: EC Diffie-Hellman, ECDSA, ...
- Operands: 160 – 256 Bits

Observation: All asymm. algorithms require heavy computation
### How many key bits do I need?

<table>
<thead>
<tr>
<th>symmetric</th>
<th>ECC</th>
<th>RSA, DL</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 bit</td>
<td>128 bit</td>
<td>≈ 700 bit</td>
<td>only short term security (breakable with some effort)</td>
</tr>
<tr>
<td>80 bit</td>
<td>160 bit</td>
<td>≈ 1024 bit</td>
<td>medium term security (excl. government attacks)</td>
</tr>
<tr>
<td>128 bit</td>
<td>256 bit</td>
<td>≈ 2048-3072 bits</td>
<td>long term security (not assuming quantum computers)</td>
</tr>
</tbody>
</table>

- Exact complexity of RSA (factorization) and DL (index-calculus) attacks is hard to determine
- Quantum computer would probably be the death of ECC, RSA & DL (but don’t hold your breath – at least a few decades away)
Arithmetic requirements of PK algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>typ. operand length (mult)</th>
<th># multipl. / group op</th>
<th># multipl. / crypto fct.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA</td>
<td>1024 bit</td>
<td>1</td>
<td>17 (verify)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>≈ 1300 (sign)</td>
</tr>
<tr>
<td>Discrete log</td>
<td>1024 bit</td>
<td>1</td>
<td>≈ 200</td>
</tr>
<tr>
<td>Elliptic Curves</td>
<td>160 bit</td>
<td>≈ 10</td>
<td>≈ 2000</td>
</tr>
</tbody>
</table>

Observations:
- RSA is „best“ for signature verification
- ECC is „best“ for signature generation
- ECC has other advantages (bandwidth etc)
- RSA by far outnumbers ECC implementations in practice (but ECC is slowly catching up)!
Hierarchical System Design of RSA and DL Engines

RSA, DL engines are mainly exponentiation units

Protocol Layer: RSA, D-H, …

Exponentiation Layer: $x^d \mod m$

Modular Arithmetic Level: $\times, +, -$  

Rem: > 90% of computation time is spent on modular multiplication
Hierarchical System Design of ECC Engines

- Modular Arithmetic Layer: $\times, \div, +, -$
- Group Operation Layer: $P+Q, P+P$
- Point Multiplication Layer: $eP$
- Protocol Layer: ECDSA, ECDH

Rem: Still > 90% of computation time is spent on modular multiplication (and on inversion, if affine coordinates are used)
1. Why do we need public-key cryptography?
2. Overview on public-key crypto schemes
3. Arithmetic
   1. Modular arithmetic
   2. Generalized Mersenne Primes
   3. Binary Fields $\mathbb{GF}(2^m)$
4. Open research problems
Arithmetic proposed for use in public-key schemes

finite fields

prime fields

GF(p)

- general primes
  - GF(p)

- special form primes
  - GF(2^n-c)
  - GF(2^n-2^m...-1)

extension fields

GF(p^m)

- char = 2
  - char = 2
    - binary primes
      - GF(2^n)

- char > 2
  - composite primes
    - GF((2^n)^m)
  - OEF primes
    - GF((2^n-c)^m)

- DL, ECC are based on finite fields ( = Galois fields)
- RSA arithmetic similar to GF(p) arithmetic
Prime Fields GF(p)

Relevance
DL: GF(p) is the only field type used in practice
ECC: GF(p) somewhat more popular than GF(2^m)
RSA: modular m=p q arithmetic, but algorithms almost identical

⇒ GF(p) is most important field in practice

Basics about GF(p) arithmetic
• addition, subtraction is cheap
• inversion is much slower than multiplication
  (hence, ECC is often used with projective coordinates)
• “Remaining" problem:

Efficient modular multiplication methods for 160-4096 bit numbers?
Prime Fields GF(p): Software I

