Breaking ECC2K-130

Daniel V. Bailey, Lejla Batina, Daniel J. Bernstein,
Peter Birkner, Joppe W. Bos, Hsieh-Chung Chen,
Chen-Mou Cheng, Gauthier van Damme,
Giacomo de Meulenaer, Luis Julian Dominguez Perez,
Junfeng Fan, Tim Güneysu, Frank Gürkaynak,
Thorsten Kleinjung, Tanja Lange, Nele Mentens,
Ruben Niederhagen, Christof Paar, Francesco Regazzoni,
Peter Schwabe, Leif Uhsadel, Anthony Van Herrewege,
Bo-Yin Yang,
and several individuals and institutions donating computer time

2010.01.12

The Certicom challenges

1997: Certicom announces several ECDLP prizes:

> *The Challenge is to compute the ECC private keys from the given list of ECC public keys and associated system parameters. This is the type of problem facing an adversary who wishes to completely defeat an elliptic curve cryptosystem.*

Objectives stated by Certicom:

- Increase community's understanding of ECDLP difficulty.
- Confirm theoretical comparisons of ECC and RSA.
- Help users select suitable key sizes.
- Compare ECDLP difficulty for $\mathbf{F}_{2^m}$ and $\mathbf{F}_p$.
- Compare $\mathbf{F}_{2^m}$ ECDLP difficulty for random and Koblitz.
- Stimulate research in algorithmic number theory.

The Certicom challenges, level 0: exercises

Bits	Name	"Estimated number of machine days"	Prize
79	ECCp-79	146	book
79	ECC2-79	352	book
89	ECCp-89	4360	book
89	ECC2-89	11278	book
97	ECC2K-95	8637	$5000
97	ECCp-97	71982	$5000
97	ECC2-97	180448	$5000

Certicom believes that it is feasible that the 79-bit exercises could be solved in a matter of hours, the 89-bit exercises could be solved in a matter of days, and the 97-bit exercises in a matter of weeks using a network of 3000 computers.

The Certicom challenges, level 1

Bits	Name	"Estimated number of machine days"	Prize
109	ECC2K-108	1300000	$10000
109	ECCp-109	9000000	$10000
109	ECC2-109	21000000	$10000
131	ECC2K-130	2700000000	$20000
131	ECCp-131	23000000000	$20000
131	ECC2-131	66000000000	$20000

The 109-bit Level I challenges are feasible using a very large network of computers. The 131-bit Level I challenges are expected to be infeasible against realistic software and hardware attacks, unless of course, a new algorithm for the ECDLP is discovered.

The Certicom challenges, level 2

Bits	Name	“Estimated number of machine days”	Prize
163	ECC2K-163	320000000000000	$30000
163	ECCp-163	2300000000000000	$30000
163	ECC2-163	6200000000000000	$30000
191	ECCp-191	48000000000000000000	$40000
191	ECC2-191	100000000000000000000	$40000
239	ECC2K-238	92000000000000000000000000	$50000
239	ECCp-239	1400000000000000000000000000	$50000
239	ECC2-238	2100000000000000000000000000	$50000
359	ECCp-359	$\approx \infty$	$100000

The Level II challenges are infeasible given today’s computer technology and knowledge.

Broken challenges

1997: Baisley and Harley break ECCp-79.
1997: Harley et al. break ECC2-79.
1998: Harley et al. break ECCp-89.
1998: Harley et al. break ECC2-89.
1998: Harley et al. (1288 computers) break ECCp-97.
1998: Harley et al. (200 computers) break ECC2K-95.
1999: Harley et al. (740 computers) break ECC2-97.
2000: Harley et al. (9500 computers) break ECC2K-108.

Updated `cert_ecc_challenge.pdf` still says "109-bit Level I challenges are feasible using a very large network . . . 131-bit Level I challenges are expected to be infeasible" etc.

2002: Monico et al. (10000 computers) break ECCp-109.
2004: Monico et al. (2600 computers) break ECC2-109.

The target: ECC2K-130

The Koblitz curve $y^2 + xy = x^3 + 1$ over
$\mathbf{F}_{2^{131}} = \mathbf{F}_2[z]/(z^{131} + z^{13} + z^2 + z + 1)$
has 4ℓ points, where ℓ is the prime
$680564733841876926932320129493409985129 \approx 2^{129}$.

Certicom generated two random points on the curve
and multiplied them by 4, obtaining the following points P, Q:

```
x(P) = 05 1C99BFA6 F18DE467 C80C23B9 8C7994AA
y(P) = 04 2EA2D112 ECEC71FC F7E000D7 EFC978BD
x(Q) = 06 C997F3E7 F2C66A4A 5D2FDA13 756A37B1
y(Q) = 04 A38D1182 9D32D347 BD0C0F58 4D546E9A
```

The challenge:
Find an integer $k \in \{0, 1, \ldots, \ell - 1\}$ such that $[k]P = Q$.

