Implementing Multivariate Public-Key Cryptosystems

Some Lessons from the Last 7 Years

Bo-Yin Yang

Institute of Information Science
Academia Sinica
Taipei, Taiwan
by@crypto.tw

SPEED-CC workshop, October 12, 2009

Outline

- Multivariate public key cryptosystems (MPKCs)
- History, trends, and factoids
- Vector instruction sets (*SSE*)
- MPKCs over odd prime fields
- Some counter-intuitive techniques
- Performance results
- Other platforms

Multivariate PKC (MPKC)

Public map of a typical multivariate PKC over base field $K = \mathbb{F}_q$:

$$\mathcal{P} : \mathbf{w} \in K^n \stackrel{S}{\mapsto} \mathbf{x} = \mathbf{M}_S\mathbf{w} + \mathbf{c}_S \stackrel{\mathcal{Q}}{\mapsto} \mathbf{y} \stackrel{T}{\mapsto} \mathbf{z} = \mathbf{M}_T\mathbf{y} + \mathbf{c}_T \in K^m$$

- S and T affine and invertible
- $\mathcal{Q}$ quadratic, known as as the *central map* (and the components of $\mathcal{Q}$ are *central polynomials*)
- For encryption schemes, $n < m$
- For signature schemes, $n > m$
- Most often $q = 2$ or a lower power of 2.

Why are MPKCs Worth Studying?

- Diversification
- Efficiency

Why are MPKCs Worth Studying?

- Diversification: Future-proof against quantum computers.
- Efficiency: Faster than “traditional” PKCs.

Why are MPKCs Worth Studying?

- Diversification: Future-proof against quantum computers.
- Efficiency: Faster than "traditional" PKCs.
  ... Maybe.

Design of Current MPKCs

Basic Trapdoor

Ways for the legitimate user to invert $\mathcal{Q}$:

Design of Current MPKCs

Basic Trapdoor

Ways for the legitimate user to invert $\mathcal{Q}$:

- Big-Field: C^*, HFE, ℓIC,

Design of Current MPKCs

Basic Trapdoor

Ways for the legitimate user to invert $\mathcal{Q}$:

- Big-Field: C^*, HFE, ℓIC,
- Small-Field: UOV, Triangular

Design of Current MPKCs

Basic Trapdoor

Ways for the legitimate user to invert $\mathcal{Q}$:

- Big-Field: C^*, HFE, ℓIC,
- Small-Field: UOV, Triangular

Modifiers

Ways to guard against an attacker finding $\mathcal{Q}$

Design of Current MPKCs

Basic Trapdoor

Ways for the legitimate user to invert $\mathcal{Q}$:

- Big-Field: C^*, HFE, ℓIC,
- Small-Field: UOV, Triangular

Modifiers

Ways to guard against an attacker finding $\mathcal{Q}$: $+$, $-$, p, i, v, ...

MPKC Modifiers

- All vanilla trapdoors have been broken
- Need modifiers to address attacks
  - Minus (-): throw away some polynomials
  - Plus (+): add central polynomials
  - Prefix or postfix (p): force some $w_i = 0$
  - Vinegar (v): perturbation in a small subspace
  - Internal perturbation (i): equal to p+v.
- A few others; not discussed here.

UOV (Unbalanced Oil and Vinegar)

Patarin, 1997

We can write the quadratic part of a polynomial in $\mathbf{w}$ as a symmetric matrix M.

UOV (Unbalanced Oil and Vinegar)

Patarin, 1997

We can write the quadratic part of a polynomial in $\mathbf{w}$ as a symmetric matrix M. If dealing with $\mathbb{F}_{2^k}$, let $f(\mathbf{w}) = \mathbf{w}^T \bar{M} \mathbf{w} +$ (lower parts), then $M = \bar{M} + \bar{M}^T$ is the matrix we want.

UOV (Unbalanced Oil and Vinegar)

Patarin, 1997

We can write the quadratic part of a polynomial in **w** as a symmetric matrix M. Matrices corresponding to central polynomials of UOV schemes have a distinctive form:

$$M_i := \left[\begin{array}{ccc|ccc} \alpha_{11}^{(i)} & \cdots & \alpha_{1,v}^{(i)} & \alpha_{1,v+1}^{(i)} & \cdots & \alpha_{1,n}^{(i)} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{v,1}^{(i)} & \cdots & \alpha_{v,v}^{(i)} & \alpha_{v,v+1}^{(i)} & \cdots & \alpha_{v,n}^{(i)} \\ \hline \alpha_{v+1,1}^{(i)} & \cdots & \alpha_{v+1,v}^{(i)} & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{n,1}^{(i)} & \cdots & \alpha_{n,v}^{(i)} & 0 & \cdots & 0 \end{array}\right]$$

Hence given **y** and $x_1, \ldots, x_v$ we can solve for $x_{v+1}, \ldots, x_n$.

