Main goal of this course:
We are the attackers.
We want to break ECC and RSA.

First need to understand ECC.

Main motivation for ECC:
Avoid index-calculus attacks
that plague finite-field DL.
Main goal of this course: We are the attackers. We want to break ECC and RSA.
First need to understand ECC.
Main motivation for ECC: Avoid index-calculus attacks that plague finite-field DL.

Diffie-Hellman key exchange
Pick some generator $P$, i.e. some group element (using additive notation here).
Alice's secret key $a$
Bob's secret key $b$
Alice's public key $aP$
Bob's public key $bP$

\{$\text{Alice}$; $\text{Bob}$\}'s shared secret $abP$ = $\{$$\text{Bob}; \text{Alice}$\}'s shared secret $baP$
Main goal of this course: We are the attackers. We want to break ECC and RSA. First need to understand ECC. Main motivation for ECC: Avoid index-calculus attacks that plague finite-field DL.
Main goal of this course:
We are the attackers.
We want to break ECC and RSA.
First need to understand ECC.
Main motivation for ECC:
Avoid index-calculus attacks
that plague finite-field DL.

Diffie-Hellman key exchange

Pick some generator $P$, i.e. some group element
(using additive notation here)

Alice’s secret key $a$
\[ \downarrow \]
Alice’s public key $aP$

Bob’s secret key $b$
\[ \downarrow \]
Bob’s public key $bP$

\[ \{Alice, Bob\}'s \text{ shared secret} = abP = \{Bob, Alice\}'s \text{ shared secret} = baP \]
Main goal of this course:
We are the attackers.
We want to break ECC and RSA.
First need to understand ECC.
Main motivation for ECC:
Avoid index-calculus attacks that plague finite-field DL.

Diffie-Hellman key exchange
Pick some generator $P$, i.e. some group element
(using additive notation here).

- Alice’s secret key $a$
  - Downward arrow
  - Alice’s public key $aP$
- Bob’s secret key $b$
  - Downward arrow
  - Bob’s public key $bP$

$\{\text{Alice, Bob}\}$’s shared secret $abP$
$\{\text{Bob, Alice}\}$’s shared secret $baP$
Main goal of this course:
We are the attackers.
We want to break ECC and RSA.
First need to understand ECC.

Main motivation for ECC:
Avoid index-calculus attacks
that plague finite-field DL.

Diffie-Hellman key exchange

Pick some generator $P$, i.e. some group element
(using additive notation here).

- Alice’s secret key $a$
- Bob’s secret key $b$
- Alice’s public key $aP$
- Bob’s public key $bP$

$\{\text{Alice, Bob}\}$’s shared secret $abP$ = $\{\text{Bob, Alice}\}$’s shared secret $baP$

What does $P$ look like & how to compute $P + Q$?
Main goal of this course:
We are the attackers.
We want to break ECC and RSA.
First need to understand ECC.

Main motivation for ECC:
Avoid index-calculus attacks
that plague finite-field DL.

Diffie-Hellman key exchange
Pick some generator $P$,
i.e. some group element
(using additive notation here).

Alice's secret key $a$
↓
↓
↘
Alice's public key $aP$

Bob's secret key $b$
↓
↓
↙
Bob's public key $bP$

{Alice, Bob}'s shared secret $abP$

What does $P$ look like &
how to compute $P + Q$?

The clock

This is the curve $x^2 + y^2 = 1$.

Warning:
This is not an elliptic curve.
"Elliptic curve" $\neq$ "ellipse."
Main goal of this course: We are the attackers.
ECC and RSA.
We want to break ECC.
First need to understand ECC.
Main motivation for ECC:
Avoid index-calculus attacks that plague finite-field DL.

Diffie-Hellman key exchange

Pick some generator $P$, i.e. some group element
(using additive notation here).

Alice’s secret key $a$

Bob’s secret key $b$

Alice’s public key $aP$

Bob’s public key $bP$

{Alice, Bob}’s shared secret $abP$

{Bob, Alice}’s shared secret $baP$

What does $P$ look like & how to compute $P + Q$?

Warning:
This is not an elliptic curve.
“Elliptic curve” ≠ “ellipse.”

This is the curve $x^2 + y^2 = 1$. 

The clock
Diffie-Hellman key exchange

Pick some generator $P$, i.e. some group element (using additive notation here).

Alice’s secret key $a$

Alice’s public key $aP$

{Alice, Bob}’s shared secret $abP$

What does $P$ look like & how to compute $P + Q$?

Bob’s secret key $b$

Bob’s public key $bP$

{Bob, Alice}’s shared secret $baP$
Diffie-Hellman key exchange

Pick some *generator* $P$, i.e. some group element (using additive notation here).

Alice’s secret key $a$

\[ \downarrow \]

Alice’s public key $aP$

$\{\text{Alice, Bob}\}$’s shared secret $abP$

Bob’s secret key $b$

\[ \downarrow \]

Bob’s public key $bP$

$\{\text{Bob, Alice}\}$’s shared secret $baP$

What does $P$ look like & how to compute $P + Q$?

Warning:
This is not an elliptic curve.
“Elliptic curve” $\neq$ “ellipse.”

The clock

This is the curve $x^2 + y^2 = 1$.
Diffie-Hellman key exchange

Pick some generator $P$, i.e. some group element (using additive notation here).

Alice’s secret key $a$

Bob’s secret key $b$

Alice’s public key $aP$

Bob’s public key $bP$

{Bob, Alice}’s shared secret $baP$

What does $P$ look like & how to compute $P + Q$?

The clock

This is the curve $x^2 + y^2 = 1$.

Warning:
This is not an elliptic curve.
“Elliptic curve” $\neq$ “ellipse.”
Diffie-Hellman key exchange

Pick some generator \( P \), i.e. some group element (using additive notation here).

Bob’s secret key \( b \)

Bob’s public key \( bP \)

\( \{\text{Bob, Alice}\}’s \) shared secret \( baP \)

What does \( P \) look like & how to compute \( P + Q \)?

This is the curve \( x^2 + y^2 = 1 \).

Warning:
This is not an elliptic curve.
“Elliptic curve” ≠ “ellipse.”
Diffie-Hellman key exchange

Pick some generator $P$, i.e. some group element (using additive notation here).

**Alice**'s secret key $a$

**Bob**'s secret key $b$

**Alice**'s public key $aP$

**Bob**'s public key $bP$

**Alice**'s and **Bob**'s shared secret $abP$

Questions:

- What does $P$ look like?
- How to compute $P + Q$?

---

```
The clock

y

x

This is the curve $x^2 + y^2 = 1$.

Warning:
This is *not* an elliptic curve.
“Elliptic curve” $\neq$ “ellipse.”
```

---

Examples of points on this curve:

- Point A
- Point B
- Point C
- Point D
- Point E
- Point F
- Point G
- Point H
- Point I
- Point J
- Point K
- Point L
- Point M
- Point N
- Point O
- Point P
- Point Q
- Point R
- Point S
- Point T
- Point U
- Point V
- Point W
- Point X
- Point Y
- Point Z

---

```
Examples of points on this curve:

- Point A
- Point B
- Point C
- Point D
- Point E
- Point F
- Point G
- Point H
- Point I
- Point J
- Point K
- Point L
- Point M
- Point N
- Point O
- Point P
- Point Q
- Point R
- Point S
- Point T
- Point U
- Point V
- Point W
- Point X
- Point Y
- Point Z
```
The clock

This is the curve $x^2 + y^2 = 1$.

Warning:
This is \textit{not} an elliptic curve.

“Elliptic curve” $\neq$ “ellipse.”
The clock

This is the curve $x^2 + y^2 = 1$.

Warning:
This is not an elliptic curve.
“Elliptic curve” $\neq$ “ellipse.”

Examples of points on this curve:
$(0, 1) = “12:00”.$
This is the curve $x^2 + y^2 = 1$.

Warning:
This is not an elliptic curve.
“Elliptic curve” ̸= “ellipse.”

Examples of points on this curve:
$(0, 1) = “12:00”.$
$(0, −1) = “6:00”.
The clock

This is the curve $x^2 + y^2 = 1$.

Warning:
This is not an elliptic curve.
“Elliptic curve” $\neq$ “ellipse.”

Examples of points on this curve:
$(0, 1) = \text{“12:00”}$.
$(0, -1) = \text{“6:00”}$.
$(1, 0) = \text{“3:00”}$.
This is the curve $x^2 + y^2 = 1$.

Warning:
This is not an elliptic curve.
“Elliptic curve” ≠ “ellipse.”
The clock

This is the curve $x^2 + y^2 = 1$.

Warning:
This is not an elliptic curve.
“Elliptic curve” $\neq$ “ellipse.”

Examples of points on this curve:
$(0, 1) = \text{“12:00”}$.
$(0, -1) = \text{“6:00”}$.
$(1, 0) = \text{“3:00”}$.
$(-1, 0) = \text{“9:00”}$.
$(\sqrt{3/4}, 1/2) =$
The clock

This is the curve $x^2 + y^2 = 1$.

Warning:
This is not an elliptic curve.
“Elliptic curve” $\neq$ “ellipse.”

Examples of points on this curve:
$(0, 1) = \text{“12:00”}$.
$(0, -1) = \text{“6:00”}$.
$(1, 0) = \text{“3:00”}$.
$(-1, 0) = \text{“9:00”}$.
$(\sqrt{3/4}, 1/2) = \text{“2:00”}$.
The clock

![Circle](image)

This is the curve $x^2 + y^2 = 1$.

**Warning:**
This is **not** an elliptic curve.

“Elliptic curve” $\neq$ “ellipse.”

Examples of points on this curve:

- $(0, 1) = “12:00”$. 
- $(0, −1) = “6:00”$. 
- $(1, 0) = “3:00”$. 
- $(-1, 0) = “9:00”$. 
- $(\sqrt{3/4}, 1/2) = “2:00”$. 
- $(1/2, −\sqrt{3/4}) = “…”$
The clock

This is the curve $x^2 + y^2 = 1$.

Warning:
This is not an elliptic curve.
“Elliptic curve” $\neq$ “ellipse.”

Examples of points on this curve:

$(0, 1) = \text{“12:00”}$.
$(0, -1) = \text{“6:00”}$.
$(1, 0) = \text{“3:00”}$.
$(-1, 0) = \text{“9:00”}$.
$(\sqrt{3/4}, 1/2) = \text{“2:00”}$.
$(1/2, -\sqrt{3/4}) = \text{“5:00”}$.
$(-1/2, -\sqrt{3/4}) =$
The clock

This is the curve $x^2 + y^2 = 1$.

Warning:

This is not an elliptic curve.

“Elliptic curve” ≠ “ellipse.”

Examples of points on this curve:

$(0, 1) = “12:00”. $(0, −1) = “6:00”.

$(1, 0) = “3:00”. $(−1, 0) = “9:00”.

$(\sqrt{3/4}, 1/2) = “2:00”. $(1/2, −\sqrt{3/4}) = “5:00”.

$(−1/2, −\sqrt{3/4}) = “7:00”.$
The clock

This is the curve $x^2 + y^2 = 1$.

Warning:
This is not an elliptic curve.
“Elliptic curve” $\neq$ “ellipse.”

Examples of points on this curve:
(0, 1) = “12:00”.
(0, −1) = “6:00”.
(1, 0) = “3:00”.
(−1, 0) = “9:00”.
$(\sqrt{3/4}, 1/2) = “2:00”.$
$(1/2, −\sqrt{3/4}) = “5:00”.$
$(-1/2, -\sqrt{3/4}) = “7:00”.$
$(\sqrt{1/2}, \sqrt{1/2}) = “1:30”.$
$(3/5, 4/5), (−3/5, 4/5).$
The clock

This is the curve $x^2 + y^2 = 1$.

Warning:
This is not an elliptic curve.
“Elliptic curve” ≠ “ellipse.”

Examples of points on this curve:

$(0, 1) = “12:00”$.
$(0, −1) = “6:00”$.
$(1, 0) = “3:00”$.
$(-1, 0) = “9:00”$.
$(\sqrt{3/4}, 1/2) = “2:00”$.
$(1/2, −\sqrt{3/4}) = “5:00”$.
$(-1/2, −\sqrt{3/4}) = “7:00”$.
$(\sqrt{1/2}, \sqrt{1/2}) = “1:30”$.
$(3/5, 4/5)$. $(-3/5, 4/5)$.
$(3/5, −4/5)$. $(-3/5, −4/5)$.
$(4/5, 3/5)$. $(-4/5, 3/5)$.
$(4/5, −3/5)$. $(-4/5, −3/5)$.
Many more.
The clock

\[ x^2 + y^2 = 1. \]

Warning:

This is not an elliptic curve.

"Elliptic curve" \( \neq \) "ellipse."

Examples of points on this curve:

\[
\begin{align*}
(0, 1) &= \text{“12:00”}. \\
(0, -1) &= \text{“6:00”}. \\
(1, 0) &= \text{“3:00”}. \\
(-1, 0) &= \text{“9:00”}. \\
\left(\sqrt{3/4}, 1/2\right) &= \text{“2:00”}. \\
\left(1/2, -\sqrt{3/4}\right) &= \text{“5:00”}. \\
\left(-1/2, -\sqrt{3/4}\right) &= \text{“7:00”}. \\
\left(\sqrt{1/2}, \sqrt{1/2}\right) &= \text{“1:30”}. \\
(3/5, 4/5) &\text{, } (-3/5, 4/5). \\
(3/5, -4/5) &\text{, } (-3/5, -4/5). \\
(4/5, 3/5) &\text{, } (-4/5, 3/5). \\
(4/5, -3/5) &\text{, } (-4/5, -3/5). \\
\end{align*}
\]

Many more.

Addition on the clock:

\[
\begin{align*}
\text{neutral } &= (0, 1) \\
\cdot P_1 &= (x_1, y_1) \\
\cdot P_2 &= (x_2, y_2) \\
\cdot P_3 &= (x_3, y_3) \\
\end{align*}
\]

\[
x^2 + y^2 = 1, \text{ parametrized by } \quad x = \sin \theta, \quad y = \cos \theta.
\]
Examples of points on this curve:
(0, 1) = “12:00”.
(0, −1) = “6:00”.
(1, 0) = “3:00”.
(−1, 0) = “9:00”.
(√3/4, 1/2) = “2:00”.
(1/2, −√3/4) = “5:00”.
(−1/2, −√3/4) = “7:00”.
(√1/2, √1/2) = “1:30”.
(3/5, 4/5). (−3/5, 4/5).
(3/5, −4/5). (−3/5, −4/5).
(4/5, 3/5). (−4/5, 3/5).
(4/5, −3/5). (−4/5, −3/5).
Many more.

Addition on the clock:
•
•
•

$x^2 + y^2 = 1$, parametrized by
$x = \sin \alpha, \quad y = \cos \alpha$. 
Examples of points on this curve:

(0, 1) = "12:00".
(0, −1) = "6:00".
(1, 0) = "3:00".
(−1, 0) = "9:00".
(√3/4, 1/2) = "2:00".
(1/2, −√3/4) = "5:00".
(−1/2, −√3/4) = "7:00".
(√1/2, √1/2) = "1:30".
(3/5, 4/5). (−3/5, 4/5).
(3/5, −4/5). (−3/5, −4/5).
(4/5, 3/5). (−4/5, 3/5).
(4/5, −3/5). (−4/5, −3/5).
Many more.

Addition on the clock:

\[ x^2 + y^2 = 1, \text{ parametrized by } x = \sin \alpha, \quad y = \cos \alpha. \]
Examples of points on this curve:
(0, 1) = “12:00”.
(0, −1) = “6:00”.
(1, 0) = “3:00”.
(−1, 0) = “9:00”.
( √3/4, 1/2) = “2:00”.
(1/2, − √3/4) = “5:00”.
(−1/2, − √3/4) = “7:00”.
( √1/2, √1/2) = “1:30”.
(3/5, 4/5). (−3/5, 4/5).
(3/5, −4/5). (−3/5, −4/5).
(4/5, 3/5). (−4/5, 3/5).
(4/5, −3/5). (−4/5, −3/5).
Many more.

Addition on the clock:

\[ x^2 + y^2 = 1, \text{ parametrized by } x = \sin \alpha, \quad y = \cos \alpha. \]
Examples of points on this curve:

- \((0, 1) = \text{“12:00”}\).
- \((0, -1) = \text{“6:00”}\).
- \((1, 0) = \text{“3:00”}\).
- \((-1, 0) = \text{“9:00”}\).
- \((\sqrt{3}/4, 1/2) = \text{“2:00”}\).
- \((1/2, -\sqrt{3}/4) = \text{“5:00”}\).
- \((-1/2, -\sqrt{3}/4) = \text{“7:00”}\).
- \((\sqrt{1/2}, \sqrt{1/2}) = \text{“1:30”}\).

- \((3/5, 4/5)\), \((-3/5, 4/5)\).
- \((3/5, -4/5)\), \((-3/5, -4/5)\).
- \((4/5, 3/5)\), \((-4/5, 3/5)\).
- \((4/5, -3/5)\), \((-4/5, -3/5)\).

Many more.

Addition on the clock:

\[
x^2 + y^2 = 1, \text{ parametrized by } x = \sin \alpha, \quad y = \cos \alpha.
\]

Recall \((\sin(\alpha_1 + \alpha_2), \cos(\alpha_1 + \alpha_2)) = \)
Examples of points on this curve:

(0, 1) = “12:00”.

(0, −1) = “6:00”.

(1, 0) = “3:00”.

(−1, 0) = “9:00”.

(\sqrt{3/4}, 1/2) = “2:00”.

(1/2, −\sqrt{3/4}) = “5:00”.

(−1/2, −\sqrt{3/4}) = “7:00”.

(\sqrt{1/2}, \sqrt{1/2}) = “1:30”.

(3/5, 4/5). (−3/5, 4/5).

(3/5, −4/5). (−3/5, −4/5).

(4/5, 3/5). (−4/5, 3/5).

(4/5, −3/5). (−4/5, −3/5).

Many more.

Addition on the clock:

\[
\begin{align*}
\text{neutral} & = (0, 1) \\
P_1 & = (x_1, y_1) \\
P_2 & = (x_2, y_2) \\
P_3 & = (x_3, y_3)
\end{align*}
\]

\[x^2 + y^2 = 1,\] parametrized by \(x = \sin \alpha, \ y = \cos \alpha.\) Recall

\[
(sin(\alpha_1 + \alpha_2), cos(\alpha_1 + \alpha_2)) = (sin \alpha_1 \cos \alpha_2 + cos \alpha_1 \sin \alpha_2,)
\]
Examples of points on this curve:

(0, 1) = “12:00”.
(0, -1) = “6:00”.
(1, 0) = “3:00”.
(-1, 0) = “9:00”.
(\sqrt{3}/4, 1/2) = “2:00”.
(1/2, -\sqrt{3}/4) = “5:00”.
(-1/2, -\sqrt{3}/4) = “7:00”.
(\sqrt{1}/2, \sqrt{1}/2) = “1:30”.
(3/5, 4/5). (-3/5, 4/5).
(3/5, -4/5). (-3/5, -4/5).
(4/5, 3/5). (-4/5, 3/5).
(4/5, -3/5). (-4/5, -3/5).
Many more.

Addition on the clock:

\[ x^2 + y^2 = 1, \text{ parametrized by } x = \sin \alpha, \ y = \cos \alpha. \]
Recall
\[ (\sin(\alpha_1 + \alpha_2), \cos(\alpha_1 + \alpha_2)) = (\sin \alpha_1 \cos \alpha_2 + \cos \alpha_1 \sin \alpha_2, \cos \alpha_1 \cos \alpha_2 - \sin \alpha_1 \sin \alpha_2). \]
Examples of points on this curve:

- $(0; 1) = "12:00"$.
- $(0; -1) = "6:00"$.
- $(1; 0) = "3:00"$.
- $(-1; 0) = "9:00"$.
- $(p_3 = 4; 1) = "2:00"$.
- $(1 = 2; -p_3 = 4) = "5:00"$.
- $(-1 = 2; -p_3 = 4) = "7:00"$.
- $(p_1 = 2; p_1 = 2) = "1:30"$.
- $(3 = 5; 4 = 5) = (3 = 5; -4 = 5)$.
- $(4 = 5; 3 = 5) = (4 = 5; -3 = 5)$.

Many more.

Adding two points corresponds to adding the angles $\alpha_1$ and $\alpha_2$. Angles modulo 360° are a group, so points on the clock are a group.

Neutral element: angle $\alpha = 0$; point (0; 1); "12:00".

The point with $\alpha = 180°$ has order 2 and equals 6:00.

3:00 and 9:00 have order 4.

Inverse of point with $\alpha$ is point with $-\alpha$ since $\alpha + (-\alpha) = 0$.

There are many more points where angle $\alpha$ is not "nice."
Examples of points on this curve:

$(0; 1) = "12:00".$

$(0; -1) = "6:00".$

$(1; 0) = "3:00".$

$(1; -p) = "5:00".$

$(p; 3) = "2:00".$

$(p; -3) = "7:00".$

$(4; 3) = "1:30".$

$(4; -3) = "7:30".$

$(4; 5) = "5:00".$

$(4; 5) = "9:00".$

$(4; -3) = "11:30".$

$(4; -3) = "3:30".$

$(3; 5) = "3:00".$

$(3; 5) = "9:00".$

$(3; -4) = "6:00".$

$(3; -4) = "12:00".$

$(4; -3) = "9:30".$

$(4; -3) = "3:30".$

$(4; 5) = "5:00".$

$(4; 5) = "9:00".$

$(4; -3) = "11:30".$

$(4; -3) = "3:30".$

$(4; 5) = "5:00".$

$(4; 5) = "9:00".$

$(4; -3) = "11:30".$

$(4; -3) = "3:30".$

Many more.

Addition on the clock:

$x^2 + y^2 = 1$, parametrized by
$x = \sin \alpha, \ y = \cos \alpha$. Recall
$(\sin(\alpha_1 + \alpha_2), \cos(\alpha_1 + \alpha_2)) =$
$(\sin \alpha_1 \cos \alpha_2 + \cos \alpha_1 \sin \alpha_2, \cos \alpha_1 \cos \alpha_2 - \sin \alpha_1 \sin \alpha_2)$.

Adding two points corresponds to adding the angles $\alpha_1$ and $\alpha_2$. Angles modulo 360° are a group, so points on clock are a group.

Neutral element: angle $\alpha = 0$; point $(0; 1); "12:00".$

The point with $\alpha = 180°$ has order 2 and equals 6:00.

3:00 and 9:00 have order 4.

Inverse of point with $\alpha$ is point with $-\alpha$ since $\alpha + (-\alpha) = 0$.

There are many more points where angle $\alpha$ is not "nice."
Examples of points on this curve:

\((0; 1) = \text{"12:00".}\)

\((0; -1) = \text{"6:00".}\)

\((1; 0) = \text{"3:00".}\)

\((-1; 0) = \text{"9:00".}\)

\(p_3 = 4; p_1 = 2\) = \"2:00\".

\((1 = 2; -p_3 = 4) = \text{"5:00".}\)

\((-1 = 2; -p_3 = 4) = \text{"7:00".}\)

\(p_1 = 2; p_1 = 2\) = \"1:30\".

\((3 = 5; 4 = 5). (-3 = 5; -4 = 5).\)

\((3 = 5; -4 = 5). (-3 = 5; -4 = 5).\)

\((4 = 5; 3 = 5). (-4 = 5; 3 = 5).\)

\((4 = 5; -3 = 5). (-4 = 5; -3 = 5).\)

Many more.

---

**Addition on the clock:**

\[ y \uparrow \uparrow \rightarrow \text{neutral} = (0; 1) \]

\[ P_1 = (x_1, y_1) \]

\[ P_2 = (x_2, y_2) \]

\[ P_3 = (x_3, y_3) \]

\[ x^2 + y^2 = 1, \text{ parametrized by } x = \sin \alpha, \ y = \cos \alpha. \]

Recall \((\sin(\alpha_1 + \alpha_2), \cos(\alpha_1 + \alpha_2)) = (\sin \alpha_1 \cos \alpha_2 + \cos \alpha_1 \sin \alpha_2, \cos \alpha_1 \cos \alpha_2 - \sin \alpha_1 \sin \alpha_2).\)

Adding two points corresponds to adding the angles \(\alpha_1\) and \(\alpha_2\).

Angles modulo 360° are a group, so points on clock are a group.

Neutral element: angle \(\alpha = 0;\) point \((0, 1); \) “12:00”.

The point with \(\alpha = 180°\) has order 2 and equals 6:00.

3:00 and 9:00 have order 4.

Inverse of point with \(\alpha\) is point with \(-\alpha\) since \(\alpha + (-\alpha) = 0.\)

There are many more points where angle \(\alpha\) is not “nice.”
Addition on the clock:

\[ x^2 + y^2 = 1, \] parametrized by \( x = \sin \alpha, \quad y = \cos \alpha. \)

Recall

\[ (\sin(\alpha_1 + \alpha_2), \cos(\alpha_1 + \alpha_2)) = \]

\[ (\sin \alpha_1 \cos \alpha_2 + \cos \alpha_1 \sin \alpha_2, \quad \cos \alpha_1 \cos \alpha_2 - \sin \alpha_1 \sin \alpha_2). \]

Adding two points corresponds to adding the angles \( \alpha_1 \) and \( \alpha_2 \). Angles modulo 360° are a group, so points on clock are a group.

Neutral element: angle \( \alpha = 0; \) point \((0, 1); \) “12:00”.

The point with \( \alpha = 180° \) has order 2 and equals 6:00.

3:00 and 9:00 have order 4.

Inverse of point with \( \alpha \) is point with \( -\alpha \) since \( \alpha + (-\alpha) = 0. \)

There are many more points where angle \( \alpha \) is not “nice.”
Adding two points corresponds to adding the angles $\alpha_1$ and $\alpha_2$. Angles modulo $360^\circ$ are a group, so points on clock are a group.

Neutral element: angle $\alpha = 0$; point $(0, 1)$; “12:00”.

The point with $\alpha = 180^\circ$ has order 2 and equals 6:00.

3:00 and 9:00 have order 4.

Inverse of point with $\alpha$ is point with $-\alpha$ since $\alpha + (-\alpha) = 0$.

There are many more points where angle $\alpha$ is not “nice.”
Adding two points corresponds to adding the angles $\alpha_1$ and $\alpha_2$. Angles modulo $360^\circ$ are a group, so points on clock are a group.

Neutral element: angle $\alpha = 0$; point $(0, 1)$; "12:00".

The point with $\alpha = 180^\circ$ has order 2 and equals 6:00.

3:00 and 9:00 have order 4.

Inverse of point with $\alpha$ is point with $-\alpha$ since $\alpha + (-\alpha) = 0$.

There are many more points where angle $\alpha$ is not "nice."

$x^2 + y^2 = 1$, parametrized by $x = \sin \alpha$, $y = \cos \alpha$.

$(\sin(\alpha_1 + \alpha_2), \cos(\alpha_1 + \alpha_2)) = (\sin \alpha_1 \cos \alpha_2 + \cos \alpha_1 \sin \alpha_2, \cos \alpha_1 \cos \alpha_2 - \sin \alpha_1 \sin \alpha_2)$. 

Addition on the clock:

$P_1 = (x_1, y_1)$
$P_2 = (x_2, y_2)$
$P_3 = (x_3, y_3)$

$\alpha_1$
$\alpha_2$
Adding two points corresponds to adding the angles $\alpha_1$ and $\alpha_2$. Angles modulo $360^\circ$ are a group, so points on clock are a group.

Neutral element: angle $\alpha = 0$; point $(0, 1)$; “12:00”.

The point with $\alpha = 180^\circ$ has order 2 and equals 6:00.

3:00 and 9:00 have order 4.

Inverse of point with $\alpha$ is point with $-\alpha$ since $\alpha + (-\alpha) = 0$.

There are many more points where angle $\alpha$ is not “nice.”
Adding two points corresponds to adding the angles $\alpha_1$ and $\alpha_2$. Angles modulo $360^\circ$ are a group, so points on clock are a group.

Neutral element: angle $\alpha = 0$; point $(0, 1)$; “12:00”.

The point with $\alpha = 180^\circ$ has order 2 and equals 6:00.

3:00 and 9:00 have order 4.

Inverse of point with $\alpha$ is point with $-\alpha$ since $\alpha + (-\alpha) = 0$.

There are many more points where angle $\alpha$ is not “nice.”

Addition on the clock:

$x^2 + y^2 = 1$, parametrized by $x = \sin \alpha, \ y = \cos \alpha$. Recall

$(\sin(\alpha_1 + \alpha_2), \cos(\alpha_1 + \alpha_2)) =
(sin \alpha_1 \cos \alpha_2 + \cos \alpha_1 \sin \alpha_2,
\cos \alpha_1 \cos \alpha_2 - \sin \alpha_1 \sin \alpha_2)$.
Adding two points corresponds to adding the angles \( \alpha_1 \) and \( \alpha_2 \). Angles modulo 360° are a group, so points on the clock are a group.

Neutral element: angle \( \alpha = 0 \); point \((0, 1)\); “12:00”.

Point with \( \alpha = 180° \) has order 2 and equals 6:00. 3:00 and 9:00 have order 4.

Inverse of point with \( \alpha \) is point with \(-\alpha \) since \( \alpha + (-\alpha) = 0 \).

There are many more points where angle \( \alpha \) is not “nice.”

Addition on the clock:

Use Cartesian coordinates for addition. Addition formula for the clock \( x^2 + y^2 = 1 \): for addition. Addition formula \( x = \sin \alpha, \ y = \cos \alpha \). Recall \( (\sin(\alpha_1 + \alpha_2), \cos(\alpha_1 + \alpha_2)) = (\sin \alpha_1 \cos \alpha_2 + \cos \alpha_1 \sin \alpha_2, \cos \alpha_1 \cos \alpha_2 - \sin \alpha_1 \sin \alpha_2) \).
Adding two points corresponds to adding the angles $\alpha_1$ and $\alpha_2$. Angles modulo $360^\circ$ are a group, so points on clock are a group.

Neutral element: angle $\alpha = 0$; point $(0, 1)$; “12:00”.

$\alpha = 180^\circ$ equals 6:00. The point with $\alpha = 180^\circ$ has order 2 and equals 6:00.

$3:00$ and $9:00$ have order 4.

Inverse of point with $\alpha$ is point with $-\alpha$ since $\alpha + (\alpha) = 0$.

There are many more points where angle $\alpha$ is not “nice.”

Addition on the clock:

Use Cartesian coordinates for addition. Addition formula for the clock $x^2 + y^2 = 1$: sum $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$.

$x^2 + y^2 = 1$, parametrized by $x = \sin \alpha, \ y = \cos \alpha$. Recall

$$(\sin(\alpha_1 + \alpha_2), \cos(\alpha_1 + \alpha_2)) = (\sin \alpha_1 \cos \alpha_2 + \cos \alpha_1 \sin \alpha_2, \cos \alpha_1 \cos \alpha_2 - \sin \alpha_1 \sin \alpha_2).$$
Adding two points corresponds to adding the angles $\alpha_1$ and $\alpha_2$. Angles modulo $360^\circ$ are a group, so points on clock are a group.

Neutral element: angle $\alpha = 0$; point $(0; 1)$; "12:00".

The point with $\alpha = 180^\circ$ has order 2 and equals 6:00.

3:00 and 9:00 have order 4.

Inverse of point with $\alpha$ is point with $-\alpha$ since $\alpha + (-\alpha) = 0$.

There are many more points where angle $\alpha$ is not "nice."

