New speed records
for point multiplication

D. J. Bernstein

Thanks to:
University of Illinois at Chicago
NSF CCR–9983950
Alfred P. Sloan Foundation

640838 Pentium M cycles
to compute a 32-byte secret
shared by Dan and Tanja,
given Dan’s 32-byte secret key \( n \)
and Tanja’s 32-byte public key \( K \).

All known attacks: \( > 2^{128} \) cycles.

This is the new speed record
for high-security Diffie-Hellman.

Encrypt and authenticate messages
using hash of shared secret as key.
Diffie-Hellman is the bottleneck
if total message length is short.
640838 Pentium M cycles to compute a 32-byte secret shared by Dan and Tanja, given Dan’s 32-byte secret key \( n \) and Tanja’s 32-byte public key \( K \).

All known attacks: \( > 2^{128} \) cycles.

This is the new speed record for high-security Diffie-Hellman.

Encrypt and authenticate messages using hash of shared secret as key. Diffie-Hellman is the bottleneck if total message length is short.

Curve25519 is the elliptic curve
\[ y^2 = x^3 + 486662x + 1 \]
mod the prime 2^{255} – 19.

640838 Pentium M (695) cycles to compute \( x \)-coordinate of \( t \) multiple of \((K, n)\), given \( K \in \{0, 1, \ldots, 2^{256} - 1\} \) and \( n \in 2^{254} + 8\{0, 1, \ldots, 2^{254}\} \).

624786 Athlon (622) cycles; 832457 Pentium III (686) cycles; 957904 Pentium 4 (f12) cycles.

I anticipate similar cycle counts for UltraSPARC, PowerPC, etc.
640838 Pentium M cycles to compute a 32-byte secret shared by Dan and Tanja, given Dan’s 32-byte secret key \( n \) and Tanja’s 32-byte public key \( K \).

All known attacks: \( > 2^{128} \) cycles.

This is the new speed record for high-security Diffie-Hellman.

Encrypt and authenticate messages using hash of shared secret as key. Diffie-Hellman is the bottleneck if total message length is short.

640838 Pentium M (695) cycles to compute \( x \)-coordinate of \( n \)th multiple of \( (K, \ldots) \) on Curve25519, given \( K \in \{0, 1, \ldots, 2^{256} - 1\} \) and \( n \in 2^{254} + 8\{0, 1, \ldots, 2^{251} - 1\} \).

Curve25519 is the elliptic curve \( y^2 = x^3 + 486662x^2 + x \) mod the prime \( 2^{255} - 19 \).

624786 Athlon (622) cycles; 832457 Pentium III (686) cycles; 957904 Pentium 4 (f12) cycles. I anticipate similar cycle counts for UltraSPARC, PowerPC, etc.
640838 Pentium M (695) cycles to compute \( x \)-coordinate of \( n \)th multiple of \((K, \ldots)\) on Curve25519, given \( K \in \{0, 1, \ldots, 2^{256} - 1\} \) and \( n \in 2^{254} + 8\{0, 1, \ldots, 2^{251} - 1\} \).

Curve25519 is the elliptic curve

\[
y^2 = x^3 + 486662x^2 + x
\]

mod the prime \( 2^{255} - 19 \).

624786 Athlon (622) cycles;
832457 Pentium III (686) cycles;
957904 Pentium 4 (f12) cycles.

I anticipate similar cycle counts for UltraSPARC, PowerPC, etc.

Immune to timing attacks, including cache-timing attacks, including hyperthreading attacks.
No data-dependent branches; no data-dependent indexing.

Software is in public domain.

16 kilobytes when compiled.

cr.yp.to/ecdh.html

No known patent problems.

For comparison, Brown et al.:
much smaller prime, \( 2^{192} - 2^{64} - 1 \);
780000 PII cycles; given; no timing-attack protection.
640838 Pentium M (695) cycles to compute $x$-coordinate of $n$th multiple of $(K,\ldots)$ on Curve25519, given $K \in \{0, 1, \ldots, 2^{256} - 1\}$ and $n \in 2^{254} + 8\{0, 1, \ldots, 2^{251} - 1\}$.

Curve25519 is the elliptic curve $y^2 = x^3 + 486662x^2 + x$ mod the prime $2^{255} - 19$.

624786 Athlon (622) cycles; 832457 Pentium III (686) cycles; 957904 Pentium 4 (f12) cycles. I anticipate similar cycle counts for UltraSPARC, PowerPC, etc.

Immune to timing attacks, including cache-timing attacks, including hyperthreading attacks. No data-dependent branches; no data-dependent indexing.

Software is in public domain. 16 kilobytes when compiled. cr.yp.to/ecdh.html

No known patent problems.

For comparison, Brown et al.: much smaller prime, $2^{192} - 2^{64} - 1$; 780000 PII cycles; $y$ given; no timing-attack protection.
Immune to timing attacks, including cache-timing attacks, including hyperthreading attacks. No data-dependent branches; no data-dependent indexing.

Software is in public domain. 16 kilobytes when compiled. cr.yp.to/ecdh.html

No known patent problems.

Where are the cycles going?
Focus today on Pentium M.
Fastest arithmetic on Pentium M uses floating-point operations: fp adds, fp subs, fp mults.

Each Pentium M cycle does \leq 1$ fp op.