Rainbow-Type Signatures

or Stage-wise UOV, Ding 2005

- For $0 < v_1 < v_2 < \cdots < v_{u+1} = n$
  - $S_l := \{1, 2, \ldots, v_l\}$
  - $O_l := \{v_l + 1, \ldots, v_{l+1}\}$
  - $o_l := v_{l+1} - v_l = |O_l|$
- $\mathcal{Q} : \mathbf{x} = (x_1, \ldots, x_n) \mapsto \mathbf{y} = (y_{v_1+1}, \ldots, y_n)$
  - $y_k := q_k(\mathbf{x})$, with following form if $v_l < k \leq v_{l+1}$

$$q_k = \sum_{i \leq j \leq v_l} \alpha_{ij}^{(k)} x_i x_j + \sum_{i \leq v_l < j < v_{l+1}} \alpha_{ij}^{(k)} x_i x_j + \sum_{i < v_{l+1}} \beta_i^{(k)} x_i$$

- Given all y_i with $v_l < i \leq v_{l+1}$ and all x_j with $j \leq v_l$, we can compute $x_{v_l+1}, \ldots, x_{v_{l+1}}$ via elimination

Rainbow Variants

TTS: Chen+Yang 2004

- Uses a sparse $\mathcal{Q}$
- $\mathcal{Q}^{-1}$ only need solving linear systems like Rainbow

Example from 2004: TTS(20,28)

$$
\begin{aligned}
y_i &= x_i + \textstyle\sum_{j=1}^{7} p_{ij} x_j x_{8+(i+j \bmod 9)},\ i = 8, \ldots, 16 \\
y_{17} &= x_{17} + p_{17,1} x_1 x_6 + p_{17,2} x_2 x_5 + p_{17,3} x_3 x_4 \\
&\quad + p_{17,4} x_9 x_{16} + p_{17,5} x_{10} x_{15} + p_{17,6} x_{11} x_{14} + p_{17,7} x_{12} x_{13} \\
y_{18} &= x_{18} + p_{18,1} x_2 x_7 + p_{18,2} x_3 x_6 + p_{18,3} x_4 x_5 \\
&\quad + p_{18,4} x_{10} x_{17} + p_{18,5} x_{11} x_{16} + p_{18,6} x_{12} x_{15} + p_{18,7} x_{13} x_{14} \\
y_i &= x_i + p_{i,0} x_{i-11} x_{i-9} + \textstyle\sum_{j=19}^{i} p_{i,j-18} x_{2(i-j)-(i \bmod 2)} x_j \\
&\quad + \textstyle\sum_{j=i+1}^{27} p_{i,j-18} x_{i-j+19} x_j,\ i = 19, \ldots, 27
\end{aligned}
$$

Rainbow Variants

TTS: Chen+Yang 2004

- Uses a sparse $\mathcal{Q}$
- $\mathcal{Q}^{-1}$ only need solving linear systems like Rainbow

TRMS: Wang-*-Yang, 2005

- Each UOV stage is

$$\text{piece of } \mathbf{y} = \text{quadratic}(\mathbf{x}_{\text{vinegar}}) + \text{linear}(\mathbf{x}_{\text{vinegar}}) \times_{\mathbb{F}_{q^k}} \text{linear}(\mathbf{x}_{\text{oil}})$$

- To invert the central map do divisions in various $\mathbb{F}_{q^k}$

Rainbow Variants

TTS: Chen+Yang 2004

- Uses a sparse $\mathcal{Q}$
- $\mathcal{Q}^{-1}$ only need solving linear systems like Rainbow

TRMS: Wang-*-Yang, 2005

- Each UOV stage is

$$\text{piece of } \mathbf{y} = \text{quadratic}(\mathbf{x}_{\text{vinegar}}) + \text{linear}(\mathbf{x}_{\text{vinegar}}) \times_{\mathbb{F}_{q^k}} \text{linear}(\mathbf{x}_{\text{oil}})$$

- To invert the central map do divisions in various $\mathbb{F}_{q^k}$

Rainbow-type Parameters Today

Suggested examples are $q = 16$ or 31 and layers sizes $(24, 20, 20)$.

UOV matrices look like this

$$M_i := \begin{bmatrix} * & * \\ * & 0 \end{bmatrix} = \left[\begin{array}{ccc|ccc} \alpha_{11}^{(i)} & \cdots & \alpha_{1v}^{(i)} & \alpha_{1,v+1,}^{(i)} & \cdots & \alpha_{1n}^{(i)} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{v1}^{(i)} & \cdots & \alpha_{vv}^{(i)} & \alpha_{v,v+1,}^{(i)} & \cdots & \alpha_{vn}^{(i)} \\ \hline \alpha_{v+1,1,}^{(i)} & \cdots & \alpha_{v+1,v,}^{(i)} & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{n1}^{(i)} & \cdots & \alpha_{nv}^{(i)} & 0 & \cdots & 0 \end{array}\right]$$