Ex: A, B ∈ GF(p), p < 2^{4096}, word size w = 32

Element representation (on 32 bit machine):
A = a_{127}2^{127×32} + ... + a_1 2^{32} + a_0, a_i ∈ \{0,1,...,2^{32}-1\}
B = b_{127}2^{127×32} + ... + b_1 2^{32} + b_0, b_i ∈ \{0,1,...,2^{32}-1\}

Goal: Compute A \times B \mod p efficiently

For the beginning, a simple approach:
1. Step: Multi-precision multiplication
2. Step: Modular reduction
1. Step: Multi-precision Multiplication

\[
C' = A \times B
\]

**Complexity**

\(n^2\) integer multiplications

(Ex: \(n^2 = 128^2 = 16,384\) int. mult.)

**Remark**

Quadratic complexity can be reduced to \(n^{1.58}\) using Karatsuba’s algorithm
2. Step: Modular Reduction

\[ C \equiv C' = A \times B \mod p \]

1. (naive) approach: long division of \( C' \) by \( p \)
2. (better) approach: fast modulo reduction techniques, avoiding division:
   2.1. Montgomery
   2.2. Barrett
   2.3. Sedlack
   2.4. …

reduction compl. \( \approx n^2 \) integer mult.

Note: fast mult. methods à la Karatsuba not applicable!

\[ \Rightarrow \text{total modular mult. compl.} \approx 2 \times n^2 \text{ integer mult.} \]

Rem: Multi-precision mult (Step 1) and modular reduction (Step 2) are often interleaved. Complexity does not change.
Montgomery Reduction in Hardware I

p is an n-bit number: \( n = \lceil \log_2 p \rceil \)

**Idea:** Compute \( n \) inner products in parallel

**Best studied architecture:** Montgomery multiplication

\[
\begin{align*}
\text{Input: } A, B, \text{ where } A &= \sum_{i=0}^{n+2} a_i 2^i, \quad B = \sum_{i=0}^{n+1} b_i 2^i \\
\text{Output: } A \cdot B \mod N
\end{align*}
\]

1. \( R_0 = 0 \)
2. for \( i = 0 \) to \( n + 2 \) do
3. \( q_i = R_i(0) \)
4. \( R_{i+1} = (R_i + a_i \cdot B + q_i \cdot N)/2 \) \(*\)

**time complexity** (radix 2):
\( n \) clock cycles

**time complexity** (radix \( r \)):
\( n/r \) clock cycles

\( \Rightarrow O(n) \) times faster than software (which has \( n^2 \))

**area complexity:**
\( \text{cnst} \times n \) gates
Montgomery Reduction in Hardware II

Remarks

1. modular reduction is reduced to **addition of long numbers**:

   \[ R_{i+1} = \frac{(R_i + a_i B + q_i N)}{2} \]

2. Use redundant representation or systolic array to avoid long carry chains

3. Division only by 2 (or \(2^r\)) \(\Rightarrow\) only right shifts
1. Why do we need public-key cryptography?
2. Overview on public-key crypto schemes

3. Arithmetic
   1. Modular arithmetic
   2. Generalized Mersenne Primes
   3. Binary Fields GF($2^m$)

4. Open research problems
Generalized Mersenne Primes

finite fields

prime fields

GF(p)

GF(p)

general primes

special form primes

GF(2^n-c)

GF(2^n-2^m...-1)

extension fields

GF(p^m)

char = 2

char > 2

binary

composite

OEF

GF(2^n)

GF((2^n)^m)

GF((2^n-c)^m)

very attractive for ECC!
Generalized Mersenne Primes: Example

Prime: \( p = 2^{192} - 2^{64} - 1 \), \( w = 64 \)

\( A = a_2 2^{128} + a_1 2^{64} + a_0 \)
\( B = b_2 2^{128} + b_1 2^{64} + b_0 \)

\[ A \times B = c_5 2^{320} + c_4 2^{256} + c_3 2^{192} + c_2 2^{128} + c_1 2^{64} + c_0 \]