Point multiplication:

For comparison, Brown et al.: much smaller prime, $2^{192} - 2^{64} - 1$; 780000 PII cycles; $y$ given; no timing-attack protection.

Understand cycle counts fairly well by simply counting fp ops.
Immune to timing attacks, including cache-timing attacks, including hyperthreading attacks. No data-dependent branches; no data-dependent indexing.

Software is in public domain. 16 kilobytes when compiled. cr.yp.to/ecdh.html

No known patent problems.

For comparison, Brown et al.: much smaller prime, $2^{192} - 2^{64} - 1$; 780000 PII cycles; $y$ given; no timing-attack protection.

Where are the cycles going?

Focus today on Pentium M.

Fastest arithmetic on Pentium M uses floating-point operations: fp adds, fp subs, fp mults.

Each Pentium M cycle does $\leq 1$ fp op.

Point multiplication: 640838 cycles. 589825 fp ops; $\approx 0.92$ per cycle.

Understand cycle counts fairly well by simply counting fp ops.
Immune to timing attacks, including cache-timing attacks, including hyperthreading attacks. No data-dependent branches; no data-dependent indexing.

Software is in public domain. 16 kilobytes when compiled.

Noknown patent problems.

For comparison, Brown et al.:
much smaller prime, \(2^{192} - 2^{64} - 1\);
780000 PII cycles;
given;
no timing-attack protection.

Avoiding all time variability to stop timing attacks:

1. For \(b \in \{0, 1\}\), compute as \(bx[1] + (1 - b)x[0] \) or similar.

Avoids data-dependent indexing.
Costs 36210 fp ops (6%).

2. Compute final reciprocal by Fermat, not extended Euclid.

Avoids data-dependent branching.

3. Don’t branch for remainders.
Allow non-least remainders.
No cost—this saves time!

Where are the cycles going?
Focus today on Pentium M.

Fastest arithmetic on Pentium M uses floating-point operations:
fp adds, fp subs, fp mults.

Each Pentium M cycle does \(\leq 1\) fp op.

Point multiplication: 640838 cycles.
589825 fp ops; \(\approx 0.92\) per cycle.

Understand cycle counts fairly well by simply counting fp ops.
Where are the cycles going?
Focus today on Pentium M.
Fastest arithmetic on Pentium M uses floating-point operations:
fp adds, fp subs, fp mults.
Each Pentium M cycle does \( \leq 1 \) fp op.
Point multiplication: 640838 cycles.
589825 fp ops; \( \approx 0.92 \) per cycle.
Understand cycle counts fairly well by simply counting fp ops.

Avoiding all time variability to stop timing attacks:
1. For \( b \in \{0, 1\} \), compute \( x[b] \) as \( bx[1] + (1 - b)x[0] \) or similar.
   Avoids data-dependent indexing.
   Costs 36210 fp ops (6%).
2. Compute final reciprocal by Fermat, not extended Euclid.
   Avoids data-dependent branching.
3. Don’t branch for remainders.
   Allow non-least remainders.
   No cost—this saves time!
Avoiding all time variability to stop timing attacks:

1. For $b \in \{0, 1\}$, compute $x[b]$ as $bx[1] + (1 - b)x[0]$ or similar. Avoids data-dependent indexing. Costs 36210 fp ops (6%).

2. Compute final reciprocal by Fermat, not extended Euclid. Avoids data-dependent branching.

3. Don’t branch for remainders. Allow non-least remainders. No cost—this saves time!

Main loop: 545700 fp ops (92.5%).
2140 times 255 iterations.
Reciprocal: 43821 fp ops (7.4%).
$41148 = 254 \cdot 162$ for 254 squarings;
$2673 = 11 \cdot 243$ for 11 more mults.
Additional work: 304 fp ops.

Inside one main-loop iteration:
$80 = 8 \cdot 10$ for 8 adds/subs;
$55$ for mult by 121665;
$648 = 4 \cdot 162$ for 4 squarings;
$1215 = 5 \cdot 243$ for 5 more mults;
$142$ for $bx[1] + (1 - b)x[0]$ etc.
Avoiding all time variability to stop timing attacks:

1. For $b \in \{0, 1\}$, compute $x[b]$ as $bx[1] + (1 - b)x[0]$ or similar. Avoids data-dependent indexing. Costs 36210 fp ops (6%).

2. Compute final reciprocal by Fermat, not extended Euclid. Avoids data-dependent branching.

3. Don’t branch for remainders. Allow non-least remainders. No cost—this saves time!

Main loop: 545700 fp ops (92.5%). 2140 times 255 iterations.

Reciprocal: 43821 fp ops (7.4%).
$41148 = 254 \cdot 162$ for 254 squarings;
$2673 = 11 \cdot 243$ for 11 more mults.

Additional work: 304 fp ops.

Inside one main-loop iteration:
$80 = 8 \cdot 10$ for 8 adds/subs;
$55$ for mult by 121665;
$648 = 4 \cdot 162$ for 4 squarings;
$1215 = 5 \cdot 243$ for 5 more mults;
$142$ for $bx[1] + (1 - b)x[0]$ etc.
Avoiding all time variability to stop timing attacks:

1. For $0, 1$, compute $x[b] + (1 - b)x[0]$ or similar. Avoids data-dependent indexing. Costs 36210 fp ops (6%).

2. Compute final reciprocal by Fermat, not extended Euclid. Avoids data-dependent branching.

3. Don't branch for remainders. Allow non-least remainders. No cost—this saves time!

Main loop: 545700 fp ops (92.5%). 2140 times 255 iterations.