Arithmetic on ECC2K-130

Elements of the Koblitz curve: a special point P_∞, and each $(x_1, y_1) \in \mathbf{F}_{2^{131}} \times \mathbf{F}_{2^{131}}$ satisfying $y_1^2 + x_1 y_1 = x_1^3 + 1$.

Arithmetic on ECC2K-130

Elements of the Koblitz curve: a special point P_∞, and each $(x_1, y_1) \in \mathbf{F}_{2^{131}} \times \mathbf{F}_{2^{131}}$ satisfying $y_1^2 + x_1 y_1 = x_1^3 + 1$.

How to add P_1, P_2:

- $P_1 + P_\infty = P_\infty + P_1 = P_1$; $(x_1, y_1) + (x_1, y_1 + x_1) = P_\infty$.
- If $x_1 \neq 0$ the double $[2](x_1, y_1) = (x_3, y_3)$ is given by

$$x_3 = \lambda^2 + \lambda, \;\; y_3 = \lambda(x_1 + x_3) + y_1 + x_3, \;\; \text{where } \lambda = x_1 + \frac{y_1}{x_1}.$$

- If $x_1 \neq x_2$ the sum $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$ is given by

$$x_3 = \lambda^2 + \lambda + x_1 + x_2, \;\; y_3 = \lambda(x_1 + x_3) + y_1 + x_3, \;\; \text{where } \lambda = \frac{y_1 + y_2}{x_1 + x_2}.$$

  Cost: 1**I** (inversion), 2**M** (multiplications), 1**S** (squaring).
- For an overview of how to perform these operations in other coordinate systems see the EFD:

  `http://hyperelliptic.org/EFD/`

Koblitz curves – the Frobenius endomorphism

- In 1991 Koblitz pointed out that scalar multiplications $[m]P$ can be computed faster on curves

  $$E_a : y^2 + xy = x^3 + ax^2 + 1,$$

  where a is restricted to $\{0, 1\}$.
- The main observation is that if $(x_1, y_1) \in E_a(\mathbf{F}_{2^n})$ then also the point $\sigma(P) = (x_1^2, y_1^2)$ is in $E_a(\mathbf{F}_{2^n})$ and these points are related by

  $$\sigma^2(P) + [\mu]\sigma(P) + [2]P = P_\infty,$$

  where $\mu = 1$ for $a = 0$ and $\mu = -1$ for $a = 1$. The map σ extends the Frobenius automorphism of $\mathbf{F}_{2^n}$ to $E_a(\mathbf{F}_{2^n})$ and is thus called the Frobenius endomorphism of E_a.

Koblitz curves – usage of σ

- Koblitz, Meyer–Staffelbach, and Solinas showed that in the computation of $[m]P$ the double-and-add method can be replaced by a σ-and-add method. Instead of needing $\log_2 m$ doublings the Frobenius-based method needs $\log_2 m$ applications of σ.
  - This means that instead of 1 **I**+ 2**M**+ 1 **S** per bit of m only 2**S** are needed per bit of m.
- The cost per addition does not change for these curves.
  - A NAF version reduces the number of additions to $\log_2 m/3$ on average without needing any precomputations.
  - Analogues of (binary) windowing methods exist in Frobenius variants.

ECRYPT vs. ECC2K-130

2009.02: VAMPIRE VAM1 sets its sights on ECC2K-130.
Optimizing ECC attacks isn't far from optimizing ECC.
Exactly how difficult is breaking ECC2K-130?

ECRYPT vs. ECC2K-130

2009.02: VAMPIRE VAM1 sets its sights on ECC2K-130.
Optimizing ECC attacks isn't far from optimizing ECC.
Exactly how difficult is breaking ECC2K-130?

2009.12: With our latest implementations,
ECC2K-130 is breakable in a year on average

- by 3039 3GHz Core 2 CPUs,
- or by 2716 GTX 295 GPUs,
- or by 2466 Cell CPUs,
- or by 2026 XC3S5000 FPGAs,
- or by (estimated) 200 ASICs costing 60000 EUR,
- or by any combination thereof.

ECRYPT vs. ECC2K-130

2009.02: VAMPIRE VAM1 sets its sights on ECC2K-130.
Optimizing ECC attacks isn't far from optimizing ECC.
Exactly how difficult is breaking ECC2K-130?

2009.12: With our latest implementations,
ECC2K-130 is breakable in a year on average

- by 3039 3GHz Core 2 CPUs,
- or by 2716 GTX 295 GPUs,
- or by 2466 Cell CPUs,
- or by 2026 XC3S5000 FPGAs,
- or by (estimated) 200 ASICs costing 60000 EUR,
- or by any combination thereof.

This is what Certicom called "infeasible"?