Rainbow and variants also have some matrices like this

$$M_i := \begin{bmatrix} * & 0 \\ 0 & 0 \end{bmatrix} = \left[\begin{array}{ccc|ccc} \alpha_{11}^{(i)} & \cdots & \alpha_{1v}^{(i)} & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{v1}^{(i)} & \cdots & \alpha_{vv}^{(i)} & 0 & \cdots & 0 \\ \hline 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & \cdots & 0 \end{array}\right]$$

The C^* Trapdoor
Matsumoto and Imai, 1988

- The central map is a monomial over $\mathbb{F}_{q^n}$

$$\mathcal{Q}(x) = x^{1+q^\theta} = x \cdot x^{q^\theta}$$

  - $\mathbb{F}_{q^n}$ is an *n*-dimension vector space over $\mathbb{F}_q$
  - Since $x \mapsto x^q$ is linear, $\mathcal{Q}$ is quadratic
  - Requires that $\gcd(1 + q^\theta, q^n - 1) = 1$
  - $\mathcal{Q}$ is inverted by raising to the inverse power of $1 + q^\theta$
- Basic scheme broken by Patarin in 1995
- $C^* - p$ and $C^* + i$ not yet broken

HFE: Hidden Field Equations
Patarin 1998

- Generalization of C^*
- Central map is a *Dembowski-Ostrom polynomial* in $\mathbb{F}_{q^n}$

$$\mathcal{Q}(x) = \sum_{q^i+q^j \le D} a_{i,j<r} x^{q^i+q^j} + \sum_{q^i \le D} b_i x^{q^i} + c$$

  - Inversion using *Berlekamp Algorithm*, much slower than C^*
  - Basic scheme is breakable if r too small
  - QUARTZ (a HFE-v) still standing

Variant Trapdoors with Smaller "Big Fields"

ℓ-invertible Cycles

- Like C^*, ℓIC also uses an intermediate field $\mathbb{L}^* = \mathbb{K}^k$
- Extends C^* by using the following central map from $(\mathbb{L}^*)^\ell$ to itself

$$\mathcal{Q} : (X_1, \ldots, X_\ell) \mapsto (Y_1, \ldots, Y_\ell) := (X_1X_2, X_2X_3, \ldots, X_{\ell-1}X_\ell, X_\ell X_1^{q^\alpha})$$

- "Standard 3IC," $\ell = 3, \alpha = 0$

$$\mathcal{Q} : (X_1, X_2, X_3) \in (\mathbb{L}^*)^3 \mapsto (X_1X_2, X_2X_3, X_3X_1)$$

HFE with intermediate fields for speed

- $\mathcal{Q}$ is a random quadratic maps in $\mathbb{L}^k \mapsto \mathbb{L}^k$, called 3HFE if $k = 3$, etc.
- To do $\mathcal{Q}^{-1}$ convert by elimination (Gröbner basis computation) to univariate equation of degree 2^k.

Note: 3HFEp, 3IC-p, and 2IC+i still standing.

Variant Trapdoors with Smaller "Big Fields"

ℓ-invertible Cycles

- Like C^*, ℓIC also uses an intermediate field $\mathbb{L}^* = \mathbb{K}^k$
- Extends C^* by using the following central map from $(\mathbb{L}^*)^\ell$ to itself

$$\mathcal{Q} : (X_1, \ldots, X_\ell) \mapsto (Y_1, \ldots, Y_\ell) := (X_1X_2, X_2X_3, \ldots, X_{\ell-1}X_\ell, X_\ell X_1^{q^\alpha})$$

- "Standard 3IC," $\ell = 3, \alpha = 0$, $\mathcal{Q}^{-1}$ in $(\mathbb{L}^*)^3$ is easy:

$$\mathcal{Q}^{-1} : (Y_1, Y_2, Y_3) \in (\mathbb{L}^*)^3 \mapsto (\sqrt{Y_1Y_3/Y_2}, \sqrt{Y_1Y_2/Y_3}, \sqrt{Y_2Y_3/Y_1},)$$

HFE with intermediate fields for speed

- $\mathcal{Q}$ is a random quadratic maps in $\mathbb{L}^k \mapsto \mathbb{L}^k$, called 3HFE if $k = 3$, etc.
- To do $\mathcal{Q}^{-1}$ convert by elimination (Gröbner basis computation) to univariate equation of degree 2^k.

Note: 3HFEp, 3IC-p, and 2IC+i still standing.

Rate-Determining Mechanisms for MPKCs

Key Generation

Evaluation of coefficients

Public Maps

Evaluating a generic set of quadratic polynomials in $\mathbb{K} = \mathbb{F}_q$

Private Maps

Rate-Determining Mechanisms for MPKCs

Key Generation

Evaluation of coefficients:

- Often as differentials of public map.
- Sometimes, by brute force!