Reciprocal: 43821 fp ops (7.4%). $41148 = 254 \cdot 162$ for 254 squarings; $2673 = 11 \cdot 243$ for 11 more mults.

Additional work: 304 fp ops.

Inside one main-loop iteration:
$80 = 8 \cdot 10$ for 8 adds/subs;
$55$ for mult by 121665;
$648 = 4 \cdot 162$ for 4 squarings;
$1215 = 5 \cdot 243$ for 5 more mults;
$142$ for $bx[1] + (1 - b)x[0]$ etc.

An integer mod $2^{255} 19$ is represented in radix $2^{25}$ as a sum of 10 fp numbers in specified ranges.

Add/sub: 10 fp adds/subs.
Delay reductions and carries!

Mult: poly mult using $10^2$ fp mults, $9^2$ fp adds; reduce using 9 fp mults, 9 fp adds; carry 11 times, each 4 fp adds; overall $2 \cdot 10^2 + 4 \cdot 9 + 3$ fp ops.

Squaring: start with $9^2$ fp doublings; then eliminate $9^2 + 9$ fp ops; overall $1 \cdot 10^2 + 6$ fp ops.
Main loop: 545700 fp ops (92.5%).
2140 times 255 iterations.

Reciprocal: 43821 fp ops (7.4%).
$41148 = 254 \cdot 162$ for 254 squarings;
$2673 = 11 \cdot 243$ for 11 more mults.

Additional work: 304 fp ops.

Inside one main-loop iteration:
$80 = 8 \cdot 10$ for 8 adds/subs;
55 for mult by 121665;
$648 = 4 \cdot 162$ for 4 squarings;
$1215 = 5 \cdot 243$ for 5 more mults;
142 for $bx[1] + (1 - b)x[0]$ etc.

An integer mod $2^{255} - 19$ is represented in radix $2^{25.5}$
as a sum of 10 fp numbers in specified ranges.

Add/sub: 10 fp adds/subs.
Delay reductions and carries!

Mult: poly mult using
$10^2$ fp mults, $9^2$ fp adds;
reduce using $9$ fp mults, $9$ fp adds;
carry 11 times, each 4 fp adds;
overall $2 \cdot 10^2 + 4 \cdot 10 + 3$ fp ops.

Squaring: start with $9$ fp doublings;
then eliminate $9^2 + 9$ fp ops;
overall $1 \cdot 10^2 + 6 \cdot 10 + 2$ fp ops.
545700 fp ops (92.5%).
2140 times 255 iterations.
Reciprocal: 43821 fp ops (7.4%).
41148 = 254 162 for 254 squarings;
2673 = 11 243 for 11 more mults.
Additional work: 304 fp ops.

Inside one main-loop iteration:
80 = 8 10 for 8 adds/subs;
55 for mult by 121665;
648 = 4 162 for 4 squarings;
1215 = 5 243 for 5 more mults;
142
An integer mod $2^{255} - 19$ is represented in radix $2^{25.5}$ as a sum of 10 fp numbers in specified ranges.

Add/sub: 10 fp adds/subs.
Delay reductions and carries!

Mult: poly mult using $10^2$ fp mults, $9^2$ fp adds;
reduce using 9 fp mults, 9 fp adds;
carry 11 times, each 4 fp adds;
overall $2 \cdot 10^2 + 4 \cdot 10 + 3$ fp ops.

Squaring: start with 9 fp doublings;
then eliminate $9^2 + 9$ fp ops;
overall $1 \cdot 10^2 + 6 \cdot 10 + 2$ fp ops.

How was the prime chosen?
Use prime close to power of 2 to save time in field operations.
Also reduces NFS exponent, so would need larger prime for traditional discrete-log systems; but doesn’t seem to affect ECDL.

Use prime not far below $2^{32}$ to avoid wasting bandwidth.

Comfortable security:
$2^{253} + 39$, $2^{253} + 51$, $2^{254} + 79$,
$2^{255} - 31$, $2^{255} - 95$. 
An integer mod $2^{255} - 19$ is represented in radix $2^{25.5}$ as a sum of 10 fp numbers in specified ranges.

Add/sub: 10 fp adds/subs. Delay reductions and carries!

Mult: poly mult using $10^2$ fp mults, $9^2$ fp adds; reduce using 9 fp mults, 9 fp adds; carry 11 times, each 4 fp adds; overall $2 \cdot 10^2 + 4 \cdot 10 + 3$ fp ops.

Squaring: start with 9 fp doublings; then eliminate $9^2 + 9$ fp ops; overall $1 \cdot 10^2 + 6 \cdot 10 + 2$ fp ops.

How was the prime chosen?

Use prime close to power of 2 to save time in field operations. Also reduces NFS exponent, so would need larger prime for traditional discrete-log systems; but doesn’t seem to affect ECDL.

Use prime not far below $2^{32k}$ to avoid wasting bandwidth.