The most important ECDL algorithms

No known index-calculus attack applies to ECC2K-130.
But can still use generic attacks that work in any group:

- The Pohlig–Hellman attack reduces the hardness of the ECDLP to the hardness of the ECDLP in the largest subgroup of prime order: in this case order ℓ.
- The Baby-Step Giant-Step attack finds the logarithm in $\sqrt{\ell}$ steps and $\sqrt{\ell}$ storage by comparing $Q - [jt]P$ (the giant steps) to a sorted list of all $[i]P$ (the baby steps), where $0 \leq i, j \leq \lceil \sqrt{\ell} \rceil$ and $t = \lceil \sqrt{\ell} \rceil$.
- Pollard's rho and kangaroo methods also use $O(\sqrt{\ell})$ steps but require constant memory—much less expensive! The kangaroo method would be faster if the logarithm were known to lie in a short interval; for us rho is best.
- Multiple-target attacks: not relevant here.

Pollard's rho method

Make a pseudo-random walk in $\langle P \rangle$, where the next step depends on current point: $P_{i+1} = f(P_i)$.

Birthday paradox: Randomly choosing from ℓ elements picks one element twice after about $\sqrt{\pi\ell/2}$ draws.

The walk has now entered a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.

Pollard's rho method

Make a pseudo-random walk in $\langle P \rangle$, where the next step depends on current point: $P_{i+1} = f(P_i)$.

Birthday paradox: Randomly choosing from ℓ elements picks one element twice after about $\sqrt{\pi\ell/2}$ draws.

The walk has now entered a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.

Assume that for each point we know $a_i, b_i \in \mathbf{Z}/\ell\mathbf{Z}$ so that $P_i = [a_i]P + [b_i]Q$. Then $P_i = P_j$ means that

$$[a_i]P + [b_i]Q = [a_j]P + [b_j]Q \quad \text{so} \quad [b_i - b_j]Q = [a_j - a_i]P.$$

If $b_i \neq b_j$ the ECDLP is solved: $k = (a_j - a_i)/(b_i - b_j)$.

Pollard's rho method

Make a pseudo-random walk in $\langle P \rangle$, where the next step depends on current point: $P_{i+1} = f(P_i)$.

Birthday paradox: Randomly choosing from ℓ elements picks one element twice after about $\sqrt{\pi\ell/2}$ draws.

The walk has now entered a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.

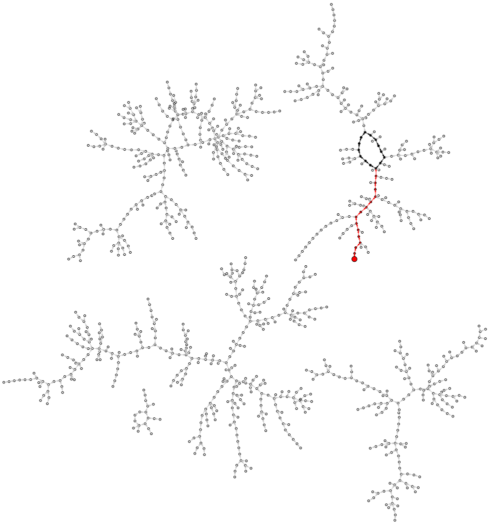
Assume that for each point we know $a_i, b_i \in \mathbf{Z}/\ell\mathbf{Z}$ so that $P_i = [a_i]P + [b_i]Q$. Then $P_i = P_j$ means that

$$[a_i]P + [b_i]Q = [a_j]P + [b_j]Q \quad \text{so} \quad [b_i - b_j]Q = [a_j - a_i]P.$$

If $b_i \neq b_j$ the ECDLP is solved: $k = (a_j - a_i)/(b_i - b_j)$.

e.g. "Adding walk": Start with $P_0 = P$ and put $f(P_i) = P_i + [c_r]P + [d_r]Q$ where $r = h(P_i)$.

A rho within a random walk on 1024 elements



Method is called rho method because of the shape.

Parallel collision search

Running Pollard's rho method on N computers gives speedup of $\approx \sqrt{N}$ from increased likelihood of finding collision.

Want better way to spread computation across clients.
Want to find collisions between walks on different machines, without frequent synchronization!

Parallel collision search

Running Pollard's rho method on N computers gives speedup of $\approx \sqrt{N}$ from increased likelihood of finding collision.

Want better way to spread computation across clients.
Want to find collisions between walks on different machines, without frequent synchronization!

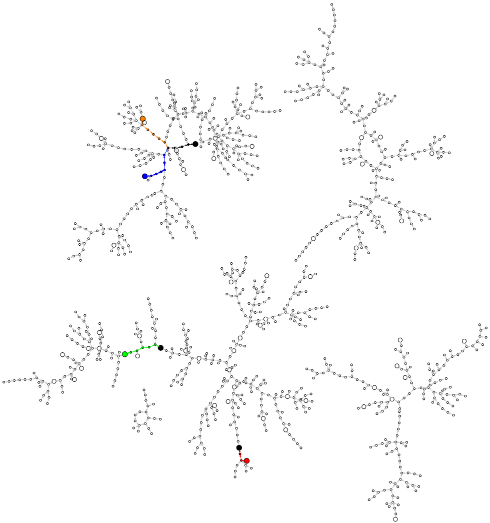
Perform walks with different starting points but same update function on all computers. If same point is found on two different computers also the following steps will be the same.