Public Maps

Evaluating a generic set of quadratic polynomials in $\mathbb{K} = \mathbb{F}_q$

Private Maps

Rate-Determining Mechanisms for MPKCs

Key Generation

Evaluation of coefficients

Public Maps

Evaluating a generic set of quadratic polynomials in $\mathbb{K} = \mathbb{F}_q$
usually as a matrix multiplying the vector of monomials

Private Maps

Rate-Determining Mechanisms for MPKCs

Key Generation

Evaluation of coefficients

Public Maps

Evaluating a generic set of quadratic polynomials in $\mathbb{K} = \mathbb{F}_q$

Private Maps

UOV	Solving linear systems of equations in $\mathbb{K} = \mathbb{F}_q$
Rainbow	Like UOV plus mini "Public Map"
TTS	Like UOV except public map is sparse
C^*	High powers in $\mathbb{L} = \mathbb{F}_{q^n}$
HFE	Equation solving in $\mathbb{L} = \mathbb{F}_{q^n}$ (general arithmetic)
TRMS	Inverse and multiplication in various $\mathbb{L} = \mathbb{F}_{q^k}$
ℓIC	Inverses and roots in $\mathbb{L}$
kHFE	Like HFE plus an elimination in $\mathbb{L}$

Practical Side of Computing

Moore's law

Transistor budget doubles every 18–24 months

Memory Latencies vs Clock Speeds

Year	Hi-End CPU	MHz	DRAM
1979	Z80	2	500ns
1984	80286	10	400ns
1989	80486	40	300ns
1994	Pentium	100	250ns
1999	Athlon	750	200ns
2004	Pentium 4	3800	160ns
2009	Core i7	*3200*	130ns

Are MPKCs Still Fast?

- Progress in high-precision arithmetic
  - In 80's, CPUs computed one 32-bit integer product every 15–20 cycles
  - In 2000, x86 CPUs computed one 64-bit product every 3–10 cycles
  - K10's and Core i7's today produces one 128-bit product every 2 cycles
  - Marvelous for ECC (and RSA)
- In contrast, progress in $\mathbb{F}_{2^q}$ arithmetic is *slow*
  - 6502 or 8051: a dozen cycles via three table look-ups
  - Modern x86: roughly same that many cycles
- Moore's law favors computation, not so much memories
  - Memory access speed increased at a snail's pace
- Wang et al. made life even harder for MPKCs
  - Forcing longer message digests
  - RSA untouched

Questions We Want to Answer

- Can all the extras on modern commodity CPUs help MPKCs as well?
- How have architectural changes affected implementation choices?
- If so, how do MPKCs compare to traditional PKCs today?

Arithmetic in $\mathbb{F}_{2^k}$

Multiplication Tables in Memory

Log/Exp Tables to a generator g

Bit-Slicing

Arithmetic in $\mathbb{F}_{2^k}$

Multiplication Tables in Memory

- One lookup per multiply
- Can result in large tables and pressure on cache
- Some parallelism can be achieved for $\mathbb{F}_4$ and $\mathbb{F}_{16}$.

Log/Exp Tables to a generator g

Bit-Slicing

Arithmetic in $\mathbb{F}_{2^k}$

Multiplication Tables in Memory

Log/Exp Tables to a generator g

- Compute xy as $g^{\log_g x + \log_g y}$ if neither is zero.
- Maximum of 3 lookups per mult, some logs can be pre-computed
- Require conditionals (bad!)

Bit-Slicing

Arithmetic in $\mathbb{F}_{2^k}$

Multiplication Tables in Memory

Log/Exp Tables to a generator g

Bit-Slicing

- Highly parallel — 32/64/128 multiplies at the same time
- Often requires rearranging of data
- Parameters can result in awkward dimensions like $1 + (\text{word size})$
- Require Conditionals or jump tables.

Arithmetic in $\mathbb{F}_{2^k}$

Multiplication Tables in Memory

Becomes attractive again if parallel lookups available.

Log/Exp Tables to a generator g

Bit-Slicing

- Highly parallel — 32/64/128 multiplies at the same time
- Often requires rearranging of data
- Parameters can result in awkward dimensions like $1 +$ (word size)
- Require Conditionals or jump tables.

SSE, the X86 Vector Instruction Set Extensions

- SSE: Streaming SIMD Extensions
  - SIMD: Single Instruction Multiple Data
- Most useful: SSE2 integer instructions
  - Work on 16 `xmm` 128-bit registers
  - As packed 8-, 16-, 32- or 64-bit operands
  - Move `xmm` to/from `xmm`, memory (even unaligned), x86 registers
  - Shuffle data and pack/unpack on vector data
  - Bit-wise logical operations like AND, OR, NOT, XOR
  - Shift left, right logical/arithmetic by units, or entire `xmm` byte-wise
  - Add/subtract on 8-, 16-, 32- and 64-bits
  - Multiply 16-bit and 32-bits in various ways
- SSSE3's PSHUFB (16 nibble-to-byte lookup in 1 cycle) and PALIGNR (256-bit bytewise rotation) quite powerful

PSHUFB in SSSE3

- "Packed Shuffle Bytes"
  - Source: $(x_0, \ldots, x_{15})$
  - Destination: $(y_0, \ldots, y_{15})$
  - Result: $(y_{x_0 \bmod 32}, \ldots, y_{x_{15} \bmod 32})$, treating $x_{16}, \ldots, x_{31}$ as 0