Comfortable security, $k = 8$: $2^{253} + 39, 2^{253} + 51, 2^{254} + 79, 2^{255} - 31, 2^{255} - 19, 2^{255} + 95$. 
An integer $\mod 2^{255}$

19 is represented in radix $2^{255}$ as a sum of 10 fp numbers in specified ranges.

Add/sub: 10 fp adds/subs. Delay reductions and carries!

Mult: poly mult using $2^{10}$ fp mults, $92$ fp adds; reduce using $9$ fp mults, $9$ fp adds; carry $11$ times, each $4$ fp adds; overall $2 \cdot 10 + 4 \cdot 10 + 3$ fp ops.

Squaring: start with $9$ fp doublings; then eliminate $9$ fp ops; overall $1 \cdot 102 + 6 \cdot 10 + 2$ fp ops.

How was the prime chosen?

Use prime close to power of 2 to save time in field operations. Also reduces NFS exponent, so would need larger prime for traditional discrete-log systems; but doesn’t seem to affect ECDL.

Use prime not far below $2^{32k}$ to avoid wasting bandwidth.

Comfortable security, $k = 8$:

$2^{253} + 39$, $2^{253} + 51$, $2^{254} + 79$, $2^{255} - 31$, $2^{255} - 19$, $2^{255} + 95$.

Bender, Castagnoli, CRYPTO '89:

"$2^{127} + 24933$ is prime. ... For this curve which is convenient in computer arithmetic we also give ..."

I use the prime $2^{255} - 19$, convenient for the same reasons. No trouble from “shift and add” patent 5159632 filed 1991.09.17.
How was the prime chosen?

Use prime close to power of 2 to save time in field operations.

Also reduces NFS exponent, so would need larger prime for traditional discrete-log systems; but doesn’t seem to affect ECDL.

Use prime not far below $2^{32k}$ to avoid wasting bandwidth.

Comfortable security, $k = 8$:

$$2^{253} + 39, 2^{253} + 51, 2^{254} + 79,$$

$$2^{255} - 31, 2^{255} - 19, 2^{255} + 95.$$
How was the prime chosen?

Use prime close to power of 2 to save time in field operations. Also reduces NFS exponent, so would need larger prime for traditional discrete-log systems; but doesn't seem to affect ECDL. Use prime not far below $2^{32k}$ to avoid wasting bandwidth.

Bender, Castagnoli, CRYPTO '89: $2^{127} + 24933$ is prime.

... For this curve which is convenient in computer arithmetic we also give ...

I use the prime $2^{255} - 19$, convenient for the same reasons. No trouble from "shift and add" patent 5159632 filed 1991.09.17.

How was the curve chosen?

Use Montgomery shape $y^2 = x^3 + Ax^2 + x$ to save time in curve operations and to avoid square roots.

Choose $(A - 2)/4 = 0$, to save time in curve operations.

Montgomery's recursion

$z_1 = 1; \quad x_{2m} = (x_m^2 \mod p)$
$z_{2m} = 4x_mz_m(x_m^2 \mod p)$
$x_{2m+1} = 4(x_mx_m \mod p)$
$z_{2m+1} = 4(x_mz_m \mod p)$

then $n(K, \ldots) = (\ldots)$.
Bender, Castagnoli, CRYPTO '89:
“$2^{127} + 24933$ is prime.
... For this curve which is
convenient in computer arithmetic
we also give ...”

I use the prime $2^{255} - 19$,
convenient for the same reasons.
No trouble from “shift and add”

How was the curve chosen?
Use Montgomery shape
$y^2 = x^3 + Ax^2 + x$
to save time in curve operations
and to avoid square roots.

Choose $(A - 2)/4$ as small integer
to save time in curve operations.

Montgomery’s recursion: $x_1 = K;
z_1 = 1; x_{2m} = (x_m^2 - z_m^2)^2;
z_{2m} = 4x_mz_m(x_m^2 + Ax_mz_m + z_m^2);
x_{2m+1} = 4(x_mx_{m+1} - z_mz_{m+1})^2;
z_{2m+1} = 4(x_mz_{m+1} - z_mx_{m+1})^2K$;
then $n(K, \ldots) = (x_n/z_n, \ldots)$. 
How was the curve chosen?

Use Montgomery shape
\[ y^2 = x^3 + Ax^2 + x \]
to save time in curve operations and to avoid square roots.

Choose \((A - 2)/4\) as small integer to save time in curve operations.

Montgomery's recursion:
\[
\begin{align*}
x_1 &= K; \\
z_1 &= 1; \\
x_{2m} &= (x_m^2 - z_m^2)^2; \\
z_{2m} &= 4x_m z_m(x_m^2 + Ax_m z_m + z_m^2); \\
x_{2m+1} &= 4(x_m x_{m+1} - z_m z_{m+1})^2; \\
z_{2m+1} &= 4(x_m z_{m+1} - z_m x_{m+1})^2 K; \\
\end{align*}
\]
then \(n(K, \ldots) = (x_n/z_n, \ldots)\).
How was the curve chosen?

Use Montgomery shape
\[ y^2 = x^3 + Ax^2 + x \]

to save time in curve operations and to avoid square roots.

Choose \((A - 2)/4\) as small integer to save time in curve operations.