Terminate each walk once it hits a distinguished point.
Attacker chooses definition of distinguished points;
can be more or less frequent. Do not wait for cycle.

Collect all distinguished points in central database.

Expect collision within $O(\sqrt{\ell}/N)$ iterations. Speedup $\approx N$.

Short walks ending in distinguished points



Blue and orange paths found the same distinguished point!

Equivalence classes

P and $-P$ have same x-coordinate. Search for x-coordinate collision. Search space for collisions is only $\ell/2$; this gives factor $\sqrt{2}$ speedup … provided that $f(P_i) = f(-P_i)$.

Solution: $f(P_i) = |P_i| + [c_r]P + [d_r]Q$ where $r = h(|P_i|)$. Define $|P_i|$ as, e.g., lexicographic minimum of $P_i, -P_i$.

Equivalence classes

P and $-P$ have same x-coordinate. Search for x-coordinate collision. Search space for collisions is only $\ell/2$; this gives factor $\sqrt{2}$ speedup . . . provided that $f(P_i) = f(-P_i)$.

Solution: $f(P_i) = |P_i| + [c_r]P + [d_r]Q$ where $r = h(|P_i|)$. Define $|P_i|$ as, e.g., lexicographic minimum of $P_i, -P_i$.

Problem: this walk can run into fruitless cycles! If there are S different steps $[c_r]P + [d_r]Q$ then with probability $1/(4S^2)$ the following happens:

$$\begin{aligned} P_{i+2} &= -P_{i+1} + [c_r]P + [d_r]Q \\ &= -(-P_i + [c_r]P + [d_r]Q) + [c_r]P + [d_r]Q = P_i. \end{aligned}$$

Get $P_{i+3} = P_{i+1}$, $P_{i+4} = P_i$, etc.
Can detect and fix. Some effort; not exactly $\sqrt{2}$ speedup.

Equivalence classes for Koblitz curves

More savings: P and $\sigma^i(P)$ have $x(\sigma^j(P)) = x(P)^{2^j}$.

Reduce number of iterations by another factor $\sqrt{n}$ by considering equivalence classes under Frobenius and $\pm$.

Need to ensure that the iteration function satisfies $f(P_i) = f(\pm\sigma^j(P_i))$.

Equivalence classes for Koblitz curves

More savings: P and $\sigma^i(P)$ have $x(\sigma^j(P)) = x(P)^{2^j}$.

Reduce number of iterations by another factor $\sqrt{n}$ by considering equivalence classes under Frobenius and ±.

Need to ensure that the iteration function satisfies $f(P_i) = f(\pm\sigma^j(P_i))$.

Could again define adding walk starting from $|P_i|$.
Redefine $|P_i|$ as canonical representative of class containing P_i: e.g., lexicographic minimum of P_i, $-P_i$, $\sigma(P_i)$, etc.

Iterations now involve many squarings,
but squarings are not so expensive in characteristic 2.

Our choice of iteration function

In normal basis, $x(P)$ and $x(P)^{2^j}$ have same Hamming weight $\mathrm{HW}(x(P))$. Convenient to use this to determine iteration.

Our iteration function—note that $\mathrm{HW}(x(P))$ is always even:

$$P_{i+1} = P_i + \sigma^j(P_i),$$

where $j = (\mathrm{HW}(x(P))/2 \bmod 8) + 3$.

This nicely avoids short, fruitless cycles.

Iteration consists of

- computing the Hamming weight $\mathrm{HW}(x(P))$ of the normal-basis representation of $x(P)$;
- checking for distinguished points (is $\mathrm{HW}(x(P)) \leq 34$?);
- computing j and $P + \sigma^j(P)$.

Analysis of our choice of iteration function

For a perfectly random walk $\sqrt{\pi\ell/2}$ iterations are expected on average. Have $\ell \approx 2^{131}/4$ for ECC2K-130.

A perfectly random walk on classes under $\pm$ and Frobenius would reduce number of iterations by $\sqrt{2 \cdot 131}$.

Analysis of our choice of iteration function

For a perfectly random walk $\sqrt{\pi\ell/2}$ iterations are expected on average. Have $\ell \approx 2^{131}/4$ for ECC2K-130.

A perfectly random walk on classes under $\pm$ and Frobenius would reduce number of iterations by $\sqrt{2 \cdot 131}$.

Loss of randomness from having only 8 choices of j.
Further loss from non-randomness of Hamming weights: Hamming weights around 66 are much more likely than at the edges; effect still noticeable after reduction to 8 choices.