Addition on the clock:

$$x^2 + y^2 = 1,$$ parametrized by $x = \sin \alpha, \quad y = \cos \alpha$. Recall

$$(\sin(\alpha_1 + \alpha_2), \cos(\alpha_1 + \alpha_2)) = ($$

$$(\sin \alpha_1 \cos \alpha_2 + \cos \alpha_1 \sin \alpha_2,$$

$$\cos \alpha_1 \cos \alpha_2 - \sin \alpha_1 \sin \alpha_2).$$

Clock addition without sin, cos:

Use Cartesian coordinates for addition. Addition formula for the clock $x^2 + y^2 = 1$: 

$$\sum (x_1, y_1) + (x_2, y_2) = (x_3, y_3).$$
Addition on the clock:

\[ P_1 = (x_1, y_1) \]
\[ P_2 = (x_2, y_2) \]
\[ P_3 = (x_3, y_3) \]

\[ x^2 + y^2 = 1, \text{ parametrized by } x = \sin \alpha, \quad y = \cos \alpha. \]

Recall \((\sin(\alpha_1 + \alpha_2), \cos(\alpha_1 + \alpha_2)) = \)
\((\sin \alpha_1 \cos \alpha_2 + \cos \alpha_1 \sin \alpha_2, \cos \alpha_1 \cos \alpha_2 - \sin \alpha_1 \sin \alpha_2).\]

Clock addition without sin, cos:

\[ P_1 = (x_1, y_1) \]
\[ P_2 = (x_2, y_2) \]
\[ P_3 = (x_3, y_3) \]

Use Cartesian coordinates for addition. Addition formula for the clock \(x^2 + y^2 = 1:\)
\[ \text{sum } (x_1, y_1) + (x_2, y_2) = (x_3, y_3) \]
Addition on the clock:

\[ \text{neutral} = (0, 1) \]

\[ P_1 = (x_1, y_1) \]
\[ P_2 = (x_2, y_2) \]
\[ P_3 = (x_3, y_3) \]

\[ x^2 + y^2 = 1, \text{ parametrized by} \]
\[ x = \sin \alpha, \quad y = \cos \alpha. \]

Recall

\[ (\sin(\alpha_1 + \alpha_2), \cos(\alpha_1 + \alpha_2)) = \]
\[ (\sin \alpha_1 \cos \alpha_2 + \cos \alpha_1 \sin \alpha_2, \]
\[ \cos \alpha_1 \cos \alpha_2 - \sin \alpha_1 \sin \alpha_2). \]

Clock addition without sin, cos:

\[ \text{neutral} = (0, 1) \]

\[ P_1 = (x_1, y_1) \]
\[ P_2 = (x_2, y_2) \]
\[ P_3 = (x_3, y_3) \]

Use Cartesian coordinates for addition. Addition formula for the clock \( x^2 + y^2 = 1 \):

\[ \text{sum } (x_1, y_1) + (x_2, y_2) = (x_3, y_3) = (x_1y_2 + y_1x_2, y_1y_2 - x_1x_2). \]

Note \( (x_1, y_1) + (-x_1, y_1) = (0, 1). \)

\[ kP = P + P + \cdots + P \text{ for } k \geq 0. \]
Clock addition:

**y**

neutral = (0, 1)

$P_1 = (x_1, y_1)$

$P_2 = (x_2, y_2)$

$P_3 = (x_3, y_3)$

$x^2 + y^2 = 1$, parametrized by $x = \sin \alpha$, $y = \cos \alpha$. Recall $\cos(\alpha_1 + \alpha_2) = \cos \alpha_2 + \cos \alpha_1 \sin \alpha_2$, $\sin(\alpha_1 + \alpha_2) = \cos \alpha_2 - \sin \alpha_1 \sin \alpha_2$.

Clock addition without sin, cos:

**y**

neutral = (0, 1)

$P_1 = (x_1, y_1)$

$P_2 = (x_2, y_2)$

$P_3 = (x_3, y_3)$

Use Cartesian coordinates for addition. Addition formula for the clock $x^2 + y^2 = 1$:

$\text{sum } (x_1, y_1) + (x_2, y_2) = (x_3, y_3)$

$= (x_1y_2 + y_1x_2, y_1y_2 - x_1x_2)$.

Note $(x_1, y_1) + (-x_1, y_1) = (0, 1)$.

$kP = P + P + \cdots + P$ for $k \geq 0$.

Examples of clock addition:

"2:00" + "5:00"

= $(3/5, 4/5) + (-3/5, -4/5)$

= $(0, 1)$.

"5:00" + "9:00"

= $(1/2, -\sqrt{3}/2) + (1/2, \sqrt{3}/2)$

= $(1, 0)$.

$kP = P + P + \cdots + P$ for $k \geq 0$.
Clock:

neutral = (0, 1)

$P_1 = (x_1, y_1)$

$P_2 = (x_2, y_2)$

$P_3 = (x_3, y_3)$

Parametrized by

$\cos \alpha$. Recall

$(\alpha_1 + \alpha_2)) =

\cos \alpha_1 \sin \alpha_2,

\sin \alpha_1 \sin \alpha_2$.

Clock addition without sin, cos:

neutral = (0, 1)

$P_1 = (x_1, y_1)$

$P_2 = (x_2, y_2)$

$P_3 = (x_3, y_3)$

Use Cartesian coordinates for addition. Addition formula

for the clock $x^2 + y^2 = 1$:

$\text{sum } (x_1, y_1) + (x_2, y_2) = (x_3, y_3)$

$= (x_1y_2 + y_1x_2, y_1y_2 - x_1x_2)$.

Note $(x_1, y_1) + (-x_1, y_1) = (0, 1)$.

$kP = P + P + \cdots + P$ for $k \geq 0$.

Examples of clock:

"2:00" + "5:00"

$= (\sqrt{3}/4, 1/2) +$

$= (-1/2, -\sqrt{3}/4) =$

"5:00" + "9:00"

$= (1/2, -\sqrt{3}/4) +$

$= (\sqrt{3}/4, 1/2) =$

$2\left(\frac{3}{5}, \frac{4}{5}\right) = \left(\frac{24}{25}, \frac{24}{25}\right).$
Clock addition without sin, cos:

\[ \text{neutral} = (0, 1) \]

\[ P_1 = (x_1, y_1) \]

\[ P_2 = (x_2, y_2) \]

\[ P_3 = (x_3, y_3) \]

Use Cartesian coordinates for addition. Addition formula for the clock \( x^2 + y^2 = 1 \): 

\[ \text{sum} \ (x_1, y_1) + (x_2, y_2) = (x_3, y_3) \]

\[ = (x_1y_2 + y_1x_2, y_1y_2 - x_1x_2). \]

Note \((x_1, y_1) + (-x_1, y_1) = (0, 1)\).

\[ kP = P + P + \cdots + P \text{ for } k \geq 0. \]

Examples of clock addition:

“2:00” + “5:00”

\[ = (\sqrt{3}/4, 1/2) + (1/2, -\sqrt{3}/4) = (7/8, -1/4) = “7:00”. \]

“5:00” + “9:00”

\[ = (1/2, -\sqrt{3}/4) + (-1, 0) = (\sqrt{3}/4, 1/2) = “2:00”. \]

2 \( \begin{pmatrix} 3 & 4 \\ 5' & 5 \end{pmatrix} \) = \( \begin{pmatrix} 24 & 7 \\ 25' & 25 \end{pmatrix} \).
Clock addition without sin, cos:

$\text{neutral} = (0, 1)$

$P_1 = (x_1, y_1)$

$P_2 = (x_2, y_2)$

$P_3 = (x_3, y_3)$

Use Cartesian coordinates for addition. Addition formula for the clock $x^2 + y^2 = 1$:

$\text{sum } (x_1, y_1) + (x_2, y_2) = (x_3, y_3)$

$= (x_1 y_2 + y_1 x_2, y_1 y_2 - x_1 x_2)$. 

Note $(x_1, y_1) + (-x_1, y_1) = (0, 1)$. 

$kP = P + P + \cdots + P$ for $k \geq 0$.

Examples of clock addition:

“2:00” + “5:00”

$= (\sqrt{\frac{3}{4}}, 1/2) + (1/2, -\sqrt{3/4})$

$= (-1/2, -\sqrt{3/4}) = “7:00”$. 

“5:00” + “9:00”

$= (1/2, -\sqrt{3/4}) + (-1, 0)$

$= (\sqrt{3/4}, 1/2) = “2:00”$. 

$2 \left( \begin{array}{cc} 3 \ 4 \\ 5 \ 5 \end{array} \right) = \left( \begin{array}{cc} 24 \ 7 \\ 25 \ 25 \end{array} \right)$. 

Clock addition without sin, cos:

Neutral = (0, 1)

P_1 = (x_1, y_1)

P_2 = (x_2, y_2)

P_3 = (x_3, y_3)

Use Cartesian coordinates for addition. Addition formula for the clock x^2 + y^2 = 1:

\[
\text{sum } (x_1, y_1) + (x_2, y_2) = (x_3, y_3) = (x_1 y_2 + y_1 x_2, y_1 y_2 - x_1 x_2).
\]

Note \((x_1, y_1) + (-x_1, y_1) = (0, 1)\).

\(kP = P + P + \cdots + P\) for \(k \geq 0\).

Examples of clock addition:

“2:00” + “5:00”

= \((\sqrt{3}/4, 1/2) + (1/2, -\sqrt{3}/4)\)

= \((-1/2, -\sqrt{3}/4) = “7:00”.\)

“5:00” + “9:00”

= \((1/2, -\sqrt{3}/4) + (-1, 0)\)

= \((\sqrt{3}/4, 1/2) = “2:00”.\)

2\(\left(\frac{3}{5}, \frac{4}{5}\right) = \left(\frac{24}{25}, \frac{7}{25}\right)\).

3\(\left(\frac{3}{5}, \frac{4}{5}\right) = \left(\frac{117}{125}, \frac{-44}{125}\right).\)
Clock addition without sin, cos:

**neutral** = (0, 1)

\[ P_1 = (x_1, y_1) \quad \quad P_2 = (x_2, y_2) \quad \quad P_3 = (x_3, y_3) \]

Use Cartesian coordinates for addition. Addition formula for the clock \( x^2 + y^2 = 1 \):

\[ \text{sum} \ (x_1, y_1) + (x_2, y_2) = (x_3, y_3) = (x_1 y_2 + y_1 x_2, y_1 y_2 - x_1 x_2). \]

Note \( (x_1, y_1) + (-x_1, y_1) = (0, 1). \)

\[ kP = P + P + \cdots + P \quad \text{for } k \geq 0. \]

Examples of clock addition:

“2:00” + “5:00”

\[ = (\sqrt{3}/4, 1/2) + (1/2, -\sqrt{3}/4) \]

\[ = (-1/2, -\sqrt{3}/4) = “7:00”. \]

“5:00” + “9:00”

\[ = (1/2, -\sqrt{3}/4) + (-1, 0) \]

\[ = (\sqrt{3}/4, 1/2) = “2:00”. \]

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 [(3/5, 4/5)]</td>
<td>[(24/25, 7/25)]</td>
</tr>
<tr>
<td>3 [(3/5, 4/5)]</td>
<td>[(117/125, -44/125)]</td>
</tr>
<tr>
<td>4 [(3/5, 4/5)]</td>
<td>[(336/625, -527/625)]</td>
</tr>
</tbody>
</table>
Clock addition without sin, cos:

\[
\begin{align*}
\text{neutral} &= (0, 1) \\
P_1 &= (x_1, y_1) \\
P_2 &= (x_2, y_2) \\
P_3 &= (x_3, y_3)
\end{align*}
\]

Use Cartesian coordinates for addition. Addition formula for the clock \( x^2 + y^2 = 1 \):

\[
\text{sum } (x_1, y_1) + (x_2, y_2) = (x_3, y_3) = (x_1 y_2 + y_1 x_2, y_1 y_2 - x_1 x_2).
\]

Note \((x_1, y_1) + (-x_1, y_1) = (0, 1)\).

\[
kP = P + P + \cdots + P \text{ for } k \geq 0.
\]

Example of clock addition:

“2:00” + “5:00”

\[
= (\sqrt{3}/4, 1/2) + (1/2, -\sqrt{3}/4) \\
= (-1/2, -\sqrt{3}/4) = “7:00”.
\]

“5:00” + “9:00”

\[
= (1/2, -\sqrt{3}/4) + (-1, 0) \\
= (\sqrt{3}/4, 1/2) = “2:00”.
\]

\[
2\left(\frac{3}{5}, \frac{4}{5}\right) = \left(\frac{24}{25}, \frac{7}{25}\right).
\]

\[
3\left(\frac{3}{5}, \frac{4}{5}\right) = \left(\frac{117}{125}, -\frac{44}{125}\right).
\]

\[
4\left(\frac{3}{5}, \frac{4}{5}\right) = \left(\frac{336}{625}, -\frac{527}{625}\right).
\]

\[
(x_1, y_1) + (0, 1) =
\]
Clock addition without sin, cos:

\[
\text{neutral} = (0, 1)
\]

\[
P_1 = (x_1, y_1)
\]

\[
P_2 = (x_2, y_2)
\]

\[
P_3 = (x_3, y_3)
\]

Use Cartesian coordinates for addition. Addition formula for the clock \( x^2 + y^2 = 1 \):

\[
\text{sum } (x_1, y_1) + (x_2, y_2) = (x_3, y_3)
\]

\[
= (x_1y_2 + y_1x_2, y_1y_2 - x_1x_2).
\]

Note \((x_1, y_1) + (-x_1, y_1) = (0, 1)\).

\[kP = P + P + \cdots + P\text{ for } k \geq 0.
\]

Examples of clock addition:

“2:00” + “5:00”

\[
= (\sqrt{3}/4, 1/2) + (1/2, -\sqrt{3}/4)
\]

\[= (-1/2, -\sqrt{3}/4) = “7:00”.\]

“5:00” + “9:00”

\[
= (1/2, -\sqrt{3}/4) + (-1, 0)
\]

\[= (\sqrt{3}/4, 1/2) = “2:00”.
\]

\[
2\left(\begin{array}{c}
3 \\
5
\end{array}\right)
\]

\[
= \left(\begin{array}{c}
24 \\
25
\end{array}\right).
\]

\[
3\left(\begin{array}{c}
3 \\
5
\end{array}\right)
\]

\[
= \left(\begin{array}{c}
117 \\
125
\end{array}\right).
\]

\[
4\left(\begin{array}{c}
3 \\
5
\end{array}\right)
\]

\[
= \left(\begin{array}{c}
336 \\
625
\end{array}\right).
\]

\[(x_1, y_1) + (0, 1) = (x_1, y_1).\]
Clock addition without sin, cos:

neutral = (0; 1)

\[ P_1 = (x_1, y_1) \]

\[ P_2 = (x_2, y_2) \]

\[ P_3 = (x_3, y_3) \]

Use Cartesian coordinates for addition. Addition formula for the clock \( x^2 + y^2 = 1 \):

\[ \text{sum } (x_1, y_1) + (x_2, y_2) = (x_3, y_3) \]

\[ = (x_1 y_2 + y_1 x_2, y_1 y_2 - x_1 x_2) \]

Note \((x_1, y_1) + (-x_1, y_1) = (0, 1)\).

\[ kP = P + P + \cdots + P \text{ for } k \geq 0. \]

Examples of clock addition:

“2:00” + “5:00”

\[ = (\sqrt{3}/4, 1/2) + (1/2, -\sqrt{3}/4) \]

\[ = (-1/2, -\sqrt{3}/4) = “7:00”. \]

“5:00” + “9:00”

\[ = (1/2, -\sqrt{3}/4) + (-1, 0) \]

\[ = (\sqrt{3}/4, 1/2) = “2:00”. \]

2 \( \left( \frac{3}{5}, \frac{4}{5} \right) \) = \( \left( \frac{24}{25}, \frac{7}{25} \right) \).

3 \( \left( \frac{3}{5}, \frac{4}{5} \right) \) = \( \left( \frac{117}{125}, \frac{-44}{125} \right) \).

4 \( \left( \frac{3}{5}, \frac{4}{5} \right) \) = \( \left( \frac{336}{625}, \frac{-527}{625} \right) \).

\((x_1, y_1) + (0, 1) = (x_1, y_1)\).

\((x_1, y_1) + (-x_1, y_1) = \)
Clock addition without sin, cos:

\[ \text{neutral} = (0, 1) \]

\[ P_1 = (x_1, y_1) \]

\[ P_2 = (x_2, y_2) \]

\[ P_3 = (x_3, y_3) \]

Use Cartesian coordinates for addition. Addition formula for the clock \( x^2 + y^2 = 1 \):

\[ \text{sum} \ (x_1, y_1) + (x_2, y_2) = (x_3, y_3) \]

\[ = (x_1y_2 + y_1x_2, y_1y_2 - x_1x_2) \]

Note \( (x_1, y_1) + (-x_1, y_1) = (0, 1) \).

\[ kP = P + P + \cdots + P \text{ for } k \geq 0. \]

\[ k \text{ copies} \]

Examples of clock addition:

“2:00” + “5:00”

\[ = (\sqrt{3}/4, 1/2) + (1/2, -\sqrt{3}/4) \]

\[ = (-1/2, -\sqrt{3}/4) = “7:00”. \]

“5:00” + “9:00”

\[ = (1/2, -\sqrt{3}/4) + (-1, 0) \]

\[ = (\sqrt{3}/4, 1/2) = “2:00”. \]

2 \[ \begin{pmatrix} 3 \\ 5 \end{pmatrix} \]

\[ \begin{pmatrix} 4 \\ 5 \end{pmatrix} \]

\[ = \begin{pmatrix} 24 \\ 25 \end{pmatrix} \]

3 \[ \begin{pmatrix} 3 \\ 5 \end{pmatrix} \]

\[ \begin{pmatrix} 4 \\ 5 \end{pmatrix} \]

\[ = \begin{pmatrix} 117 \\ 125 \end{pmatrix} \]

4 \[ \begin{pmatrix} 3 \\ 5 \end{pmatrix} \]

\[ \begin{pmatrix} 4 \\ 5 \end{pmatrix} \]

\[ = \begin{pmatrix} 336 \\ 625 \end{pmatrix} \]

\[ \begin{pmatrix} -527 \\ 625 \end{pmatrix} \]

\[ (x_1, y_1) + (0, 1) = (x_1, y_1). \]

\[ (x_1, y_1) + (-x_1, y_1) = (0, 1). \]
Addition without sin, cos:

\[ y \]
\[ \uparrow \uparrow \]
\[ \rightarrow \rightarrow \]
neutral = (0, 1)

\[ \begin{align*}
P_1 &= (x_1, y_1) \\
P_2 &= (x_2, y_2) \\
P_3 &= (x_3, y_3)
\end{align*} \]

Cartesian coordinates for

Addition formula

for the clock

\[ x^2 + y^2 = 1: \]
\[ (x_1, y_1) + (x_2, y_2) = (x_3, y_3) + y_1 x_2, y_1 y_2 - x_1 x_2). \]
\[ (x_1, y_1) + (-x_1, y_1) = (0, 1). \]

\[ P + \ldots + P \text{ for } k \geq 0. \]

Examples of clock addition:

“2:00” + “5:00”

\[ (\sqrt{3}/4, 1/2) + (1/2, -\sqrt{3}/4) \]
\[ = (-1/2, -\sqrt{3}/4) = \text{“7:00”}. \]

“5:00” + “9:00”

\[ (1/2, -\sqrt{3}/4) + (-1, 0) \]
\[ = (\sqrt{3}/4, 1/2) = \text{“2:00”}. \]

2 \[
\begin{pmatrix}
3 \\
5
\end{pmatrix}
\] \[
\begin{pmatrix}
4 \\
5
\end{pmatrix}
\]

= \[
\begin{pmatrix}
24 \\
25
\end{pmatrix}
\] \[
\begin{pmatrix}
7 \\
25
\end{pmatrix}
\]

= \[
\begin{pmatrix}
117 \\
125
\end{pmatrix}
\] \[
\begin{pmatrix}
-44 \\
125
\end{pmatrix}
\]

3 \[
\begin{pmatrix}
3 \\
5
\end{pmatrix}
\] \[
\begin{pmatrix}
4 \\
5
\end{pmatrix}
\]

= \[
\begin{pmatrix}
336 \\
625
\end{pmatrix}
\] \[
\begin{pmatrix}
-527 \\
625
\end{pmatrix}
\]

Clocks over finite fields

\[ \mathbb{F}_7 = \{0, 1, 2, 3, 4, 5, 6\} \]
\[ \{x, y\} \in \mathbb{F}_7 \times \mathbb{F}_7: \]
\[ x^2 + y^2 = 1 \bar{\sim}. \]

Clock(\mathbb{F}_7)

Here \( \mathbb{F}_7 \)

= \{0, 1, 2, 3, 4, 5, 6\}

with +, \(-\), \(\times\) modulo 7.

E.g. \[ \begin{pmatrix}
2 \\
5
\end{pmatrix} \times \begin{pmatrix}
5 \\
5
\end{pmatrix} = \begin{pmatrix}
3 \\
6
\end{pmatrix} \times \begin{pmatrix}
1 \\
1
\end{pmatrix} = \begin{pmatrix}
2 \\
2
\end{pmatrix} \times \begin{pmatrix}
1 \\
1
\end{pmatrix} = \begin{pmatrix}
3 \\
3
\end{pmatrix} \times \begin{pmatrix}
1 \\
1
\end{pmatrix} = \begin{pmatrix}
2 \\
2
\end{pmatrix}. \]

Examples of clock addition:

“2:00” + “5:00”

= (p_3 = 4, 1) + (1, -p_3 = 4)

= (-1, -p_3 = 4) = “7:00”.

“5:00” + “9:00”

= (1, -p_3 = 4) + (-1, 0)

= (p_3 = 4, 1) = “2:00”.

Clocks over finite fields
Clock addition without sin, cos:

Neutral = (0, 1)

\[ P_1 = (x_1, y_1) \]

\[ P_2 = (x_2, y_2) \]

\[ P_3 = (x_3, y_3) \]

Coordinates for addition:

- \( y^2 = 1 \):
  - \( (x_2, y_2) = (x_3, y_3) \)
  - \( y_2^2 = 1 \)
  - \( x_3_1 y_2 - x_1x_2 \).
- \( (x_1, y_1) = (0, 1) \).

\( k \cdot P \) for \( k \geq 0 \).

Examples of clock addition:

- \( "2:00" + "5:00" \)
  
  \[ (\sqrt{3}/4, 1/2) + (1/2, -\sqrt{3}/4) = (\sqrt{3}/4, 1/2) \]

- \( "5:00" + "9:00" \)
  
  \[ (1/2, -\sqrt{3}/4) + (-1, 0) = \]

Clocks over finite fields:

- \( \text{Clock}(F_7) = \{ (x, y) \in F_7 \times F_7 \mid x^2 + y^2 = 1 \} \)

Here \( F_7 = \{ 0, 1, 2, 3, -3, -2, -1 \} \)

with \( +, -, \times \) mod 7.

E.g. \( 2 \cdot 5 = 3 \) and \( 3 = 2 = 5 \) in \( F_7 \).
Clock addition without sin, cos:

$\mathbf{y} = (\nu; 1)$

$x = (x_1; y_1)$

$\mathbf{x} = (x_2, y_2)$

$\mathbf{x} = (x_3, y_3)$

$\mathbf{y} = (0; 1)$

$k \geq 0.$

**Examples of clock addition:**

"2:00" + "5:00"

= $(\sqrt{3/4}, 1/2) + (1/2, -\sqrt{3/4})$

= $(-1/2, -\sqrt{3/4}) = "7:00".$

"5:00" + "9:00"

= $(1/2, -\sqrt{3/4}) + (-1, 0)$

= $(\sqrt{3/4}, 1/2) = "2:00".$

$2(\begin{array}{c}
3 \\
5 \\
\end{array}, \begin{array}{c}
4 \\
5 \\
\end{array}) = \begin{array}{c}
24 \\
25 \\
\end{array}, \begin{array}{c}
7 \\
25 \\
\end{array}.$

$3(\begin{array}{c}
3 \\
5 \\
\end{array}, \begin{array}{c}
4 \\
5 \\
\end{array}) = \begin{array}{c}
117 \\
125 \\
\end{array}, \begin{array}{c}
-44 \\
125 \\
\end{array}.$

$4(\begin{array}{c}
3 \\
5 \\
\end{array}, \begin{array}{c}
4 \\
5 \\
\end{array}) = \begin{array}{c}
336 \\
625 \\
\end{array}, \begin{array}{c}
-527 \\
625 \\
\end{array}.$

$(x_1, y_1) + (0, 1) = (x_1, y_1).

(x_1, y_1) + (-x_1, y_1) = (0, 1).

Clock($F_7$) =

$\{(x, y) \in F_7 \times F_7 : x^2 + y^2 \}$

Here $F_7 = \{0, 1, 2, 3, 4, 5, 6\}$

= \{0, 1, 2, 3, -3, -2, -1\}

with $+, -, \times \mod 7.$

E.g. $2 \cdot 5 = 3$ and $3/2 = 5$ in $F_7.$
Examples of clock addition:

“2:00” + “5:00”

= \((\sqrt{3}/4, 1/2) + (1/2, -\sqrt{3}/4)\)

= \((-1/2, -\sqrt{3}/4) = “7:00”\).

“5:00” + “9:00”

= \((1/2, -\sqrt{3}/4) + (-1, 0)\)

= \((\sqrt{3}/4, 1/2) = “2:00”\).

2 \(\begin{pmatrix} 3 \\ 5 \end{pmatrix} = \begin{pmatrix} 24 \\ 25 \end{pmatrix} \).

3 \(\begin{pmatrix} 3 \\ 5 \end{pmatrix} = \begin{pmatrix} 117 \\ 125 \end{pmatrix} \).

4 \(\begin{pmatrix} 3 \\ 5 \end{pmatrix} = \begin{pmatrix} 336 \\ 625 \end{pmatrix} \).

\((x_1, y_1) + (0, 1) = (x_1, y_1)\).

\((x_1, y_1) + (-x_1, y_1) = (0, 1)\).

Clocks over finite fields

\(\text{Clock}(F_7) = \{(x, y) \in F_7 \times F_7 : x^2 + y^2 = 1\}\).

Here \(F_7 = \{0, 1, 2, 3, 4, 5, 6\}\)

= \{0, 1, 2, 3, -3, -2, -1\}

with +, −, × modulo 7.

E.g. 2 · 5 = 3 and 3/2 = 5 in \(F_7\).
Examples of clock addition:

- “5:00”
  \[
  \left(\frac{1}{4}, \frac{1}{2}\right) + (1/2, -\sqrt{3}/4)
  \]
  \(= (2, -\sqrt{3}/4) = “7:00”\).

- “9:00”
  \[
  \left(\frac{1}{4}, \frac{1}{2}\right) + (-1, 0)
  \]
  \(= (0, -\sqrt{3}/4) = “2:00”\).

Clocks over finite fields

\[
\text{Clock}(\mathbb{F}_7) = \{(x, y) \in \mathbb{F}_7 \times \mathbb{F}_7 : x^2 + y^2 = 1\}.
\]

Here \(\mathbb{F}_7 = \{0, 1, 2, 3, 4, 5, 6\}\)
\(= \{0, 1, 2, 3, -3, -2, -1\}\)
with \(+, -, \times\) modulo 7.

E.g. \(2 \cdot 5 = 3\) and \(3/2 = 5\) in \(\mathbb{F}_7\).
Examples of clock addition:

(1/2, \( -\sqrt{3/4} \)) + (\( -1, 0 \)) = “7:00”.

Clocks over finite fields

\[
\begin{array}{cccccccc}
   &   &   &   &   &   &   &   \\
   &   &   &   &   &   &   &   \\
   &   &   &   &   &   &   &   \\
   &   &   &   &   &   &   &   \\
   &   &   &   &   &   &   &   \\
   &   &   &   &   &   &   &   \\
   &   &   &   &   &   &   &   \\
   &   &   &   &   &   &   &   \\
   \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
   \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
   \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
   \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
   \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
   \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
   \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
   \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
   \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\end{array}
\]

Clock(\( F_7 \)) = \{ (x, y) \in F_7 \times F_7 : x^2 + y^2 = 1 \}.

Here \( F_7 = \{ 0, 1, 2, 3, 4, 5, 6 \} = \{ 0, 1, 2, 3, -3, -2, -1 \} \)

with +, −, × modulo 7.

E.g. 2 · 5 = 3 and 3/2 = 5 in \( F_7 \).
Examples of clock addition:

"2:00" + "5:00" = (p_3 = 4; 1 = 2) + (1 = 2; −p_3 = 4) = (−1 = 2; −p_3 = 4) = "7:00".

"5:00" + "9:00" = (1 = 2; −p_3 = 4) + (−1 = 2; 0) = (p_3 = 4; 1 = 2) = "2:00".

Clocks over finite fields

Clock(\(F_7\)) = \{(x, y) \in F_7 \times F_7 : x^2 + y^2 = 1\}.

Here \(F_7 = \{0, 1, 2, 3, 4, 5, 6\}\) = \{0, 1, 2, 3, −3, −2, −1\} with +, −, × modulo 7.

E.g. \(2 \cdot 5 = 3\) and \(3/2 = 5\) in \(F_7\).

>>> for x in range(7):
    ...     for y in range(7):
    ...         if (x*x+y*y) % 7 == 1:
    ...             print (x,y)
    ...
(0, 1)
(0, 6)
(1, 0)
(2, 2)
(2, 5)
(5, 2)
(5, 5)
(6, 0)

>>>
Clocks over finite fields

Clock\(\mathbb{F}_7\) = 
\{(x, y) \in \mathbb{F}_7 \times \mathbb{F}_7 : x^2 + y^2 = 1\}.
Here \(\mathbb{F}_7 = \{0, 1, 2, 3, 4, 5, 6\}\)
= \{0, 1, 2, 3, −3, −2, −1\}
with +, −, × modulo 7.
E.g. \(2 \cdot 5 = 3\) and \(3/2 = 5\) in \(\mathbb{F}_7\).

```python
>>> for x in range(7):
    ... for y in range(7):
    ... if (x*x+y*y) % 7 == 1:
    ...     print (x,y)
...
(0, 1)
(0, 6)
(1, 0)
(2, 2)
(2, 5)
(5, 2)
(5, 5)
(6, 0)
```
Clocks over finite fields

\[ \{ x, y \in \mathbb{F}_7 : x^2 + y^2 = 1 \} \]

\[ \{ 0, 1, 2, 3, 4, 5, 6 \} \]

\[ 2, 3, -3, -2, -1 \]

- + \text{ modulo } 7.

5 = 3 and 3/2 = 5 in \( \mathbb{F}_7 \).

```python
>>> for x in range(7):
...     for y in range(7):
...         if (x*x+y*y) % 7 == 1:
...             print (x,y)
...
(0, 1)
(0, 6)
(1, 0)
(2, 2)
(2, 5)
(5, 2)
(5, 5)
(6, 0)
```

```python
>>> for x in range(7):
...     for y in range(7):
...         if (x*x+y*y) % 7 == 1:
...             print (x,y)
...
(0, 1)
(0, 6)
(1, 0)
(2, 2)
(2, 5)
(5, 2)
(5, 5)
(6, 0)
```

```python
class F7:
    def __init__(self,x):
        self.int = x % 7
    def __str__(self):
        return str(self.int)
    __repr__ = __str__

>>> print F7(2)
2
>>> print F7(6)
6
>>> print F7(7)
0
>>> print F7(10)
3
```
Clocks over finite fields

\[
\text{Clock} \left( F_7 \right) = \\overline{\left( x, y \right)} \in F_7 \times F_7 : x^2 + y^2 = 1
\]

Here \( F_7 = \{0, 1, 2, 3, 4, 5, 6\} \) with addition and multiplication modulo 7.

E.g. \( 2 \cdot 5 = 3 \) and \( 3 = 2 = 5 \) in \( F_7 \).

```python
>>> for x in range(7):
...     for y in range(7):
...         if (x*x+y*y) % 7 == 1:
...             print (x,y)
...
(0, 1)
(0, 6)
(1, 0)
(2, 2)
(2, 5)
(5, 2)
(5, 5)
(6, 0)
```

```python
>>> class F7:
...     def __init__(self,x):
...         self.int = x % 7
...     def __str__(self):
...         return str(self.int)
...     __repr__ = __str__
...
>>> print F7(2)
2
>>> print F7(6)
6
>>> print F7(7)
0
>>> print F7(10)
3
```
Clocks over finite fields

Clock\((F_7)\) = \(\mathbb{F}(x;y) \in F_7 \times F_7: x^2 + y^2 = 1\).

Here \(F_7 = \{0; 1; 2; 3; 4; 5; 6\}\) with \(+\); \(-\); \(\times\) modulo 7.

