Batch binary Edwards

D. J. Bernstein
University of Illinois at Chicago

NSF ITR–0716498

Nonnegative elements of $\mathbf{Z}$:

$$\begin{array}{rll}
0 & \text{meaning} & 0 \\
1 & \text{meaning} & 2^0 \\
10 & \text{meaning} & 2^1 \\
11 & \text{meaning} & 2^0 + 2^1 \\
100 & \text{meaning} & 2^2 \\
101 & \text{meaning} & 2^0 + 2^2 \\
110 & \text{meaning} & 2^1 + 2^2 \\
111 & \text{meaning} & 2^0 + 2^1 + 2^2 \\
1000 & \text{meaning} & 2^3 \\
1001 & \text{meaning} & 2^0 + 2^3 \\
1010 & \text{meaning} & 2^1 + 2^3
\end{array}$$

etc.

Addition: $2^e + 2^e = 2^{e+1}$.

Multiplication: $2^e 2^f = 2^{e+f}$.

Elements of $\mathbf{F}_2[t]$:

0	meaning	0
1	meaning	t^0
10	meaning	t^1
11	meaning	$t^0 + t^1$
100	meaning	t^2
101	meaning	$t^0 + t^2$
110	meaning	$t^1 + t^2$
111	meaning	$t^0 + t^1 + t^2$
1000	meaning	t^3
1001	meaning	$t^0 + t^3$
1010	meaning	$t^1 + t^3$

etc.

Addition: $t^e + t^e = 0$.

Multiplication: $t^e t^f = t^{e+f}$.

Modular arithmetic in $\mathbf{Z}$:
e.g., $\mathbf{Z}/12 = \{0, 1, \ldots, 11\}$
with $+$, $\cdot$ reduced mod 12.
Modular arithmetic in $\mathbf{F}_2[t]$:
e.g., $\mathbf{F}_2[t]/(t^4 + t) =$
$\{0, 1, \ldots, t^3 + t^2 + t + 1\}$
with $+$, $\cdot$ reduced mod $t^4 + t$.

Primes of $\mathbf{Z}$: $2, 3, 5, 7, 11, \ldots$.
Primes of $\mathbf{F}_2[t]$: $t, t + 1,$
$t^2 + t + 1, t^3 + t + 1, \ldots$.

Can build finite fields from
arithmetic modulo primes.
e.g. $\mathbf{Z}/(2^{127} - 1)$.
e.g. $\mathbf{F}_2[t]/(t^{127} + t + 1)$.

Many decades of literature have explored number-theoretic analogies between $\mathbf{Z}$ and $\mathbf{F}_2[t]$.

Often $\mathbf{F}_2[t]$ is simpler than $\mathbf{Z}$.

e.g. Breaking $\mathbf{F}_2[t]$ RSA is much faster than breaking $\mathbf{Z}$ RSA.

Fastest known algorithm to compute prime factors of a b-bit element of $\mathbf{Z}$: worst-case time $2^{b^{1/3+o(1)}}$.

Fastest known algorithm to compute prime factors of a b-bit element of $\mathbf{F}_2[t]$: time $2^{(c+o(1))\lg b}$ with $c < 2$.

In *some* cryptographic contexts, $\mathbf{F}_2[t]$ and $\mathbf{Z}$ have same security.

e.g. Message authentication using shared secret key.

Take $k = \mathbf{Z}/(2^{127} - 1)$
or $k = \mathbf{F}_2[t]/(t^{127} + t + 1)$.

Message $m \in k[x]$.
One-time key $(r, s) \in k^2$:
use for only one message!
Authenticator $s + rm(r) \in k$.

Standard security proof $\Rightarrow$
chance of successful forgery
$< 2^{-128} \cdot \#\{\text{attack bits}\}$.

Hardware designers prefer $\mathbf{F}_2[t]$ because its costs are lower for the same security level. Example: GMAC, inside GCM.