Analysis of our choice of iteration function

For a perfectly random walk $\sqrt{\pi\ell/2}$ iterations are expected on average. Have $\ell \approx 2^{131}/4$ for ECC2K-130.

A perfectly random walk on classes under $\pm$ and Frobenius would reduce number of iterations by $\sqrt{2 \cdot 131}$.

Loss of randomness from having only 8 choices of j.
Further loss from non-randomness of Hamming weights: Hamming weights around 66 are much more likely than at the edges; effect still noticeable after reduction to 8 choices.

Our analysis shows that the total loss is 6.9993%.
This loss is justified by the very fast iteration function.

Average number of iterations for our attack against ECC2K-130: $\sqrt{\pi\ell/(2 \cdot 2 \cdot 131)} \cdot 1.069993 \approx 2^{60.9}$.

Field arithmetic required

Look more closely at costs of iteration function:

- one normal-basis Hamming-weight computation;
- one application of σ^j for some $j \in \{3, 4, \ldots, 10\}$: $\leq 20\mathbf{S}$ if computed as a series of squarings;
- one elliptic-curve addition: $1\mathbf{I} + 2\mathbf{M} + 1\mathbf{S} + 7\mathbf{a}$ in affine coordinates.

Field arithmetic required

Look more closely at costs of iteration function:

- one normal-basis Hamming-weight computation;
- one application of σ^j for some $j \in \{3, 4, \ldots, 10\}$:
  $\leq 20\mathbf{S}$ if computed as a series of squarings;
- one elliptic-curve addition:
  $1\mathbf{I} + 2\mathbf{M} + 1\mathbf{S} + 7\mathbf{a}$ in affine coordinates.

"Montgomery's trick": handle N iterations in parallel;
batch $N\mathbf{I}$ into $1\mathbf{I} + (3N - 3)\mathbf{M}$.

Summary: Each iteration costs
$\leq (1/N)(\mathbf{I} - 3\mathbf{M}) + 5\mathbf{M} + 21\mathbf{S} + 7\mathbf{a}$
plus a Hamming-weight computation in normal basis.

How to perform these operations most efficiently?

Bit operations

We can compute an iteration using a straight-line (branchless) sequence of $70467 + 70263/N$ two-input bit operations.
e.g. 71880 bit operations/iteration for $N = 51$.

Bit operations: "AND" and "XOR";
i.e., multiplication and addition in $\mathbf{F}_2$.

Bit operations

We can compute an iteration using a straight-line (branchless) sequence of $70467 + 70263/N$ two-input bit operations. e.g. 71880 bit operations/iteration for $N = 51$.

Bit operations: "AND" and "XOR"; i.e., multiplication and addition in $\mathbf{F}_2$.

Compare to 34061 bit operations (131^2 ANDs + 130^2 XORs) for one schoolbook multiplication of two 131-bit polynomials.

Bit operations

We can compute an iteration using a straight-line (branchless) sequence of $70467 + 70263/N$ two-input bit operations.
e.g. 71880 bit operations/iteration for $N = 51$.

Bit operations: “AND” and “XOR”;
i.e., multiplication and addition in $\mathbf{F}_2$.

Compare to 34061 bit operations (131^2 ANDs + 130^2 XORs) for one schoolbook multiplication of two 131-bit polynomials.

Fortunately, there are faster multiplication methods.
`http://binary.cr.yp.to/m.html`: $M(131) \le 11961$ where $M(n)$ is minimum # bit operations for n-bit multiplication.

Polynomial basis vs. normal basis

We could use the polynomial basis $1, z, z^2, \ldots, z^{130}$ of $\mathbf{F}_{2^{131}} = \mathbf{F}_2[z]/(z^{131} + z^{13} + z^2 + z + 1)$.

Or we could use the "type-2 optimal normal basis" $\zeta + 1/\zeta, \zeta^2 + 1/\zeta^2, \zeta^4 + 1/\zeta^4, \ldots, \zeta^{2^{130}} + 1/\zeta^{2^{130}}$ where ζ is a primitive 263rd root of 1.

Polynomial basis vs. normal basis

We could use the polynomial basis $1, z, z^2, \ldots, z^{130}$ of $\mathbf{F}_{2^{131}} = \mathbf{F}_2[z]/(z^{131} + z^{13} + z^2 + z + 1)$.

Or we could use the "type-2 optimal normal basis" $\zeta + 1/\zeta, \zeta^2 + 1/\zeta^2, \zeta^4 + 1/\zeta^4, \ldots, \zeta^{2^{130}} + 1/\zeta^{2^{130}}$ where ζ is a primitive 263rd root of 1.

Well-known advantages of normal basis:

- The 21**S** are free.
- The conversion to normal basis is free.

Polynomial basis vs. normal basis

We could use the polynomial basis $1, z, z^2, \ldots, z^{130}$ of $\mathbf{F}_{2^{131}} = \mathbf{F}_2[z]/(z^{131} + z^{13} + z^2 + z + 1)$.