E.g. \(2 \cdot 5 = 3\) and \(3 = 2 = 5\) in \(F_7\).

```python
>>> for x in range(7):
...     for y in range(7):
...         if (x*x+y*y) % 7 == 1:
...             print (x,y)
...
(0, 1)
(0, 6)
(1, 0)
(2, 2)
(2, 5)
(5, 2)
(5, 5)
(6, 0)
```

```python
>>> class F7:
...     def __init__(self,x):
...         self.int = x % 7
...     def __str__(self):
...         return str(self.int)
...     __repr__ = __str__
...
>>> print F7(2)
2
>>> print F7(6)
6
>>> print F7(7)
0
>>> print F7(10)
3
```
>>> for x in range(7):
...     for y in range(7):
...         if (x*x+y*y) % 7 == 1:
...             print (x,y)
...
(0, 1)
(0, 6)
(1, 0)
(2, 2)
(2, 5)
(5, 2)
(5, 5)
(6, 0)

>>> class F7:
...     def __init__(self, x):
...         self.int = x % 7
...     def __str__(self):
...         return str(self.int)
...     __repr__ = __str__
...
>>> print F7(2)
2
>>> print F7(6)
6
>>> print F7(7)
0
>>> print F7(10)
3
>>> for x in range(7):
...     for y in range(7):
...         if (x*x+y*y) % 7 == 1:
...             print (x,y)
...
(0, 1)
(0, 6)
(1, 0)
(2, 2)
(2, 5)
(5, 2)
(5, 5)
(6, 0)

>>> class F7:
...     def __init__(self,x):
...         self.int = x % 7
...     def __str__(self):
...         return str(self.int)
...     __repr__ = __str__
...
>>> print F7(2)
2
>>> print F7(6)
6
>>> print F7(7)
0
>>> print F7(10)
3

>>> F7.__eq__ = lambda a,b: \n...     a.int == b.int

>>> print F7(7) == F7(0)
True
>>> print F7(10) == F7(3)
True
>>> print F7(-3) == F7(4)
True
>>> print F7(0) == F7(1)
False
>>> print F7(0) == F7(2)
False
>>> print F7(0) == F7(3)
False
>> for x in range(7):
... for y in range(7):
... if (x*x+y*y) % 7 == 1:
... print (x,y)
...
(0, 1)
(0, 6)
(1, 0)
(2, 2)
(2, 5)
(5, 2)
(5, 5)
(6, 0)

>>> class F7:
...    def __init__(self, x):
...        self.int = x % 7
...    def __str__(self):
...        return str(self.int)
...    __repr__ = __str__
...
>>> print F7(2)
2
>>> print F7(6)
6
>>> print F7(7)
0
>>> print F7(10)
3

>>> F7.__eq__ = lambda a, b: a.int == b.int

>>> print F7(7) == F7(0)
True
>>> print F7(10) == F7(3)
True
>>> print F7(-3) == F7(4)
True
>>> print F7(0) == F7(1)
False
>>> print F7(0) == F7(2)
False
>>> print F7(0) == F7(3)
False
>>> class F7:
...     def __init__(self, x):
...         self.int = x % 7
...     def __str__(self):
...         return str(self.int)
...     __repr__ = __str__
...
>>> print F7(2)
2
>>> print F7(6)
6
>>> print F7(7)
0
>>> print F7(10)
3

>>> F7.__eq__ = lambda a, b: a.int == b.int
>>> print F7(7) == F7(0)
True
>>> print F7(10) == F7(3)
True
>>> print F7(-3) == F7(4)
True
>>> print F7(0) == F7(1)
False
>>> print F7(0) == F7(2)
False
>>> print F7(0) == F7(3)
False
```python
class F7:
    def __init__(self, x):
        self.int = x % 7
    def __str__(self):
        return str(self.int)
    __repr__ = __str__

>>> print F7(2)
2
>>> print F7(6)
6
>>> print F7(7)
0
>>> print F7(10)
3
>>> F7.__eq__ = lambda a, b: a.int == b.int

>>> print F7(7) == F7(0)
True
>>> print F7(10) == F7(3)
True
>>> print F7(-3) == F7(4)
True
>>> print F7(0) == F7(1)
False
>>> print F7(0) == F7(2)
False
>>> print F7(0) == F7(3)
False
```
class F7:
    def __init__(self, x):
        self.int = x % 7
    def __str__(self):
        return str(self.int)
    __repr__ = __str__

print F7(2)
print F7(6)
print F7(7)
print F7(10)

>>> F7.__eq__ = lambda a, b: a.int == b.int
>>> print F7(7) == F7(0)
True
>>> print F7(10) == F7(3)
True
>>> print F7(-3) == F7(4)
True
>>> print F7(0) == F7(1)
False
>>> print F7(0) == F7(2)
False
>>> print F7(0) == F7(3)
False

>>> F7.__add__ = lambda a, b: F7(a.int + b.int)
>>> F7.__sub__ = lambda a, b: F7(a.int - b.int)
>>> F7.__mul__ = lambda a, b: F7(a.int * b.int)

>>> print F7(2) + F7(5)
0
>>> print F7(2) - F7(5)
4
>>> print F7(2) * F7(5)
3
```python
>>> class F7:
...     def __init__(self, x):
...         self.int = x % 7
...     def __str__(self):
...         return str(self.int)
...     __repr__ = __str__
...     __eq__ = lambda a, b: a.int == b.int
...     __add__ = lambda a, b: F7(a.int + b.int)
...     __sub__ = lambda a, b: F7(a.int - b.int)
...     __mul__ = lambda a, b: F7(a.int * b.int)
...>>> print F7(2)
2
>>> print F7(6)
6
>>> print F7(7)
0
>>> print F7(10)
3
```
```python
>>> class F7:
...     def __init__(self, x):
...         self.int = x % 7
...     def __str__(self):
...         return str(self.int)
...     __repr__ = __str__
... >>> print F7(2)
2
>>> print F7(6)
6
>>> print F7(7)
0
>>> print F7(10)
3

>>> F7.__eq__ = lambda a, b: a.int == b.int
>>> print F7(7) == F7(0)
True
>>> print F7(10) == F7(3)
True
>>> print F7(-3) == F7(4)
True
>>> print F7(0) == F7(1)
False
>>> print F7(0) == F7(2)
False
>>> print F7(0) == F7(3)
False

>>> F7.__add__ = lambda a, b: F7(a.int + b.int)
>>> F7.__sub__ = lambda a, b: F7(a.int - b.int)
>>> F7.__mul__ = lambda a, b: F7(a.int * b.int)
>>> print F7(2) + F7(5)
0
>>> print F7(2) - F7(5)
4
>>> print F7(2) * F7(5)
3
```
>>> F7.__eq__ = lambda a,b: a.int == b.int
>>> print F7(7) == F7(0)
True
>>> print F7(10) == F7(3)
True
>>> print F7(-3) == F7(4)
True
>>> print F7(0) == F7(1)
False
>>> print F7(0) == F7(2)
False
>>> print F7(0) == F7(3)
False

>>> F7.__add__ = lambda a,b: F7(a.int + b.int)
>>> F7.__sub__ = lambda a,b: F7(a.int - b.int)
>>> F7.__mul__ = lambda a,b: F7(a.int * b.int)

>>> print F7(2) + F7(5)
0
>>> print F7(2) - F7(5)
4
>>> print F7(2) * F7(5)
3
```python
>>> F7.__eq__ = lambda a,b:
...    a.int == b.int
>>> F7(7) == F7(0)
True
>>> F7(10) == F7(3)
True
>>> F7(-3) == F7(4)
True
>>> F7(0) == F7(1)
False
>>> F7(0) == F7(2)
False
>>> F7(0) == F7(3)
False
```

```python
>>> F7.__add__ = lambda a,b:
...    F7(a.int + b.int)
>>> F7(2) + F7(5)
0
>>> F7.__sub__ = lambda a,b:
...    F7(a.int - b.int)
>>> F7(2) - F7(5)
4
>>> F7.__mul__ = lambda a,b:
...    F7(a.int * b.int)
>>> F7(2) * F7(5)
3
```

Larger example: Clock( F 1000003 ).

```python
class Fp:
...    @staticmethod
...    def clockadd(P1,P2):
...        x1,y1 = P1
...        x2,y2 = P2
...        x3 = x1*y2+y1*x2
...        y3 = y1*y2-x1*x2
...        return x3,y3
```

```python
p = 1000003
def clockadd(P1,P2):
    x1,y1 = P1
    x2,y2 = P2
    x3 = x1*y2+y1*x2
    y3 = y1*y2-x1*x2
    return x3,y3
```
Larger example: Clock(1000003).

```python
p = 1000003
class Fp:
    def clock_add(P1, P2):
        x1, y1 = P1
        x2, y2 = P2
        x3 = x1 * y2 + y1 * x2
        y3 = y1 * y2 - x1 * x2
        return x3, y3
```

```python
>>> print F7(2) + F7(5)
0
>>> print F7(2) - F7(5)
4
>>> print F7(2) * F7(5)
3
```
>>> F7.__eq__ = lambda a,b: 
... a.int == b.int
>>> print F7(7) == F7(0)
True
>>> print F7(10) == F7(3)
True
>>> print F7(-3) == F7(4)
True
>>> print F7(0) == F7(1)
False
>>> print F7(0) == F7(2)
False
>>> print F7(0) == F7(3)
False

>>> F7.__add__ = lambda a,b: 
... F7(a.int + b.int)
>>> F7.__sub__ = lambda a,b: 
... F7(a.int - b.int)
>>> F7.__mul__ = lambda a,b: 
... F7(a.int * b.int)
>>> print F7(2) + F7(5)
0
>>> print F7(2) - F7(5)
4
>>> print F7(2) * F7(5)
3

Larger example: Clock(F1000003).

p = 1000003
class Fp:
...
def clockadd(P1,P2):
    x1,y1 = P1
    x2,y2 = P2
    x3 = x1*y2+y1*x2
    y3 = y1*y2-x1*x2
    return x3,y3
Larger example: Clock($F_{1000003}$).

```python
p = 1000003
class Fp:
    ...
def clockadd(P1,P2):
        x1,y1 = P1
        x2,y2 = P2
        x3 = x1*y2+y1*x2
        y3 = y1*y2-x1*x2
        return x3,y3
```

```python
>>> print F7(2) + F7(5)
0
>>> print F7(2) - F7(5)
4
>>> print F7(2) * F7(5)
3
```
def clockadd(P1, P2):
    x1, y1 = P1
    x2, y2 = P2
    x3 = x1 * y2 + y1 * x2
    y3 = y1 * y2 - x1 * x2
    return x3, y3
Larger example: Clock($\mathbf{F}_{1000003}$).

```python
p = 1000003
class Fp:
    ...
def clockadd(P1, P2):
        x1, y1 = P1
        x2, y2 = P2
        x3 = x1*y2+y1*x2
        y3 = y1*y2-x1*x2
        return x3, y3
```

```python
>>> P = (Fp(1000), Fp(2))
>>> P2 = clockadd(P, P)
>>> print P2
(4000, 7)
>>> P3 = clockadd(P2, P)
>>> print P3
(15000, 26)
>>> P4 = clockadd(P3, P)
>>> P5 = clockadd(P4, P)
>>> P6 = clockadd(P5, P)
>>> print P6
(780000, 1351)
>>> print clockadd(P3, P3)
(780000, 1351)
```
Larger example: \text{Clock}(\mathbb{F}_{1000003}).

\begin{Verbatim}
\begin{verbatim}
p = 1000003
class Fp:
    ... 
def clockadd(P1, P2):
    x1, y1 = P1
    x2, y2 = P2
    x3 = x1*y2 + y1*x2
    y3 = y1*y2 - x1*x2
    return x3, y3
\end{verbatim}
\end{Verbatim}

\begin{Verbatim}
>>> P = (Fp(1000), Fp(2))
>>> P2 = clockadd(P, P)
>>> print P2
(4000, 7)
>>> P3 = clockadd(P2, P)
>>> print P3
(15000, 26)
>>> P4 = clockadd(P3, P)
>>> P5 = clockadd(P4, P)
>>> P6 = clockadd(P5, P)
>>> print P6
(780000, 1351)
>>> print clockadd(P3, P3)
(780000, 1351)
\end{Verbatim}
Larger example: Clock($F_{1000003}$).

```python
p = 1000003
class Fp:
    ...
def clockadd(P1, P2):
    x1, y1 = P1
    x2, y2 = P2
    x3 = x1*y2 + y1*x2
    y3 = y1*y2 - x1*x2
    return x3, y3
```

```python
>>> P = (Fp(1000), Fp(2))
>>> P2 = clockadd(P, P)
>>> print P2
(4000, 7)
>>> P3 = clockadd(P2, P)
>>> print P3
(15000, 26)
>>> P4 = clockadd(P3, P)
>>> print P4
(780000, 1351)
>>> print clockadd(P3, P3)
(780000, 1351)
```
Example: Clock(F_{1000003}).

\[ p = 1000003 \]

```
class Fp:
    ...
def clockadd(P1, P2):
        x1, y1 = P1
        x2, y2 = P2
        x3 = x1*y2 + y1*x2
        y3 = y1*y2 - x1*x2
        return x3, y3
```

```
>>> P = (Fp(1000), Fp(2))
>>> P2 = clockadd(P, P)
>>> print P2
(4000, 7)
>>> P3 = clockadd(P2, P)
>>> print P3
(15000, 26)
>>> P4 = clockadd(P3, P)
>>> print P4
(780000, 1351)
>>> print clockadd(P3, P3)
(780000, 1351)
>>> def scalarmult(n, P):
    ...
    if n == 0:
        return (Fp(0), Fp(1))
    if n == 1:
        return P
    Q = scalarmult(n//2, P)
    Q = clockadd(Q, Q)
    if n % 2:
        Q = clockadd(P, Q)
    return Q
    ...
```

```
>>> n = ourSixDigitSecret
>>> scalarmult(n, P)
(947472, 736284)
```
Clock($F_{1000003}$).

```python
>>> P = (Fp(1000), Fp(2))
>>> P2 = clockadd(P, P)
>>> print P2
(4000, 7)
>>> P3 = clockadd(P2, P)
>>> print P3
(15000, 26)
>>> P4 = clockadd(P3, P)
>>> P5 = clockadd(P4, P)
>>> P6 = clockadd(P5, P)
>>> print P6
(780000, 1351)
>>> print clockadd(P3, P3)
(780000, 1351)
```

```python
>>> def scalarmult(n, P):
...    if n == 0:
...        return (Fp(0), Fp(1))
...    if n == 1:
...        return P
...    Q = scalarmult(n // 2, P)
...    Q = clockadd(Q, Q)
...    if n % 2:
...        Q = clockadd(P, Q)
...    return Q
...
```

```python
>>> n = oursixdigitsecret
>>> scalarmult(n, P)
(947472, 736284)
```

Can you figure out our secret $n$?
>>> p = 1000003
>>> class Fp:
...     def clockadd(P1, P2):
...         x1, y1 = P1
...         x2, y2 = P2
...         x3 = x1 * y2 + y1 * x2
...         y3 = y1 * y2 - x1 * x2
...         return x3, y3

>>> P = (Fp(1000), Fp(2))
>>> P2 = clockadd(P, P)
>>> print P2
(4000, 7)
>>> P3 = clockadd(P2, P)
>>> print P3
(15000, 26)
>>> P4 = clockadd(P3, P)
>>> print P4
(350000, 52)
>>> P5 = clockadd(P4, P)
>>> print P5
(700000, 105)
>>> P6 = clockadd(P5, P)
>>> print P6
(1400000, 210)
>>> print clockadd(P3, P3)
(780000, 1351)

>>> def scalarmult(n, P):
...     if n == 0:
...         return (Fp(0), Fp(1))
...     if n == 1:
...         return P
...     Q = scalarmult(n // 2, P)
...     Q = clockadd(Q, Q)
...     if n % 2:
...         Q = clockadd(P, Q)
...     return Q

>>> n = yoursixdigitsecret
>>> scalarmult(n, P)
(947472, 736284)

Can you figure out our secret n ?
>>> P = (Fp(1000),Fp(2))
>>> P2 = clockadd(P,P)
>>> print P2
(4000, 7)
>>> P3 = clockadd(P2,P)
>>> print P3
(15000, 26)
>>> P4 = clockadd(P3,P)
>>> P5 = clockadd(P4,P)
>>> P6 = clockadd(P5,P)
>>> print P6
(780000, 1351)
>>> print clockadd(P3,P3)
(780000, 1351)

>>> def scalarmult(n,P):
... if n == 0: \
... return (Fp(0),Fp(1))
... if n == 1: return P
... Q = scalarmult(n//2,P)
... Q = clockadd(Q,Q)
... if n % 2: Q = clockadd(P,Q)
... return Q
...

>>> n = oursixdigitsecret
>>> scalarmult(n,P)
(947472, 736284)

Can you figure out our secret \( n \)?
Clock cryptography

The “Clock Diffie–Hellman protocol”:

Standardize large prime \( p \) &
base point \((x; y) \in \text{Clock}(F_p)\).

Alice chooses big secret \( a \),
computes her public key \( a(x; y) \).

Bob chooses big secret \( b \),
computes his public key \( b(x; y) \).

Alice computes \( a(b(x; y)) \).

Bob computes \( b(a(x; y)) \).

They use this shared secret
to encrypt with AES-GCM etc.

Can you figure out our secret \( n \)?
Clock cryptography
The “Clock Diffie–Hellman protocol”:
Standardize large prime $p$ &
base point $(x, y) \in \text{Clock}(\mathbb{F}_p)$.

Alice chooses big secret $a$,
computes her public key $a(x, y)$.

Bob chooses big secret $b$,
computes his public key $b(x, y)$.

Alice computes $a(b(x, y))$.
Bob computes $b(a(x, y))$.
They use this shared secret to encrypt with AES-GCM etc.
```python
>>> def scalarmult(n,P):
...    if n == 0: \
...        return (Fp(0),Fp(1))
...    if n == 1: return P
...    Q = scalarmult(n//2,P)
...    Q = clockadd(Q,Q)
...    if n % 2: Q = clockadd(P,Q)
...    return Q
...
>>> n = oursixdigitsecret
>>> scalarmult(n,P)
(947472, 736284)
```

Can you figure out our secret $n$?

Clock cryptography

The “Clock Diffie–Hellman protocol”:

Standardize large prime $p$ & base point $(x, y) \in \text{Clock}(\mathbb{F}_p)$.

Alice chooses big secret $a$, computes her public key $a(x, y)$.

Bob chooses big secret $b$, computes his public key $b(x, y)$.

Alice computes $a(b(x, y))$.
Bob computes $b(a(x, y))$.

They use this shared secret to encrypt with AES-GCM etc.
Can you figure out our secret $n$?

Clock cryptography

The “Clock Diffie–Hellman protocol”:

Standardize large prime $p$ &
**base point** $(x, y) \in \text{Clock}(\mathbb{F}_p)$.

Alice chooses big secret $a$, computes her public key $a(x, y)$.

Bob chooses big secret $b$, computes his public key $b(x, y)$.

Alice computes $a(b(x, y))$. Bob computes $b(a(x, y))$.

They use this shared secret to encrypt with AES-GCM etc.
def scalarmult(n,P):
    n == 0: 
        return (Fp(0),Fp(1))
    n == 1: return P
    Q = scalarmult(n//2,P)
    Q = clockadd(Q,Q)
    n % 2: Q = clockadd(P,Q)
    return Q

n = oursixdigitsecret
scalarmult(n,P)
(947472, 736284)

Clock cryptography

The “Clock Diffie–Hellman protocol”:

Standardize large prime \( p \) &
base point \((x, y) \in \text{Clock}(\mathbb{F}_p)\).

Alice chooses big secret \( a \),
computes her public key \( a(x, y) \).

Bob chooses big secret \( b \),
computes his public key \( b(x, y) \).

Alice computes \( a(b(x, y)) \).
Bob computes \( b(a(x, y)) \).
They use this shared secret
to encrypt with AES-GCM etc.
Clock cryptography

The “Clock Diffie–Hellman protocol”:

Standardize large prime $p$ & base point $(x, y) \in \text{Clock}(\mathbb{F}_p)$.

Alice chooses big secret $a$, computes her public key $a(x, y)$.

Bob chooses big secret $b$, computes his public key $b(x, y)$.

Alice computes $a(b(x, y))$.

Bob computes $b(a(x, y))$.

They use this shared secret to encrypt with AES-GCM etc.

Can you figure out our secret $n$?
Clock cryptography

The “Clock Diffie–Hellman protocol”:

Standardize large prime $p$ &
base point $(x, y) \in \text{Clock}(\mathbb{F}_p)$.

Alice chooses big secret $a$,
computes her public key $a(x, y)$.

Bob chooses big secret $b$,
computes his public key $b(x, y)$.

Alice computes $a(b(x, y))$.
Bob computes $b(a(x, y))$.

They use this shared secret
to encrypt with AES-GCM etc.

Can you figure out our secret $n$?
Clock cryptography

The “Clock Diffie–Hellman protocol”:

Standardize large prime $p$ & base point $(x, y) \in \text{Clock}(\mathbb{F}_p)$.

Alice chooses big secret $a$, computes her public key $a(x, y)$.

Bob chooses big secret $b$, computes his public key $b(x, y)$.

Alice computes $a(b(x, y))$.
Bob computes $b(a(x, y))$.

They use this shared secret to encrypt with AES-GCM etc.

$$
\begin{align*}
\text{Alice's secret key } a \\
\downarrow \\
\text{Alice's public key } a(X, Y) \\
\{\text{Alice, Bob} \}'s \text{shared secret } ab(X, Y)
\end{align*}
$$

$$
\begin{align*}
\text{Bob's secret key } b \\
\downarrow \\
\text{Bob's public key } b(X, Y) \\
\{\text{Bob, Alice} \}'s \text{shared secret } ba(X, Y)
\end{align*}
$$
Clock cryptography

The “Clock Diffie–Hellman protocol”:

Standardize large prime $p$ & base point $(x, y) \in \text{Clock}(\mathbb{F}_p)$.

Alice chooses big secret $a$, computes her public key $a(x, y)$.

Bob chooses big secret $b$, computes his public key $b(x, y)$.

Alice computes $a(b(x, y))$. Bob computes $b(a(x, y))$.

They use this shared secret to encrypt with AES-GCM etc.

Warning #1:
Many $p$ are unsafe!

Warning #2:
Clocks aren’t elliptic!
To match RSA-3072 security need $p \approx 2^{1536}$. 
Clock cryptography

The "Clock Diffie–Hellman protocol":

- Standardize large prime \( p \) &
  base point \((x; y) \in \text{Clock}(F_p)\).

- Alice chooses big secret \( a \), computes her public key \( a(x; y) \).
- Bob chooses big secret \( b \), computes his public key \( b(x; y) \).
- Alice computes \( a(b(x; y)) \).
- Bob computes \( b(a(x; y)) \).

They use this shared secret to encrypt with AES-GCM etc.

Warning #1:
Many \( p \) are unsafe!

Warning #2:
Clocks aren’t elliptic!
To match RSA-3072 security need \( p \approx 2^{1536} \).

Warning #3:
Attacker sees more than public keys \( a(x; y) \) and \( b(x; y) \).
Attacker sees how much time Alice uses to compute \( a(b(x; y)) \).
Often attacker can see time for each operation performed by Alice, not just total time.
This reveals secret scalar \( a \).
Break by timing attacks, e.g., 2011 Brumley–Tuveri.
The "Clock Diffie–Hellman protocol":

Standardize a large prime $p$ & base point $(x; y) \in \text{Clock}(\mathbb{F}_p)$.

Alice chooses a big secret $a$, computes her public key $a(x; y)$.

Bob chooses a big secret $b$, computes his public key $b(x; y)$.

Alice computes $a(b(x; y))$.

Bob computes $b(a(x; y))$.

They use this shared secret to encrypt with AES-GCM etc.

Warning #1:
Many $p$ are unsafe!

Warning #2:
Clocks aren’t elliptic!
To match RSA-3072 security need $p \approx 2^{1536}$.

Warning #3:
Attacker sees more than public keys $a(x, y)$ and $b(x, y)$.
Attacker sees how much time Alice uses to compute $a(b(x, y))$.
Often attacker can see how much time for each operation performed by Alice, not just total time.
This reveals secret scalar $a$.
Break by timing attacks, e.g., 2011 Brumley–Tuveri.
Alice chooses big secret $a$, computes her public key $a(x, y)$. 
Bob chooses big secret $b$, computes his public key $b(x, y)$.

Alice computes $a(b(x, y))$. 
Bob computes $b(a(x, y))$. 

They use this shared secret to encrypt with AES-GCM etc.

Warning #1: 
Many $p$ are unsafe!

Warning #2: 
Clocks aren’t elliptic! 
To match RSA-3072 security need $p \approx 2^{1536}$.

Warning #3: 
Attacker sees more than public keys $a(x, y)$ and $b(x, y)$.
Attacker sees how much time Alice uses to compute $a(b(x, y))$. 
Often attacker can see time for each operation performed by Alice, not just total time. 
This reveals secret scalar $a$.

Break by timing attacks, e.g., 2011 Brumley–Tuveri.
Alice’s secret key $a$

↓ ↓ 

Alice’s public key $a(X, Y)$

Bob’s secret key $b$

↓ ↓ 

Bob’s public key $b(X, Y)$

\{Alice, Bob\}'s shared secret $ab(X, Y)$

$=$ \{Bob, Alice\}'s shared secret $ba(X, Y)$

Warning #1: Many $p$ are unsafe!

Warning #2: Clocks aren’t elliptic!

To match RSA-3072 security need $p \approx 2^{1536}$.

Warning #3: Attacker sees more than public keys $a(x, y)$ and $b(x, y)$.

Attacker sees how much time Alice uses to compute $a(b(x, y))$.

Often attacker can see time for each operation performed by Alice, not just total time.

This reveals secret scalar $a$.

Break by timing attacks, e.g., 2011 Brumley–Tuveri.
Alice's secret key $a$
\[ \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \]
Alice's public key $a(X, Y)$

Bob's secret key $b$
\[ \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \]
Bob's public key $b(X, Y)$

\{Alice, Bob\}'s shared secret $ab(X, Y)$

\{Bob, Alice\}'s shared secret $ba(X, Y)$

Warning #1:
Many $p$ are unsafe!

Warning #2:
Clocks aren’t elliptic!
To match RSA-3072 security need $p \approx 2^{1536}$.

Warning #3:
Attacker sees more than public keys $a(x, y)$ and $b(x, y)$.
Attacker sees how much time Alice uses to compute $a(b(x, y))$.
Often attacker can see time for each operation performed by Alice, not just total time.
This reveals secret scalar $a$.

Break by timing attacks, e.g., 2011 Brumley–Tuveri.

Fix: constant-time code, performing same operations no matter what scalar is.
Alice’s secret key $a$

↓

Bob’s secret key $b$

Alice’s public key $a(X, Y)$

↘

Bob’s public key $b(X, Y)$

Warning #1:
Many $p$ are unsafe!

Warning #2:
Clocks aren’t elliptic!

To match RSA-3072 security
need $p \approx 2^{1536}$.

Warning #3:
Attacker sees more than public keys $a(x, y)$ and $b(x, y)$.

Attacker sees how much time Alice uses to compute $a(b(x, y))$.

Often attacker can see time for each operation performed by Alice, not just total time.

This reveals secret scalar $a$.

Break by timing attacks, e.g., 2011 Brumley–Tuveri.

Fix: constant-time code, performing same operations no matter what scalar is.

Exercise
How many multiplications do you need to compute $(x_1y_2 + y_1x_2; y_1y_2 - x_1x_2)$?

How many multiplications do you need to double a point, i.e., to compute $(x_1y_1 + y_1x_1; y_1y_1 - x_1x_1)$?

How can you optimize the computation if squarings are cheaper than multiplications?

Assume $S < M < 2^S$. 

Exercise
How many multiplications do you need to compute $(x_1y_2 + y_1x_2; y_1y_2 - x_1x_2)$?
Warning #3:
Attacker sees more than public keys $a(x, y)$ and $b(x, y)$.
Attacker sees how much time Alice uses to compute $a(b(x, y))$.
Often attacker can see time for each operation performed by Alice, not just total time.
This reveals secret scalar $a$.

Break by timing attacks, e.g., 2011 Brumley–Tuveri.

Fix: constant-time code, performing same operations no matter what scalar is.

Exercise
How many multiplications do you need to compute
$(x_1y_2 + y_1x_2, y_1y_2 - x_1x_2)$?
How many multiplications do you need to double a point, i.e. to compute
$(x_1y_1 + y_1x_1, y_1y_1)$?
How can you optimize the computation if squarings are cheaper than multiplications?
Assume $S < M < 2S$. 

Exercise
How many multiplications do you need to compute $(x_1y_2 + y_1x_2, y_1y_2 - x_1x_2)$?
Warning #3:
Attacker sees more than public keys \(a(x, y)\) and \(b(x, y)\).

Attacker sees how much time Alice uses to compute \(a(b(x, y))\).

Often attacker can see time for each operation performed by Alice, not just total time.
This reveals secret scalar \(a\).

Break by timing attacks, e.g., 2011 Brumley–Tuveri.

Fix: \textbf{constant-time} code, performing same operations no matter what scalar is.

Exercise
How many multiplications do you need to compute \((x_1y_2 + y_1x_2, y_1y_2 - x_1x_2)\)?

How many multiplications do you need to double a point, i.e., to compute \((x_1y_1 + y_1x_1, y_1y_1 - x_1x_1)\)?

How can you optimize the computation if squarings are cheaper than multiplications?
Assume \(S < M < 2S\).

Exercise
How many multiplications do you need to compute \((x_1y_2 + y_1x_2, y_1y_2 - x_1x_2)\)?

How many multiplications do you need to double a point, i.e., to compute \((x_1y_1 + y_1x_1, y_1y_1 - x_1x_1)\)?

How can you optimize the computation if squarings are cheaper than multiplications?
Assume \(S < M < 2S\).
Warning #3:
Attacker sees more than public keys $a(x, y)$ and $b(x, y)$.
Attacker sees how much time Alice uses to compute $a(b(x, y))$.
Often attacker can see time for each operation performed by Alice, not just total time.
This reveals secret scalar $a$.

Break by timing attacks, e.g., 2011 Brumley–Tuveri.

Fix: constant-time code, performing same operations no matter what scalar is.

Exercise
How many multiplications do you need to compute $(x_1y_2 + y_1x_2, y_1y_2 - x_1x_2)$?

How many multiplications do you need to double a point, i.e. to compute $(x_1y_1 + y_1x_1, y_1y_1 - x_1x_1)$?

How can you optimize the computation if squarings are cheaper than multiplications?
Assume $S < M < 2S$. 

Warning #3:
Attacker sees more than public keys $a(x, y)$ and $b(x, y)$.
Attacker sees how much time Alice uses to compute $a(b(x, y))$.
Attacker can see time for operation performed by Alice, not just total time.
This reveals secret scalar $a$.

By timing attacks, e.g., 2011 Brumley–Tuveri.

Fix: constant-time code, performing same operations no matter what scalar is.

Exercise
How many multiplications do you need to compute $(x_1y_2 + y_1x_2, y_1y_2 - x_1x_2)$?

How many multiplications do you need to double a point, i.e. to compute $(x_1y_1 + y_1x_1, y_1y_1 - x_1x_1)$?

How can you optimize the computation if squarings are cheaper than multiplications?
Assume $\mathbf{S} < \mathbf{M} < 2\mathbf{S}$.

Addition on an Edwards curve
Change the curve on which Alice and Bob work.

$$\begin{align*}
\text{neutral} &= (0, 1) \\
\cdot P_1 &= (x_1, y_1) \\
\cdot P_2 &= (x_2, y_2) \\
\cdot P_3 &= (x_3, y_3) \\
\cdot \ldots &
\end{align*}$$

$$x^2 + y^2 = 1$$

Sum of $(x_1, y_1)$ and $(x_2, y_2)$ is
$$(x_1y_2 + y_1x_2, y_1y_2 - x_1x_2) = (1 - 30x_1x_2y_1y_2), (1 + 30x_1x_2y_1y_2).$$
Exercise

How many multiplications do you need to compute
\((x_1y_2 + y_1x_2, y_1y_2 - x_1x_2)\)?

How many multiplications do you need to double a point, i.e. to compute
\((x_1y_1 + y_1x_1, y_1y_1 - x_1x_1)\)?

How can you optimize the computation if squarings are cheaper than multiplications?

Assume \(S < M < 2S\).

Addition on an Edwards curve

Change the curve on which Alice and Bob work.

Sum of \((x_1, y_1)\) and \((x_2, y_2)\) is
\[
\left(\frac{(x_1y_2+y_1x_2)}{1-30x_1x_2y_1y_2}, \frac{(y_1y_2-x_1x_2)}{1+30x_1x_2y_1y_2}\right).
\]
Exercise

How many multiplications do you need to compute 
\((x_1y_2 + y_1x_2, y_1y_2 - x_1x_2)\)?

How many multiplications do you need to double a point, i.e. to compute 
\((x_1y_1 + y_1x_1, y_1y_1 - x_1x_1)\)?

How can you optimize the computation if squarings are cheaper than multiplications? Assume \(S < M < 2S\).

Addition on an Edwards curve

Change the curve on which Alice and Bob work.

\[ x^2 + y^2 = 1 - 30x^2y^2. \]

Sum of \((x_1, y_1)\) and \((x_2, y_2)\):

- \((x, y) = ((x_1y_2 + y_1x_2)/(1-30x_1x_2y_1y_2), (y_1y_2 - x_1x_2)/(1+30x_1x_2y_1y_2))\).
Exercise

How many multiplications do you need to compute
\((x_1y_2 + y_1x_2, y_1y_2 - x_1x_2)\)?

How many multiplications do you need to double a point, i.e. to compute
\((x_1y_1 + y_1x_1, y_1y_1 - x_1x_1)\)?

How can you optimize the computation if squarings are cheaper than multiplications?
Assume \(S < M < 2S\).

Addition on an Edwards curve

Change the curve on which Alice and Bob work.

\[ x^2 + y^2 = 1 - 30x^2y^2. \]

Sum of \((x_1, y_1)\) and \((x_2, y_2)\) is
\[
\left( \frac{(x_1y_2 + y_1x_2)}{(1-30x_1x_2y_1y_2)}, \frac{(y_1y_2 - x_1x_2)}{(1+30x_1x_2y_1y_2)} \right).
\]
Exercise
How many multiplications do you need to compute \((x_1 y_2 + y_1 x_2; y_1 y_2 - x_1 x_2)\)?

How many multiplications do you need to double a point, i.e. to compute \((x_1 y_1 + y_1 x_1; y_1 y_1 - x_1 x_1)\)?