Speeding Up MPKCs over $\mathbb{F}_{16}$

- TT : 16×16 table, with $TT_{i,j} = i * j, 0 \leq i, j < 16$
- To compute $a\mathbf{v}$, $a \in \mathbb{F}_{16}, \mathbf{v} \in (\mathbb{F}_{16})^{16}$
  - `xmm` $\leftarrow$ a-th row of TT
  - $a\mathbf{v} \leftarrow$ PSHUFB `xmm`,$\mathbf{v}$
- Works similarly for $\mathbf{a} \in (\mathbb{F}_{16})^2, \mathbf{v} \in (\mathbb{F}_{16})^{32}$
  - Need to unpack, do PSHUFBs, then pack
- Delivers $2\times$ performance over simple bit slicing in private map evaluation of rainbow and TTS
- Some other platforms also have similar instructions
  - AMD's SSE5: PPERM (superset of PSHUFB)
  - IBM POWER AltiVec/VMX: PERMU

- TL : 256×16 table, with $TL_{i,j} = i * j, 0 \le i < 256,\ 0 \le j < 16$
- TH : 256×16 table, with $TH_{i,j} = i * (16j), 0 \le i < 256,\ 0 \le j < 16$
- To compute $a\mathbf{v}$, $a \in \mathbb{F}_{256}, \mathbf{v} \in (\mathbb{F}_{256})^{16}$
  - $a\mathbf{v}_i = a(16\lfloor \mathbf{v}_i/16 \rfloor) + a(\mathbf{v}_i \bmod 16), 0 \le i < 16$
- $\mathbf{v}'_i \leftarrow a(16\lfloor \mathbf{v}_i/16 \rfloor)$
  - $\mathbf{v}'_i \leftarrow \lfloor \mathbf{v}_i/16 \rfloor$ (SHIFT)
  - `xmm` $\leftarrow$ a-th row of TH
  - $\mathbf{v}' \leftarrow$ PSHUFB `xmm`,$\mathbf{v}'$
- $\mathbf{v}_i \leftarrow a(\mathbf{v}_i \bmod 16)$
  - $\mathbf{v}_i \leftarrow \mathbf{v}_i \bmod 16$ (AND)
  - `xmm` $\leftarrow$ a-th row of TL
  - $\mathbf{v} \leftarrow$ PSHUFB `xmm`,$\mathbf{v}$
- $a\mathbf{v} \leftarrow \mathbf{v} + \mathbf{v}'$ (OR)

Some Interesting Design Choices

System and Architecture-Dependent Stuff

- Key Generation
- Matrix-to-Vector-Multiply and Evaluating Public Maps
- Tower Field Arithmetic
- System- and Equation-Solving
  - Pre-scripted Gröbner Basis Computation
  - Iterative Methods instead of Gaussian Eliminations
  - Cantor-Zassenhaus instead of Berlekamp

Key Generation

Matsumoto-Imai's notaton: $z_k := \sum_i w_i \left[P_{ik} + Q_{ik} w_i + \sum_{j<i} R_{ijk} w_j \right].$

Usual Way: as differentials of public map $\mathcal{P} = (p_1, \ldots, p_m)$

for $q > 2$, we choose any $a \neq 0, 1$ and get

$$\begin{aligned} Q_{ik} &:= (a(a-1))^{-1} (p_k(a\mathbf{v}_i) - a p_k(\mathbf{v}_i)) \\ P_{ik} &:= p_k(\mathbf{v}_i) - Q_{ik} \\ R_{ijk} &:= p_k(\mathbf{v}_i + \mathbf{v}_j) - Q_{ik} - Q_{jk} - P_{ik} - P_{jk} \end{aligned}$$

For $\mathbb{F}_2$, it becomes

$$\begin{aligned} P_{ik} &:= p_k(\mathbf{v}_i) \\ R_{ijk} &:= p_k(\mathbf{v}_i + \mathbf{v}_j) - P_{ik} - P_{jk} \end{aligned}$$

($\mathbf{v}_i$ means the unit vector on the i-th direction)

Key Generation

Matsumoto-Imai's notaton: $z_k := \sum_i w_i \left[P_{ik} + Q_{ik} w_i + \sum_{j<i} R_{ijk} w_j \right].$