Reduction equations

\[ 2^{320} \equiv 2^{192} + 2^{128} \mod p \]
\[ 2^{256} \equiv 2^{128} + 2^{64} \mod p \]
\[ 2^{192} \equiv 2^{64} + 1 \mod p \]

\[ A \times B \equiv c_5 (2^{192} + 2^{128}) + c_4 (2^{128} + 2^{64}) + c_3 (2^{64} + 1) + c_2 2^{128} + c_1 2^{64} + c_0 \mod p \]
\[ A \times B \equiv [c_5 + c_4 + c_2] 2^{128} + [c_5 + c_4 + c_3 + c_1] 2^{64} + [c_5 + c_3 + c_0] \mod p \]

Modular reduction is realized with a few additions!
(no multiplications, no inversions)
Generalized Mersenne Primes and ECC

- Specific primes recommended by NIST: 192, 224, 256, 384, 521 bit
- Reduction requires no multiplication, only additions
- Roughly twice as fast as modular multipl. with general primes
- Very popular for ECC in practice
Contents

1. Why do we need public-key cryptography?
2. Overview on public-key crypto schemes
3. Arithmetic
   1. Modular arithmetic
   2. Generalized Mersenne Primes
   3. Binary Fields GF(2^m)
4. Open research problems
Binary Fields $GF(2^m)$

finite fields

prime fields

$GF(p)$

- general primes
- $GF(p)$
  - pseudo Mersenne
  - $GF(2^n-c)$
  - $GF(2^n-2^m\ldots-1)$

extension fields

$GF(p^m)$

- special form primes
- generalized Mersenne
- $GF(2^n)$
- $GF((2^n)^m)$
- $GF((2^n-c)^m)$

Multiplication is the most critical operation in most applications

Summer School on Crypto HW, Louvain-la-Neuve, June 2006
Basic Facts about Binary Fields GF(2^m)

1. main application in modern PK: **Elliptic Curve Cryptosystems**

2. also applicable for DL, but index-calculs attack works somewhat better in GF(2^m)* than in GF(p)*

   ⇒ *rarely used anymore for DL problems*

3. **very well studied** compared to other extension fields since 1960s (applications in channel coding for early space missions)

4. choice of char = 2 was traditionally driven by **hardware implementations**

5. arithmetic is greatly influenced by choice of basis
   – polynomial basis
   – normal basis
   – other (dual basis, triangular basis, …)

**polynomial basis most attractive for PK crypto in practice**
A Big Question:
GF(2^m) vs GF(p) for ECC?

A long story made short

1. **Software:** GF(p) is somewhat faster if carefully implemented. (Note that the vast majority of implementations run in software)
2. **Hardware:** GF(2^m) has a much better time-area product than GF(p)
3. It is believed that the **patent situation** is less messy in the GF(p) case
4. There is a trend that **GF(p) is more common in practice** (due to national standards in the US and Europe & patent situation)
5. GF(2^m) in hardware is highly attractive for **light-weight crypto** (RFID and such)
**GF(2\(^m\)) Multipliers for Hardware**

- many proposed architectures
- classification according to time-area trade-off

<table>
<thead>
<tr>
<th>architecture</th>
<th>#clocks (time)</th>
<th>#gates (area)</th>
<th>(m)</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit parallel</td>
<td>1</td>
<td>(O(m^2))</td>
<td>any</td>
<td>usually „too big“ for PK crypto</td>
</tr>
<tr>
<td>hybrid</td>
<td>(m/D)</td>
<td>(O(mD))</td>
<td>(D</td>
<td>m)</td>
</tr>
<tr>
<td>digit serial</td>
<td>(m/D)</td>
<td>(O(mD))</td>
<td>any</td>
<td>digit size (D) allows scaling</td>
</tr>
<tr>
<td>bit serial</td>
<td>(m)</td>
<td>(O(m))</td>
<td>any</td>
<td>classical arch.</td>
</tr>
<tr>
<td>super serial</td>
<td>(ms)</td>
<td>(O(m/s))</td>
<td>any</td>
<td>SW-like, only if RAM cheap</td>
</tr>
</tbody>
</table>