How can you optimize the computation if squarings are cheaper than multiplications? Assume \(S < M < 2S\).

### Addition on an Edwards curve

Change the curve on which Alice and Bob work.

![Diagram of Edwards curve](image)

neutral = \((0, 1)\)

\[ P_1 = (x_1, y_1) \]

\[ P_2 = (x_2, y_2) \]

\[ P_3 = (x_3, y_3) \]

\[ x^2 + y^2 = 1 - 30x^2y^2. \]

Sum of \((x_1, y_1)\) and \((x_2, y_2)\) is

\[ ((x_1 y_2 + y_1 x_2)/(1 - 30x_1 x_2 y_1 y_2), \]
\[ (y_1 y_2 - x_1 x_2)/(1 + 30x_1 x_2 y_1 y_2)) \].

The clock again, for comparison:

\[ x^2 + y^2 = 1. \]

Sum of \((x_1 y_2 + y_1 x_2; y_1 y_2 - x_1 x_2)\):

\[ (x_1 y_2 + y_1 x_2)/(1 - 30x_1 x_2 y_1 y_2), \]
\[ (y_1 y_2 - x_1 x_2)/(1 + 30x_1 x_2 y_1 y_2)) \].
How many multiplications do you need to compute \((x_1 y_2 + y_1 x_2; y_1 y_2 - x_1 x_2)\)?

How many multiplications do you need to double a point, i.e. to compute \((x_1 y_1 + y_1 x_1; y_1 y_1 - x_1 x_1)\)?

How can you optimize the computation if squarings are cheaper than multiplications?

Assume \(S < M < 2S\).

**Addition on an Edwards curve**

Change the curve on which Alice and Bob work.

\[ x^2 + y^2 = 1 - 30x^2y^2. \]

Sum of \((x_1, y_1)\) and \((x_2, y_2)\) is
\[
\left( ((x_1 y_2 + y_1 x_2)/(1 - 30x_1 x_2 y_1 y_2)),
(y_1 y_2 - x_1 x_2)/(1 + 30x_1 x_2 y_1 y_2) \right).
\]

The clock again, for comparison:
Exercise
How many multiplications do you need to compute \((x_1 y_2 + y_1 x_2; y_1 y_2 - x_1 x_2)\)?

How many multiplications do you need to double a point, i.e. to compute \((x_1 y_1 + y_1 x_1; y_1 y_1 - x_1 x_1)\)?

How can you optimize the computation if squarings are cheaper than multiplications?

Assume \(S < M < 2S\).

Addition on an Edwards curve

Change the curve on which Alice and Bob work.

The clock again, for comparison:
Addition on an Edwards curve

Change the curve on which Alice and Bob work.

\[ x^2 + y^2 = 1 - 30x^2y^2. \]

Sum of \((x_1, y_1)\) and \((x_2, y_2)\) is
\[
\left(\frac{x_1y_2 + y_1x_2}{1 - 30x_1x_2y_1y_2}, \frac{y_1y_2 - x_1x_2}{1 + 30x_1x_2y_1y_2}\right).
\]
Addition on an Edwards curve

Change the curve on which Alice and Bob work.

\[
\text{neutral} = (0, 1)
\]

\[
P_1 = (x_1, y_1)
\]

\[
P_2 = (x_2, y_2)
\]

\[
P_3 = (x_3, y_3)
\]

\[
x^2 + y^2 = 1.
\]

Sum of \((x_1, y_1)\) and \((x_2, y_2)\) is
\[
(x_1y_2 + y_1x_2, y_1y_2 - x_1x_2).
\]

The clock again, for comparison:

\[
\text{neutral} = (0, 1)
\]

\[
P_1 = (x_1, y_1)
\]

\[
P_2 = (x_2, y_2)
\]

\[
P_3 = (x_3, y_3)
\]

"Hey, there were divisions in the Edwards addition law! What if the denominators are 0?"

Answer: They aren't!

If \(x_i = 0\) or \(y_i = 0\) then 
\[
1 \pm 30x_1x_2y_1y_2 \neq 0.
\]

If \(x^2 + y^2 = 1 - 30x_2y_2\) then 
\[
30x_2y_2 < 1
\]
so \(\sqrt{30} | xy| < 1\).
Addition on an Edwards curve

Change the curve on which Alice and Bob work.

neutral = (0; 1)

\[ P_1 = (x_1; y_1) \]

\[ P_2 = (x_2; y_2) \]

\[ P_3 = (x_3; y_3) \]

\[ x^2 + y^2 = 1. \]

Sum of \((x_1; y_1)\) and \((x_2; y_2)\) is

\[ (x_1y_2 + y_1x_2, \ y_1y_2 - x_1x_2). \]

The clock again, for comparison:

Hey, there were divisions in the Edwards addition law! What if the denominators are 0?

Answer: They aren't!

If \(x_i = 0\) or \(y_i = 0\) then

\[ 1 \pm 30x_1x_2y_1y_2 \neq 0. \]

If \(x^2 + y^2 = 1 - 30x^2y^2\) then \(30x^2y^2 < 1\) so \(\sqrt{30} \ |xy| < 1\).
Addition on an Edwards curve

Change the curve on which Alice and Bob work.

Neutral = (0;1)

\[ P_1 = (x_1, y_1) \]
\[ P_2 = (x_2, y_2) \]
\[ P_3 = (x_3, y_3) \]

\[ x^2 + y^2 = 1. \]

Sum of \((x_1, y_1)\) and \((x_2, y_2)\) is
\[ (x_1y_2 + y_1x_2, y_1y_2 - x_1x_2). \]

The clock again, for comparison:

Hey, there were divisions in the Edwards addition law! What if the denominators are 0?

Answer: They aren’t!

If \(x_i = 0\) or \(y_i = 0\) then
\[ 1 \pm 30x_1x_2y_1y_2 = 1 \neq 0. \]

If \(x^2 + y^2 = 1 - 30x^2y^2\) then \(30x^2y^2 < 1\) so \(\sqrt{30} |xy| < 1.\)
The clock again, for comparison:

\[ y \]

\[ \text{neutral} = (0, 1) \]

\[ P_1 = (x_1, y_1) \]

\[ P_2 = (x_2, y_2) \]

\[ P_3 = (x_3, y_3) \]

\[ x^2 + y^2 = 1. \]

Sum of \((x_1, y_1)\) and \((x_2, y_2)\) is

\[ (x_1y_2 + y_1x_2, \quad y_1y_2 - x_1x_2). \]

“Hey, there were divisions in the Edwards addition law!
What if the denominators are 0?”
Answer: They aren’t!
If \(x_i = 0\) or \(y_i = 0\) then
\[ 1 \pm 30x_1x_2y_1y_2 = 1 \neq 0. \]
If \(x^2 + y^2 = 1 - 30x^2y^2\) then \(30x^2y^2 < 1\)
so \(\sqrt{30} |xy| < 1.\)
The clock again, for comparison:

```
  y
↑
neutral = (0, 1)
P1 = (x1, y1)
P2 = (x2, y2)
P3 = (x3, y3)
```

\[ x^2 + y^2 = 1. \]

Sum of \((x_1, y_1)\) and \((x_2, y_2)\) is

\[
(x_1y_2 + y_1x_2, \\
y_1y_2 - x_1x_2).
\]

“Hey, there were divisions in the Edwards addition law! What if the denominators are 0?”

Answer: They aren’t!

If \(x_i = 0\) or \(y_i = 0\) then

\[ 1 \pm 30x_1x_2y_1y_2 = 1 \neq 0. \]

If \(x^2 + y^2 = 1 - 30x^2y^2\) then \(30x^2y^2 < 1\) so \(\sqrt{30} \ |xy| < 1\).

If \(x_1^2 + y_1^2 = 1 - 30x_1^2y_1^2\) and \(x_2^2 + y_2^2 = 1 - 30x_2^2y_2^2\) then \(\sqrt{30} \ |x_1y_1| < 1\) and \(\sqrt{30} \ |x_2y_2| < 1\).
The clock again, for comparison:

\[
\begin{align*}
\text{neutral} &= (0, 1) \\
P_1 &= (x_1, y_1) \\
P_2 &= (x_2, y_2) \\
P_3 &= (x_3, y_3)
\end{align*}
\]

\[x^2 + y^2 = 1.\]

Sum of \((x_1, y_1)\) and \((x_2, y_2)\) is \((x_1y_2 + y_1x_2, y_1y_2 - x_1x_2)\).

“Hey, there were divisions in the Edwards addition law! What if the denominators are 0?”

Answer: They aren’t!

If \(x_i = 0\) or \(y_i = 0\) then

\[1 \pm 30x_1x_2y_1y_2 = 1 \neq 0.\]

If \(x^2 + y^2 = 1 - 30x^2y^2\) then \(30x^2y^2 < 1\)

so \(\sqrt{30} |xy| < 1.\)

If \(x_1^2 + y_1^2 = 1 - 30x_1^2y_1^2\) and \(x_2^2 + y_2^2 = 1 - 30x_2^2y_2^2\) then \(\sqrt{30} |x_1y_1| < 1\)

and \(\sqrt{30} |x_2y_2| < 1\)

so \(30 |x_1y_1x_2y_2| < 1\)

so \(1 \pm 30x_1x_2y_1y_2 > 0.\)
The contact again, for comparison:

\[
\begin{align*}
\text{neutral} &= (0; 1) \\
\text{P}_1 &= (x_1; y_1) \\
\text{P}_2 &= (x_2; y_2) \\
\text{P}_3 &= (x_3; y_3)
\end{align*}
\]

\[x^2 + y^2 = 1.\]

\[
(x_1, y_1) \text{ and } (x_2, y_2) \text{ is } y_1x_2, x_1y_2.\]

“Hey, there were divisions in the Edwards addition law! What if the denominators are 0?”

Answer: They aren’t!

If \(x_i = 0\) or \(y_i = 0\) then

\[1 \pm 30x_1x_2y_1y_2 = 1 \neq 0.\]

If \(x^2 + y^2 = 1 - 30x^2y^2\) then \(30x^2y^2 < 1\)

so \(\sqrt{30} |xy| < 1.\)

If \(x_1^2 + y_1^2 = 1 - 30x_1^2y_1^2\) and \(x_2^2 + y_2^2 = 1 - 30x_2^2y_2^2\) then \(\sqrt{30} |x_1y_1| < 1\)

and \(\sqrt{30} |x_2y_2| < 1\)

so \(30|x_1y_1x_2y_2| < 1\)

so \(1 \pm 30x_1x_2y_1y_2 > 0.\)

The Edwards addition law

\((x_1, y_1) + (x_2, y_2) = ((x_1y_2 + y_1x_2) = (1 - 30x_1x_2y_1y_2),
(\text{of proof are easy:}
addition law is commutative;
(0; 1) is neutral element;
(\text{addition law is associative.})\)

Other parts of proof are easy:

\((x_1, y_1) + (x_1, -y_1) = (0; 1).\)
“Hey, there were divisions in the Edwards addition law! What if the denominators are 0?”

Answer: They aren’t!

If \( x_i = 0 \) or \( y_i = 0 \) then
\[
1 \pm 30x_1x_2y_1y_2 = 1 \neq 0.
\]

If \( x^2 + y^2 = 1 - 30x^2y^2 \) then \( 30x^2y^2 < 1 \)
so \( \sqrt{30} |xy| < 1 \).

If \( x_1^2 + y_1^2 = 1 - 30x_1^2y_1^2 \)
and \( x_2^2 + y_2^2 = 1 - 30x_2^2y_2^2 \)
then \( \sqrt{30} |x_1y_1| < 1 \)
and \( \sqrt{30} |x_2y_2| < 1 \)
so \( 30 |x_1y_1x_2y_2| < 1 \)
so \( 1 \pm 30x_1x_2y_1y_2 > 0. \)

The Edwards addition law
\[
(x_1, y_1) + (x_2, y_2) = ((x_1y_2 + y_1x_2)/(1 - 30x_1x_2y_1y_2)),
(y_1y_2 - x_1x_2)/(1 - 30x_1x_2y_1y_2))
\]
is a group law for the curve \( x^2 + y^2 = 1 - 30x^2y^2 \).

Some calculation required: the addition result is on curve; addition law is associative.

Other parts of proof are easy: addition law is commutative;
\((0 ; 1)\) is neutral element;
\((x_1, y_1) + (-x_1, y_1) = (0 ; 1)\).
“Hey, there were divisions in the Edwards addition law! What if the denominators are 0?”

Answer: They aren’t!

If \( x_i = 0 \) or \( y_i = 0 \) then
\[
1 \pm 30x_1 x_2 y_1 y_2 = 1 \neq 0.
\]

If \( x^2 + y^2 = 1 - 30x^2 y^2 \) then \( 30x^2 y^2 < 1 \)
so \( \sqrt{30} |xy| < 1. \)

If \( x_1^2 + y_1^2 = 1 - 30x_1^2 y_1^2 \) and \( x_2^2 + y_2^2 = 1 - 30x_2^2 y_2^2 \) then \( \sqrt{30} |x_1 y_1| < 1 \)
and \( \sqrt{30} |x_2 y_2| < 1 \)
so \( 30 |x_1 y_1 x_2 y_2| < 1 \)
so \( 1 \pm 30x_1 x_2 y_1 y_2 > 0. \)

The Edwards addition law
\[
(x_1, y_1) + (x_2, y_2) =
((x_1 y_2 + y_1 x_2) / (1 - 30x_1 x_2 y_1 y_2),
(y_1 y_2 - x_1 x_2) / (1 + 30x_1 x_2 y_1 y_2))
\]
is a group law for the curve
\[
x^2 + y^2 = 1 - 30x^2 y^2.
\]

Some calculation required: addition result is on curve;
addition law is associative.

Other parts of proof are easy: addition law is commutative;
(0, 1) is neutral element;
\((x_1, y_1) + (-x_1, y_1) = (0, 1)\).
“Hey, there were divisions in the Edwards addition law! What if the denominators are 0?”
Answer: They aren’t!

If $x_i = 0$ or $y_i = 0$ then
\[ 1 \pm 30x_1x_2y_1y_2 = 1 \neq 0. \]
If $x^2 + y^2 = 1 - 30x^2y^2$ then $30x^2y^2 < 1$
so $\sqrt{30}|xy| < 1$.

If $x_1^2 + y_1^2 = 1 - 30x_1^2y_1^2$ and $x_2^2 + y_2^2 = 1 - 30x_2^2y_2^2$
then $\sqrt{30}|x_1y_1| < 1$
and $\sqrt{30}|x_2y_2| < 1$
so $30|x_1y_1x_2y_2| < 1$
so $1 \pm 30x_1x_2y_1y_2 > 0$.

The Edwards addition law
\[
(x_1, y_1) + (x_2, y_2) = 
\left( \frac{x_1y_2 + y_1x_2}{1 - 30x_1x_2y_1y_2}, \right.
\left. \frac{y_1y_2 - x_1x_2}{1 + 30x_1x_2y_1y_2} \right)
\]
is a group law for the curve
\[ x^2 + y^2 = 1 - 30x^2y^2. \]

Some calculation required:
addition result is on curve;
addition law is associative.

Other parts of proof are easy:
addition law is commutative;
(0, 1) is neutral element;
\[(x_1, y_1) + (-x_1, y_1) = (0, 1).\]
Hey, there were divisions in the Edwards addition law! What if the denominators are 0?"

They aren’t!

If \( x_i = 0 \) or \( y_i = 0 \) then

\[
\begin{align*}
1 \pm 30 x_1 x_2 y_1 y_2 &= 1 \\
30 x_1 x_2 y_1 y_2 &< 1
\end{align*}
\]

so

\[
\sqrt{30} |xy| < 1.
\]

If \( x_1 + y_1 = 0 \) and \( y_2 - x_2 = 0 \) then

\[
\sqrt{30} |x_1 y_1 x_2 y_2| < 1.
\]

so

\[
1 \pm 30 x_1 x_2 y_1 y_2 > 0.
\]

The Edwards addition law

\[
(x_1, y_1) + (x_2, y_2) = (x_1 y_2 + y_1 x_2)/(1 - 30 x_1 x_2 y_1 y_2),
\]

\[
(y_1 y_2 - x_1 x_2)/(1 + 30 x_1 x_2 y_1 y_2))
\]

is a group law for the curve

\[
x^2 + y^2 = 1 - 30 x^2 y^2.
\]

Some calculation required:

addition result is on curve;
addition law is associative.

Other parts of proof are easy:

addition law is commutative;
(0, 1) is neutral element;
(\( x_1, y_1 \) + (\( -x_1, y_1 \) = (0, 1).

Edwards curves mod \( p \)

Choose an odd prime \( p \).
Choose a non-square \( d \in F_p \).

\[
\{ (x, y) \in F_p \times F_p : x^2 + y^2 = 1 + dx^2 y^2 \}
\]

is a “complete Edwards curve”.

Roughly \( p + 1 \) pairs \((x, y)\).

```python
def edwardsadd(P1, P2):
    x1, y1 = P1
    x2, y2 = P2
    x3 = (x1*y2+y1*x2)/(1+d*x1*x2*y1*y2)
    y3 = (y1*y2-x1*x2)/(1-d*x1*x2*y1*y2)
    return x3, y3
```

If \( x = 0 \) or \( y = 0 \) then

\[
\begin{align*}
x^2 y^2 &= 1 - 30 x^2 y^2 \\
x_1 x_2 y_1 y_2 &= 1 \neq 0.
\end{align*}
\]

\[
y_1^2 = 1 - 30 x_1^2 y_1^2
\]

\[
y_2^2 = 1 - 30 x_2^2 y_2^2
\]

\[
0 |x_1 y_1| < 1
\]

\[
0 |x_2 y_2| < 1
\]

\[
|y_1 y_2 x_1 x_2| < 1
\]

\[
0 x_1 x_2 y_1 y_2 > 0.
\]
Hey, there were divisions
in the Edwards addition law!
What if the denominators are 0?”
Answer: They aren’t!

If \( x_1 = 0 \) or \( y_1 = 0 \) then

\[
\frac{1}{1 + 30x_1x_2y_1y_2} > 0.
\]

The Edwards addition law
\( (x_1, y_1) + (x_2, y_2) = ((x_1y_2 + y_1x_2)/(1-30x_1x_2y_1y_2),
(y_1y_2 - x_1x_2)/(1+30x_1x_2y_1y_2)) \)
is a group law for the curve
\( x^2 + y^2 = 1 - 30x^2y^2 \).

Some calculation required:
addition result is on curve;
addition law is associative.

Other parts of proof are easy:
addition law is commutative;
(0, 1) is neutral element;
\( (x_1, y_1) + (-x_1, y_1) = (0, 1) \).

Edwards curves mod \( p \)
Choose an odd prime \( p \).
Choose a non-square \( d \in \mathbb{F}_p \).
\{ (x, y) \in \mathbb{F}_p \times \mathbb{F}_p : x^2 + y^2 = 1 - 30x^2y^2 \}
is a “complete Edwards curve”.

Roughly \( p + 1 \) pairs

def edwardsadd(P1, P2):
x1,y1 = P1
x2,y2 = P2
x3 = (x1*y2+y1*x2)/(1+d*x1*x2*y1*y2)
y3 = (y1*y2-x1*x2)/(1-d*x1*x2*y1*y2)
return x3,y3
Hey, there were divisions in the Edwards addition law! What if the denominators are 0?

Answer: They aren’t!

If \( x_i = 0 \) or \( y_i = 0 \) then \( 1 \pm \frac{30}{x_1 x_2 y_1 y_2} > 0 \).

The Edwards addition law
\[
(x_1, y_1) + (x_2, y_2) = \\
\left(\frac{x_1 y_2 + y_1 x_2}{1 - 30 x_1 x_2 y_1 y_2}, \frac{y_1 y_2 - x_1 x_2}{1 + 30 x_1 x_2 y_1 y_2}\right)
\]
is a group law for the curve
\[
x^2 + y^2 = 1 - 30 x^2 y^2.
\]

Some calculation required:
- addition result is on curve;
- addition law is associative.

Other parts of proof are easy:
- addition law is commutative;
- \((0, 1)\) is neutral element;
- \((x_1, y_1) + (-x_1, y_1) = (0, 1)\).

Edwards curves mod \( p \)
Choose an odd prime \( p \).
Choose a non-square \( d \in \mathbb{F}_p \).
\[
\{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p : x^2 + y^2 = 1 + dx^2 y^2\}
\]
is a “complete Edwards curve.”

Roughly \( p + 1 \) pairs \((x, y)\).

```python
def edwardsadd(P1, P2):
    x1, y1 = P1
    x2, y2 = P2
    x3 = (x1*y2+y1*x2)/(1+d*x1*x2*y1*y2)
    y3 = (y1*y2-x1*x2)/(1-d*x1*x2*y1*y2)
    return x3, y3
```

Edwards curves mod \( p \)
Choose an odd prime \( p \).
Choose a non-square \( d \in \mathbb{F}_p \).
\[
\{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p : x^2 + y^2 = 1 + dx^2 y^2\}
\]
is a “complete Edwards curve.”

Roughly \( p + 1 \) pairs \((x, y)\).

```python
def edwardsadd(P1, P2):
    x1, y1 = P1
    x2, y2 = P2
    x3 = (x1*y2+y1*x2)/(1+d*x1*x2*y1*y2)
    y3 = (y1*y2-x1*x2)/(1-d*x1*x2*y1*y2)
    return x3, y3
```
The Edwards addition law
\((x_1, y_1) + (x_2, y_2) =
((x_1 y_2 + y_1 x_2)/(1 - 30 x_1 x_2 y_1 y_2),
(y_1 y_2 - x_1 x_2)/(1 + 30 x_1 x_2 y_1 y_2))\)

is a group law for the curve
\(x^2 + y^2 = 1 - 30 x^2 y^2.\)

Some calculation required:
addition result is on curve;
addition law is associative.

Other parts of proof are easy:
addition law is commutative;
(0, 1) is neutral element;
\((x_1, y_1) + (\pm x_1, y_1) = (0, 1)\).

Edwards curves mod \(p\)

Choose an odd prime \(p\).
Choose a non-square \(d \in \mathbb{F}_p\).
\(\{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p : x^2 + y^2 = 1 + dx^2 y^2\}\)
is a “complete Edwards curve”.
Roughly \(p + 1\) pairs \((x, y)\).

```python
def edwardsadd(P1, P2):
x1, y1 = P1
x2, y2 = P2
x3 = (x1*y2+y1*x2) / \n(1+d*x1*x2*y1*y2)
y3 = (y1*y2-x1*x2) / \n(1-d*x1*x2*y1*y2)
return x3, y3
```
Edwards addition law
\[(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1 y_2 + y_1 x_2}{1 - 30x_1 x_2 y_1 y_2}, \frac{y_1 y_2 - x_1 x_2}{1 + 30x_1 x_2 y_1 y_2}\right)\]
is a group law for the curve
\[x^2 + y^2 = 1 - 30x^2 y^2.\]

Some calculation required:
- result is on curve;
- addition law is associative.

Other parts of proof are easy:
- addition law is commutative;
- \((0, 1)\) is neutral element;
- \((x_1, y_1) + (\mathbf{−}x_1, y_1) = (0, 1)\).

Edwards curves mod \(p\)

Choose an odd prime \(p\).
Choose a non-square \(d \in \mathbb{F}_p\).
\[\{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p : x^2 + y^2 = 1 + dx^2 y^2\}\]
is a “complete Edwards curve”.

Roughly \(p + 1\) pairs \((x, y)\).

```python
def edwardsadd(P1, P2):
    x1, y1 = P1
    x2, y2 = P2
    x3 = (x1*y2+y1*x2) / \
         (1+d*x1*x2*y1*y2)
    y3 = (y1*y2-x1*x2) / \
         (1-d*x1*x2*y1*y2)
    return x3, y3
```

Denominators are never 0.
But need different proof;
“\(x^2 + y^2 > 0\)” doesn’t work.
The Edwards addition law

\[
(\begin{array}{l}
(x_1; y_1) + (x_2; y_2) = \\
(x_1y_2 + y_1x_2, y_1y_2 - x_1x_2)
\end{array}) = (1 - 30x_1x_2y_1y_2, 1 + 30x_1x_2y_1y_2),
\]

is a group law for the curve

\[
x^2 + y^2 = 1 - 30x_1x_2y_1y_2.
\]

Some calculation required:

- Addition result is on curve;
- Addition law is associative.

Other parts of proof are easy:

- Addition law is commutative;
- \((0; 1)\) is neutral element;
- \((x_1; y_1) + (\ - x_1; y_1) = (0; 1)\).

---

Edwards curves mod \(p\)

Choose an odd prime \(p\).

Choose a non-square \(d \in \mathbb{F}_p\).

\[
\{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p : x^2 + y^2 = 1 + dx^2y^2\}
\]

is a “complete Edwards curve”.

Roughly \(p + 1\) pairs \((x, y)\).

```python
def edwardsadd(P1, P2):
    x1, y1 = P1
    x2, y2 = P2
    x3 = (x1*y2+y1*x2) / (1+d*x1*x2*y1*y2)
    y3 = (y1*y2-x1*x2) / (1-d*x1*x2*y1*y2)
    return x3, y3
```

Denominators are never 0.

But need different proof;

“\(x^2 + y^2 > 0\)” doesn’t work.
Edwards curves mod $p$

Choose an odd prime $p$.

Choose a non-square $d \in \mathbb{F}_p$.

$$\{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p : x^2 + y^2 = 1 + dx^2y^2\}$$

is a “complete Edwards curve”.

Roughly $p + 1$ pairs $(x, y)$.

```python
def edwardsadd(P1, P2):
    x1, y1 = P1
    x2, y2 = P2
    x3 = (x1*y2 + y1*x2) / \n         (1 + d*x1*x2*y1*y2)
    y3 = (y1*y2 - x1*x2) / \n         (1 - d*x1*x2*y1*y2)
    return x3, y3
```

Denominators are never 0.

But need different proof; “$x^2 + y^2 > 0$” doesn’t work.
Edwards curves mod $p$

Choose an odd prime $p$.
Choose a non-square $d \in \mathbb{F}_p$.

\[
\{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p : x^2 + y^2 = 1 + dx^2y^2\}
\]

is a “complete Edwards curve”.
Roughly $p + 1$ pairs $(x, y)$.

```python
def edwardsadd(P1, P2):
x1, y1 = P1
x2, y2 = P2
x3 = (x1*y2+y1*x2)/\n(1+d*x1*x2*y1*y2)
y3 = (y1*y2-x1*x2)/\n(1-d*x1*x2*y1*y2)
return x3, y3```

Denominators are never 0. But need different proof; “$x^2 + y^2 > 0$” doesn’t work.
Edwards curves mod $p$

Choose an odd prime $p$.
Choose a non-square $d \in \mathbb{F}_p$.

$\{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p : x^2 + y^2 = 1 + dx^2y^2 \}$

is a “complete Edwards curve”.

Roughly $p + 1$ pairs $(x, y)$.

```python
def edwardsadd(P1, P2):
    x1, y1 = P1
    x2, y2 = P2
    x3 = (x1*y2+y1*x2)/(1+d*x1*x2*y1*y2)
    y3 = (y1*y2-x1*x2)/(1-d*x1*x2*y1*y2)
    return x3, y3
```

Denominators are never 0.

But need different proof;
“$x^2 + y^2 > 0$” doesn’t work.

Answer: Can prove that the denominators are never 0.

Addition law is complete.
Edwards curves mod $p$

Choose an odd prime $p$.
Choose a non-square $d \in \mathbb{F}_p$.

$\{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p : x^2 + y^2 = 1 + dx^2y^2\}$

is a “complete Edwards curve”.
Roughly $p + 1$ pairs $(x, y)$.

```python
def edwardsadd(P1, P2):
    x1, y1 = P1
    x2, y2 = P2
    x3 = (x1*y2+y1*x2)/ (1+d*x1*x2*y1*y2)
    y3 = (y1*y2-x1*x2)/ (1-d*x1*x2*y1*y2)
    return x3, y3
```

Denominators are never 0.

But need different proof;
“$x^2 + y^2 > 0$” doesn’t work.

Answer: Can prove that
the denominators are never 0.
Addition law is complete.

This proof relies on
choosing non-square $d$. 

Edwards curves mod $p$

Choose an odd prime $p$.
Choose a non-square $d \in \mathbb{F}_p$.
\[
\{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p : 
    x^2 + y^2 = 1 + dx^2y^2\}
\]
is a “complete Edwards curve”.
Roughly $p + 1$ pairs $(x, y)$.

```python
def edwardsadd(P1, P2):
    x1, y1 = P1
    x2, y2 = P2
    x3 = (x1*y2+y1*x2)/\n        (1+d*x1*x2*y1*y2)
    y3 = (y1*y2-x1*x2)/\n        (1-d*x1*x2*y1*y2)
    return x3, y3
```

Denominators are never 0.
But need different proof;
“$x^2 + y^2 > 0$” doesn’t work.

Answer: Can prove that
the denominators are never 0.
Addition law is complete.

This proof relies on
choosing non-square $d$.

If we instead choose square $d$:
curve is still elliptic, and
addition seems to work,
but there are failure cases,
often exploitable by attackers.
Safe code is more complicated.
Edwards curves mod $p$

Choose an odd prime $p$.

Choose a non-square $d \in \mathbf{F}_p$.

$\{ (x;y) \in \mathbf{F}_p \times \mathbf{F}_p : x^2 + y^2 = 1 + dx^2y^2 \}$

Is a “complete Edwards curve”.

$p + 1$ pairs $(x, y)$.

Edwardsadd(P1,P2):

\[
\begin{align*}
    x1, y1 & = P1 \\
    x2, y2 & = P2 \\
    x3 & = (x1*y2 + y1*x2) / (1 + x1*x2*y1*y2) \\
    y3 & = (y1*y2 - x1*x2) / (1 + x1*x2*y1*y2) \\
    x3, y3 & 
\end{align*}
\]

Denominators are never 0.

But need different proof; “$x^2 + y^2 > 0$” doesn’t work.

Answer: Can prove that the denominators are never 0.

Addition law is \textbf{complete}.

This proof relies on choosing \textit{non-square} $d$.

If we instead choose square $d$:

curve is still elliptic, and addition \textit{seems to work},

but there are failure cases, often exploitable by attackers.

Safe code is more complicated.
Choose an odd prime $p$. Choose a non-square $d \in \mathbb{F}_p$. 

\[ \{ (x; y) \in \mathbb{F}_p \times \mathbb{F}_p : x^2 + y^2 = 1 + dx^2y^2 \} \]

is a “complete Edwards curve”.

Roughly $p + 1$ pairs $(x; y)$.