Usual Way: as differentials of public map $\mathcal{P} = (p_1, \ldots, p_m)$

For TTS and other sparse central $\mathcal{Q}$: by brute force

$$
\begin{aligned}
P_{ik} &= \sum_{h=0}^{m-1} \left[(\mathrm{M}_T)_{kh} \left((\mathrm{M}_S)_{hi} + \sum_{p\, x_\alpha x_\beta \text{ in } q_h} p \left((\mathrm{M}_S)_{\alpha i} (\mathbf{c}_S)_\beta + (\mathbf{c}_S)_\alpha (\mathrm{M}_S)_{\beta i} \right) \right) \right] \\
Q_{ik} &= \sum_{h=0}^{m-1} \left[(\mathrm{M}_T)_{kh} \left(\sum_{p\, x_\alpha x_\beta \text{ in } q_h} p\, (\mathrm{M}_S)_{\alpha i} (\mathrm{M}_S)_{\beta i} \right) \right] \\
R_{ijk} &= \sum_{h=0}^{m-1} \left[(\mathrm{M}_T)_{kh} \left(\sum_{p\, x_\alpha x_\beta \text{ in } q_h} p \left((\mathrm{M}_S)_{\alpha i} (\mathrm{M}_S)_{\beta j} + (\mathrm{M}_S)_{\alpha j} (\mathrm{M}_S)_{\beta i} \right) \right) \right]
\end{aligned}
$$

Evaluating Public Maps

Naive Way (and on μP's still)

$$z_k = \sum_i w_i \left[P_{ik} + Q_{ik} w_i + \sum_{i<j} R_{ijk} w_j \right]$$

For better memory access pattern

1. $\mathbf{c} \leftarrow [\mathbf{w}^T, (w_i w_j)_{i \le j}]^T$
2. $\mathbf{z} \leftarrow \mathbf{Pc}$, where $\mathbf{P}$ is the $m \times n(n+3)/2$ public-key matrix

How to do Matrix-to-Vector mults

Microcontrollers Naively

Somewhat newer CPUs Bit-slicing for $\mathbb{F}_{2^k}$

With more cache Big look-up tables (with nibble-slicing)

Newest architectures More or less naively, with SSE*

MPKCs over Odd Prime Fields

MPKCs over Odd Prime Fields

Are you out of your mind?

- `XOR` is easy, addition mod q is not.
- How can it possibly be faster?

MPKCs over Odd Prime Fields

Are you out of your mind?

- XOR is easy, addition mod q is not.
- How can it possibly be faster?

It's more than about speed

- Good for defending against Gröbner basis attacks
  - The field equation $X^q - X = 0$ becomes much less useful
- SSE* gives you parallel arithmetic on small integers,
  - and you only need to parallelize 4 or 8 at a time.
- Do you know how many 18-bit multipliers there are on an FPGA?

Basic Building Blocks for Speeding Up Odd MPKCs

- IMULHIb: the upper half in a signed product of two b-bit words
- Useful for computing $\lfloor xy/2^b \rfloor$
  - For $-2^{b-1} \leq x \leq 2^{b-1} - (q-1)/2$
  - $t \leftarrow$ IMULHIb $\lfloor 2^b/q \rfloor, x + \lfloor (q-1)/2 \rfloor$
  - $y \leftarrow x - qt$ computes $y = x \bmod q, |y| \leq q$
- For $q = 31$ and $b = 16$, we can do even better
  - For $-32768 \leq x \leq 32752$
  - $t \leftarrow$ IMULHI16 $2114, x + 15$
  - $y \leftarrow x - 31t$ computes $y = x \bmod 31, -16 \leq y \leq 15$

Speeding Up Matrix-to Vector Mults

- PMADDWD: Packed Multiply and Add, Word to Double-word
  - Source: $(x_0, \ldots, x_7)$
  - Destination: $(y_0, \ldots, y_7)$
  - Result: $(x_0 y_0 + x_1 y_1, x_2 y_2 + x_3 y_3, x_4 y_4 + x_5 y_5, x_6 y_6 + x_7 y_7)$
- Helpful in evaluating $\mathbf{z} = \mathbf{Pc}$, piece by piece
  - Let $\mathbf{Q}$ be a 4×2 submatrix of $\mathbf{P}$
  - $\mathbf{d}^T$ be the corresponding 2×1 submatrix of $\mathbf{c}$
  - r1 $\leftarrow (Q_{11}, Q_{12}, Q_{21}, Q_{22}, Q_{31}, Q_{32}, Q_{41}, Q_{42})$
  - r2 $\leftarrow (d_1, d_2, d_1, d_2, d_1, d_2, d_1, d_2)$
  - PMADDWD r1, r2 computes $\mathbf{Qd}$
  - Continue in 32-bits until reduction $\bmod q$
- Saves a few $\bmod q$ operations and delivers $1.5\times$ performance

Big look-up tables for matrix multiplication

As suggested by Berbain *et al*, SAC 2006

- Pre-compute $a\mathbf{v}$ for each column $\mathbf{v}$ in any constant matrix
- Read off the appropriately offset vector as needed
- Can nibble-slice $\mathbb{F}_{16}/\mathbb{F}_{256}$ into $\mathbb{F}_{16}/\mathbb{F}_4$
- Obviously minimizes the need for operations

Big look-up tables for matrix multiplication

As suggested by Berbain *et al*, SAC 2006

- Pre-compute $a\mathbf{v}$ for each column $\mathbf{v}$ in any constant matrix
- Read off the appropriately offset vector as needed
- Can nibble-slice $\mathbb{F}_{16}/\mathbb{F}_{256}$ into $\mathbb{F}_{16}/\mathbb{F}_4$
- Obviously minimizes the need for operations

Unbelievably ...