Main relevance in cryptography: **bit serial and digit serial**
Bit Serial Multiplication

Polynomial-basis multiplication

$$A \times B = (a_0 + \ldots + a_{m-1} x^{m-1}) \times (b_0 + \ldots + b_{m-1} x^{m-1}) \mod P(x)$$

where $a_i, b_i \in \text{GF}(2)$

In practice: $P(x)$ is almost always trinomial or pentanomial

Two traditional architectures
- least significant bit-first (LSB) multiplier
- most significant bit-first (MSB) multiplier
Least Significant Bit GF(2^m) Multiplier

A × B = a_0 B(x)
+ a_1 B mod P(x)
+ ⋯
+ a_{m-1} B mod P(x)

Time: m clock cycles
Area: \text{cnst} × m gates (\text{cnst} small)
Digit Multipliers for GF(2<sup>m</sup>)

1. generalization of bit-serial multipliers
2. fundamental idea: process \( D > 1 \) bit at a time
3. works for any \( m \)
4. trades space for speed: faster but larger than bit-serial architectures
5. time-area product is constant (at least under big-O notation)
6. LSD (least significant digit) and MSD (most significant digit) are possible
Least Significant Digit Architecture

Idea: Break $A(x)$ down into $s$ digit polynomials

$$A(x) = a_{m-1}x^{m-1} + \ldots + a_1 x + a_0 , \quad a_i \in \text{GF}(2)$$

\[
\begin{array}{cccccc}
  a_{m-1} & \cdots & a_{m-D-2} & \cdots & a_{D-1} & \cdots & a_0 \\
\end{array}
\]

$$s = \left\lceil \frac{m}{D} \right\rceil \quad \tilde{a}_{s-1} \quad \tilde{a}_0$$

$$A \left( x^D \right) = \tilde{a}_{s-1}x^{(s-1)D} + \ldots + \tilde{a}_1 x^D + \tilde{a}_0$$

where $\tilde{a}_i = \tilde{a}_i(x) = a_{i,D-1}x^{D-1} + \ldots + a_{i,1} x + a_{i,0}, \quad a_{i,j} \in \text{GF}(2)$
Least Significant Digit GF(2^m) Multiplier

\[ A \times B = \tilde{a}_0 \, B(x) + \tilde{a}_0 \cdot [x^D \, B \mod P(x)] + \cdots + \tilde{a}_{m-1} \cdot [x^D \cdot x^{Dm-2} \, B \mod P(x)] \]

\[ \tilde{a}_{s-1}, ..., \tilde{a}_1, \tilde{a}_0 \]

Time: \( \approx \frac{m}{D} \) clock cycles

Area: \( \text{cnst} \times m \, D \) gates (cnst small)

Watch out: optimum \( D = 2^i - 1 \) (and not \( 2^i \))

\[ A(x) \times B(x) \mod P(x) \]
Contents

1. Why do we need public-key cryptography?
2. Overview on public-key crypto schemes
3. Arithmetic
4. Open research problems
Challenges in Applied Public Key Cryptography

1. Highly efficient implementations of established alg. (RSA, DL, ECC) for light-weight crypto

2. New PK algorithms with low implementation complexities

3. GF(p^m) (“OEF”) has nice implementation properties in software: Security of such fields for discrete log and ECC?

4. Special-purpose hardware for PK cryptanalysis

5. Better understanding of side channel and tamper resistance
Related Workshops

RFIDSec
July 2006, Graz

CHES – Cryptographic Hardware and Embedded Systems (+ FDTC)
October 2006, Yokohama

escar – Embedded Security in Cars
November 2006, Berlin