```python
def edwardsadd(P1, P2):
    x1, y1 = P1
    x2, y2 = P2
    x3 = (x1*y2+y1*x2)/(1+d*x1*x2*y1*y2)
    y3 = (y1*y2-x1*x2)/(1-d*x1*x2*y1*y2)
    return x3, y3
```

Denominators are never 0.

But need different proof; “$x^2 + y^2 > 0$” doesn’t work.

Answer: Can prove that the denominators are never 0. Addition law is complete.

This proof relies on choosing non-square $d$.

If we instead choose square $d$: curve is still elliptic, and addition seems to work, but there are failure cases, often exploitable by attackers. Safe code is more complicated.
Denominators are never 0.
But need different proof;
“$x^2 + y^2 > 0$” doesn’t work.

Answer: Can prove that
the denominators are never 0.
Addition law is complete.

This proof relies on
choosing non-square $d$.

If we instead choose square $d$:
curve is still elliptic, and
addition seems to work,
but there are failure cases,
often exploitable by attackers.
Safe code is more complicated.
Denominators are never 0.
But need different proof;
“$x^2 + y^2 > 0$” doesn’t work.

Answer: Can prove that
the denominators are never 0.
Addition law is complete.

This proof relies on
choosing non-square $d$.

If we instead choose square $d$:
curve is still elliptic, and
addition seems to work,
but there are failure cases,
often exploitable by attackers.
Safe code is more complicated.

Edwards curves are cool
Denominators are never 0. But need different proof; “x^2 + y^2 > 0” doesn’t work.

Can prove that denominators are never 0.

Addition law is complete.

Proof relies on choosing non-square d.

Instead choose square d: curve is still elliptic, and addition seems to work, but there are failure cases, often exploitable by attackers.

Safe code is more complicated.

Edwards curves are cool

ECDSA

Users can sign messages using Edwards curves.

Take a point P on an Edwards curve modulo a prime p > 2.

ECDSA signer needs to know the order of P.

There are only finitely many other points; about p in total.

Adding P to itself will eventually reach (0 ; 1); let ℓ be the smallest integer > 0 with ℓP = (0 ; 1).

This ℓ is the order of P.
Denominators are never 0. But need different proof; "x^2 + y^2 > 0" doesn’t work.

Answer: Can prove that the denominators are never 0. Addition law is complete. This proof relies on choosing non-square d. If we instead choose square d:

- curve is still elliptic, and addition seems to work,
- but there are failure cases, often exploitable by attackers.
- Safe code is more complicated.

**Edwards curves are cool**

**ECDSA**

Users can sign messages using Edwards curves.

Take a point P on an Edwards curve modulo a prime.

ECDSA signer needs to know the order of P.

There are only finitely many other points; about p in total.

Adding P to itself will eventually reach (0, 1); let l be the smallest integer > 0 with ℓP = (0, 1).

This l is the order of P.
Denominators are never 0. But need different proof; \( x^2 + y^2 > 0 \) doesn't work. Answer: Can prove that the denominators are never 0. Addition law is complete. This proof relies on choosing non-square \( d \). If we instead choose square \( d \): curve is still elliptic, and addition seems to work, but there are failure cases, often exploitable by attackers. Safe code is more complicated.

**ECDSA**

Users can sign messages using Edwards curves. Take a point \( P \) on an Edwards curve modulo a prime \( p > 2 \). ECDSA signer needs to know the order of \( P \). There are only finitely many points; about \( p \) in total. Adding \( P \) to itself will eventually reach \((0,1)\); let \( \ell \) be the smallest integer \( > 0 \) with \( \ell P = (0,1) \). This \( \ell \) is the order of \( P \).
Edwards curves are cool

ECDSA

Users can sign messages using Edwards curves.

Take a point $P$ on an Edwards curve modulo a prime $p > 2$.

ECDSA signer needs to know the order of $P$.

There are only finitely many other points; about $p$ in total. Adding $P$ to itself will eventually reach $(0, 1)$; let $\ell$ be the smallest integer $> 0$ with $\ell P = (0, 1)$. This $\ell$ is the order of $P$. 
Edwards curves are cool

**ECDSA**

Users can sign messages using Edwards curves.

Take a point $P$ on an Edwards curve modulo a prime $p > 2$.

ECDSA signer needs to know the order of $P$.

There are only finitely many other points; about $p$ in total.

Adding $P$ to itself will eventually reach $(0, 1)$; let $\ell$ be the smallest integer $> 0$ with $\ell P = (0, 1)$.

This $\ell$ is the order of $P$.

The signature scheme has as system parameters a curve $E$; a base point $P$; and a hash function $h$ with output length at least $\lceil \log_2 \ell \rceil + 1$.

Alice’s secret key is an integer $a$ and her public key is $P_A = aP$.

To sign message $m$, Alice computes $h(m)$; picks random $k$; computes $R = kP = (x_1, y_1)$; puts $r \equiv y_1 \mod \ell$; computes $s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell$.

The signature on $m$ is $(r; s)$. 

The signature scheme has as system parameters a curve $E$; a base point $P$; and a hash function $h$ with output length at least $\lceil \log_2 \ell \rceil + 1$.

Alice’s secret key is an integer $a$ and her public key is $P_A = aP$.

To sign message $m$, Alice computes $h(m)$; picks random $k$; computes $R = kP = (x_1, y_1)$; puts $r \equiv y_1 \mod \ell$; computes $s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell$.

The signature on $m$ is $(r; s)$. 

The signature scheme has as system parameters a curve $E$; a base point $P$; and a hash function $h$ with output length at least $\lceil \log_2 \ell \rceil + 1$.

Alice’s secret key is an integer $a$ and her public key is $P_A = aP$.

To sign message $m$, Alice computes $h(m)$; picks random $k$; computes $R = kP = (x_1, y_1)$; puts $r \equiv y_1 \mod \ell$; computes $s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell$.

The signature on $m$ is $(r; s)$.
ECDSA

Users can sign messages using Edwards curves.

Take a point $P$ on an Edwards curve modulo a prime $p > 2$.

ECDSA signer needs to know the order of $P$.

There are only finitely many other points; about $p$ in total.
Adding $P$ to itself will eventually reach $(0,1)$; let $\ell$ be the smallest integer $> 0$ with $\ell P = (0,1)$.
This $\ell$ is the order of $P$.

The signature scheme has as system parameters a curve $E$; a base point $P$; and a hash function $h$ with output length $\lceil \log_2 \ell \rceil + 1$.
Alice’s secret key is an integer $a$ and her public key is $P_A = aP$.

To sign message $m$,
Alice computes $h(m)$;
picks random $k$;
computes $R = kP = (x_1,y_1)$;
puts $r \equiv y_1 \mod \ell$;
computes $s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell$.
The signature on $m$ is $(r,s)$.
ECDSA

Users can sign messages using Edwards curves.

Take a point $P$ on an Edwards curve modulo a prime $p > 2$.

ECDSA signer needs to know the order of $P$.

There are only finitely many other points; about $p$ in total.

Adding $P$ to itself will eventually reach $(0, 1)$; let $\ell$ be the smallest integer $> 0$ with $\ell P = (0, 1)$.
This $\ell$ is the order of $P$.

The signature scheme has as system parameters a curve $E$; a base point $P$; and a hash function $h$ with output length at least $\lceil \log_2 \ell \rceil + 1$.

Alice’s secret key is an integer $a$ and her public key is $P_A = aP$.

To sign message $m$,
Alice computes $h(m)$;
picks random $k$;
computes $R = kP = (x_1, y_1)$;
puts $r \equiv y_1 \mod \ell$; computes $s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell$.

The signature on $m$ is $(r, s)$. 
ECDSA

Users can sign messages using Edwards curves.

Take a point $P$ on an Edwards curve modulo a prime $p > 2$.

ECDSA signer needs to know the order of $P$.

There are only finitely many other points; about $p$ in total.

Adding $P$ to itself will eventually reach $(0, 1)$; let $\ell$ be the smallest integer $> 0$ with $\ell P = (0, 1)$.

This $\ell$ is the order of $P$.

The signature scheme has as system parameters a curve $E$; a base point $P$; and a hash function $h$ with output length at least $\lceil \log_2 \ell \rceil + 1$.

Alice’s secret key is an integer $a$ and her public key is $P_A = aP$.

To sign message $m$,

Alice computes $h(m)$;
picks random $k$;
computes $R = kP = (x_1, y_1)$;
puts $r \equiv y_1 \mod \ell$; computes $s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell$.

The signature on $m$ is $(r, s)$. 
Users can sign messages using Edwards curves. Take a point $P$ on an Edwards curve modulo a prime $p > 2$. ECDSA signer needs to know the order of $P$. There are only finitely many other points; about $p$ in total. Adding $P$ to itself will eventually reach $(0; 1)$; let $\ell$ be the smallest integer $> 0$ with $\ell P = (0; 1)$. This $\ell$ is the order of $P$.

The signature scheme has as system parameters a curve $E$; a base point $P$; and a hash function $h$ with output length at least $\lceil \log_2 \ell \rceil + 1$.

Alice's secret key is an integer $a$ and her public key is $P_A = aP$.

To sign message $m$, Alice computes $h(m)$; picks random $k$; computes $R = kP = (x_1, y_1)$; puts $r \equiv y_1 \mod \ell$; computes $s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell$.

The signature on $m$ is $(r, s)$.

Anybody can verify signature given $m$ and $(r, s)$:
Compute $w_1 \equiv s^{-1}h(m) \mod \ell$ and $w_2 \equiv s^{-1}r \mod \ell$.
Check whether the $y$-coordinate of $w_1 P + w_2 P_A = kP$ and if so, accept signature.

Alice's signatures are valid: $w_1 P + w_2 P_A = kP$ and so the $y$-coordinate of this expression equals $r$, the $y$-coordinate of $kP$. 
Users can sign messages using Edwards curves. Take a point \( P \) on an Edwards curve modulo a prime \( p > 2 \). ECDSA signer needs to know the order of \( P \). There are only finitely many other points; about \( p \) in total. Adding \( P \) to itself will eventually reach \((0, 1)\); let \( \ell \) be the smallest integer \( > 0 \) with \( \ell P = (0, 1) \). This \( \ell \) is the order of \( P \).

The signature scheme has as system parameters a curve \( E \); a base point \( P \); and a hash function \( h \) with output length at least \( \lfloor \log_2 \ell \rfloor + 1 \). Alice's secret key is an integer \( a \) and her public key is \( P_A = aP \).

To sign message \( m \), Alice computes \( h(m) \); picks random \( k \); computes \( R = kP = (x_1, y_1) \); puts \( r \equiv y_1 \mod \ell \); computes \( s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell \). The signature on \( m \) is \((r, s)\).

Anybody can verify given \( m \) and \((r, s)\):

Compute \( w_1 \equiv s^{-1}h(m) \mod \ell \) and \( w_2 \equiv s^{-1} \cdot r \mod \ell \). Check whether the \( y \)-coordinate of \( w_1P + w_2P_A \) equals \( r \mod \ell \) and if so, accept signature.

Alice's signatures \( w_1P + w_2P_A = (s^{-1}h(m))P + (s^{-1}(h(m) + ra))P = kP \) and so the \( y \)-coordinate of this expression equals \( r \), the \( y \)-coordinate of \( kP \).
The signature scheme has as system parameters a curve $E$; a base point $P$; and a hash function $h$ with output length at least $\lceil \log_2 \ell \rceil + 1$.

Alice’s secret key is an integer $a$ and her public key is $P_A = aP$.

To sign message $m$,
Alice computes $h(m)$;
picks random $k$;
computes $R = kP = (x_1, y_1)$;
puts $r \equiv y_1 \mod \ell$; computes $s \equiv k^{-1}(h(m) + ra) \mod \ell$.

The signature on $m$ is $(r, s)$.

Anybody can verify signature given $m$ and $(r, s)$:
Compute $w_1 \equiv s^{-1}h(m) \mod \ell$ and $w_2 \equiv s^{-1} \cdot r \mod \ell$.
Check whether the $y$-coordinate of $w_1P + w_2P_A$ equals $r$ modulo $\ell$, and if so, accept signature.

Alice’s signatures are valid:

$$w_1P + w_2P_A = (s^{-1}h(m))P + (s^{-1} \cdot r)P = (s^{-1}(h(m) + ra))P = kP$$
and so the $y$-coordinate of this expression equals $r$, the $y$-coordinate of $kP$. 

Users can sign messages using Edwards curves. Take a point $P$ on an Edwards curve modulo a prime $p > 2$. The ECDSA signer needs to know the order of $P$. There are only finitely many other points; about $p$ in total. Adding $P$ to itself will eventually reach $(0, 1)$; let $'P = (0, 1)$.

This $'$ is the order of $P$. 

ECDSA
Users can sign messages
using Edwards curves.
Take a point $P$ on an Edwards
curve modulo a prime $p > 2$.
ECDSA signer needs to know
the order of $P$. There are only finitely many other
points; about $p$ in total.
Adding $P$ to itself will eventually
reach $(0, 1)$; let $'P = (0, 1)$.

This $'$ is the order of $P$. 

The signature scheme has as
system parameters a curve $E$; a
base point $P$; and a hash function
$h$ with output length at least
$\lceil \log_2 \ell \rceil + 1$.

Alice’s secret key is an integer $a$ and her public key is $P_A = aP$.

To sign message $m$,
Alice computes $h(m)$;
picks random $k$;
computes $R = kP = (x_1, y_1)$;
puts $r \equiv y_1 \mod \ell$; computes $s \equiv k^{-1}(h(m) + ra) \mod \ell$.

The signature on $m$ is $(r, s)$.
The signature scheme has as system parameters a curve $E$; a base point $P$; and a hash function $h$ with output length at least $\lceil \log_2 \ell \rceil + 1$. Alice’s secret key is an integer $a$ and her public key is $P_A = aP$.

To sign message $m$, Alice computes $h(m)$; picks random $k$; computes $R = kP = (x_1, y_1)$; puts $r \equiv y_1 \mod \ell$; computes $s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell$.

The signature on $m$ is $(r, s)$.

Anybody can verify signature given $m$ and $(r, s)$:
Compute $w_1 \equiv s^{-1}h(m) \mod \ell$ and $w_2 \equiv s^{-1} \cdot r \mod \ell$.
Check whether the $y$-coordinate of $w_1P + w_2P_A$ equals $r \mod \ell$ and if so, accept signature.

Alice’s signatures are valid:
\[
w_1P + w_2P_A = (s^{-1}h(m))P + (s^{-1} \cdot r)P_A = (s^{-1}(h(m) + ra))P = kP \]
and so the $y$-coordinate of this expression equals $r$, the $y$-coordinate of $kP$. 

The signature scheme has as system parameters a curve $E$; a base point $P$; and a hash function $h$ with output length at least $\lceil \log_2 \ell \rceil + 1$.

Alice's secret key is an integer $a$ and her public key is $P_A = aP$.

To sign message $m$, Alice computes $h(m)$; picks random $k$; computes $R = kP = (x_1, y_1)$; computes $y_1 \equiv s^{-1} \mod \ell$; computes $s \equiv k^{-1} (h(m) + ra) \mod \ell$. And so the $y$-coordinate of this expression equals $r$, the $y$-coordinate of $kP$.

The signature on $m$ is $(r, s)$.

Anybody can verify signature given $m$ and $(r, s)$:
Compute $w_1 \equiv s^{-1} h(m) \mod \ell$ and $w_2 \equiv s^{-1} \cdot r \mod \ell$.
Check whether the $y$-coordinate of $w_1 P + w_2 P_A$ equals $r \mod \ell$ and if so, accept signature.

Alice's signatures are valid:
$s^{-1} h(m) P + s^{-1} \cdot r P_A = kP$ and so the $y$-coordinate of this expression equals $r$, the $y$-coordinate of $kP$.

Attacker's view on signatures:
Anybody can produce an $R = kP$.

Alice's private key is only used in $s \equiv k^{-1} (h(m) + ra) \mod \ell$.

Can fake signatures if one can break the DLP, i.e., if one can compute $a$ from $P_A$.

Most of this course deals with methods for breaking DLPs. Sometimes attacks are easier...
The signature scheme has as system parameters a curve $E$; a base point $P$; and a hash function $h$ with output length at least $\lceil \log_2 \ell \rceil + 1$.

Alice's secret key is an integer $a$ and her public key is $P_A = aP$.

To sign message $m$, Alice computes $h(m)$; picks random $k$; computes $R = kP = (x_1, y_1)$; computes $r \equiv y_1 \mod \ell$; computes $s \equiv k^{-1}(h(m) + ra) \mod \ell$.

The signature on $m$ is $(r; s)$.

Anybody can verify signature given $m$ and $(r, s)$:

Compute $w_1 \equiv s^{-1}h(m) \mod \ell$ and $w_2 \equiv s^{-1} \cdot r \mod \ell$.
Check whether the $y$-coordinate of $w_1P + w_2P_A$ equals $r$ modulo $\ell$ and if so, accept signature.

Alice's signatures are valid:

$$w_1P + w_2P_A = (s^{-1}h(m))P + (s^{-1} \cdot r)P_A = (s^{-1}(h(m) + ra))P = kP$$

and so the $y$-coordinate of this expression equals $r$, the $y$-coordinate of $kP$.

Attacker's view on signatures

Anybody can produce an $R = kP$.
Alice's private key is only used in $s \equiv k^{-1}(h(m) + ra)$.

Can fake signatures if one can break the DLP, i.e., if one can compute $a$ from $P_A$.

Most of this course deals with methods for breaking DLP.

Sometimes attacks are easier...
Anybody can verify signature given \( m \) and \((r, s)\):
Compute \( w_1 \equiv s^{-1} h(m) \mod \ell \)
and \( w_2 \equiv s^{-1} \cdot r \mod \ell \).
Check whether the \( y \)-coordinate of \( w_1 P + w_2 P_A \) equals \( r \) modulo \( \ell \)
and if so, accept signature.

Alice’s signatures are valid:
\[
\begin{align*}
w_1 P + w_2 P_A &= (s^{-1} h(m)) P + (s^{-1} \cdot r) P_A \\
&= (s^{-1} (h(m) + ra)) P = k P
\end{align*}
\]
and so the \( y \)-coordinate of this expression equals \( r \),
the \( y \)-coordinate of \( k P \).

Attacker’s view on signature verification:

Anybody can produce an \( R = kP \).
Alice’s private key is only used in \( s \equiv k^{-1} (h(m) + r \cdot a) \mod \ell \).

Can fake signatures if one can break the DLP, i.e., if one can compute \( a \) from \( P_A \).

Most of this course deals with methods for breaking DLPs.
Sometimes attacks are easier...
Anybody can verify signature given $m$ and $(r, s)$:
Compute $w_1 \equiv s^{-1}h(m) \mod \ell$ and $w_2 \equiv s^{-1}\cdot r \mod \ell$.
Check whether the $y$-coordinate of $w_1P + w_2P_A$ equals $r \mod \ell$ and if so, accept signature.

Alice’s signatures are valid:

$$w_1P + w_2P_A = (s^{-1}h(m))P + (s^{-1}\cdot r)P_A = (s^{-1}(h(m) + ra))P = kP$$
and so the $y$-coordinate of this expression equals $r$, the $y$-coordinate of $kP$.

Attacker’s view on signatures

Anybody can produce an $R = kP$.
Alice’s private key is only used in $s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell$.
Can fake signatures if one can break the DLP, i.e., if one can compute $a$ from $P_A$.
Most of this course deals with methods for breaking DLPs.
Sometimes attacks are easier…
Anybody can verify signature and \((r, s)\):
Compute \(w_1 \equiv s^{-1}h(m) \mod \ell\)
and \(w_2 \equiv s^{-1} \cdot r \mod \ell\).
Check whether the \(y\)-coordinate
of \(w_1P + w_2PA\) equals \(r\) modulo \(\ell\),
accept signature.

Alice’s signatures are valid:
\[
w_1P + w_2PA = \left( s^{-1}h(m) \right)P + \left( s^{-1} \cdot r \right)PA =
\left( k^{-1}(h(m) + r \cdot a) \right)P = kP
\]
the \(y\)-coordinate of this
expression equals \(r\),
the \(y\)-coordinate of \(kP\).

Attacker’s view on signatures

Anybody can produce an \(R = kP\).
Alice’s private key is only used in
\(s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell\).

Can fake signatures if one can
break the DLP, i.e., if one can
compute \(a\) from \(PA\).
Most of this course deals with
methods for breaking DLPs.
Sometimes attacks are easier…

If \(k\) is known for some \(m; (r, s)\),
then \(a \equiv \left( s \cdot k - h(m) \right) \mod \ell\).

If two signatures \(m_1, (r, s_1)\) and
\(m_2, (r, s_2)\) have the same value
for \(r\): assume \(k_1 = k_2\);
observe \(s_1 - s_2 = k_1^{-1}(h(m_1) + ra -
(h(m_2) + ra));\) compute
\(k = \left( s_1 - s_2 \right) \mod \ell\).
Continue as above.

If bits of many \(k\)’s are known
(biased PRNG) can attack
\(s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell\)
as hidden number problem
using lattice basis reduction.
Attacker’s view on signatures

Anybody can produce an $R = kP$. Alice’s private key is only used in
$s ≡ k^{-1}(h(m) + r \cdot a) \mod \ell$.

Can fake signatures if one can break the DLP, i.e., if one can compute $a$ from $P_A$.
Most of this course deals with methods for breaking DLPs.
Sometimes attacks are easier…

If $k$ is known for some $m; (r,s)$, then $a \equiv (sk - h(m)) \mod \ell$.

If two signatures $m_1, (r,s_1)$ and $m_2, (r,s_2)$ have the same value for $r$: assume $k_1 = k_2$; observe
$s_1 - s_2 = k_1^{-1}(h(m_1) - h(m_2))$; compute $k = (s_1 - s_2)/(h(m_1) - h(m_2))$.
Continue as above.

If bits of many $k$’s are known (biased PRNG) can attack
$s \equiv k^{-1}(h(m) + r \cdot a)$ as hidden number problem using lattice basis reduction.
Anybody can verify signature given \( m \) and \((r;s)\):

Compute \( w_1 \equiv s^{-1}h(m) \mod ℓ \) and \( w_2 \equiv s^{-1}r \mod ℓ \).

Check whether the \( y \)-coordinate of \( w_1 P + w_2 P_A \) equals \( r \) modulo \( ℓ \) and if so, accept signature.

Alice's signatures are valid:

\[
 w_1 P + w_2 P_A = (s^{-1}h(m)) P + (s^{-1}(h(m) + ra)) P = kP
\]

and so the \( y \)-coordinate of this expression equals \( r \), the \( y \)-coordinate of \( kP \).

Attacker's view on signatures:

Anybody can produce an \( R = kP \).

Alice's private key is only used in \( s \equiv k^{-1}(h(m) + r \cdot a) \mod ℓ \).

Can fake signatures if one can break the DLP, i.e., if one can compute \( a \) from \( P_A \).

Most of this course deals with methods for breaking DLPs.

Sometimes attacks are easier.

If \( k \) is known for some \( m; (r;s) \), then \( a \equiv (sk - h(m))/r \mod ℓ \).

If two signatures \( m_1; (r_1;s_1) \) and \( m_2; (r_2;s_2) \) have the same value for \( r \): assume \( k_1 = k_2 \); observe \( s_1 - s_2 = k_1^{-1}(h(m_1) - h(m_2))/(h(m_1) - h(m_2)) \).

Continue as above.

If bits of many \( k \)'s are known as hidden number problem (biased PRNG) can attack by using lattice basis reduction.
Attacker’s view on signatures

Anybody can produce an $R = kP$. Alice’s private key is only used in
$s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell$.

Can fake signatures if one can break the DLP, i.e., if one can compute $a$ from $P_A$.

Most of this course deals with methods for breaking DLPs.

Sometimes attacks are easier...

If $k$ is known for some $m, (r, s)$ then $a \equiv (sk - h(m))/r \mod \ell$.

If two signatures $m_1, (r, s_1)$ and $m_2, (r, s_2)$ have the same value for $r$: assume $k_1 = k_2$; observe
$s_1 - s_2 = k_1^{-1}(h(m_1) + ra - (h(m_2) + ra))$; compute $k = (s_1 - s_2)/(h(m_1) - h(m_2))$.
Continue as above.

If bits of many $k$’s are known (biased PRNG) can attack
$s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell$ as hidden number problem using lattice basis reduction.
Attacker's view on signatures

Anybody can produce an $R = kP$. Alice's private key is only used in $s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell$.

Can fake signatures if one can break the DLP, i.e., if one can compute $a$ from $P_A$.

Most of this course deals with methods for breaking DLPs. Sometimes attacks are easier.

If $k$ is known for some $m$, $(r, s)$ then $a \equiv (sk - h(m))/r \mod \ell$.

If two signatures $m_1, (r, s_1)$ and $m_2, (r, s_2)$ have the same value for $r$: assume $k_1 = k_2$; observe $s_1 - s_2 = k_1^{-1}(h(m_1) + ra - (h(m_2) + ra))$; compute $k = (s_1 - s_2)/(h(m_1) - h(m_2))$. Continue as above.

If bits of many $k$'s are known (biased PRNG) can attack $s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell$ as hidden number problem using lattice basis reduction.

Malicious signer

Alice can set up her public key so that two messages of her choice share the same signature, i.e., she can claim to have signed $m_1$ or $m_2$ at will:

$R = (x_1; y_1)$ and $-R = (x_2; y_2)$ have the same $y$-coordinate.

Thus, $(r; s)$ fits $R = kP$, $s \equiv k^{-1}(h(m_1) + ra - (h(m_2) + ra)) \mod \ell$.

If $a \equiv -(h(m_1) + h(m_2)) = 2r \mod \ell$.
Attacker's view on signatures

Anybody can produce an $R = kP$. Alice's private key is only used in $s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell$.

Can fake signatures if one can break the DLP, i.e., if one can compute $a$ from $P_A$.

Most of this course deals with methods for breaking DLPs.

Sometimes attacks are easier...

Malicious signer

Alice can set up her public key so that two messages of her choice share the same signature, i.e., she can claim to have signed $m_1$ or $m_2$ at will:

If $k$ is known for some $m$, $(r, s)$ then $a \equiv (sk - h(m))/r \mod \ell$.

If two signatures $m_1, (r, s_1)$ and $m_2, (r, s_2)$ have the same value for $r$: assume $k_1 = k_2$; observe $s_1 - s_2 = k_1^{-1}(h(m_1) + ra - (h(m_2) + ra))$; compute $k = (s_1 - s_2)/(h(m_1) - h(m_2))$.

Continue as above.

If bits of many $k$'s are known (biased PRNG) can attack as hidden number problem using lattice basis reduction.

Thus, $(r, s)$ fits $R = (x_1, y_1)$ and $-R = (-k)P$,

$s \equiv k^{-1}(h(m_1) + r \cdot a) \mod \ell$,

$a \equiv -(h(m_1) + h(m_2))/r \mod \ell$. 

Attacker's view on signatures

Anybody can produce an \( R = kP \).

Alice's private key is only used in
\[ s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell. \]

Can fake signatures if one can break the DLP, i.e., if one can compute \( a \) from \( P_A \).

Most of this course deals with methods for breaking DLPs.

Sometimes attacks are easier...
If $k$ is known for some $m$, $(r, s)$ then $a \equiv (sk - h(m))/r \mod \ell$.

If two signatures $m_1, (r, s_1)$ and $m_2, (r, s_2)$ have the same value for $r$: assume $k_1 = k_2$; observe
$s_1 - s_2 = k_1^{-1}(h(m_1) + ra - (h(m_2) + ra))$; compute $k = (s_1 - s_2)/(h(m_1) - h(m_2))$.
Continue as above.

If bits of many $k$'s are known (biased PRNG) can attack
$s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell$ as hidden number problem
using lattice basis reduction.

Malicious signer

Alice can set up her public key so that two messages of her choice
share the same signature, i.e., she can claim to have
signed $m_1$ or $m_2$ at will:
$R = (x_1, y_1)$ and $-R = (-x_1, y_1)$

Thus, $(r, s)$ fits $R = kP$,
$s \equiv k^{-1}(h(m_1) + ra) \mod \ell$ and
$-R = (-k)P$,
$s \equiv -k^{-1}(h(m_2) + ra) \mod \ell$ if
$a \equiv -(h(m_1) + h(m_2))/2r \mod \ell.$
If \( k \) is known for some \( m, (r, s) \) then \( a \equiv (sk - h(m))/r \mod \ell \).

If two signatures \( m_1, (r, s_1) \) and \( m_2, (r, s_2) \) have the same value for \( r \): assume \( k_1 = k_2 \); observe
\[
s_1 - s_2 = k_1^{-1}(h(m_1) + ra - (h(m_2) + ra));
\]
compute \( k = (s_1 - s_2)/(h(m_1) - h(m_2)) \).

Continue as above.

If bits of many \( k \)'s are known (biased PRNG) can attack
\[
s \equiv k^{-1}(h(m) + r \cdot a) \mod \ell
\]
as hidden number problem using lattice basis reduction.

Malicious signer

Alice can set up her public key so that two messages of her choice share the same signature, i.e., she can claim to have signed \( m_1 \) or \( m_2 \) at will:
\[
R = (x_1, y_1) \quad \text{and} \quad -R = (-x_1, y_1)
\]
have the same \( y \)-coordinate.

Thus, \((r, s)\) fits \( R = kP \),
\[
s \equiv k^{-1}(h(m_1) + ra) \mod \ell \quad \text{and} \quad
-\text{R} = (-k)P,
\]
\[
s \equiv -k^{-1}(h(m_2) + ra) \mod \ell
\]
if
\[
a \equiv -(h(m_1) + h(m_2))/2r \mod \ell.
\]
(Easy tweak: include bit of \( x_1 \).)
If $k$ is known for some $m$, then $a \equiv (sk - h(m))/r \mod \ell$.

If two signatures $m_1, (r, s_1)$ and $m_2, (r, s_2)$ have the same value for $r$, assume $k_1 = k_2$; observe $s_1 - s_2 \equiv k^{-1}(h(m_1) + ra - h(m_2))$; compute $k \equiv (s_1 - s_2)/(h(m_1) - h(m_2))$.