Or we could use the "type-2 optimal normal basis" $\zeta + 1/\zeta, \zeta^2 + 1/\zeta^2, \zeta^4 + 1/\zeta^4, \ldots, \zeta^{2^{130}} + 1/\zeta^{2^{130}}$ where ζ is a primitive 263rd root of 1.

Well-known advantages of normal basis:

- The 21**S** are free.
- The conversion to normal basis is free.

Well-known disadvantage:

- Normal-basis multipliers are painfully slow.

Polynomial basis vs. normal basis

We could use the polynomial basis $1, z, z^2, \ldots, z^{130}$ of $\mathbf{F}_{2^{131}} = \mathbf{F}_2[z]/(z^{131} + z^{13} + z^2 + z + 1)$.

Or we could use the "type-2 optimal normal basis" $\zeta + 1/\zeta, \zeta^2 + 1/\zeta^2, \zeta^4 + 1/\zeta^4, \ldots, \zeta^{2^{130}} + 1/\zeta^{2^{130}}$ where ζ is a primitive 263rd root of 1.

Well-known advantages of normal basis:

- The 21**S** are free.
- The conversion to normal basis is free.

Well-known disadvantage:

- Normal-basis multipliers are painfully slow.

Harley et al. tried normal basis for ECC2K-95 and ECC2K-108 but reported that polynomial basis was much faster.

The Shokrollahi multiplier

2007 Shokrollahi, von zur Gathen–Shokrollahi–Shokrollahi:
Can convert from a length-n type-2 optimal normal basis
$\zeta + 1/\zeta, \zeta^2 + 1/\zeta^2, \zeta^4 + 1/\zeta^4, \ldots$
to $1, \zeta + 1/\zeta, (\zeta + 1/\zeta)^2, (\zeta + 1/\zeta)^3, \ldots$
using $\approx (1/2)(n \lg n)$ bit operations; similar for inverse.
$\approx M(n+1) + 2n \lg n$ bit operations for normal-basis mult.

The Shokrollahi multiplier

2007 Shokrollahi, von zur Gathen–Shokrollahi–Shokrollahi:
Can convert from a length-n type-2 optimal normal basis
$\zeta + 1/\zeta, \zeta^2 + 1/\zeta^2, \zeta^4 + 1/\zeta^4, \ldots$
to $1, \zeta + 1/\zeta, (\zeta + 1/\zeta)^2, (\zeta + 1/\zeta)^3, \ldots$
using $\approx (1/2)(n \lg n)$ bit operations; similar for inverse.
$\approx M(n+1) + 2n \lg n$ bit operations for normal-basis mult.

New: Save bit operations by streamlining the conversion.
$M(131) + 1559$ for size-131 normal-basis multiplication.

The Shokrollahi multiplier

2007 Shokrollahi, von zur Gathen–Shokrollahi–Shokrollahi: Can convert from a length-n type-2 optimal normal basis $\zeta + 1/\zeta, \zeta^2 + 1/\zeta^2, \zeta^4 + 1/\zeta^4, \ldots$ to $1, \zeta + 1/\zeta, (\zeta + 1/\zeta)^2, (\zeta + 1/\zeta)^3, \ldots$ using $\approx (1/2)(n \lg n)$ bit operations; similar for inverse. $\approx M(n+1) + 2n \lg n$ bit operations for normal-basis mult.

New: Save bit operations by streamlining the conversion. $M(131) + 1559$ for size-131 normal-basis multiplication.

Save even more bit operations by mixing type-2 optimal normal basis, type-2 optimal polynomial basis. $\approx M(n) + n \lg n$ to multiply; $M(131) + 917$ for $n = 131$. $\approx (1/2) n \lg n$ before and after squarings; 325 for $n = 131$.

For more details: 2009 Bernstein–Lange, forthcoming.

Bit operations vs. reality

Conventional wisdom: Counting bit operations is too simple. Analyzing and optimizing performance on Phenom II, Core 2, GTX 295, Cell, XC3S5000, etc. is much more work:

- Natural units are words (32 bits or more), not bits.
- Not limited to AND, XOR. Can use, e.g., array lookups.
- Extracting individual bits is feasible but relatively slow.
- Need to copy code into small "cache". Copies are slow.
- Need to copy data into small "cache"—and then into tiny "register set". ("Load" into register; "store" from register.)
- On "two-operand" CPUs, each arithmetic operation overwrites one of its input registers.
- Can perform several *independent* operations at once.

Example: The Cell implementation(s)

Cell teams: Joppe Bos and Thorsten Kleinjung, Lausanne;
Peter Schwabe, Eindhoven, later joined by Ruben Niederhagen.

Tried two different implementation strategies for the Cell:

- traditional “non-bitsliced” (using polynomial basis);
- “bitsliced” (polynomial basis at first, then normal basis).