Slower than SSE on Core 2 45nm and Core i7 (or K10 45nm for mod31)!

Big look-up tables for matrix multiplication

As suggested by Berbain *et al*, SAC 2006

- Pre-compute $a\mathbf{v}$ for each column $\mathbf{v}$ in any constant matrix
- Read off the appropriately offset vector as needed
- Can nibble-slice $\mathbb{F}_{16}/\mathbb{F}_{256}$ into $\mathbb{F}_{16}/\mathbb{F}_4$
- Obviously minimizes the need for operations

Unbelievably ...

Slower than SSE on Core 2 45nm and Core i7 (or K10 45nm for mod31)!

When L2 isn't fast enough

- SSE instructions have a reverse throughput of 1 cycle today
- memory access is linear when using SSE
- L2 latency 20+ cycles; LUT reads not regular enough
- We are still trying to amend this with manual pre-fetching

Inversion in $\mathbb{F}_{31}$

- On C2 and Ci7: can use two SSE lookups with some extra work.
- On K8/K10: $x \mapsto x^{29}$
  - $y \leftarrow x * x * x \bmod 31$ $(y = x^3)$
  - $y \leftarrow x * y * y \bmod 31$ $(y = x^7)$
  - $y \leftarrow y * y \bmod 31$ $(y = x^{14})$
  - $y \leftarrow x * y * y \bmod 31$ $(y = x^{29})$
- Deliver $2\times$ performance over serial table look-ups!

Remarks on Getting More Performance

Laziness often leads to optimality

- Do not always need the tightest range
- The less reductions, the better!
- The less memory access, the better!
- The more regular memory access, the better!
- Packing $\mathbb{F}_q$-blocks into binary can use more bits than necessary as long as the map is injective and convenient to compute

Wiedemann vs. Gauss Elimination mod q

- How to solve a medium-sized dense linear system?
  - Wiedemann iterative solver for $\mathbf{A}\mathbf{x} = \mathbf{b}$
    - ★ Compute $\mathbf{z}\mathbf{A}^i\mathbf{b}$ for some $\mathbf{z}$
    - ★ Compute minimal polynomial using Berlekamp-Massey
  - Requires $O(2n^3)$ field multiplications
  - Straightforward Gauss elimination requires $O(n^3/3)$
- However, Wiedemann involves much less reductions modulo q
- Result: Wiedemann beats Gauss by a factor of 2!

Big Tower Fields mod q

- $\mathbb{F}_{q^k}$ isomorphic to $\mathbb{F}_q\,[t]/p(t)$, $\deg p = k$ and p irreducible
- For $k|(q-1)$ and a few other cases, $p(t) = t^k - a$ for a small a.
  - $> 2\times$ reduction performance over cases where p has 3 terms
  - $X \mapsto X^q$ becomes trivial to compute
  - Multiplication is straightforward, S:M ratio is between 0.75 and 0.92.
  - Inversion: (again) raising to the $(q^k - 2)$-th power!
  - For some tower of tower fields such as $\mathbb{F}_{31^{30}}$, can use Karatsuba.
- Square roots computed via Tonelli-Shanks Example: in $\mathbb{F}_{31^9}$ we raise to the $\frac{1}{4}\left(31^9+1\right)$-th power

$$\begin{array}{llll} i. & \mathtt{temp1} & := & (((\mathtt{input})^2)^2)^2, \\ ii. & \mathtt{t2} & := & (\mathtt{t1})^2 * ((\mathtt{t1})^2)^2, \\ iii. & \mathtt{t2} & := & \left[\mathtt{t2} * ((\mathtt{t2})^2)^2\right]^{31}, \\ iv. & \mathtt{t2} & := & \mathtt{t2} * (\mathtt{t2})^{31}, \\ v. & \mathtt{result} & := & \mathtt{t1} * \mathtt{t2} * \left((\mathtt{t2})^{31}\right)^{31}; \end{array}$$

and note that this shares some steps with inversion.

Some Performance numbers

Microarchitecture	MULT	SQ	INV	SQRT	INV+SQRT
C2 (65nm)	234	194	2640	4693	6332
C2+ (45nm)	145	129	1980	3954	5244
K8 (Athlon 64)	397	312	5521	8120	11646
K10 (Phenom)	242	222	2984	5153	7170

As an illustration of how we are doing, 128-way bitsliced multiplication with multi-stage Karatsuba and Toom in $\mathbb{F}_{2^{88}}$ with `djb`-class code is about 4 times faster on the K10.

To Solve Equation(s) in a Big Tower Field mod q

Scripted Gröbner Basis Computation

From 3 quadratic equations in 3 variables, we in succession run Gaussian eliminations on matrices of dimensions 3×10, 11×19, 8×16, 5×13, with many coefficients that we know to be zero in advance, to reach a degree-8 equation. You can call this a tailored matrix-$\mathbf{F_4}$.