Malicious signer

Alice can set up her public key so that two messages of her choice share the same signature, i.e., she can claim to have signed $m_1$ or $m_2$ at will:

$R = (x_1, y_1)$ and $-R = (-x_1, y_1)$ have the same $y$-coordinate.

Thus, $(r, s)$ fits $R = kP$, $s \equiv k^{-1}(h(m_1) + ra) \mod \ell$ and $-R = (-k)P$, $s \equiv -k^{-1}(h(m_2) + ra) \mod \ell$ if $a \equiv -(h(m_1) + h(m_2))/2r \mod \ell$.

(Easy tweak: include bit of $x_1$.)

More elliptic curves

Edwards curves are elliptic.

Easiest way to understand elliptic curves is Edwards.

Geometrically, all elliptic curves are Edwards.

Algebraically, more elliptic curves exist (not always point of order 4).

Every odd-char curve can be expressed as Weierstrass curve $v^2 = u^3 + a_2 u^2 + a_4 u + a_6$.

Warning: "Weierstrass" has different meaning in char 2.
Malicious signer

Alice can set up her public key so that two messages of her choice share the same signature, i.e., she can claim to have signed $m_1$ or $m_2$ at will:

Let $R = (x_1, y_1)$ and $−R = (−x_1, y_1)$ have the same $y$-coordinate. Thus, $(r, s)$ fits $R = kP$,

$$s \equiv k^{-1}(h(m_1) + ra) \mod ℓ$$

and

$$−R = (−k)P,$$

$$s \equiv −k^{-1}(h(m_2) + ra) \mod ℓ$$

if $a \equiv −(h(m_1) + h(m_2))/2r \mod ℓ$.

(Easy tweak: include bit of $x_1$.)

More elliptic curves

Edwards curves are elliptic. Easiest way to understand elliptic curves is Edwards.

Geometrically, all elliptic curves are Edwards curves.

Algebraically, more elliptic curves exist (not always point of order 4).

Every odd-char curve can be expressed as Weierstrass curve

$$v^2 = u^3 + a_2 u^2 + a_4 u + a_6.$$  

Warning: “Weierstrass” has different meaning in char 2.
If \( k \) is known for some \( m \); then
\[
(a \equiv (sk - h(m)) = r \mod \ell).
\]

If two signatures \( m_1; (r;s_1) \) and \( m_2; (r;s_2) \) have the same value for \( r \); assume \( k_1 = k_2 \); observe
\[
s_1 - s_2 \\equiv k - 1 (h(m_1) + ra - (h(m_2) + ra)) \mod \ell.
\]

Continue as above.

If bits of many \( k \)'s are known (biased PRNG) can attack
\[
s \equiv k - 1 (h(m) + ra) \mod \ell
\]
as hidden number problem using lattice basis reduction.

Malicious signer

Alice can set up her public key so that two messages of her choice share the same signature, i.e., she can claim to have signed \( m_1 \) or \( m_2 \) at will:

\[
R = (x_1, y_1) \text{ and } -R = (-x_1, y_1)
\]
have the same \( y \)-coordinate.

Thus, \((r, s)\) fits \( R = kP \),
\[
s \equiv k^{-1}(h(m_1) + ra) \mod \ell \text{ and } -R = (-k)P,
\]
\[
s \equiv -k^{-1}(h(m_2) + ra) \mod \ell \text{ if } a \equiv -(h(m_1) + h(m_2))/2r \mod \ell.
\]

(Easy tweak: include bit of \( x_1 \).)

More elliptic curves

Edwards curves are elliptic. Easiest way to understand elliptic curves is Edwards.

Geometrically, all elliptic curves are Edwards curves.

Algebraically, more elliptic curves exist (not always point of order 4).

Every odd-char curve can be expressed as Weierstrass curve
\[
v^2 = u^3 + a_2u^2 + a_4u + a_6.
\]

Warning: “Weierstrass” has different meaning in char 2.
Malicious signer

Alice can set up her public key so that two messages of her choice share the same signature, i.e., she can claim to have signed \( m_1 \) or \( m_2 \) at will:

\[
R = (x_1, y_1) \quad \text{and} \quad -R = (-x_1, y_1)
\]

have the same y-coordinate. Thus, \((r, s)\) fits

\[
R = kP \quad \Rightarrow \quad s \equiv k^{-1}(h(m_1) + ra) \mod \ell
\]

and

\[
-R = (-k)P \quad \Rightarrow \quad s \equiv -k^{-1}(h(m_2) + ra) \mod \ell
\]

if

\[
a \equiv -\frac{h(m_1) + h(m_2)}{2r} \mod \ell.
\]

(Easy tweak: include bit of \( x_1 \).)

More elliptic curves

Edwards curves are elliptic. Easiest way to understand elliptic curves is Edwards.

Geometrically, all elliptic curves are Edwards curves.

Algebraically, more elliptic curves exist (not always point of order 4).

Every odd-char curve can be expressed as Weierstrass curve

\[
v^2 = u^3 + a_2u^2 + a_4u + a_6.
\]

Warning: “Weierstrass” has different meaning in char 2.
Malicious signer

Alice can set up her public key so that two messages of her choice share the same signature, i.e., she can claim to have signed $m_1$ or $m_2$ at will:

$$R = (x_1, y_1)$$

and

$$-R = (-x_1, y_1)$$

have the same $y$-coordinate.

Thus, $(r, s)$ fits

$$R = kP,$$

$s ≡ k^{-1}(h(m_1) + ra) \text{ mod } ℓ$ and

$$-R = (-k)P,$$

$s ≡ -k^{-1}(h(m_2) + ra) \text{ mod } ℓ$ if

$$a ≡ - (h(m_1) + h(m_2)) = 2r \text{ mod } ℓ.$$

(Easy tweak: include bit of $x_1$.)

More elliptic curves

Edwards curves are elliptic.

Easiest way to understand elliptic curves is Edwards.

Geometrically, all elliptic curves are Edwards curves.

Algebraically, more elliptic curves exist (not always point of order 4).

Every odd-char curve can be expressed as Weierstrass curve

$$v^2 = u^3 + a_2u^2 + a_4u + a_6.$$

Warning: “Weierstrass” has different meaning in char 2.

Addition on Weierstrass curve

Let $P_1 = (u_1, v_1), P_2 = (u_2, v_2)$.

The slope $λ$ is given by

$$λ = \frac{v_2 - v_1}{u_2 - u_1}.$$

Note that $u_1 \neq u_2$. 

Warning: “Weierstrass” has different meaning in char 2.
Malicious signer
Alice can set up her public key so that two messages of her choice share the same signature, i.e., she can claim to have signed $m_1$ or $m_2$ at will:
\[ R = (x_1; y_1) \text{ and } -R = (-x_1; y_1) \]
have the same $y$-coordinate.

Thus, \((r; s)\) fits $R = kP$, $s \equiv k^{-1}(h(m_1) + ra) \mod \ell$ and $-R = (-kP)$, $s \equiv -(k^{-1}(h(m_2) + ra)) \mod \ell$ if $a \equiv -(h(m_1) + h(m_2)) \equiv 2r \mod \ell$.

(Easy tweak: include bit of $x_1$.)

More elliptic curves

Edwards curves are elliptic.

Easiest way to understand elliptic curves is Edwards.

Geometrically, all elliptic curves are Edwards curves.

Algebraically, more elliptic curves exist (not always point of order 4).

Every odd-char curve can be expressed as Weierstrass curve $v^2 = u^3 + a_2u^2 + a_4u + a_6$.

Warning: “Weierstrass” has different meaning in char 2.

Addition on Weierstrass curve $v^2 = u^3 + u^2 + u + 1$.

\[ \text{Slope } \lambda = (v_2 - v_1) / (u_2 - u_1). \]
Note that $u_1 \neq u_2$.
Malicious signer

Alice can set up her public key so that two messages of her choice share the same signature, i.e., she can claim to have signed $m_1$ or $m_2$ at will:

$R = (x_1; y_1)$ and $-R = (-x_1; y_1)$

have the same $y$-coordinate.

Thus, $(r; s)$ fits $R = kP$, $s \equiv k^{-1}(h(m_1) + ra)$ mod $\ell$ and $-R = -(kP)$, $s \equiv -(k^{-1}(h(m_2) + ra))$ mod $\ell$ if $a \equiv -(h(m_1) + h(m_2)) = 2r$ mod $\ell$.

(Easy tweak: include bit of $x_1$.)

More elliptic curves

Edwards curves are elliptic. Easiest way to understand elliptic curves is Edwards.

Geometrically, all elliptic curves are Edwards curves.

Algebraically, more elliptic curves exist (not always point of order 4).

Every odd-char curve can be expressed as Weierstrass curve $v^2 = u^3 + a_2 u^2 + a_4 u + a_6$.

Warning: “Weierstrass” has different meaning in char 2.

Addition on Weierstrass curve

$v^2 = u^3 + u^2 + u + 1$

Slope $\lambda = (v_2 - v_1)/(u_2 - u_1)$. Note that $u_1 \neq u_2$. 

More elliptic curves

Edwards curves are elliptic.

Easiest way to understand elliptic curves is Edwards.

Geometrically, all elliptic curves are Edwards curves.

Algebraically, more elliptic curves exist (not always point of order 4).

Every odd-char curve can be expressed as Weierstrass curve $v^2 = u^3 + a_2 u^2 + a_4 u + a_6$.

Warning: “Weierstrass” has different meaning in char 2.
More elliptic curves

Edwards curves are elliptic. Easiest way to understand elliptic curves is Edwards.

Geometrically, all elliptic curves are Edwards curves.

Algebraically, more elliptic curves exist (not always point of order 4).

Every odd-char curve can be expressed as Weierstrass curve

\[ v^2 = u^3 + a_2 u^2 + a_4 u + a_6. \]

Warning: “Weierstrass” has different meaning in char 2.

Addition on Weierstrass curve

\[ v^2 = u^3 + u^2 + u + 1 \]

\[ P_1 + P_2 \]

\[ -(P_1 + P_2) \]

\[ P_1 \]

\[ P_2 \]

Slope \( \lambda = (v_2 - v_1)/(u_2 - u_1). \) Note that \( u_1 \neq u_2. \)
Elliptic curves

- Curves are elliptic.
- Easiest way to understand elliptic curves is Edwards.
- Geometrically, all elliptic curves are Edwards curves.
- Algebraically, more elliptic curves exist (not always point of order 4).
- Every odd-char curve can be expressed as Weierstrass curve
  \[ v^2 = u^3 + a_2 u^2 + a_4 u + a_6. \]
- Note: “Weierstrass” has different meaning in char 2.

### Addition on Weierstrass curve

\[ v^2 = u^3 + u^2 + u + 1 \]

Slope \( \lambda = (v_2 - v_1)/(u_2 - u_1) \).
Note that \( u_1 \neq u_2 \).

### Doubling on Weierstrass curve

\[ v^2 = u^3 - u \]

Slope \( \lambda = (3u_2^2 - 1)/(2v_2) \).
More elliptic curves
Edwards curves are elliptic.
Easiest way to understand elliptic curves is Edwards.
Geometrically, all elliptic curves are Edwards curves.
Algebraically, more elliptic curves exist (not always point of order 4).
Every odd-char curve can be expressed as Weierstrass curve
\[ v^2 = u^3 + a_2 u^2 + a_4 u + a_6. \]
"Weierstrass" has different meaning in char 2.

### Addition on Weierstrass curve
\[ v^2 = u^3 + u^2 + u + 1 \]

Slope \( \lambda = (v_2 - v_1)/(u_2 - u_1) \).
Note that \( u_1 \neq u_2 \).

### Doubling on Weierstrass curve
\[ v^2 = u^3 - u \]

Slope \( \lambda = (3u_1^2 - 1)/(2v_1) \).
Addition on Weierstrass curve

\[ v^2 = u^3 + u^2 + u + 1 \]

Slope \( \lambda = (v_2 - v_1)/(u_2 - u_1) \).
Note that \( u_1 \neq u_2 \).

Doubling on Weierstrass curve

\[ v^2 = u^3 - u \]

Slope \( \lambda = (3u_1^2 - 1)/(2v_1) \).
Addition on Weierstrass curve

\[ v^2 = u^3 + u^2 + u + 1 \]

Slope \( \lambda = (v_2 - v_1)/(u_2 - u_1) \).
Note that \( u_1 \neq u_2 \).

Doubling on Weierstrass curve

\[ v^2 = u^3 - u \]

Slope \( \lambda = (3u_1^2 - 1)/(2v_1) \).
Addition on Weierstrass curve

\[ v^2 = u^3 + u^2 + u + 1 \]

Then

\[ P_1 + P_2 = (v_2 - v_1)/(u_2 - u_1). \]

Note that \( u_1 \neq u_2 \).

Doubling on Weierstrass curve

\[ v^2 = u^3 - u \]

Slope \( \lambda = (3u_1^2 - 1)/(2v_1) \).

In most cases

\( (u_1, v_1) + (u_2, v_2) = (u_3, v_3) \)

where (\( u_3, v_3 \)) =

\[ (–2 - u_1 - u_2; –(u_1 - u_3) - v_1) \]

Also handle some exceptions:

\( (u_1, v_1) = (u_2, -v_2); \infty \) as input.

Messy to implement and test.
Addition on Weierstrass curve

\[ v^2 = u^3 + u^2 + u + 1 \]

\[ \bullet P_1 \quad \bullet P_2 \]

\[ -(P_1 + P_2) \]

\[ \bullet (P_1 + P_2) \]

\[ \bullet -2P_1 \quad \bullet 2P_1 \]

Slope \( \lambda = (3u_1^2 - 1)/(2v_1) \).

Doubling on Weierstrass curve

\[ v^2 = u^3 - u \]

In most cases

\((u_1, v_1) + (u_2, v_2) = (u_3, v_3)\) where \((u_3, v_3) = (-2 - u_1 - u_2; -(u_1 - u_3) - v_1)\):

Total cost \(1I + 2M + 1S\).

Also handle some exceptions:

\((u_1, v_1) = (u_2, -v_2); \infty\) as input.

Messy to implement and test.
Addition on Weierstrass curve
\[ v^2 = u^3 + u^2 + u + 1 \]
\[ \cdot P_1 \cdot P_2 \]
\[ - (P_1 + P_2) \]
\[ \rightarrow \]
\[ v \]
\[ \uparrow \]
Slope \[ \lambda = (v_2 - v_1)/(u_2 - u_1). \]

Doubling on Weierstrass curve
\[ v^2 = u^3 - u \]
\[ \cdot P_1 \cdot \cdot P_2 \]
Slope \[ \lambda = (3u_1^2 - 1)/(2v_1). \]

In most cases
\[(u_1, v_1) + (u_2, v_2) = (u_3, v_3) \text{ where } (u_3, v_3) = (\lambda^2 - u_1 - u_2, \lambda(u_1 - u_3) - v_1)\]
\[ u_1 \neq u_2, \text{ “addition” (alert!)} \]
\[ \lambda = (v_2 - v_1)/(u_2 - u_1). \]
Total cost \[ 1I + 2M + 1S. \]

\[(u_1, v_1) = (u_2, v_2) \text{ and } v_1 \neq 0, \text{ “doubling” (alert!):} \]
\[ \lambda = (3u_1^2 + 2a_2u_1 + a_4)/(2v_1). \]
Total cost \[ 1I + 2M + 2S. \]

Also handle some exceptions:
\[(u_1, v_1) = (u_2, -v_2); \infty \text{ as input.} \]
Messy to implement and test.
Doubling on Weierstrass curve

\[ v^2 = u^3 - u \]

Slope \( \lambda = (3u_1^2 - 1)/(2v_1) \).

In most cases

\( (u_1, v_1) + (u_2, v_2) = (u_3, v_3) \)

where \( (u_3, v_3) = (\lambda^2 - u_1 - u_2, \lambda(u_1 - u_3) - v_1) \).

\( u_1 \neq u_2 \), “addition” (alert!):

\[ \lambda = (v_2 - v_1)/(u_2 - u_1). \]

Total cost \( 1I + 2M + 1S \).

\( (u_1, v_1) = (u_2, v_2) \) and \( v_1 \neq 0 \), “doubling” (alert!):

\[ \lambda = (3u_1^2 + 2a_2u_1 + a_4)/(2v_1). \]

Total cost \( 1I + 2M + 2S \).

Also handle some exceptions:

\( (u_1, v_1) = (u_2, -v_2); \infty \) as input.

Messy to implement and test.
Doubling on Weierstrass curve

\[v^2 = u^3 - u \cdot P_1 - 2 \cdot P_1\]

\[\Rightarrow \quad v \rightarrow v^\uparrow \quad u \rightarrow u^\uparrow\]

Slope \(- = (3u_1^2 - 1) = (2v_1).

In most cases

\[(u_1, v_1) + (u_2, v_2) = (u_3, v_3) \text{ where } (u_3, v_3) = (\lambda^2 - u_1 - u_2, \lambda(u_1 - u_3) - v_1).\]

\[u_1 \neq u_2, \text{ “addition” (alert!)}:\]

\[\lambda = (v_2 - v_1)/(u_2 - u_1).\]

Total cost \(1I + 2M + 1S.\)

\[(u_1, v_1) = (u_2, v_2) \text{ and } v_1 \neq 0, \text{ “doubling” (alert!)}:\]

\[\lambda = (3u_1^2 + 2a_2u_1 + a_4)/(2v_1).\]

Total cost \(1I + 2M + 2S.\)

Also handle some exceptions:

\[(u_1, v_1) = (u_2, -v_2); \infty \text{ as input.}\]

Messy to implement and test.

Birational equivalence

Starting from point \((x; y)\) on \(x^2 + y^2 = 1 + dx^2 y^2 :\)

Define \(A = 2(1 + d) = (1 - d)\)

\(B = 4/(1 + d)\)

\[u = (1 + y)/B, \quad v = u/x.\]

(Skip a few exceptional points.)

Then \((u, v)\) is a point on a Weierstrass curve:

\[v^2 = u^3 + (A/B) u^2 + (1/B^2) u.\]

Easily invert this map:

\[x = u/v, \quad y = (Bx - 1)/B = (Bx + 1).\]
In most cases
\((u_1, v_1) + (u_2, v_2) = (u_3, v_3)\) where \((u_3, v_3) = (\lambda^2 - u_1 - u_2, \lambda(u_1 - u_3) - v_1)\).

\(u_1 \neq u_2\), “addition” (alert!):
\(\lambda = (v_2 - v_1)/(u_2 - u_1)\).
Total cost \(1I + 2M + 1S\).

\((u_1, v_1) = (u_2, v_2)\) and \(v_1 \neq 0\),
“doubling” (alert!):
\(\lambda = (3u_1^2 + 2a_2u_1 + a_4)/(2v_1)\).
Total cost \(1I + 2M + 2S\).

Also handle some exceptions:
\((u_1, v_1) = (u_2, -v_2); \infty\) as input.
Messy to implement and test.
In most cases
\((u_1, v_1) + (u_2, v_2) = (u_3, v_3)\) where \((u_3, v_3) = (\lambda^2 - u_1 - u_2, \lambda(u_1 - u_3) - v_1)\).

\(u_1 \neq u_2\), “addition” (alert!):
\(\lambda = (v_2 - v_1)/(u_2 - u_1)\).
Total cost \(1I + 2M + 1S\).

\((u_1, v_1) = (u_2, v_2)\) and \(v_1 \neq 0\),
“doubling” (alert!):
\(\lambda = (3u_1^2 + 2a_2u_1 + a_4)/(2v_1)\).
Total cost \(1I + 2M + 2S\).

Also handle some exceptions:
\((u_1, v_1) = (u_2, -v_2); \infty \text{ as input.}\)
Messy to implement and test.

Birational equivalence
Starting from point \((x, y)\)
on \(x^2 + y^2 = 1 + dx^2y^2:\)
Define \(A = 2(1 + d)/(1 - d)\);
\(B = 4/(1 - d)\);
\(u = (1 + y)/(B(1 - y))\),
\(v = u/x = (1 + y)/(Bx(1 - y))\).
(Skip a few exceptional points.)

Then \((u, v)\) is a point on
a Weierstrass curve:
\(v^2 = u^3 + (A/B)u^2 + (1/B)u\).

Easily invert this map:
\(x = u/v, y = (Bu - 1)/(Bu - 1)\).
In most cases

\((u_1, v_1) + (u_2, v_2) = (u_3, v_3)\) where

\((u_3, v_3) = (\lambda^2 - u_1 - u_2, \lambda(u_1 - u_3) - v_1)\).

\(u_1 \neq u_2\), “addition” (alert!):

\(\lambda = (v_2 - v_1)/(u_2 - u_1)\).

Total cost \(1I + 2M + 1S\).

\((u_1, v_1) = (u_2, v_2)\) and \(v_1 \neq 0\), “doubling” (alert!):

\(\lambda = (3u_1^2 + 2a_2u_1 + a_4)/(2v_1)\).

Total cost \(1I + 2M + 2S\).

Also handle some exceptions:

\((u_1, v_1) = (u_2, -v_2); \infty\) as input.

Messy to implement and test.

\[\text{Birational equivalence}\]

Starting from point \((x, y)\) on \(x^2 + y^2 = 1 + dx^2y^2\):

Define \(A = 2(1 + d)/(1 - d), B = 4/(1 - d)\);

\(u = (1 + y)/(B(1 - y)), v = u/x = (1 + y)/(Bx(1 - y))\). (Skip a few exceptional points.)

Then \((u, v)\) is a point on a Weierstrass curve:

\[v^2 = u^3 + (A/B)u^2 + (1/B^2)u.\]

Easily invert this map:

\(x = u/v, y = (Bu - 1)/(Bu + 1)\).
In most cases
\((u_1; v_1) + (u_2; v_2) = (u_3; v_3)\) where \((u_3; v_3) = (u_1 - u_2, \lambda(u_1 - u_3) - v_1)\).

, “addition” (alert!):
\[-v_1/(u_2 - u_1)\].

\text{Total cost} \(1I + 2M + 1S\).

\((u_1; v_1) = (u_2; -v_2); \infty\) as input.

\text{Handle some exceptions:}
\(\infty\) as input.

\text{Easily invert this map:}
\(x = u/v, y = (Bu - 1)/(Bu + 1)\).

Birational equivalence

Starting from point \((x, y)\)
on \(x^2 + y^2 = 1 + dx^2y^2\):

Define \(A = 2(1 + d)/(1 - d), B = 4/(1 - d)\);
\(u = (1 + y)/(B(1 - y)), v = u/x = (1 + y)/(Bx(1 - y)).\)

(Skip a few exceptional points.)

Then \((u, v)\) is a point on
a Weierstrass curve:
\(v^2 = u^3 + (A/B)u^2 + (1/B^2)u\).

\text{Easily invert this map:}
\(x = u/v, y = (Bu - 1)/(Bu + 1)\).

Attacker can transform Edwards curve to Weierstrass curve and
vice versa; \(n(x, y) \mapsto n(u, v)\).

\(\Rightarrow\) Same discrete-log security!

Can choose curve representation
so that implementation of attack
is faster.

System designer can choose curve
representation so that protocol
runs fastest; no need to worry
about security degradation.
In most cases
(\(u_1, v_1\)) + (\(u_2, v_2\)) = (\(u_3, v_3\)) where (\(u_3, v_3\)) = (−2 − u_1 − u_2; −(u_1 − u_3) − v_1).

“addition” (alert!):
− = (\(v_2 − v_1\)) = (\(u_2 − u_1\)).

\(\mathcal{M} + 1\mathcal{S}\).

Also handle some exceptions:
(\(u_1, v_1\)) = (\(u_2, −v_2\)); ∞ as input.

“doubling” (alert!):
− = (\(3u_1 + 2a_2u_1 + a_4\)) = (\(2v_1\)).

\(\mathcal{M} + 2\mathcal{S}\).

Then (\(u, v\)) is a point on a Weierstrass curve:
\(v^2 = u^3 + (A/B)u^2 + (1/B^2)u\).

Easily invert this map:
x = u/v, y = (Bu − 1)/(Bu + 1).

Birational equivalence

Starting from point (\(x, y\)) on \(x^2 + y^2 = 1 + dx^2y^2\):

Define \(A = 2(1 + d)/(1 − d)\), \(B = 4/(1 − d)\);
\(u = (1 + y)/(B(1 − y))\), \(v = u/x = (1 + y)/(Bx(1 − y))\).

(Skip a few exceptional points.)

Then (\(u, v\)) is a point on a Weierstrass curve:
\(v^2 = u^3 + (A/B)u^2 + (1/B^2)u\).

Easily invert this map:
x = u/v, y = (Bu − 1)/(Bu + 1).

Attacker can transform Edwards curve to Weierstrass curve and vice versa; \(n(x, y)\) ↦→ \(n(u, v)\).
⇒ Same discrete-log security!

Can choose curve representation so that implementation runs fastest; no need to worry about security degradation.

System designer can choose curve representation so that protocol runs fastest; no need to worry about security degradation.
In most cases \((u_1; v_1) + (u_2; v_2) = (u_3; v_3)\) where
\[(u_3; v_3) = (-2 - u_1 - u_2; - (u_1 - u_3) - v_1).
\]

Also handle some exceptions:
\[(u_1; v_1) = (u_2; -v_2); \infty \text{ as input.}
\]

Birational equivalence

Starting from point \((x, y)\) on \(x^2 + y^2 = 1 + dx^2y^2\):

Define \(A = 2(1 + d)/(1 - d), B = 4/(1 - d); u = (1 + y)/(B(1 - y)), v = u/x = (1 + y)/(Bx(1 - y))\).

(Skip a few exceptional points.)

Then \((u, v)\) is a point on a Weierstrass curve:
\[v^2 = u^3 + (A/B)u^2 + (1/B^2)u.\]

Easily invert this map:
\[x = u/v, y = (Bu - 1)/(Bu + 1).\]

Attacker can transform Edwards curve to Weierstrass curve and vice versa; \(n(x, y) \mapsto n(u, v)\).

⇒ Same discrete-log security!

Can choose curve representation so that implementation of attack is faster/easier.

System designer can choose curve representation so that protocol runs fastest; no need to worry about security degradation.
Birational equivalence

Starting from point \((x, y)\) on \(x^2 + y^2 = 1 + dx^2y^2:\)

Define \(A = 2(1 + d)/(1 - d),\) \(B = 4/(1 - d);\)
\(u = (1 + y)/(B(1 - y)),\)
\(v = u/x = (1 + y)/(Bx(1 - y)).\)
(Skip a few exceptional points.)

Then \((u, v)\) is a point on a Weierstrass curve:
\(v^2 = u^3 + (A/B)u^2 + (1/B^2)u.\)

Easily invert this map:
\(x = u/v,\) \(y = (Bu - 1)/(Bu + 1).\)

Attacker can transform Edwards curve to Weierstrass curve and vice versa; \(n(x, y) \mapsto n(u, v).\)
\(\Rightarrow\) Same discrete-log security!
Can choose curve representation so that implementation of attack is faster/easier.

System designer can choose curve representation so that protocol runs fastest; no need to worry about security degradation.
Birational equivalence

Starting from point \((x, y)\) on \(x^2 + y^2 = 1 + dx^2 y^2\):

Define \(A = 2(1 + d)/(1 - d)\), 
\(B = 4/(1 - d)\); 
\(u = (1 + y)/(B(1 - y))\), 
\(v = u/x = (1 + y)/(Bx(1 - y))\).
(Skip a few exceptional points.)

Then \((u, v)\) is a point on 
a Weierstrass curve: 
\[v^2 = u^3 + (A/B)u^2 + (1/B^2)u.\]

Easily invert this map: 
\[x = u/v, \ y = (Bu - 1)/(Bu + 1).\]
Birational equivalence

Starting from point \((x, y)\) on \(x^2 + y^2 = 1 + dx^2 y^2\):

Define \(A = 2(1 + d)/(1 - d)\), \(B = 4/(1 - d)\);
\(u = (1 + y)/(B(1 - y))\),
\(v = u/x = (1 + y)/(Bx(1 - y))\).
(Skip a few exceptional points.)

Then \((u, v)\) is a point on a Weierstrass curve:
\[v^2 = u^3 + (A/B)u^2 + (1/B^2)u.\]

Easily invert this map:
\[x = u/v, \ y = (Bu - 1)/(Bu + 1).\]

Elliptic-curve groups

Following algorithms will need a unique representative per point. For that Weierstrass curves are the speed leader.
Birational equivalence
Starting from point \((x, y)\) on \(x^2 + y^2 = 1 + dx^2y^2:\)
Define 
\[
A = 2(1 + d)/(1 - d), \quad B = 4(1 - d);
\]
\[
x = (1 + y)/(B(1 - y)), \quad u = (1 + y)/(Bx(1 - y)).
\]
Then \((u, v)\) is a point on a Weierstrass curve:
\[
v^2 = u^3 + (A/B)u^2 + (1/B^2)u.
\]
Easily invert this map:
\[
x = u/v, \quad y = (Bu - 1)/(Bu + 1).
\]

The discrete-logarithm problem
Define \(p = 1000003\) and consider the Weierstrass curve
\(y^2 = x^3 - x\) over \(F_p\).
This curve has
...
Define $A = 2(1 + d)/(1 - d)$, $B = 4/(1 - d)$, $u = (1 + y)/(B(1 - y))$, $v = u/x = (1 + y)/(Bx(1 - y))$.

Then $(u; v)$ is a point on a Weierstrass curve:
$$v^2 = u^3 + (A/B)u^2 + (1/B^2)u.$$  
Easily invert this map:
$x = u/v$, $y = (Bu - 1)/(Bu + 1)$.

Following algorithms will need a unique representative per point. For that Weierstrass curves are the speed leader.

The discrete-logarithm problem
Define $p = 1000003$ and consider the Weierstrass curve $y^2 = x^3 - x$ over $F_p$. This curve has

Elliptic-curve groups

$$P_1, P_2, -P_1 - P_2, P_1 + P_2, 2P_1, -2P_1.$$
Birational equivalence
Starting from point \((x;y)\) on \(x^2 + y^2 = 1 + dx^2y^2\):

Define \(A = 2(1 + d) = (1 - d)\), \(B = 4 = (1 - d);\)
\(u = (1 + y) = B(1 - y), \ v = \frac{u}{x} = (1 + y) = Bx(1 - y)\).

Then \((u;v)\) is a point on a Weierstrass curve:
\(v^2 = u^3 + \frac{A}{B}u^2 + \frac{1}{B^2}u\).

Easily invert this map:
\(x = \frac{u}{v}, \ y = \frac{Bu - 1}{v} = \frac{Bu + 1}{v}\).

Elliptic-curve groups

Following algorithms will need a unique representative per point. For that Weierstrass curves are the speed leader.

The discrete-logarithm problem

Define \(p = 1000003\) and consider the Weierstrass curve \(y^2 = x^3 - x\) over \(\mathbb{F}_p\).
This curve has...
Elliptic-curve groups

Following algorithms will need a unique representative per point. For that Weierstrass curves are the speed leader.

The discrete-logarithm problem

Define \( p = 1000003 \) and consider the Weierstrass curve \( y^2 = x^3 - x \) over \( \mathbb{F}_p \).

This curve has
Elliptic-curve groups

Following algorithms will need a unique representative per point. For that Weierstrass curves are the speed leader.

The discrete-logarithm problem

Define $p = 1000003$ and consider the Weierstrass curve $y^2 = x^3 - x$ over $\mathbb{F}_p$.
This curve has $1000004 = 2^2 \cdot 53^2 \cdot 89$ points and $P = (101384, 614510)$ is a point of order $2 \cdot 53^2 \cdot 89$. 

Elliptic-curve groups

Following algorithms will need a unique representative per point. For that Weierstrass curves are the speed leader.

The discrete-logarithm problem

Define $p = 1000003$ and consider the Weierstrass curve $y^2 = x^3 - x$ over $\mathbb{F}_p$.

This curve has $1000004 = 2^2 \cdot 53^2 \cdot 89$ points and $P = (101384, 614510)$ is a point of order $2 \cdot 53^2 \cdot 89$.

In general, point counting over $\mathbb{F}_p$ runs in time polynomial in $\log p$.

Number of points in $[p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}]$.

The group is isomorphic to $\mathbb{Z}/n \times \mathbb{Z}/m$, where $n \mid m$ and $n \mid (p - 1)$. 

Elliptic-curve groups

The discrete-logarithm problem

Define \( p = 1000003 \) and consider the Weierstrass curve
\[ y^2 = x^3 - x \]
over \( \mathbb{F}_p \).
This curve has
\[ 1000004 = 2^2 \cdot 53^2 \cdot 89 \]
points and \( P = (101384, 614510) \) is a point of order \( 2 \cdot 53^2 \cdot 89 \).
In general, point counting over \( \mathbb{F}_p \) runs in time polynomial in \( \log p \).

Number of points in
\[ [p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}] \]
The group is isomorphic to \( \mathbb{Z}/n \times \mathbb{Z}/m \), where \( n \mid m \) and \( n \mid (p - 1) \).

Following algorithms will need a unique representative per point.
For that Weierstrass curves are the speed leader.

Can we find an integer \( n \in \{1, 2, 3, \ldots, 500001\} \) such that \( nP = (670366, 740819) \)?
This point was generated as a multiple of \( P \); could also be outside of cyclic group.
Could find \( n \) by brute force.
Is there a faster way?
The discrete-logarithm problem

Define $p = 1000003$ and consider the Weierstrass curve $y^2 = x^3 - x$ over $\mathbb{F}_p$.

This curve has $1000004 = 2^2 \cdot 53^2 \cdot 89$ points and $P = (101384, 614510)$ is a point of order $2 \cdot 53^2 \cdot 89$.

In general, point counting over $\mathbb{F}_p$ runs in time polynomial in $\log p$.

Number of points in $[p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}]$.

The group is isomorphic to $\mathbb{Z}/n \times \mathbb{Z}/m$, where $n \mid m$ and $n \mid (p - 1)$.

Can we find an integer $n \in \{1, 2, 3, \ldots, 500001\}$ such that $nP = (670366, 740819)$?

This point was generated as a multiple of $P$; could also be outside cyclic group.

Could find $n$ by brute force. Is there a faster way?
The discrete-logarithm problem

Define $p = 1000003$ and consider the Weierstrass curve $y^2 = x^3 - x$ over $\mathbb{F}_p$.

This curve has $1000004 = 2^2 \cdot 53^2 \cdot 89$ points and $P = (101384, 614510)$ is a point of order $2 \cdot 53^2 \cdot 89$.

In general, point counting over $\mathbb{F}_p$ runs in time polynomial in $\log p$.

Number of points in $[p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}]$.

The group is isomorphic to $\mathbb{Z}/n \times \mathbb{Z}/m$, where $n \mid m$ and $n \mid (p - 1)$.

Can we find an integer $n \in \{1, 2, 3, \ldots, 500001\}$ such that $nP = (670366, 740819)$?

This point was generated as a multiple of $P$; could also be outside cyclic group.

Could find $n$ by brute force. Is there a faster way?
The discrete-logarithm problem

Define \( p = 1000003 \) and consider the Weierstrass curve \( y^2 = x^3 - x \) over \( \mathbb{F}_p \).

This curve has \( 1000004 = 2^2 \cdot 53^2 \cdot 89 \) points and \( P = (101384, 614510) \) is a point of order \( 2 \cdot 53^2 \cdot 89 \).

In general, point counting over \( \mathbb{F}_p \) runs in time polynomial in \( \log p \).

Number of points in \([p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}]\).

The group is isomorphic to \( \mathbb{Z}/n \times \mathbb{Z}/m \), where \( n \mid m \) and \( n \mid (p - 1) \).

Can we find an integer \( n \in \{1, 2, 3, \ldots, 500001\} \) such that \( nP = (670366, 740819) \)?

This point was generated as a multiple of \( P \); could also be outside cyclic group.