Montgomery’s recursion: \(x_1 = K;\)
\(z_1 = 1;\)
\(x_{2m} = (x_m^2 - z_m^2)^2;\)
\(z_{2m} = 4x_m z_m(x_m^2 + Ax_m z_m + z_m^2);\)
\(x_{2m+1} = 4(x_m x_{m+1} - z_m z_{m+1})^2;\)
\(z_{2m+1} = 4(x_m z_{m+1} - z_m x_{m+1})^2 K;\)
then \(n(K, \ldots) = (x_n/z_n, \ldots).\)
How was the curve chosen?
Use Montgomery shape
\[ 2 = 3 + 2 \]
to save time in curve operations and to avoid square roots.
Choose \( (x)^{4} \) as small integer to save time in curve operations.

Montgomery's recursion:
\[
1 = 1; \\
2 = (x^4)^2; \\
2 = 4(x^4)^2; \\
2 + 1 = 4(x^4)^2; \\
\text{then} \ (2) = (x^4). \\
\]

Reject A unless curve and twist orders are \( \{4 \cdot \text{prime} 8 \cdot \text{prime} \} \).
Montgomery shape forces 4; characteristic in \( 4 \mathbb{Z} + 1 \) forces 8.

For \( A = 486662 \): Curve has order 8 times prime \( p_1 = 2^{252} \).
The twist has order 4 times prime \( p_2 = 2^{253} \).
Reject $A$ unless curve and twist orders are $\{4 \cdot \text{prime}, 8 \cdot \text{prime}\}$. Montgomery shape forces $4$; characteristic in $4\mathbb{Z} + 1$ forces $4, 8$.

For $A = 486662$: Curve has order $8$ times prime $p_1 = 2^{252} + \cdots$. The twist has order $4$ times prime $p_2 = 2^{253} - \cdots$. 
Reject $A$ unless curve and twist orders are $\{4 \cdot \text{prime}, 8 \cdot \text{prime}\}$.
Montgomery shape forces 4; characteristic in $4\mathbb{Z} + 1$ forces 4, 8.

For $A = 486662$: Curve has order 8 times prime $p_1 = 2^{252} + \cdots$.
The twist has order 4 times prime $p_2 = 2^{253} - \cdots$.

For $A = 358990$: One prime is $2^{252}$, so user’s secret key $n \in 2^{254} + 8\{0, 1, \ldots\}$ could be 8 times that prime. Extremely unlikely, but annoys implementors, so reject this $A$. 
Reject $A$ unless curve and twist orders are $\{4 \cdot \text{prime}, 8 \cdot \text{prime}\}$.
Montgomery shape forces 4;
characteristic in $4\mathbb{Z} + 1$ forces 4, 8.

For $A = 486662$: Curve has order
8 times prime $p_1 = 2^{252} + \cdots$.
The twist has order
4 times prime $p_2 = 2^{253} - \cdots$.

For $A = 358990$:
One prime is $2^{252} - \cdots$,
so user’s secret key
$n \in 2^{254} + 8\{0, 1, \ldots, 2^{251} - 1\}$
could be 8 times that prime.
Extremely unlikely,
but annoys implementors,
so reject this $A$. 
For $A = 358990$:
One prime is $2^{252} - \ldots$,
so user’s secret key

$n \in 2^{254} + 8\{0, 1, \ldots, 2^{251} - 1\}$
could be 8 times that prime.
Extremely unlikely,
but annoys implementors,
so reject this $A$.

Note on comparing curves
and comparing coordinate systems:
Count fp ops, not field ops!
Otherwise you make bad choices.

Reality: mult by small constant
is as expensive as several adds.

Reality: square-to-multiply ratio
is $2^3$ for this field, not $4^5$.

Reality: $a^2 + b^2 + \ldots$ is faster than $(a^2, b^2, \ldots)$. 

Reject
unless curve and twist
orders are
4 prime 8 prime.
Montgomery shape forces 4;
characteristic in $4\mathbb{Z} + 1$ forces 4, 8.

Curve has order

$= 2^{252} + \ldots$.

Twist has order

$= 2^{253} - \ldots$.

For $A = 358990$:
One prime is $2^{252} - \ldots$,
so user’s secret key

$n \in 2^{254} + 8\{0, 1, \ldots, 2^{251} - 1\}$
could be 8 times that prime.
Extremely unlikely,
but annoys implementors,
so reject this $A$.

Note on comparing curves
and comparing coordinate systems:
Count fp ops, not field ops!
Otherwise you make bad choices.

Reality: mult by small constant
is as expensive as several adds.

Reality: square-to-multiply ratio
is $2^3$ for this field, not $4^5$.

Reality: $a^2 + b^2 + \ldots$ is faster than $(a^2, b^2, \ldots)$. 

Reject
unless curve and twist
orders are
4 prime 8 prime.
Montgomery shape forces 4;
characteristic in $4\mathbb{Z} + 1$ forces 4, 8.

Curve has order

$= 2^{252} + \ldots$.

Twist has order

$= 2^{253} - \ldots$.
For $A = 358990$:
One prime is $2^{252} - \cdots$,
so user’s secret key
$n \in 2^{254} + 8\{0, 1, \ldots, 2^{251} - 1\}$
could be 8 times that prime.
Extremely unlikely,
but annoys implementors,
so reject this $A$.

Note on comparing curves
and comparing coordinate systems:
Count fp ops, not field ops!
Otherwise you make bad choices.