Cell inside PlayStation 3 has 6 accessible “SPU” cores.
Each core runs at 3.2GHz: i.e., 3.2 billion cycles/second.
$\leq$ 1 arithmetic operation/cycle. $\leq$ 1 load or store/cycle.
Large register set: 128 registers, each 128 bits.
Small memory on each core: 256 kilobytes.

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)

- 06 Aug: 6488 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 19 Aug: 1426 (non-bitsliced)

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)

- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)
- 30 Aug: 1180 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)
- 30 Aug: 1180 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)

- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)

- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)

- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)

- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)

- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)

- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)
- 07 Oct: 956 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)
- We surrender!

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)
- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)
- 07 Oct: 956 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)
- We surrender!

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)
- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)
- 07 Oct: 956 (bitsliced)
- 12 Oct: 903 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)

- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)

- We surrender!
- Please stop!

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)

- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)
- 07 Oct: 956 (bitsliced)
- 12 Oct: 903 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)

- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)

- We surrender!
- Please stop!

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)

- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)
- 07 Oct: 956 (bitsliced)
- 12 Oct: 903 (bitsliced)
- 13 Oct: 871 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)

- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)

- We surrender!
- Please stop!
- Ow, ow, it hurts!

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)

- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)
- 07 Oct: 956 (bitsliced)
- 12 Oct: 903 (bitsliced)
- 13 Oct: 871 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)

- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)

- We surrender!
- Please stop!
- Ow, ow, it hurts!

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)

- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)
- 07 Oct: 956 (bitsliced)
- 12 Oct: 903 (bitsliced)
- 13 Oct: 871 (bitsliced)
- 14 Oct: 844 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)
- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)
- We surrender!
- Please stop!
- Ow, ow, it hurts!
- Okay, go on, it's for the good of the project . . . [urk]

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)
- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)
- 07 Oct: 956 (bitsliced)
- 12 Oct: 903 (bitsliced)
- 13 Oct: 871 (bitsliced)
- 14 Oct: 844 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)

- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)

- We surrender!
- Please stop!
- Ow, ow, it hurts!
- Okay, go on, it's for the good of the project . . . [urk]

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)

- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)
- 07 Oct: 956 (bitsliced)
- 12 Oct: 903 (bitsliced)
- 13 Oct: 871 (bitsliced)
- 14 Oct: 844 (bitsliced)
- 15 Oct: 789 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)

- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)

- We surrender!
- Please stop!
- Ow, ow, it hurts!
- Okay, go on, it's for the good of the project . . . [urk]

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)

- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)
- 07 Oct: 956 (bitsliced)
- 12 Oct: 903 (bitsliced)
- 13 Oct: 871 (bitsliced)
- 14 Oct: 844 (bitsliced)
- 15 Oct: 789 (bitsliced)
- 29 Oct: 749 (bitsliced)

Cycles per iteration on each SPU

- 31 Jul: 2565 (non-bitsliced)
- 03 Aug: 1735 (non-bitsliced)

- 19 Aug: 1426 (non-bitsliced)
- 19 Aug: 1293 (non-bitsliced)
- 04 Sep: 1157 (non-bitsliced)

- We surrender!
- Please stop!
- Ow, ow, it hurts!
- Okay, go on, it's for the good of the project . . . [urk]

- 06 Aug: 6488 (bitsliced)
- 10 Aug: 1587 (bitsliced)
- 13 Aug: 1389 (bitsliced)

- 30 Aug: 1180 (bitsliced)
- 05 Sep: 1051 (bitsliced)
- 07 Sep: 1047 (bitsliced)
- 07 Oct: 956 (bitsliced)
- 12 Oct: 903 (bitsliced)
- 13 Oct: 871 (bitsliced)
- 14 Oct: 844 (bitsliced)
- 15 Oct: 789 (bitsliced)
- 29 Oct: 749 (bitsliced)
- 10 Jan: 746 (bitsliced)

Overall speedup

2466 Cell CPUs for a year: i.e., 900000 machine days.
Recall Certicom's estimate: 2700000000 machine days.

Overall speedup

2466 Cell CPUs for a year: i.e., 900000 machine days.
Recall Certicom's estimate: 2700000000 machine days.

"That's unfair! Computers ten years ago were very slow!"

Overall speedup

2466 Cell CPUs for a year: i.e., 900000 machine days.
Recall Certicom's estimate: 2700000000 machine days.

"That's unfair! Computers ten years ago were very slow!"
Indeed, Certicom's "machine" was a 100MHz Pentium.
Today's Cell has several cores, each running at 3.2GHz.
Scale by counting cycles . . . and we're still 15× faster.

Overall speedup

2466 Cell CPUs for a year: i.e., 900000 machine days.
Recall Certicom's estimate: 2700000000 machine days.

"That's unfair! Computers ten years ago were very slow!"
Indeed, Certicom's "machine" was a 100MHz Pentium.
Today's Cell has several cores, each running at 3.2GHz.
Scale by counting cycles . . . and we're still 15× faster.