Could find \( n \) by brute force.

Is there a faster way?
The discrete-logarithm problem

Define $p = 1000003$ and consider the Weierstrass curve $y^2 = x^3 - x$ over $\mathbb{F}_p$.

The curve has $1000004 = 2^2 \cdot 53^2 \cdot 89$ points and $P = (101384, 614510)$ is a point of order $2 \cdot 53^2 \cdot 89$.

In general, point counting over $\mathbb{F}_p$ runs in time polynomial in $\log p$.

Number of points in $[p + 1 - 2\sqrt{p}; p + 1 + 2\sqrt{p}]$.

The group is isomorphic to $\mathbb{Z}/n \times \mathbb{Z}/m$, where $n | m$ and $n | (p - 1)$.

Can we find an integer $n \in \{1, 2, 3, \ldots, 500001\}$ such that $nP = (670366, 740819)$?

This point was generated as a multiple of $P$; could also be outside cyclic group.

Could find $n$ by brute force.

Is there a faster way?

Understanding brute force

Can compute successively $1P = (101384; 614510)$, $2P = (102361; 628914)$, $3P = (77571; 87643)$, $4P = (650289; 31313)$, $500001P = -P$, $500002P = \infty$.

At some point we'll find $n$ with $nP = (670366; 740819)$.

Maximum cost of computation:

$\leq 500001$ additions of $P$;

$\leq 500001$ nanoseconds on a CPU that does 1 ADD/nanosecond.
The discrete-logarithm problem

Define \( p = 1000003 \) and consider the Weierstrass curve \( y^2 = x^3 - x \) over \( \mathbb{F}_p \).

This curve has \( 1000004 = 2 \cdot 53^2 \cdot 89 \) points and \( P = (101384, 614510) \) is a point of order \( 2 \cdot 53^2 \cdot 89 \).

In general, point counting over \( \mathbb{F}_p \) runs in time polynomial in \( \log p \).

Number of points in \( [p + 1 - 2\sqrt{p}; p + 1 + 2\sqrt{p}] \).

The group is isomorphic to \( \mathbb{Z} = n \times \mathbb{Z} = m \), where \( n \mid m \) and \( n \mid (p - 1) \).

Can we find an integer \( n \in \{1, 2, 3, \ldots, 500001\} \) such that \( nP = (670366, 740819) \)?

This point was generated as a multiple of \( P \); could also be outside cyclic group.

Could find \( n \) by brute force.

Is there a faster way?

Understanding brute force

Can compute successively
\[ 1P = (101384, 614510) \]
\[ 2P = (102361, 628914) \]
\[ 3P = (77571, 87643) \]
\[ 4P = (650289, 31313) \]
\[ 500001P = -P \]
\[ 500002P = \infty \].

At some point we’ll find \( n \) with \( nP = (670366, 740819) \).

Maximum cost of computation:
\[ \leq 500001 \text{ additions of } P \]
\[ \leq 500001 \text{ nanoseconds on a CPU that does } 1 \text{ ADD/nanosecond} \].
Can we find an integer $n \in \{1, 2, 3, \ldots, 500001\}$ such that $nP = (670366, 740819)$?

This point was generated as a multiple of $P$; could also be outside cyclic group.

Could find $n$ by brute force. Is there a faster way?

Maximum cost of computation:
$\leq 500001$ additions of $P$;
$\leq 500001$ nanoseconds on a CPU that does 1 ADD/nanosecond.
Can we find an integer \( n \in \{1, 2, 3, \ldots, 500001\} \) such that \( nP = (670366, 740819) \)?

This point was generated as a multiple of \( P \); could also be outside cyclic group.

Could find \( n \) by brute force. Is there a faster way?

---

Understanding brute force

Can compute successively

\[
\begin{align*}
1P &= (101384, 614510), \\
2P &= (102361, 628914), \\
3P &= (77571, 87643), \\
4P &= (650289, 31313), \\
500001P &= -P, \\
500002P &= \infty.
\end{align*}
\]

At some point we’ll find \( n \) with \( nP = (670366, 740819) \).

Maximum cost of computation:

\[
\begin{align*}
&\leq 500001 \text{ additions of } P; \\
&\leq 500001 \text{ nanoseconds on a CPU that does 1 ADD/nanosecond.}
\end{align*}
\]
Can we find an integer 
\( n \in \{1, 2, 3, \ldots, 500001\} \)
such that \( nP = (670366, 740819) \)?

This point was generated as
a multiple of \( P \); could also be
outside cyclic group.

Could find \( n \) by brute force.
Is there a faster way?

Understanding brute force

Can compute successively
\( 1P = (101384, 614510), \)
\( 2P = (102361, 628914), \)
\( 3P = (77571, 87643), \)
\( 4P = (650289, 31313), \)
\( 500001P = -P. \)
\( 500002P = \infty. \)

At some point we’ll find \( n \)
with \( nP = (670366, 740819) \).

Maximum cost of computation:
\( \leq 500001 \) additions of \( P \);
\( \leq 500001 \) nanoseconds on a CPU
that does 1 ADD/nanosecond.

This is negligible work
for \( p \approx 2^{20} \).
But users can
standardize a larger \( p \),
making the attack slower.

Attack cost scales linearly:
\( \approx 2^{50} \) ADDs for \( p \approx 2^{50} \),
\( \approx 2^{100} \) ADDs for \( p \approx 2^{100} \), etc.
(Not exactly linearly:
cost of ADDs grows with \( p \).
But this is a minor effect.)
Can we find an integer $n \in \{1, 2, 3, \ldots, 500001\}$ such that $nP = (670366, 740819)$? This point was generated as a multiple of $P$; could also be outside cyclic group. Could find $n$ by brute force. Is there a faster way?

Understanding brute force

Can compute successively $1P = (101384, 614510)$, $2P = (102361, 628914)$, $3P = (77571, 87643)$, $4P = (650289, 31313)$, $500001P = -P$. $500002P = \infty$.

At some point we’ll find $n$ with $nP = (670366, 740819)$.

Maximum cost of computation:
- $\leq 500001$ additions of $P$;
- $\leq 500001$ nanoseconds on a CPU that does 1 ADD/nanosecond.

This is negligible work for $p \approx 2^{20}$.

But users can standardize a larger $p$, making the attack slower.

Attack cost scales:
- $\approx 2^{50}$ ADDs for $p \approx 2^{50}$,
- $\approx 2^{100}$ ADDs for $p \approx 2^{100}$, etc.

(Not exactly linearly: cost of ADDs grows with $p$. But this is a minor effect.)
Can we find an integer $n \in \{1; 2; 3; \ldots; 500001\}$ such that $nP = (670366; 740819)$? This point was generated as a multiple of $P$; could also be outside cyclic group. Could find $n$ by brute force. Is there a faster way?

**Understanding brute force**

Can compute successively

$1P = (101384; 614510)$,
$2P = (102361; 628914)$,
$3P = (77571; 87643)$,
$4P = (650289; 31313)$,
$500001P = -P$,
$500002P = \infty$.

At some point we’ll find $n$ with $nP = (670366; 740819)$.

Maximum cost of computation:

$\leq 500001$ additions of $P$;

$\leq 500001$ nanoseconds on a CPU that does 1 ADD/nanosecond.

This is negligible work for $p \approx 2^{20}$.

But users can standardize a larger $p$, making the attack slower.

Attack cost scales linearly:

$\approx 2^{50}$ ADDs for $p \approx 2^{50}$,

$\approx 2^{100}$ ADDs for $p \approx 2^{100}$, etc.

(Not exactly linearly: cost of ADDs grows with $p$.
But this is a minor effect.)
Understanding brute force

Can compute successively

1 \( P = (101384, 614510) \),
2 \( P = (102361, 628914) \),
3 \( P = (77571, 87643) \),
4 \( P = (650289, 31313) \),
500001 \( P = -P \).
500002 \( P = \infty \).

At some point we’ll find \( n \) with \( nP = (670366, 740819) \).

Maximum cost of computation:
\( \leq 500001 \) additions of \( P \);
\( \leq 500001 \) nanoseconds on a CPU that does 1 ADD/nanosecond.

This is negligible work for \( p \approx 2^{20} \).

But users can standardize a larger \( p \), making the attack slower.

Attack cost scales linearly:
\( \approx 2^{50} \) ADDs for \( p \approx 2^{50} \),
\( \approx 2^{100} \) ADDs for \( p \approx 2^{100} \), etc.

(Not exactly linearly: cost of ADDs grows with \( p \).
But this is a minor effect.)
Understanding brute force

Can compute successively

1 $P = (101384; 614510)$,
2 $P = (102361; 628914)$,
3 $P = (77571; 87643)$,
4 $P = (650289; 31313)$,
500001 $P = -P$.
500002 $P = \infty$.

At some point we’ll find $n$ with $nP = (670366; 740819)$.

Maximum cost of computation:

$\leq 500001$ additions of $P$;
$\leq 500001$ nanoseconds on a CPU that does 1 ADD/nanosecond.

This is negligible work for $p \approx 2^{20}$.

But users can standardize a larger $p$, making the attack slower.

Attack cost scales linearly:

$\approx 2^{50}$ ADDs for $p \approx 2^{50}$,
$\approx 2^{100}$ ADDs for $p \approx 2^{100}$, etc.

(Not exactly linearly: cost of ADDs grows with $p$.
But this is a minor effect.)

Computation has a good chance of finishing earlier.
Chance scales linearly:

$1 = 2$ chance of $1 = 2$ cost;
$1 = 10$ chance of $1 = 10$ cost; etc.

“So users should choose large $n$.”
Understanding brute force

Can compute successively

1. \( P = (101384; 614510) \),
2. \( P = (102361; 628914) \),
3. \( P = (77571; 87643) \),
4. \( P = (650289; 31313) \),
5. \( P = \) (other values due to subtraction)

At some point we'll find \( n \) with \( nP = (670366; 740819) \).

Maximum cost of computation:

\[ \leq 500001 \text{ additions of } P \]
\[ \leq 500001 \text{ nanoseconds on a CPU that does 1 ADD/nanosecond.} \]

This is negligible work for \( p \approx 2^{20} \).

But users can standardize a larger \( p \), making the attack slower.

Attack cost scales linearly:

\[ \approx 2^{50} \text{ ADDs for } p \approx 2^{50} \]
\[ \approx 2^{100} \text{ ADDs for } p \approx 2^{100} \]

(Not exactly linearly: cost of ADDs grows with \( p \).
But this is a minor effect.)

Computation has a good chance of finishing earlier.

Chance scales linearly:

1/2 chance of 1/2 cost;
1/10 chance of 1/10 cost; etc.

“So users should choose large \( n \).”
Understanding brute force
Can compute successively
1. $P_1 = (101384; 614510)$,
2. $P_2 = (102361; 628914)$,
3. $P_3 = (77571; 87643)$,
4. $P_4 = (650289; 31313)$,
5. $P_{500001} = -P_{500002}$.

At some point we'll find $n_P$ with $n_P P = (670366; 740819)$.

Maximum cost of computation:
- $\leq 500001$ additions of $P$;
- $\leq 500001$ nanoseconds on a CPU that does 1 ADD/nanosecond.

This is negligible work for $p \approx 2^{20}$.

But users can standardize a larger $p$, making the attack slower.

Attack cost scales linearly:
- $\approx 2^{50}$ ADDs for $p \approx 2^{50}$,
- $\approx 2^{100}$ ADDs for $p \approx 2^{100}$, etc.

(Not exactly linearly: cost of ADDs grows with $p$.
But this is a minor effect.)

Computation has a good chance of finishing earlier.
Chance scales linearly:
- $1/2$ chance of $1/2$ cost;
- $1/10$ chance of $1/10$ cost; etc.

“So users should choose large $n$."

This is negligible work for $p \approx 2^{20}$.

But users can standardize a larger $p$, making the attack slower.

Attack cost scales linearly:
$\approx 2^{50}$ ADDs for $p \approx 2^{50}$,
$\approx 2^{100}$ ADDs for $p \approx 2^{100}$, etc.

(Not exactly linearly: cost of ADDs grows with $p$. But this is a minor effect.)

Computation has a good chance of finishing earlier.
Chance scales linearly:
$1/2$ chance of $1/2$ cost;
$1/10$ chance of $1/10$ cost; etc.

“So users should choose large $n$.”
This is negligible work for $p \approx 2^{20}$.

But users can standardize a larger $p$, making the attack slower.

Attack cost scales linearly:
$\approx 2^{50}$ ADDs for $p \approx 2^{50}$,
$\approx 2^{100}$ ADDs for $p \approx 2^{100}$, etc.
(Not exactly linearly: cost of ADDs grows with $p$.
But this is a minor effect.)

Computation has a good chance of finishing earlier.
Chance scales linearly:
$1/2$ chance of $1/2$ cost;
$1/10$ chance of $1/10$ cost; etc.

“So users should choose large $n$.”

That’s pointless. We can apply “random self-reduction”:
choose random $r$, say 69961;
compute $rP = (593450, 987590)$;
compute $(r + n)P$ as $(593450, 987590)+(670366, 740819)$;
compute discrete log;
subtract $r \mod 500002$; obtain $n$. 
negligible work for $p \approx 2^{20}$.

Users can standardize a larger $p$, making the attack slower.

Cost scales linearly:
- $\approx 2^{50}$ ADDs for $p \approx 2^{50}$
- $\approx 2^{100}$ ADDs for $p \approx 2^{100}$, etc.

(Not exactly linearly: cost of ADDs grows with $p$. But this is a minor effect.)

Computation has a good chance of finishing earlier.
Chance scales linearly:
- $1/2$ chance of $1/2$ cost
- $1/10$ chance of $1/10$ cost; etc.

“So users should choose large $n$.”

That’s pointless. We can apply “random self-reduction”:
choose random $r$, say 69961;
compute $rP = (593450, 987590)$;
compute $(r + n)P$ as $(593450, 987590) + (670366, 740819)$;
compute discrete log;
subtract $r \mod 500002$; obtain $n$.

Computation can be parallelized.
One low-cost chip can run many parallel searches.
Example, $2^{6}$ e: one chip, $2^{10}$ cores on the chip, each $2^{30}$ ADDs/second?
Maybe; see SHARCS workshops for detailed cost analyses.

Attacker can run many parallel chips.
Example, $2^{30}$ e: $2^{24}$ chips, so $2^{34}$ cores, so $2^{64}$ ADDs/second, so $2^{89}$ ADDs/year.
Computation has a good chance of finishing earlier.
Chance scales linearly:
1/2 chance of 1/2 cost;
1/10 chance of 1/10 cost; etc.

“So users should choose large $n$.”

That’s pointless. We can apply “random self-reduction”:
choose random $r$, say 69961;
compute $rP = (593450, 987590)$;
compute $(r + n)P$ as
$(593450, 987590) + (670366, 740819)$;
compute discrete log;
subtract $r \mod 500002$; obtain $n$.

Computation can be parallelized.
One low-cost chip can run many parallel searches.
Example, $2^6 \in$: one chip, $2^{10}$ cores on the chip,
each $2^{30}$ ADDs/second.
Maybe; see SHARCS workshops for detailed cost analysis.

Attacker can run many parallel chips.
Example, $2^{30} \in$: 2^{24} chips,
so $2^{34}$ cores,
so $2^{64}$ ADDs/second,
so $2^{89}$ ADDs/year.
Computation has a good chance of finishing earlier. Chance scales linearly: 1/2 chance of 1/2 cost; 1/10 chance of 1/10 cost; etc.

“So users should choose large $n$.”

That’s pointless. We can apply “random self-reduction”:
choose random $r$, say 69961;
compute $rP = (593450, 987590)$;
compute $(r + n)P$ as $(593450, 987590) + (670366, 740819)$;
compute discrete log;
subtract $r \mod 500002$; obtain $n$.

Computation can be parallelized.
One low-cost chip can run many parallel searches. Example, $2^6 \in$: one chip, $2^{10}$ cores on the chip, each $2^{30}$ ADDs/second?
Maybe; see SHARCS workshops for detailed cost analyses.

Attacker can run many parallel chips. Example, $2^{30} \in$: $2^{24}$ chips, so $2^{34}$ cores, so $2^{64}$ ADDs/second, so $2^{89}$ ADDs/year.
Computation has a good chance of finishing earlier.
Chance scales linearly:
1/2 chance of 1/2 cost;
1/10 chance of 1/10 cost; etc.

“So users should choose large $n$.”

That’s pointless. We can apply “random self-reduction”:
choose random $r$, say 69961;
compute $rP = (593450, 987590)$;
compute $(r + n)P$ as
$(593450, 987590) + (670366, 740819)$;
compute discrete log;
subtract $r$ mod 500002; obtain $n$.

Computation can be parallelized.
One low-cost chip can run many parallel searches.
Example, $2^6 \in$: one chip,
$2^{10}$ cores on the chip, each $2^{30}$ ADDs/second?
Maybe; see SHARCS workshops for detailed cost analyses.

Attacker can run many parallel chips.
Example, $2^{30} \in$: $2^{24}$ chips,
so $2^{34}$ cores,
so $2^{64}$ ADDs/second,
so $2^{89}$ ADDs/year.
Computation has a good chance of finishing earlier.

Chance scales linearly:
- 1/2 chance of 1/2 cost;
- 1/10 chance of 1/10 cost; etc.

“Users should choose large $n$.”

That’s pointless. We can apply “random self-reduction”:
- Choose random $r$, say 69961;
- Compute $rP = (593450, 987590)$;
- Compute $(r + n)P$ as
  $(593450, 987590) + (670366, 740819)$;
- Compute discrete log;
  $r \mod 500002$; obtain $n$.

Computation can be parallelized.
One low-cost chip can run many parallel searches.
Example, $2^6 \in$: one chip, $2^{10}$ cores on the chip,
each $2^{30}$ ADDs/second?
Maybe; see SHARCS workshops for detailed cost analyses.

Attacker can run many parallel chips.
Example, $2^{30} \in$: $2^{24}$ chips,
each $2^{34}$ cores,
so $2^{89}$ ADDs/year.

Multiple targets and giant steps
Computation can be applied to many targets at once.

Given $100$ DL targets $n_1P, n_2P, \ldots$:
Can find all of $n_1, \ldots, n_100$ with $\leq 500002$ ADDs.

Simplest approach: First build a sorted table containing $n_1P, \ldots, n_100P$.
Then check table for $1P, 2P, \ldots$.
Computation has a good chance of finishing earlier.

Chance scales linearly:
\[ 1 = 2 \text{ chance of } 1 = 2 \text{ cost}; \]
\[ 1 = 10 \text{ chance of } 1 = 10 \text{ cost}; \] etc.

"So users should choose large \( n \)."

We can apply "random self-reduction": choose random \( r \), say 69961; compute \( r^{P} = (593450, 987590); \) compute \( (r + n)^{P} \) as \( (593450, 987590) + (670366, 740819); \) compute discrete log; subtract \( r \) mod 500002; obtain \( n \).

Computation can be parallelized.

One low-cost chip can run many parallel searches.
Example, \( 2^6 \in \): one chip, \( 2^{10} \) cores on the chip, each \( 2^{30} \) ADDs/second?
Maybe; see SHARCS workshops for detailed cost analyses.

Attacker can run many parallel chips.
Example, \( 2^{30} \in \): \( 2^{24} \) chips, \( 2^{34} \) cores, \( 2^{64} \) ADDs/second, \( 2^{89} \) ADDs/year.

Multiple targets and giant steps
Computation can be applied to many targets at once.
Given 100 DL targets \( n_1 P, n_2 P, \ldots, n_{100} P \):
Can find all of \( n_1, n_2, \ldots, n_{100} \) with \( \leq 500002 \) ADDs.

Simplest approach: First build a sorted table containing \( n_1 P, n_2 P, \ldots, n_{100} P \). Then check table for \( 1 P, 2 P, \ldots, 100 P \).
Computation can be parallelized.

One low-cost chip can run many parallel searches.
Example, $2^6 \in$: one chip, $2^{10}$ cores on the chip, each $2^{30}$ ADDs/second? Maybe; see SHARCS workshops for detailed cost analyses.

Attacker can run many parallel chips.
Example, $2^{30} \in$: $2^{24}$ chips, so $2^{34}$ cores, so $2^{64}$ ADDs/second, so $2^{89}$ ADDs/year.

Multiple targets and giant steps

Computation can be applied to many targets at once.

Given 100 DL targets $n_1P$, $n_2P$, \ldots, $n_{100}P$:
Can find all of $n_1$, $n_2$, \ldots, $n_{100}$ with $\leq 500002$ ADDs.

Simplest approach: First build a sorted table containing $n_1P$, \ldots, $n_{100}P$.
Then check table for $1P$, $2P$, etc.
Computation can be parallelized.

One low-cost chip can run many parallel searches.
Example, $2^6 \in$: one chip, $2^{10}$ cores on the chip, each $2^{30}$ ADDs/second?
Maybe; see SHARCS workshops for detailed cost analyses.

Attacker can run many parallel chips.
Example, $2^{30} \in$: $2^{24}$ chips, so $2^{34}$ cores, so $2^{64}$ ADDs/second, so $2^{89}$ ADDs/year.

Multiple targets and giant steps

Computation can be applied to many targets at once.

Given 100 DL targets $n_1 P$, $n_2 P$, $\ldots$, $n_{100} P$:
Can find all of $n_1, n_2, \ldots, n_{100}$ with $\leq 500002$ ADDs.

Simplest approach: First build a sorted table containing $n_1 P, \ldots, n_{100} P$.
Then check table for $1P, 2P$, etc.
Computation can be parallelized. One low-cost chip can run many parallel searches.

Example, $2^6$ €: one chip, $2^{30}$ €: 24 chips, $2^{60}$ €: 2 ADDs/second?

see SHARCS workshops for detailed cost analyses.

Multiple targets and giant steps

Computation can be applied to many targets at once.

Given 100 DL targets $n_1P$, $n_2P$, $\ldots$, $n_{100}P$:
Can find all of $n_1$, $n_2$, $\ldots$, $n_{100}$ with $\leq 500002$ ADDs.

Simplest approach: First build a sorted table containing $n_1P$, $\ldots$, $n_{100}P$.
Then check table for $1P$, $2P$, etc.

Interesting consequence #1: Solving all 100 DL problems isn’t much harder than solving one.

Interesting consequence #2: Solving at least one out of 100 DL problems is much easier than solving one.

When did this computation find its first $n_i$?
Computation can be parallelized. One low-cost chip can run many parallel searches. Example: $2^{24}$ chips, so $2^{34}$ cores, so $2^{64}$ ADDs/second? Maybe; see SHARCS workshops for detailed cost analyses.

Attacker can run many parallel chips. Example: $2^{30}$: $2^{24}$ chips, so $2^{34}$ cores, so $2^{64}$ ADDs/second, so $2^{89}$ ADDs/year.

Multiple targets and giant steps

Computation can be applied to many targets at once.

Given 100 DL targets $n_1 P$, $n_2 P$, $..., n_{100} P$:
Can find all of $n_1, n_2, ..., n_{100}$ with $\leq 500002$ ADDs.

Simplest approach: First build a sorted table containing $n_1 P$, $n_2 P$, $..., n_{100} P$.
Then check table for $1 P$, $2 P$, etc.

Interesting consequence #1:
Solving all 100 DL problems isn't much harder than solving one DL problem.

Interesting consequence #2:
Solving at least one out of 100 DL problems is much easier than solving one DL problem.

When did this computation find its first $n_i$?
Multiple targets and giant steps

Computation can be applied to many targets at once.

Given 100 DL targets $n_1 P$, $n_2 P$, $\ldots$, $n_{100} P$:
Can find all of $n_1, n_2, \ldots, n_{100}$ with $\leq 500002$ ADDs.

Simplest approach: First build a sorted table containing $n_1 P$, $\ldots$, $n_{100} P$.
Then check table for $1P$, $2P$, etc.

Interesting consequence #1:
Solving all 100 DL problems isn’t much harder than solving one DL problem.

Interestingly consequence #2:
Solving at least one out of 100 DL problems is much easier than solving one DL problem.

When did this computation find its first $n_i$?
Multiple targets and giant steps

Computation can be applied to many targets at once.

Given 100 DL targets $n_1 P, n_2 P, \ldots, n_{100} P$:
Can find all of $n_1, n_2, \ldots, n_{100}$ with $\leq 500002$ ADDs.

Simplest approach: First build a sorted table containing $n_1 P, \ldots, n_{100} P$.
Then check table for $1P, 2P$, etc.

Interesting consequence #1:
Solving all 100 DL problems isn’t much harder than solving one DL problem.

Interesting consequence #2:
Solving at least one out of 100 DL problems is much easier than solving one DL problem.

When did this computation find its first $n_i$?
Multiple targets and giant steps

Computation can be applied to many targets at once.

Given 100 DL targets \( n_1 P, n_2 P, \ldots, n_{100} P \):
Can find all of \( n_1, n_2, \ldots, n_{100} \) with \( \leq 500002 \) ADDs.

Simplest approach: First build a sorted table containing \( n_1 P, \ldots, n_{100} P \).
Then check table for \( 1P, 2P, \) etc.

Interesting consequence #1:
Solving all 100 DL problems isn’t much harder than solving one DL problem.

Interesting consequence #2:
Solving at least one out of 100 DL problems is much easier than solving one DL problem.

When did this computation find its first \( n_i \)?
Typically \( \approx 500002/100 \) mults.
Multiple targets and giant steps

Computation can be applied to many targets at once.

Given 100 DL targets \( n_1 P, \ldots, n_{100} P \): Can find all of \( n_1; n_2; \ldots; n_{100} \) with \( \leq 500002 \) ADDs.

Simplest approach: First build a sorted table containing \( n_1 P, \ldots, n_{100} P \). Then check table for \( 1 P, 2 P, \ldots \).

Interesting consequence #1: Solving all 100 DL problems isn't much harder than solving one DL problem.

Interesting consequence #2: Solving at least one out of 100 DL problems is much easier than solving one DL problem.

When did this computation find its first \( n_i \)? Typically \( \approx \frac{500002}{100} \) muls.

Can use random self-reduction to turn a single target into multiple targets.

Let \( \ell \) be the order of \( P \).

Given \( n P \): Choose random \( r_1; r_2; \ldots; r_{100} \).

Compute \( r_1 P + n P, r_2 P + n P, \ldots \).

Solve these 100 DL problems.

Typically \( \approx \ell = 100 \) muls to find at least one \( r_i + n \) mod \( \ell \), immediately revealing \( n \).
Multiple targets and giant steps

Computation can be applied
to many targets at once.

Given 100 DL targets

\[ n_1^P, \quad n_2^P, \quad \ldots, \quad n_{100}^P \] :

Can find all
of \( n_1^P, \quad n_2^P, \quad \ldots, \quad n_{100}^P \)
with \( \leq 500002 \) ADDs.

Simplest approach: First build
a sorted table containing
\( n_1^P, \quad \ldots, \quad n_{100}^P \).

Then check table for
\( 1^P, \quad 2^P, \) etc.

Interesting consequence #1:
Solving all 100 DL problems
isn’t much harder than
solving one DL problem.

Interesting consequence #2:
Solving at least one
out of 100 DL problems
is much easier than
solving one DL problem.

When did this computation
find its first \( n_i \)?
Typically \( \approx 500002/100 \) mults.

Can use random self-reduction
to turn a single target
into multiple targets.

Let \( \ell \) be the order of \( P \).

Given \( n^P \):
Choose random \( r_1, \quad r_2, \ldots, \quad r_{100} \).

Compute \( r_1^P + n^P, \quad r_2^P + n^P, \) etc.

Solve these 100 DL problems.

Typically \( \approx \ell/100 \) mults
to find at least one
\( r_i + n \mod \ell \),
immediately revealing \( n \).
Interesting consequence #1: Solving all 100 DL problems isn’t much harder than solving one DL problem.

Interesting consequence #2: Solving *at least one* out of 100 DL problems is much easier than solving one DL problem.

When did this computation find its first $n_i$?
Typically $\approx 500002/100$ mults.

Can use random self-reduction to turn a single target into multiple targets.
Let $\ell$ be the order of $P$.

Given $nP$:
Choose random $r_1, r_2, \ldots, r_{100}$.
Compute $r_1 P + n P$, $r_2 P + n P$, etc.

Solve these 100 DL problems.
Typically $\approx \ell/100$ mults
to find *at least one* $r_i + n \mod \ell$,
immediately revealing $n$. 
Interesting consequence #1: Solving all 100 DL problems isn’t much harder than solving one DL problem.

Interesting consequence #2: Solving at least one out of 100 DL problems is much easier than solving one DL problem.

When did this computation find its first $n_i$?

Typically $\approx 50000^2 = 100$ mults.

Can use random self-reduction to turn a single target into multiple targets.

Let $\ell$ be the order of $P$.

Given $nP$:
Choose random $r_1, r_2, \ldots, r_{100}$.
Compute $r_1P + nP, r_2P + nP$, etc.

Solve these 100 DL problems. Typically $\approx \ell/100$ mults to find at least one $r_i + n \text{ mod } \ell$, immediately revealing $n$. 

Interesting consequence #1: Solving all 100 DL problems isn't much harder than solving one DL problem.

Interesting consequence #2: Solving at least one out of 100 DL problems is much easier than solving one DL problem.

When did this computation find its first \( n_i \)? Typically \( \approx 500002 \) mults.

Can use random self-reduction to turn a single target into multiple targets.

Let \( \ell \) be the order of \( P \).

Given \( nP \):
Choose random \( r_1, r_2, \ldots, r_{100} \).
Compute \( r_1P + nP \), \( r_2P + nP \), etc.

Solve these 100 DL problems.
Typically \( \approx \ell/100 \) mults to find at least one \( r_i + n \mod \ell \), immediately revealing \( n \).

Also spent some ADDs to compute each \( r_iP \): \( \approx \lg p \) ADDs for each \( i \).

Faster: Choose \( r_i = ir_1 \) with \( r_1 \approx \ell/100 \).
Compute \( r_1P + nP \), \( 2r_1P + nP \), \( 3r_1P + nP \), etc.
Just 1 ADD for each new \( i \).
Typically \( \approx 100 + \lg \ell + \ell/100 \) ADDs to find \( n \) given \( nP \).
Interesting consequence #1: Solving all 100 DL problems isn't much harder than solving one DL problem.

Interesting consequence #2: Solving at least one out of 100 DL problems is much easier than solving one DL problem.

When did this computation find its first $n$? Typically $\approx 50000^2 = 100$ mults.

Can use random self-reduction to turn a single target into multiple targets. Let $\ell$ be the order of $P$.