Reality: mult by small constant
is as expensive as several adds.

Reality: square-to-multiply ratio
is $2/3$ for this field, not $4/5$.

Reality: $a^2 + b^2 + c^2$ is
faster than $(a^2, b^2, c^2)$. 
Note on comparing curves and comparing coordinate systems: Count fp ops, not field ops! Otherwise you make bad choices.

Reality: mult by small constant is as expensive as several adds.

Reality: square-to-multiply ratio is 2/3 for this field, not 4/5.

Reality: $a^2 + b^2 + c^2$ is faster than $(a^2, b^2, c^2)$.

How was the key range chosen?

Public key for secret key is $x$-coordinate of $2^{251}$th multiple of standard base point.

Base-point order is $2^{252}$, so uniform random $\{0, 1, 2, \ldots, 2^{251} - 1\}$ produces almost exactly uniform random public key from among $\approx 2^{251}$ possibilities.

The addition of $2^{251}$ avoids and avoids timing attacks.
Note on comparing curves and comparing coordinate systems: Count fp ops, not field ops! Otherwise you make bad choices.

Reality: mult by small constant is as expensive as several adds.

Reality: square-to-multiply ratio is 2/3 for this field, not 4/5.

Reality: $a^2 + b^2 + c^2$ is faster than $(a^2, b^2, c^2)$.

How was the key range chosen?

Public key for secret key $n$ is $x$-coordinate of $n$th multiple of standard base point $(9, \ldots)$.

Base-point order is $p_1 \approx 2^{252}$, so uniform random $n$ in $2^{251} + \{0, 1, 2, \ldots, 2^{251} - 1\}$ produces almost exactly uniform random public key from among $\approx 2^{251}$ possibilities.

The addition of $2^{251}$ avoids $\infty$ and avoids timing attacks.
How was the key range chosen?

Public key for secret key $n$ is $x$-coordinate of $n$th multiple of standard base point $(9, \ldots)$.

Base-point order is $p_1 \approx 2^{252}$, so uniform random $n$ in $2^{251} + \{0, 1, 2, \ldots, 2^{251} - 1\}$ produces almost exactly uniform random public key from among $\approx 2^{251}$ possibilities.

The addition of $2^{251}$ avoids $\infty$ and avoids timing attacks.

Miller, CRYPTO '85:

“For the key exchange only the $x$-coordinate needs to be transmitted. The formulas for multiples of a point cited in the first section make it clear that the $x$-coordinate of a multiple depends only on the $x$-coordinate of the original point.”

This is the compression method I use. No trouble from “point compression” patent 6141420 filed 1994.07.29.
How was the key range chosen?

Public key for secret key $n$ is $x$-coordinate of $n$th multiple of standard base point $(9, \ldots)$.

Base-point order is $p_1 \approx 2^{252}$, so uniform random $n$ in $2^{251} + \{0, 1, 2, \ldots, 2^{251} - 1\}$ produces almost exactly uniform random public key from among $\approx 2^{251}$ possibilities.

The addition of $2^{251}$ avoids $\infty$ and avoids timing attacks.

Miller, CRYPTO ’85:

“For the key exchange ... only the $x$-coordinate needs to be transmitted. The formulas for multiples of a point cited in the first section make it clear that the $x$-coordinate of a multiple depends only on the $x$-coordinate of the original point.”

This is the compression method I use. No trouble from “point compression” patent 6141420 filed 1994.07.29.
How was the key range chosen?

Public key for secret key is the $x$-coordinate of the $n$th multiple of standard base point $(9, \ldots)$. The order $p_1 \approx 2^{252}$ of $n$ in \{1, $2^{251} - 1\}$ exactly uniform random in $2^{251}$ produces almost exactly uniform random public key from among $2^{251}$ possibilities. The addition of $2^{251}$ avoids $\infty$ and avoids timing attacks.

Miller, CRYPTO '85:

“For the key exchange only the $x$-coordinate needs to be transmitted. The formulas for multiples of a point cited in the first section make it clear that the $x$-coordinate of a multiple depends only on the $x$-coordinate of the original point.”

This is the compression method I use. No trouble from “point compression” patent 6141420 filed 1994.07.29.

Insert factor of 8 in case $(K, \ldots)$ is not actually in this group of order 1.

Three possibilities:

- $\infty$, output as 0;
- or a nontrivial point in the desired prime group;
- or a nontrivial point in the twist prime group.

Don’t spend time “validating” $K$, i.e., checking it’s in desired group.
Miller, CRYPTO ’85:
“For the key exchange ... only the $x$-coordinate needs to be transmitted. The formulas for multiples of a point cited in the first section make it clear that the $x$-coordinate of a multiple depends only on the $x$-coordinate of the original point.”

This is the compression method I use. No trouble from “point compression” patent 6141420 filed 1994.07.29.

Insert factor of 8 into $n$ in case $(K, \ldots)$ is not actually in this group of order $p_1$.

Three possibilities for $8(K, \ldots)$:
$\infty$, output as 0;
or a nontrivial point in the desired prime group;
or a nontrivial point in the twist prime group.

Don’t spend time “validating” $K$, i.e., checking it’s in desired group.
Insert factor of 8 into \( n \) in case \((K,\ldots)\) is not actually in this group of order \( p_1 \).