Lack of carries ($t^e + t^e = 0$) makes addition and multiplication smaller and faster; also makes squaring much smaller and faster.

Hardware designers prefer $\mathbf{F}_2[t]$ because its costs are lower for the same security level. Example: GMAC, inside GCM.

Lack of carries ($t^e + t^e = 0$) makes addition and multiplication smaller and faster; also makes squaring much smaller and faster.

But software is different! For many years, $\mathbf{Z}$ has held crypto software speed records. Examples: Poly1305, UMAC.

Why is $\mathbf{Z}$ faster than $\mathbf{F}_2[t]$?
Standard answer: CPUs are designed for video games, movie decompression, etc.

These applications rely heavily on multiplication in $\mathbf{Z}$.
CPUs devote large area to $\mathbf{Z}$ multiplication circuits, speeding up these applications.

Conventional wisdom:
Advantages of $\mathbf{F}_2[t]$ are outweighed by speed of CPU's built-in $\mathbf{Z}$ multipliers, especially big 64-bit multipliers.

Next generation of Intel CPUs devote some circuit area to $\mathbf{F}_2[t]$ multiplier "PCLMULQDQ".

Maybe still slower than $\mathbf{Z}$, but maybe fast enough to make $\mathbf{F}_2[t]$ set new speed records for some crypto applications.

Next generation of Intel CPUs devote some circuit area to $\mathbf{F}_2[t]$ multiplier "PCLMULQDQ".

Maybe still slower than $\mathbf{Z}$, but maybe fast enough to make $\mathbf{F}_2[t]$ set new speed records for some crypto applications.

This talk: New speed records for elliptic-curve cryptography on **current** Intel CPUs.
These records use $\mathbf{F}_2[t]$.

User: busy server bottlenecked by public-key cryptography.

Throughput: tens of thousands of $n, P \mapsto nP$ per second.

Latency: a few milliseconds.
Software handles input batch $(n_1, P_1), (n_2, P_2), \ldots, (n_{128}, P_{128})$.
No need for related inputs.

Security level: $\approx 2^{128}$, assuming standard conjectures; twist-secure; constant-time.

Free software: `binary.cr.yp.to`

New software is bitsliced.
Advantage: low-cost shifts.
Disadvantage: high-cost branches.

Low-cost shifts allow
very fast squarings, reductions.

Low-cost shifts minimize
overhead for Karatsuba etc.

See paper for details of
improved Karatsuba, Toom;
often 20% fewer operations
than previous literature.

What about branches?

2007 Bernstein–Lange:
The Edwards addition law

$$x_3 = \frac{x_1 y_2 + y_1 x_2}{1 + d x_1 x_2 y_1 y_2},$$

$$y_3 = \frac{y_1 y_2 - x_1 x_2}{1 - d x_1 x_2 y_1 y_2}.$$

works for *all* inputs
on the Edwards curve
$x^2 + y^2 = 1 + d x^2 y^2$ over $\mathbf{Z}/p$
if d is non-square in $\mathbf{Z}/p$.

Also extremely fast.

Completeness helps against various side-channel attacks; simplifies implementations; and helps bitslicing.

Same for binary curves?

Completeness helps against various side-channel attacks; simplifies implementations; and helps bitslicing.

Same for binary curves?

2008 B.–L.–Rezaeian Farashahi: Fast complete addition on "binary Edwards curve"
$d(x+x^2+y+y^2) = (x+x^2)(y+y^2)$
over field $\mathbf{F}_2[t]/(\cdots)$
if $x^2 + x + d$ has no roots.

Continuing work on fast $\mathbf{F}_2[t]$:

1. Subfield applications.
Maybe $\approx 1.5\times$ faster ECC?

2. Genus-2 applications.
Maybe $\approx 1.5\times$ faster than ECC?

3. Better code scheduling.
Maybe $\approx 2\times$ faster?

4. Other curve applications;
e.g., faster ECC2K-130.

5. Other crypto applications;
e.g., faster McEliece.