"Computers ten years ago didn't do much work per cycle!"

Overall speedup

2466 Cell CPUs for a year: i.e., 900000 machine days.
Recall Certicom's estimate: 2700000000 machine days.

"That's unfair! Computers ten years ago were very slow!"
Indeed, Certicom's "machine" was a 100MHz Pentium.
Today's Cell has several cores, each running at 3.2GHz.
Scale by counting cycles . . . and we're still 15× faster.

"Computers ten years ago didn't do much work per cycle!"
True for the Pentium . . . but not for Harley's Alpha,
which had 64-bit registers, 4 instructions/cycle, etc.

Harley's ECC2K-108 software uses 1651 Alpha cycles/iteration.
We ran the same software on a Core 2: 1800 cycles/iteration.
We also wrote our own polynomial-basis ECC2K-108 software:
on the same Core 2, fewer than 500 cycles/iteration.

Our servers

Is ECC2K-130 feasible for a serious attacker? Obviously.

Our servers

Is ECC2K-130 feasible for a serious attacker? Obviously.
Is ECC2K-130 feasible for a big public Internet project? Yes.

Our servers

Is ECC2K-130 feasible for a serious attacker? Obviously.
Is ECC2K-130 feasible for a big public Internet project? Yes.
Is ECC2K-130 feasible for us? We think so.
To prove it we're running the attack.

Our servers

Is ECC2K-130 feasible for a serious attacker? Obviously.
Is ECC2K-130 feasible for a big public Internet project? Yes.
Is ECC2K-130 feasible for us? We think so.
To prove it we're running the attack.

Eight central servers (at TU Eindhoven) receive points,
pre-sort the points into 8192 RAM buffers,
flush the buffers to 8192 disk files.

Periodically read each file into RAM, sort, find collisions.
Also double-check random samples for validity.

Written to disk so far: 12 gigabytes.
Reading, sorting, finding collisions: 10 seconds per server.

Client-server communication

Each report is compressed to 8-byte seed, 8-byte hash.
Negligible cost for server to recompute distinguished point.

We have clusters computing points at several sites worldwide.
Currently 3648 separate streams of data entering servers.

Client-server communication

Each report is compressed to 8-byte seed, 8-byte hash.
Negligible cost for server to recompute distinguished point.

We have clusters computing points at several sites worldwide.
Currently 3648 separate streams of data entering servers.

Lightweight data-transfer protocol:

- Client collects 64 reports into a 1024-byte block;
  adds 8 bytes identifying client site, stream, block position;
  sends resulting packet to server through UDP.
- Server has 4-byte stream state, sends back 8-byte ack.

Client-server communication

Each report is compressed to 8-byte seed, 8-byte hash.
Negligible cost for server to recompute distinguished point.

We have clusters computing points at several sites worldwide.
Currently 3648 separate streams of data entering servers.

Lightweight data-transfer protocol:

- Client collects 64 reports into a 1024-byte block;
  adds 8 bytes identifying client site, stream, block position;
  sends resulting packet to server through UDP.
- Server has 4-byte stream state, sends back 8-byte ack.

Each packet is encrypted, authenticated, verified, decrypted
using `http://nacl.cace-project.eu`; costs 16 bytes.
Total block cost: 1090-byte IP packet plus 66-byte ack.

Clients so far

2010.01.10 tally of reports to servers:

- from site “ℓ”: 7309224960 bytes; Cell (PS3), Core 2
- from site “L”: 4280884224 bytes; Opteron
- from site “G”: 2126951424 bytes; ?
- from site “j”: 987218944 bytes; Cell blade
- from site “t”: 582718464 bytes; Core 2, Phenom II
- from site “e”: 504352768 bytes; Core 2
- from site “b”: 237230080 bytes
- from site “d”: 33841152 bytes
- from site “a”: 32056320 bytes
- from site “c”: 14313472 bytes
- from site “n”: 4313088 bytes
- from site “z”: 4027392 bytes

Streams from site “p” and “i” just started. Working on collecting more clusters, building FPGA clusters, continuing to speed up the implementations (especially GPU).

Get more details, and watch our progress!

http://eprint.iacr.org/2009/466:
"The Certicom challenges ECC2-X" (SHARCS 2009)—
analysis of ECC2K-130, ECC2-131, ECC2K-163, ECC2-163
with ASIC, FPGA, Cell, Core2 implementation details.

http://eprint.iacr.org/2009/541:
"Breaking ECC2K-130"; continues to be improved;
more platforms, better speeds, running the attack.

http://ecc-challenge.info:
anonymous web page (but ours, really!), including
graph of number of points reported to the servers.

https://twitter.com/ECCchallenge:
anonymous Twitter page with the latest announcements.

Hope to finish attack in first half of 2010.