Three possibilities for 8\((K,\ldots)\):
- \( \infty \), output as 0;
- or a nontrivial point in the desired prime group;
- or a nontrivial point in the twist prime group.

Don’t spend time “validating” \( K \), i.e., checking it’s in desired group.

Even if attacker were given same \( n \) times point on twist, would still need to break hash-Diffie-Hellman for product of these two prime groups.

For uniform random exponent, provably requires breaking at least one of the prime groups. Curve and twist both seem secure.

No known way to exploit limited exponent range. Often used in Diffie-Hellman for multiplicative group.
Insert factor of 8 into $n$
in case $(K, \ldots)$ is not actually
in this group of order $p_1$.

Three possibilities for $8(K, \ldots)$:
$\infty$, output as 0;
or a nontrivial point
in the desired prime group;
or a nontrivial point
in the twist prime group.

Don’t spend time
“validating” $K$, i.e.,
checking it’s in desired group.

Even if attacker were given
same $n$ times point on twist,
would still need to break
hash-Diffie-Hellman for product
of these two prime groups.

For uniform random exponent,
provably requires breaking
at least one of the prime groups.
Curve and twist both seem secure.

No known way to exploit
limited exponent range.
Often used in Diffie-Hellman
for multiplicative group.
Even if attacker were given same $n$ times point on twist, would still need to break hash-Diffie-Hellman for product of these two prime groups.

For uniform random exponent, provably requires breaking at least one of the prime groups. Curve and twist both seem secure.

No known way to exploit limited exponent range. Often used in Diffie-Hellman for multiplicative group.

Bernstein, sci.crypt, 2001.11.09: "You can happily skip both the transmission and the square root. In fact, if both the curve and its twist have nearly prime order, you can even skip square testing."

I use a curve of this type. No trouble from rumored new "public-key validation" patent filed 2003.
Even if attacker were given same $n$ times point on twist, would still need to break hash-Diffie-Hellman for product of these two prime groups.

For uniform random exponent, provably requires breaking at least one of the prime groups. Curve and twist both seem secure.

No known way to exploit limited exponent range. Often used in Diffie-Hellman for multiplicative group.

Bernstein, sci.crypt, 2001.11.09:

“You can happily skip both the $y$ transmission and the square root. In fact, if both the curve and its twist have nearly prime order, then you can even skip square testing.”

I use a curve of this type. No trouble from rumored new “public-key validation” patent filed 2003.
Even if attacker were given the same point on twist, would still need to break hash-Diffie-Hellman for product of these two prime groups. For uniform random exponent, provably requires breaking at least one of the prime groups. Curve and twist both seem secure. No known way to exploit limited exponent range. Often used in Diffie-Hellman for multiplicative group.

Bernstein, sci.crypt, 2001.11.09:
“You can happily skip both the \( y \) transmission and the square root. In fact, if both the curve and its twist have nearly prime order, then you can even skip square testing.” I use a curve of this type. No trouble from rumored new “public-key validation” patent filed 2003. How was the software built?
Common phenomenon: Write fp op sequence in C. Feed it to C compiler to produce machine. Observe that cycles is much larger than 1, sometimes 5 or more! Have faith. Don’t accept 1. Understand and eliminate non-fp-op cycles. (I have more work to do here for Athlon et al. Expect speedups.)
Bernstein, sci.crypt, 2001.11.09:

“You can happily skip both the $y$ transmission and the square root. In fact, if both the curve and its twist have nearly prime order, then you can even skip square testing.”

I use a curve of this type. No trouble from rumored new “public-key validation” patent filed 2003.

How was the software built?

Common phenomenon:
Write fp op sequence in C. Feed it to C compiler to produce machine language. Observe that cycles/fp ops is much larger than 1: sometimes 5 or more!

Have faith. Don’t accept $>1.1$. Understand and eliminate non-fp-op cycles.

(I have more work to do here for Athlon et al. Expect speedups.)
Bernstein, sci.crypt, 2001.11.09:

You can happily skip both the $y$ transmission and the square root. In fact, if both the curve and its twist have nearly prime order, then you can even skip square testing.”

I use a curve of this type.

No trouble from rumored new “public-key validation” patent filed 2003.

---

How was the software built?

Common phenomenon:
Write fp op sequence in C.
Feed it to C compiler to produce machine language.
Observe that cycles/fp ops is much larger than 1: sometimes 5 or more!

Have faith. Don’t accept $\geq 1.1$.
Understand and eliminate non-fp-op cycles.
(I have more work to do here for Athlon et al. Expect speedups.)

Some important delays:

- 3-cycle “load” latency, copying data from “cache” to “register” for arithmetic.
- Only 8 registers.
- 3-cycle fp add latency.
- 5-cycle fp mult latency.

An op waits if its inputs aren’t ready. CPU has some ability to reorder ops, but uses greedy algorithm.
How was the software built?

Common phenomenon:
Write fp op sequence in C.
Feed it to C compiler
to produce machine language.
Observe that cycles/fp ops
is much larger than 1:
sometimes 5 or more!

Have faith. Don’t accept $\geq 1.1$.
Understand and eliminate
non-fp-op cycles.

(I have more work to do here for
Athlon et al. Expect speedups.)