Cantor-Zassenhaus (instead of Berlekamp)

1. Replace $u(X)$ by $\gcd(u(X), X^{q^k} - X)$ so that u splits in $\mathbb{L}$.
   1. Compute and tabulate $X^d \bmod u(X), \ldots, X^{2d-2} \bmod u(X)$.
   2. Compute $X^q \bmod u(X)$ via square-and-multiply.
   3. Compute and tabulate $X^{qi} \bmod u(X)$ for $i = 2, 3, \ldots, d-1$.
   4. Compute $X^{q^i} \bmod u(X)$ for $i = 2, 3, \ldots, k$, then $X^{q^k} \bmod u(X)$.
2. Do $\gcd\left(v(X)^{(q^k-1)/2} - 1, u(X)\right)$ for random $v(X)$ with $\deg v < \deg u$, to find nontrivial factor $\geq \frac{1}{2}$ of the time; repeat as needed.

Anything else New For $\mathbb{F}_{2^k}$?

Anything else New For $\mathbb{F}_{2^k}$?

Not Really.

Anything else New For $\mathbb{F}_{2^k}$?

Not Really.

Ok, So we implemented some

- Karatsuba-type implementations for tower fields
- Parallel bitslicing for $\mathbb{F}_{2^k}$ useful for MPKCs
- More SSSE3 parallelization using `PSHUB`

But no sense talking such with so many sado-masochistic bitslicers here!

Performance Comparison on Intel Q9550

Scheme	Result	PubKey	PriKey	KeyGen	PubMap	PriMap
RSA (1024 bits)	128 B	128 B	1024 B	27.2 ms	26.9 μs	806.1 μs
4HFE-p (31,10)	68 B	23 KB	8 KB	4.1 ms	6.8 μs	659.7 μs
3HFE-p (31,9)	67 B	7 KB	5 KB	0.8 ms	2.3 μs	60.5 μs
RSA (1024 bits)	128 B	128 B	1024 B	26.4 ms	22.4 μs	813.5 μs
ECDSA (160 bits)	40 B	40 B	60 B	0.3 ms	409.2 μs	357.8 μs
C^*-p (pFLASH)	37 B	72 KB	5 KB	28.7 ms	97.9 μs	473.6 μs
3IC-p (31,18,1)	36 B	35 KB	12 KB	4.2 ms	11.7 μs	256.2 μs
Rainbow (31,24,20,20)	43 B	57 KB	150 KB	120.4 ms	17.7 μs	70.6 μs
TTS (31,24,20,20)	43 B	57 KB	16 KB	13.7 ms	18.4 μs	14.2 μs

Measured using SUPERCOP: System for unified performance evaluation related to cryptographic operations and primitives. `http://bench.cr.yp.to/supercop.html`, April 2009.

Conclusions and Remarks

- Take-away point: Odd MPKCs worth studying!
  - Algebraic attacks become harder
  - Friendly to mainstream computing devices
    - ★ X86 CPUs have vector instructions
    - ★ High-end FPGAs have multiplier IPs
    - ★ Can be good for many-core GPUs (NVIDIA, ATI/AMD, Larrabee)
- It is very important to tune to your architecture.
- MPKCs still competitive speedwise, including on 8051s.
- When Intel's new vector instruction set comes out, it's likely to double the MPKC throughput per cycle too.

Future work

- Implement for new CPUs and instructions (`PCLMULQDQ`).
- Implement on Graphic cards and all that.
- Implement some side-channel-attack resistant versions?

Collaborators

- I had help from these Students/Assistants
  - Anna Inn-Tung Chen, U of Michigan
  - Chia-Hsin Owen Chen, MIT
  - Ming-Shing Chen, Nat'l Taiwan University, Taiwan
  - Tien-Ren Chen, Nat'l Immigration Agency, Taiwan
  - Yen-Hung Chen, ASUStek, Taiwan

- Colleagues I worked with
  - Chen-Mou Chen, Nat'l Taiwan University, Taiwan
  - Jiun-Ming Chen, Nat'l Cheng-Kung University, Taiwan

Collaborators

- I had help from these Students/Assistants
  - Anna Inn-Tung Chen, U of Michigan
  - Chia-Hsin Owen Chen, MIT
  - Ming-Shing Chen, Nat'l Taiwan University, Taiwan
  - Tien-Ren Chen, Nat'l Immigration Agency, Taiwan
  - Yen-Hung Chen, ASUStek, Taiwan
  - Eric Li-Hsiang Kuo, Academia Sinica, Taiwan
  - Frost Yu-Shuang Lee, U of Michigan
- Colleagues I worked with
  - Chen-Mou Cheng, Nat'l Taiwan University, Taiwan
  - Jiun-Ming Chen, Nat'l Cheng-Kung University, Taiwan
  - Jintai Ding, University of Cincinnati, USA
  - Lih-Chung Wang, Nat'l Dong-Hua University, Taiwan
  - Christopher Wolf, Ruhr University Bochum, Germany

Thanks for Listening!

- Questions or comments?