Some important delays:

- 3-cycle “load” latency,
copying data from “cache” to
“register” for arithmetic.
Only 8 registers.
- 3-cycle fp add latency.
- 5-cycle fp mult latency.

An op waits if its inputs
aren’t ready. CPU has some
ability to reorder ops, but
uses greedy algorithm; suboptimal.
Some important delays:

- 3-cycle “load” latency, copying data from “cache” to “register” for arithmetic.
- Only 8 registers.
- 3-cycle fp add latency.
- 5-cycle fp mult latency.

An op waits if its inputs aren’t ready. CPU has some ability to reorder ops, but uses greedy algorithm; suboptimal.

Can’t rely on C compiler to sensibly permute fp ops.

Sometimes \( r \leftarrow a + c \); \( r \leftarrow r + c \); \( r \leftarrow r + c \) is a sequence of exact fp adds; best done as, e.g., \( s \leftarrow a + c \); \( r \leftarrow r + s \).

But sometimes \( r \leftarrow a + c \) is a non-associative deliberately rounded fp add!

The C language has no way to express this distinction.
Some important delays:

- 3-cycle “load” latency, copying data from “cache” to “register” for arithmetic.
  Only 8 registers.
- 3-cycle fp add latency.
- 5-cycle fp mult latency.

An op waits if its inputs aren’t ready. CPU has some ability to reorder ops, but uses greedy algorithm; suboptimal.

Can’t rely on C compiler to sensibly permute fp ops.

Sometimes \( r \leftarrow a + b; r \leftarrow r + c; r \leftarrow r + d \) is a sequence of exact fp adds best done as, e.g., \( r \leftarrow b + d; s \leftarrow a + c; r \leftarrow r + s. \)

But sometimes \( r \leftarrow r + c \) is a non-associative deliberately rounded fp add!

The C language has no way to express this distinction.
Some important delays:

3-cycle "load" latency,
copying data from "cache" to
"register" for arithmetic.

3-cycle fp add latency.

5-cycle fp mult latency.

An op waits if its inputs
aren't ready. CPU has some
ability to reorder ops, but
uses greedy algorithm; suboptimal.

Can't rely on C compiler
to sensibly permute fp ops.

Sometimes \( r \leftarrow a + b \);
\( r \leftarrow r + c; \ r \leftarrow r + d \) is
a sequence of exact fp adds
best done as, e.g., \( r \leftarrow b + d; \)
\( s \leftarrow a + c; \ r \leftarrow r + s \).

But sometimes \( r \leftarrow r + c \)
is a non-associative
deliberately rounded fp add!
The C language has no way
to express this distinction.

Curve25519 implementation
is actually in qhasm,
new programming language
for high-speed computations.
Language allows declaration
and propagation of fp ranges;
guided register allocation; et al.
Lets me write desired code
with much less human
traditional asm and C compiler.
Have also used for fast AES,
fast Poly1305, fast Salsa20, etc.
Can’t rely on C compiler to sensibly permute fp ops.

Sometimes \( r \leftarrow a + b; \)
\( r \leftarrow r + c; \ r \leftarrow r + d \) is a sequence of exact fp adds best done as, e.g., \( r \leftarrow b + d; \)
\( s \leftarrow a + c; \ r \leftarrow r + s. \)

But sometimes \( r \leftarrow r + c \)
is a non-associative deliberately rounded fp add!

The C language has no way to express this distinction.

Curve25519 implementation is actually in qhasm, new programming language for high-speed computations.

Language allows declaration and propagation of fp ranges; guided register allocation; et al.

Lets me write desired code with much less human time than traditional asm and C compiler. Have also used for fast AES, fast Poly1305, fast Salsa20, etc.
Can't rely on C compiler to sensibly permute fp ops. Sometimes
\[ a + b; \]
\[ a + d; \]
\[ a + c; \]
is a sequence of exact fp adds
\[ r \leftarrow b + d; \]
\[ r \leftarrow c + s. \]
\[ r \leftarrow r + c \]
is a non-associative deliberately rounded fp add!
The C language has no way to express this distinction.

Curve25519 implementation is actually in qhasm, new programming language for high-speed computations. Language allows declaration and propagation of fp ranges; guided register allocation; et al.

Lets me write desired code with much less human time than traditional asm and C compiler. Have also used for fast AES, fast Poly1305, fast Salsa20, etc.

What's next?
Culmination of extensive work on eliminating field mults for genus-2 hyperelliptic curves: 25 mults per bit. Gaudry, eprint.iacr.org/2005/314

Half-size prime: e.g., \(2^{127} - 1\).
Select curve to make some mults easier, like choosing
\[ p \]

Should count fp ops instead. Prediction: this will beat genus 1.
Curve25519 implementation is actually in qhasm, new programming language for high-speed computations.

Language allows declaration and propagation of fp ranges; guided register allocation; et al.

Lets me write desired code with much less human time than traditional asm and C compiler.

Have also used for fast AES, fast Poly1305, fast Salsa20, etc.

What’s next?

Culmination of extensive work on eliminating field mults for genus-2 hyperelliptic curves: 25 mults per bit. Gaudry, eprint.iacr.org/2005/314

Half-size prime: e.g., $2^{127} - 1$. Select curve to make some mults easier, like choosing $A$.

Should count fp ops instead. Prediction: this will beat genus 1.