Given $nP$: Choose random $r_1, r_2, \ldots, r_{100}$. Compute $r_1P + nP$, $r_2P + nP$, etc.

Solve these 100 DL problems. Typically $\approx \ell/100$ mults to find at least one $r_i + n \mod \ell$, immediately revealing $n$.

Also spent some ADDs to compute each $r_iP$: $\approx \lg p$ ADDs for each.

Faster: Choose $r_i = ir_1$ with $r_1 \approx \ell/100$. Compute $r_1P$; $r_1P + nP$; $2r_1P + nP$; $3r_1P + nP$; etc. Just 1 ADD for each new $i$. $\approx 100 + \lg \ell + \ell/100$ mults to find $n$ given $nP$. 
Can use random self-reduction to turn a single target into multiple targets.
Let \( \ell \) be the order of \( P \).

Given \( nP \):
Choose random \( r_1, r_2, \ldots, r_{100} \).
Compute \( r_1P + nP \),
\( r_2P + nP \), etc.

Solve these 100 DL problems.
Typically \( \approx \ell/100 \) mults
to find \textit{at least one} \( r_i + n \) mod \( \ell \),
immediately revealing \( n \).

Also spent some ADDs to compute each \( r_iP \):
\( \approx \lg p \) ADDs for each \( i \).

Faster: Choose \( r_i = ir_1 \) with \( r_1 \approx \ell/100 \).

Compute \( r_1P \);
\( r_1P + nP \);
\( 2r_1P + nP \);
\( 3r_1P + nP \); etc.
Just 1 ADD for each new \( i \).
\( \approx 100 + \lg \ell + \ell/100 \) ADDs
to find \( n \) given \( nP \).
Can use random self-reduction to turn a single target into multiple targets. Let $\ell$ be the order of $P$.

Given $nP$:
Choose random $r_1, r_2, \ldots, r_{100}$.
Compute $r_1P + nP$, $r_2P + nP$, etc.

Solve these 100 DL problems.
Typically $\approx \ell/100$ mults to find at least one $r_i + n \mod \ell$, immediately revealing $n$.

Also spent some ADDs to compute each $r_iP$:
$\approx \lg p$ ADDs for each $i$.

Faster: Choose $r_i = ir_1$ with $r_1 \approx \ell/100$.
Compute $r_1P$,
$r_1P + nP$,
$2r_1P + nP$;
$3r_1P + nP$; etc.
Just 1 ADD for each new $i$.
$\approx 100 + \lg \ell + \ell/100$ ADDs to find $n$ given $nP$. 
Can use random self-reduction to turn a single target into multiple targets. Let $\ell$ be the order of $P$. Given $nP$, choose random $r_1, r_2, \ldots, r_{100}$. Compute $r_1 P + nP$, $r_2 P + nP$, etc. Solve these 100 DL problems. Typically $\approx \ell/100$ mults to find at least one $r_i + n \mod \ell$, immediately revealing $n$. Also spent some ADDs to compute each $r_i P$: $\approx \lg p$ ADDs for each $i$. Faster: Choose $r_i = i r_1$ with $r_1 \approx \ell/100$. Compute $r_1 P$; $r_1 P + nP$; $2r_1 P + nP$; $3r_1 P + nP$; etc. Just 1 ADD for each new $i$. $\approx 100 + \lg \ell + \ell/100$ ADDs to find $n$ given $nP$. Faster: Increase 100 to $\approx \sqrt{\ell}$. Only $\approx 2 \sqrt{\ell}$ ADDs to solve one DL problem! "Shanks baby-step-giant-step discrete-logarithm algorithm." Example: $p = 1000003$, $\ell = 500002$, $P = (101384, 614510)$, $Q = nP = (670366, 740819)$. Compute $708 P = (393230, 421116)$. Then compute $707$ targets: $708 P + nP = (342867, 153817)$, $2 \cdot 708 P + nP = (430321, 994742)$, $3 \cdot 708 P + nP = (423151, 635197)$, $\ldots$, $706 \cdot 708 P + nP = (534170, 450849)$. Only $\approx 2 \sqrt{\ell}$ ADDs to solve one DL problem! "Shanks baby-step-giant-step discrete-logarithm algorithm." Example: $p = 1000003$, $\ell = 500002$, $P = (101384, 614510)$, $Q = nP = (670366, 740819)$. Compute $708 P = (393230, 421116)$. Then compute $707$ targets: $708 P + nP = (342867, 153817)$, $2 \cdot 708 P + nP = (430321, 994742)$, $3 \cdot 708 P + nP = (423151, 635197)$, $\ldots$, $706 \cdot 708 P + nP = (534170, 450849)$. Only $\approx 2 \sqrt{\ell}$ ADDs to solve one DL problem! "Shanks baby-step-giant-step discrete-logarithm algorithm." Example: $p = 1000003$, $\ell = 500002$, $P = (101384, 614510)$, $Q = nP = (670366, 740819)$. Compute $708 P = (393230, 421116)$. Then compute $707$ targets: $708 P + nP = (342867, 153817)$, $2 \cdot 708 P + nP = (430321, 994742)$, $3 \cdot 708 P + nP = (423151, 635197)$, $\ldots$, $706 \cdot 708 P + nP = (534170, 450849).
Can use random self-reduction to turn a single target into multiple targets. Let \( 
\ell \) be the order of \( P \). Given \( nP \), choose random \( r_1, r_2, \ldots, r_{100} \). Compute \( r_1P, r_2P +nP, \ldots, 100 \cdot r_1P +nP \). Solve these 100 DL problems. Typically \( \approx \ell \) mults to find at least one \( r_i + n \mod \ell \), immediately revealing \( n \).

Also spent some ADDs to compute each \( r_iP \): \( \approx \lg p \) ADDs for each \( i \).

Faster: Choose \( r_i = ir_1 \) with \( r_1 \approx \ell/100 \).
Compute \( r_1P, r_1P +nP, 2r_1P +nP, 3r_1P +nP; \ldots \). Just 1 ADD for each new \( i \).

\( \approx 100 + \lg \ell + \ell/100 \) ADDs to find \( n \) given \( nP \).

Faster: Increase \( 100 \) to \( \approx \sqrt{\ell} \).
Only \( \approx 2\sqrt{\ell} \) ADDs to solve one DL problem.


Example: \( p = 1050003, \ell = 500002, P = (101384, 614510) \), \( Q = nP = (670366, 740819) \).

Compute \( 708P = (393230, 421116) \).
Then compute 707 targets:
\( 708P + Q = (342867, 153817) \),
\( 2 \cdot 708P +nP = (430321, 994742) \),
\( 3 \cdot 708P +nP = (423151, 635197) \),
\( \ldots \), \( 706 \cdot 708P +nP = (534170, 450849) \).
Can use random self-reduction to turn a single target into multiple targets. Let $\ell$ be the order of $P$. Given $nP$:

Choose random $r_1, r_2, \ldots, r_{100}$. Compute $r_1 P + nP$, $r_2 P + nP$, etc. Solve these 100 DL problems. Typically $\approx 100$ mults to find at least one $r_i + n \mod \ell$, immediately revealing $n$.

Also spent some ADDs to compute each $r_i P$:

$\approx \lg p$ ADDs for each $i$.

Faster: Choose $r_i = i r_1$ with $r_1 \approx \ell/100$.

Compute $r_1 P$; $r_1 P + nP$; $2r_1 P + nP$; $3r_1 P + nP$; etc.

Just 1 ADD for each new $i$.

$\approx 100 + \lg \ell + \ell/100$ ADDs to find $n$ given $nP$.

Faster: Increase 100 to $\approx \sqrt{\ell}$. Only $\approx 2\sqrt{\ell}$ ADDs to solve one DL problem!

“Shanks baby-step-giant-step discrete-logarithm algorithm.”

Example: $p = 1000003, \ell = 500002, P = (101384, 614510)$; $Q = nP = (670366, 740819)$. Compute $708P = (393230, 421116)$.

Then compute 707 targets:

$708P + Q = (342867, 153817)$

$2 \cdot 708P + nP = (430321, 994742)$

$3 \cdot 708P + nP = (423151, 635197)$

$\ldots$, $706 \cdot 708P + nP = (534170, 450849)$.
Also spent some ADDs to compute each $r_i P$: 
$\approx \lg p$ ADDs for each $i$.

Faster: Choose $r_i = i r_1$ with $r_1 \approx \ell/100$.
Compute $r_1 P$;
$r_1 P + n P$;
$2 r_1 P + n P$;
$3 r_1 P + n P$; etc.
Just 1 ADD for each new $i$.
$\approx 100 + \lg \ell + \ell/100$ ADDs to find $n$ given $nP$.

Faster: Increase 100 to $\approx \sqrt{\ell}$.
Only $\approx 2\sqrt{\ell}$ ADDs to solve one DL problem!

“Shanks baby-step-giant-step discrete-logarithm algorithm.”

Example: $p = 1000003, \ell = 500002, P = (101384, 614510), Q = n P = (670366, 740819)$.
Compute $708 P = (393230, 421116)$.
Then compute $707$ targets:
$708 P + Q = (342867, 153817)$,
$2 \cdot 708 P + n P = (430321, 994742)$,
$3 \cdot 708 P + n P = (423151, 635197)$,
$\ldots$, $706 \cdot 708 P + n P = (534170, 450849)$. 

Also spent some ADDs to compute each $r_i P$: $\approx \lg p$ ADDs for each $i$.
Faster: Choose $r_i = ir_1 \approx \ell/100$.
Compute $r_1 P$; $2r_1 P$; $3r_1 P$; $nP$; $2nP$; etc.
Just 1 ADD for each new $i$.
$\approx \lg \ell + \ell/100$ ADDs to solve one DL problem!

Faster: Increase 100 to $\approx \sqrt{\ell}$. Only $\approx 2\sqrt{\ell}$ ADDs to solve one DL problem!

“Shanks baby-step-giant-step discrete-logarithm algorithm.”

Example: $p = 1000003$, $\ell = 500002$, $P = (101384, 614510)$, $Q = nP = (670366, 740819)$.
Compute $708P = (393230, 421116)$.
Then compute 707 targets:
$708P + Q = (342867, 153817)$, $2 \cdot 708P + nP = (430321, 994742)$, $3 \cdot 708P + nP = (423151, 635197)$, $\ldots$, $706 \cdot 708P + nP = (534170, 450849)$.

Build a sorted table of targets:
600 $\cdot 708P + Q = (799978, 929249)$, $27 \cdot 708P + Q = (785344, 831127)$, $219 \cdot 708P + Q = (425475, 793466)$, $\ldots$, $317 \cdot 708P + Q = (599785, 189116)$.

Look up $P$, $2P$, $3P$, etc. in table.

$620P = (950652, 688508)$; find $596 \cdot 708P + Q = (950652, 688508)$ in the table of targets; so $620 = 596 \cdot 708 + n$ mod 500002; deduce $n = 78654$. 
Also spent some ADDs to compute each \( r_i \)P :
\[ \approx \lg p \] ADDs for each \( i \).

Faster: Choose \( r_i = ir_1 \) with \( r_1 \approx \sqrt{100} \).
Compute \( r_1P, r_1P + nP, 2r_1P + nP, 3r_1P + nP, \ldots \) with just 1 ADD for each new \( i \).
\[ \approx 100 + \lg \sqrt{100} + \sqrt{100} \approx 100 \text{ ADDs} \] to find \( n \) given \( nP \).

Faster: Increase 100 to \( \approx \sqrt{l} \).
Only \( \approx 2\sqrt{l} \) ADDs to solve one DL problem!

“Shanks baby-step-giant-step discrete-logarithm algorithm.”

Example: \( p = 1000003, l = 500002, P = (101384, 614510), \) \( Q = nP = (670366, 740819). \)
Compute \( 708P = (393230, 421116). \)
Then compute 707 targets:
\[ 708P + Q = (342867, 153817), \]
\[ 2 \cdot 708P + nP = (430321, 994742), \]
\[ 3 \cdot 708P + nP = (423151, 635197), \]
\[ \ldots, 706 \cdot 708P + nP = (534170, 450849). \]

Build a sorted table of targets:
\[ 600 \cdot 708P + Q = (799978, 929249), \]
\[ 27 \cdot 708P + Q = (785344, 831127), \]
\[ 219 \cdot 708P + Q = (425475, 793466), \]
\[ \ldots, 317 \cdot 708P + Q = (599785, 189116). \]

Look up \( P, 2P, 3P, \ldots \) in the table of targets:
\[ 620P = (950652, 688508); \] find \( 596 \cdot 708P + Q = (950652, 688508) \) in the table of targets;
so \( 620 = 596 \cdot 708 + n \mod 500002 \);
deduce \( n = 78654 \).
Faster: Increase 100 to $\approx \sqrt{l}$.
Only $\approx 2\sqrt{l}$ ADDs to solve one DL problem!
“Shanks baby-step-giant-step discrete-logarithm algorithm.”

Example: $p = 1000003, \ell = 500002, P = (101384, 614510), Q = nP = (670366, 740819).

Compute $708P = (393230, 421116)$. Then compute 707 targets:

$708P + Q = (342867, 153817)$,
$2 \cdot 708P + nP = (430321, 994742)$,
$3 \cdot 708P + nP = (423151, 635197)$,
$\ldots$, $706 \cdot 708P + nP = (534170, 450849)$.

Look up $P, 2P, 3P$, etc. in the table of targets:

$620P = (950652, 688508)$; find $596 \cdot 708P + Q = (950652, 688508)$ in the table of targets;
so $620 = 596 \cdot 708 + n \mod 500002$;
deduce $n = 78654$.

Build a sorted table of targets:

$600 \cdot 708P + Q = (799978, 929249)$,
$27 \cdot 708P + Q = (785344, 831127)$,
$219 \cdot 708P + Q = (425475, 793466)$,
$242 \cdot 708P + Q = (262804, 347755)$,
$\ldots$,
$317 \cdot 708P + Q = (599785, 189116)$.
Faster: Increase 100 to $\approx \sqrt{l}$.
Only $\approx 2\sqrt{l}$ ADDs to solve one DL problem!

“Shanks baby-step-giant-step discrete-logarithm algorithm.”

Example: $p = 1000003, \ell = 500002$, $P = (101384, 614510)$, $Q = nP = (670366, 740819)$.
Compute $708P = (393230, 421116)$.
Then compute 707 targets:
$708P + Q = (342867, 153817)$,
$2 \cdot 708P + nP = (430321, 994742)$,
$3 \cdot 708P + nP = (423151, 635197)$,
\ldots, $706 \cdot 708P + nP = (534170, 450849)$.

Build a sorted table of targets:
$600 \cdot 708P + Q = (799978, 929249)$,
$27 \cdot 708P + Q = (785344, 831127)$,
$219 \cdot 708P + Q = (425475, 793466)$,
\ldots
$242 \cdot 708P + Q = (262804, 347755)$,
\ldots
$317 \cdot 708P + Q = (599785, 189116)$.

Look up $P, 2P, 3P$, etc. in table.
$620P = (950652, 688508)$; find
$596 \cdot 708P + Q = (950652, 688508)$ in the table of targets;
so $620 = 596 \cdot 708 + n \mod 500002$;
deduce $n = 78654$. 
Increase 100 to $\approx \sqrt{l}$. 
$2\sqrt{l}$ ADDs to solve one DL problem!


Example: $p = 1000003$; $P = (101384, 614510)$, $Q = nP = (670366, 740819)$.

$708P = (393230, 421116)$.

Compute 707 targets:

$708P + Q = (342867, 153817)$,
$2 \cdot 708P + Q = (430321, 994742)$,
$3 \cdot 708P + Q = (423151, 635197)$,
$\ldots$,
$242 \cdot 708P + Q = (262804, 347755)$,
$\ldots$,
$317 \cdot 708P + Q = (599785, 189116)$.

Look up $P$, $2P$, $3P$, etc. in table.

$620P = (950652, 688508)$; find $596 \cdot 708P + Q = (950652, 688508)$ in the table of targets;
so $620 = 596 \cdot 708 + n$ mod 500002;
deduce $n = 78654$. 

Factors of the group order $P$ has order $2 \cdot 53^2 \cdot 89$.

Given $Q = nP$, find $n = \log_P Q$:

$R = (53^2 \cdot 89)P$ has order 2, and
$S = (53^2 \cdot 89)Q$ is multiple of $R$.

Compute $n_1 = \log_R S \equiv n$ mod 53.

This is a DLP in a group of size 53.
Faster: Increase 100 to $\approx \sqrt{l}$.

Only $\approx 2\sqrt{l}$ ADDs to solve one DL problem!


Example: $p = 1000003$; $q = 500002$, $P = (101384, 614510)$, $Q = nP = (670366, 740819)$.

Compute $708P = (393230, 421116)$.

Then compute 707 targets:

$708P + Q = (342867, 153817)$,
$2 \cdot 708P + nP = (430321, 994742)$,
$3 \cdot 708P + nP = (423151, 635197)$,

:::,
$706 \cdot 708P + nP = (534170, 450849)$.

Build a sorted table of targets:

$600 \cdot 708P + Q = (799978, 929249)$,
$27 \cdot 708P + Q = (785344, 831127)$,
$219 \cdot 708P + Q = (425475, 793466)$,

:::,
$242 \cdot 708P + Q = (262804, 347755)$,
$317 \cdot 708P + Q = (599785, 189116)$.

Look up $P$, $2P$, $3P$, etc. in table.

$620P = (950652, 688508)$; find
$596 \cdot 708P + Q = (950652, 688508)$

in the table of targets;

so $620 = 596 \cdot 708 + n \mod 500002$;

deduce $n = 78654$.

Factors of the group order

$P$ has order $2 \cdot 53^2 \cdot 89$.

Given $Q = nP$, find

$R = (53^2 \cdot 89)P$ has order 2, and

$S = (53^2 \cdot 89)Q$ is multiple of $R$.

Compute $n_1 = \log_R S \equiv n \mod 53$.

$R = (2 \cdot 53 \cdot 89)P$ has order 53,

and

$S = (2 \cdot 53 \cdot 89)Q$ is multiple of $R$.

Compute

$n_2 = \log_R S \equiv n \mod 53$.

This is a DLP in a group of size 53.
Faster: Increase $100$ to $\approx \sqrt{}$.

Only $\approx 2\sqrt{}$ ADDs to solve one DL problem!


Example:

\[ p = 1000003; \]
\[ x = 500002, \quad P = (101384, 614510), \]
\[ Q = nP = (670366, 740819). \]

Compute $708P = (393230, 421116)$. Then compute $707$ targets:

\[ 708P + Q = (342867, 153817), \]
\[ 2 \cdot 708P + nP = (430321, 994742), \]
\[ 3 \cdot 708P + nP = (423151, 635197), \]
\[ \ldots \]
\[ 706 \cdot 708P + nP = (534170, 450849). \]

Build a sorted table of targets:

\[ 600 \cdot 708P + Q = (799978, 929249), \]
\[ 27 \cdot 708P + Q = (785344, 831127), \]
\[ 219 \cdot 708P + Q = (425475, 793466), \]
\[ \ldots \]
\[ 242 \cdot 708P + Q = (262804, 347755), \]
\[ \ldots \]
\[ 317 \cdot 708P + Q = (599785, 189116). \]

Look up $P$, $2P$, $3P$, etc. in table.

\[ 620P = (950652, 688508); \;
\text{find} \;
596 \cdot 708P + Q = (950652, 688508) \]

in the table of targets;

so $620 = 596 \cdot 708 + n \mod 500002$; deduce $n = 78654$.

Factors of the group order

$P$ has order $2 \cdot 53^2 \cdot 89$.

Given $Q = nP$, find $n = \log_P Q$.

\[ R = (53^2 \cdot 89)P \text{ has order } 2, \]
\[ S = (53^2 \cdot 89)Q \text{ is multiple of } R. \]

Compute $n_1 = \log_R S \equiv n \mod 53$. and

\[ R = (2 \cdot 53 \cdot 89)P \text{ has order } 53, \]
\[ S = (2 \cdot 53 \cdot 89)Q \text{ is multiple of } R. \]

Compute $n_2 = \log_R S \equiv n \mod 53$.

This is a DLP in a group of size 53.
Build a sorted table of targets:

- $600 \cdot 708P + Q = (799978, 929249)$
- $27 \cdot 708P + Q = (785344, 831127)$
- $219 \cdot 708P + Q = (425475, 793466)$
- $\vdots$
- $242 \cdot 708P + Q = (262804, 347755)$
- $\vdots$
- $317 \cdot 708P + Q = (599785, 189116)$

Look up $P$, $2P$, $3P$, etc. in the table.

- $620P = (950652, 688508)$; find $596 \cdot 708P + Q = (950652, 688508)$ in the table of targets;
- so $620 = 596 \cdot 708P + n \mod 500002$;
- deduce $n = 78654$.

Factors of the group order

- $P$ has order $2 \cdot 53^2 \cdot 89$.
- Given $Q = nP$, find $n = \log_P Q$:
  - $R = (53^2 \cdot 89)P$ has order 2, and
  - $S = (53^2 \cdot 89)Q$ is multiple of $R$.
  - Compute $n_1 = \log_R S \equiv n \mod 2$.
  - $R = (2 \cdot 53 \cdot 89)P$ has order 53, and
  - $S = (2 \cdot 53 \cdot 89)Q$ is multiple of $R$.
  - Compute $n_2 = \log_R S \equiv n \mod 53$.
  - This is a DLP in a group of size 53.
Factors of the group order

$P$ has order $2 \cdot 53^2 \cdot 89$.

Given $Q = nP$, find $n = \log_P Q$.

$R = (53^2 \cdot 89)P$ has order 2, and $S = (53^2 \cdot 89)Q$ is multiple of $R$.

Compute $n_1 = \log_R S \equiv n \mod 2$.

$R = (2 \cdot 53 \cdot 89)P$ has order 53, and $S = (2 \cdot 53 \cdot 89)Q$ is multiple of $R$.

Compute $n_2 = \log_R S \equiv n \mod 53$.

This is a DLP in a group of size 53.

$T = (2 \cdot 89)(Q - n_2P)$ is also a multiple of $R$.

Compute $n_3 = \log_R T \equiv n \mod 53$.

Now $n_2 + 53n_3 \equiv n \mod 53^2$.

$R = (2 \cdot 53 \cdot 89)P$ has order 89, and $S = (2 \cdot 53 \cdot 89)Q$ is multiple of $R$.

Compute $n_4 = \log_R S \equiv n \mod 89$.

Use Chinese Remainder Theorem

$n \equiv n_1 \mod 2$,

$n \equiv n_2 + 53n_3 \mod 53^2$,

$n \equiv n_4 \mod 89$,

to determine $n$ modulo $2 \cdot 53^2 \cdot 89$. 

Build a sorted table of targets:

$P + Q = (799978, 929249)$,

$P + Q = (785344, 831127)$,

$P + Q = (425475, 793466)$,

::: $P + Q = (262804, 347755)$,

::: $P + Q = (599785, 189116)$.

Look up $P$, $2P$, $3P$, etc. in table.

$620P = (950652, 688508)$; find $596P + Q = (950652, 688508)$ in the table of targets;

so $620 = 596 \cdot 708 + n \mod 500002$;

deduce $n = 78654$. 

$T = (2 \cdot 89)(Q - n_2P)$ is also a multiple of $R$.

Compute $n_3 = \log_R T \equiv n \mod 53$.

Now $n_2 + ... Remainder Theorem

$n \equiv n_1 \mod 2$,

$n \equiv n_2 + 53n_3 \mod 53$,

$n \equiv n_4 \mod 89$,

to determine $n$ modulo $2 \cdot 53^2 \cdot 89$. 

Build a sorted table of targets:

$600 \cdot 708 P + Q = (799978, 929249)$,

$27 \cdot 708 P + Q = (785344, 831127)$,

$219 \cdot 708 P + Q = (425475, 793466)$,

::: $242 \cdot 708 P + Q = (262804, 347755)$,

::: $317 \cdot 708 P + Q = (599785, 189116)$. 

Factors of the group order

$P$ has order $2 \cdot 53^2 \cdot 89$.

Given $Q = nP$, find $n = \log_P Q$.

$R = (53^2 \cdot 89)P$ has order 2, and $S = (53^2 \cdot 89)Q$ is multiple of $R$.

Compute $n_1 = \log_R S \equiv n \mod 2$.

$R = (2 \cdot 53 \cdot 89)P$ has order 53, and $S = (2 \cdot 53 \cdot 89)Q$ is multiple of $R$.

Compute $n_2 = \log_R S \equiv n \mod 53$.

This is a DLP in a group of size 53.
Build a sorted table of targets:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>600 1</td>
<td>708</td>
</tr>
<tr>
<td>27</td>
<td>708</td>
</tr>
<tr>
<td>219</td>
<td>708</td>
</tr>
<tr>
<td>242</td>
<td>708</td>
</tr>
<tr>
<td>317</td>
<td>708</td>
</tr>
</tbody>
</table>

Look up \( P \), \( 2P \), \( 3P \), etc. in table.

\[ 620 \cdot 708 P + Q = (950652; 688508); \text{find } 596 \cdot 708 P + Q = (950652; 688508) \text{ in the table of targets; } \]

so \( 620 = 596 \cdot 708 + n \mod 500002; \)

deduce \( n = 78654.\)

Factors of the group order

\( P \) has order \( 2 \cdot 53^2 \cdot 89.\)

Given \( Q = nP \), find \( n = \log_P Q: \)

\[ R = (53^2 \cdot 89)P \text{ has order 2, and } S = (53^2 \cdot 89)Q \text{ is multiple of } R. \]

Compute \( n_1 = \log_R S \equiv n \mod 2. \)

\[ R = (2 \cdot 53 \cdot 89)P \text{ has order 53, and } S = (2 \cdot 53 \cdot 89)Q \text{ is multiple of } R. \]

Compute \( n_2 = \log_R S \equiv n \mod 53. \)

This is a DLP in a group of size 53.

\( T = (2 \cdot 89)(Q - n_2P) \) is also a multiple of \( R \), i.e., has order 53.

Compute \( n_3 = \log_R T \equiv n \mod 53. \)

Now \( n_2 + 53n_3 \equiv n \mod 53^2. \)

\( R = (2 \cdot 53^2)P \) has order 89, and \( S = (2 \cdot 53^2)Q \text{ is multiple of } R. \)

Compute \( n_4 = \log_R S \equiv n \mod 89. \)

Use Chinese Remainder Theorem \( n \equiv n_1 \mod 2, \)
\( n \equiv n_2 + 53n_3 \mod 53^2, \)
\( n \equiv n_4 \mod 89, \)

to determine \( n \) modulo \( 2 \cdot 53^2 \cdot 89. \)
Factors of the group order

$P$ has order $2 \cdot 53^2 \cdot 89$.

Given $Q = nP$, find $n = \log_P Q$:

$R = (53^2 \cdot 89)P$ has order 2, and

$S = (53^2 \cdot 89)Q$ is multiple of $R$.

Compute $n_1 = \log_R S \equiv n \mod 2$.

$R = (2 \cdot 53 \cdot 89)P$ has order 53, and

$S = (2 \cdot 53 \cdot 89)Q$ is multiple of $R$.

Compute $n_2 = \log_R S \equiv n \mod 53$.

This is a DLP in a group of size 53.

$T = (2 \cdot 89)(Q - n_2P)$ is also a multiple of $R$, i.e., has order 53.

Compute $n_3 = \log_R T \equiv n \mod 53$.

Now $n_2 + 53n_3 \equiv n \mod 53^2$.

$R = (2 \cdot 53^2)P$ has order $89$.

$S = (2 \cdot 53^2)Q$ is multiple of $R$.

Compute $n_4 = \log_R S \equiv n \mod 89$.

Use Chinese Remainder Theorem

$n \equiv n_1 \mod 2$,

$n \equiv n_2 + 53n_3 \mod 53^2$,

$n \equiv n_4 \mod 89$,

to determine $n$ modulo $2 \cdot 53^2 \cdot 89$. 

Subjects:

(9249),

(1127),

(3466),

(7755),

(9116),

table.

find

(88508)

900002;
Factors of the group order

\( P \) has order \( 2 \cdot 53^2 \cdot 89 \).

Given \( Q = nP \), find \( n = \log_P Q \):

\( R = (53^2 \cdot 89)P \) has order 2, and \( S = (53^2 \cdot 89)Q \) is multiple of \( R \).

Compute \( n_1 = \log_R S \equiv n \mod 2 \).

\( R = (2 \cdot 53 \cdot 89)P \) has order 53, and \( S = (2 \cdot 53 \cdot 89)Q \) is multiple of \( R \).

Compute \( n_2 = \log_R S \equiv n \mod 53 \).

This is a DLP in a group of size 53.

\( T = (2 \cdot 89)(Q - n_2P) \) is also a multiple of \( R \), i.e., has order 53.

Compute \( n_3 = \log_R T \equiv n \mod 53 \).

Now \( n_2 + 53n_3 \equiv n \mod 53^2 \).

\( R = (2 \cdot 53^2)P \) has order 89, and \( S = (2 \cdot 53^2)Q \) is multiple of \( R \).

Compute \( n_4 = \log_R S \equiv n \mod 89 \).

Use Chinese Remainder Theorem

\( n \equiv n_1 \mod 2 \),
\( n \equiv n_2 + 53n_3 \mod 53^2 \),
\( n \equiv n_4 \mod 89 \),
to determine \( n \) modulo \( 2 \cdot 53^2 \cdot 89 \).
Factors of the group order $P$ has order $2 \cdot 53^2 \cdot 89$. Given $Q = nP$, find $n = \log_P Q$:

- $(53^2 \cdot 89)P$ has order 2, and
- $(53^2 \cdot 89)Q$ is multiple of $R$.

Compute $n_1 = \log_R S \equiv n \mod 2$.

- $(2 \cdot 53 \cdot 89)P$ has order 53,
- $(2 \cdot 53 \cdot 89)Q$ is multiple of $R$.

Compute $n_4 = \log_R S \equiv n \mod 89$.

Use Chinese Remainder Theorem

$$n \equiv n_1 \mod 2,$$
$$n \equiv n_2 + 53n_3 \mod 53^2,$$
$$n \equiv n_4 \mod 89,$$

to determine $n$ modulo $2 \cdot 53^2 \cdot 89$.

This “Pohlig-Hellman method” converts an order-$ab$ DLP into an order-$a$ DLP, an order-$b$ DLP, and a few scalar multiplications.

Here $(53^2 \cdot 89)P = (1; 0)$ and $(53^2 \cdot 89)Q = \infty$, thus $n_1 = 0$.

$(2 \cdot 53 \cdot 89)P = (539296; 488875)$, $(2 \cdot 53 \cdot 89)Q = (782288; 572333)$.

A search quickly finds $n_2 = 2$.

$(2 \cdot 89)(Q - 2P) = \infty$, thus $n_3 = 0$ and $n_2 + 53n_3 = 2$. 

$T = (2 \cdot 89)(Q - n_2P)$ is also a multiple of $R$, i.e., has order 53. 

Compute $n_3 = \log_R T \equiv n \mod 53$.

Now $n_2 + 53n_3 \equiv n \mod 53^2$.

$R = (2 \cdot 53^2)P$ has order 89, and
- $(2 \cdot 53 \cdot 89)Q$ is multiple of $R$.

Compute $n_4 = \log_R S \equiv n \mod 89$.

$T = (2 \cdot 89)(Q - n_2P)$ is also a multiple of $R$, i.e., has order 53. 

Compute $n_3 = \log_R T \equiv n \mod 53$.

Now $n_2 + 53n_3 \equiv n \mod 53^2$.

$R = (2 \cdot 53^2)P$ has order 89, and
- $(2 \cdot 53 \cdot 89)Q$ is multiple of $R$.

Compute $n_4 = \log_R S \equiv n \mod 89$.

Use Chinese Remainder Theorem

$$n \equiv n_1 \mod 2,$$
$$n \equiv n_2 + 53n_3 \mod 53^2,$$
$$n \equiv n_4 \mod 89,$$

to determine $n$ modulo $2 \cdot 53^2 \cdot 89$. 

This “Pohlig-Hellman method” converts an order-$ab$ DLP into an order-$a$ DLP, an order-$b$ DLP, and a few scalar multiplications. 

Here $(53^2 \cdot 89)P = (1; 0)$ and $(53^2 \cdot 89)Q = \infty$, thus $n_1 = 0$. 

$(2 \cdot 53 \cdot 89)P = (539296; 488875)$, $(2 \cdot 53 \cdot 89)Q = (782288; 572333)$. 

A search quickly finds $n_2 = 2$. 

$(2 \cdot 89)(Q - 2P) = \infty$, thus $n_3 = 0$ and $n_2 + 53n_3 = 2$.
Given $Q = nP$, find $n = \log_P Q$.

$R = (53^2 \cdot 89)P$ has order 2, and $S = (53^2 \cdot 89)Q$ is multiple of $R$.

Compute $n_1 = \log_R T \equiv n \mod 53$.

Now $n_2 + 53n_3 \equiv n \mod 53^2$.

$R = (2 \cdot 53^2)P$ has order 89, and $S = (2 \cdot 53^2)Q$ is multiple of $R$.

Compute $n_4 = \log_R S \equiv n \mod 89$.

Use Chinese Remainder Theorem

$n \equiv n_1 \mod 2,$

$n \equiv n_2 + 53n_3 \mod 53^2,$

$n \equiv n_4 \mod 89,$

to determine $n$ modulo $2 \cdot 53^2 \cdot 89$.

This “Pohlig-Hellman method” converts an order-$ab$ DL into an order-$a$ DL, an order-$b$ DL, and a few scalar multiplications.

Here $(53^2 \cdot 89)P \equiv (1; 0)$ and $(53^2 \cdot 89)Q = \infty$, thus $n_1 = 0$.

$(2 \cdot 53 \cdot 89)P = (539296; 488875)$, $(2 \cdot 53 \cdot 89)Q = (782288; 572333)$.

A search quickly finds $n_2 = 2$.

$(2 \cdot 89)(Q - 2P) = \infty$, thus $n_3 = 0$ and $n_2 + 53n_3 = 2$. 

$T = (2 \cdot 89)(Q - n_2P)$ is also a multiple of $R$, i.e., has order 53.

$R = (2 \cdot 53^2)P$ has order 89, and $S = (2 \cdot 53^2)Q$ is multiple of $R$.

Compute $n_4 = \log_R S \equiv n \mod 89$.
Factors of the group order $P$ has order $2 \cdot 53^2 \cdot 89$.

Given $Q = nP$, find $n = \log_P Q$:

$R = (53^2 \cdot 89)P$ has order 2, and $S = (53^2 \cdot 89)Q$ is multiple of $R$.

Compute $n_3 = \log_R T \equiv n \mod 53$.

Now $n_2 + 53n_3 \equiv n \mod 53^2$.

$R = (2 \cdot 53^2)P$ has order 89, and $S = (2 \cdot 53^2)Q$ is multiple of $R$.

Compute $n_4 = \log_R S \equiv n \mod 89$.

Use Chinese Remainder Theorem

$n \equiv n_1 \mod 2$,

$n \equiv n_2 + 53n_3 \mod 53^2$,

$n \equiv n_4 \mod 89$,

to determine $n$ modulo $2 \cdot 53^2 \cdot 89$.

This “Pohlig-Hellman method” converts an order-$ab$ DL into an order-$a$ DL, an order-$b$ DL, and a few scalar multiplications.

Here $(53^2 \cdot 89)P = (1, 0)$ and $(53^2 \cdot 89)Q = \infty$, thus $n_1 = 0$.

$(2 \cdot 53 \cdot 89)P = (539296, 488875)$,

$(2 \cdot 53 \cdot 89)Q = (782288, 572333)$.

A search quickly finds $n_2 = 2$.

$(2 \cdot 89)(Q - 2P) = \infty$, thus $n_3 = 0$ and $n_2 + 53n_3 = 2$. 

$T = (2 \cdot 89)(Q - n_2P)$ is also a multiple of $R$, i.e., has order 53.

This “Pohlig-Hellman method” converts an order-$ab$ DL into an order-$a$ DL, an order-$b$ DL, and a few scalar multiplications.
\[ T = (2 \cdot 89)(Q - n_2 P) \] is also a multiple of \( R \), i.e., has order 53. Compute
\[ n_3 = \log_R T \equiv n \mod 53. \]
Now
\[ n_2 + 53n_3 \equiv n \mod 53^2. \]
\[ R = (2 \cdot 53^2)P \] has order 89, and \( S = (2 \cdot 53^2)Q \) is multiple of \( R \). Compute
\[ n_4 = \log_R S \equiv n \mod 89. \]
Use Chinese Remainder Theorem
\[ n \equiv n_1 \mod 2, \]
\[ n \equiv n_2 + 53n_3 \mod 53^2, \]
\[ n \equiv n_4 \mod 89, \]
to determine \( n \) modulo \( 2 \cdot 53^2 \cdot 89 \).

This “Pohlig-Hellman method” converts an order-\( ab \) DL into an order-\( a \) DL, an order-\( b \) DL, and a few scalar multiplications.

Here \((53^2 \cdot 89)P = (1, 0)\) and \((53^2 \cdot 89)Q = \infty\), thus \( n_1 = 0 \).

\[ (2 \cdot 53 \cdot 89)P = (539296, 488875), \]
\[ (2 \cdot 53 \cdot 89)Q = (782288, 572333). \]
A search quickly finds \( n_2 = 2 \).
\[ (2 \cdot 89)(Q - 2P) = \infty, \text{ thus } n_3 = 0 \]
and \( n_2 + 53n_3 = 2 \).
\[ T = (2 \cdot 89)(Q - n_2P) \] is also a multiple of \( R \), i.e., has order 53. \[ n_3 = \log_R T \equiv n \mod 53. \]

Now \[ n_2 + 53n_3 \equiv n \mod 53^2. \]

\((53^2 \cdot 89)P = (1, 0)\) and \((53^2 \cdot 89)Q = \infty\), thus \( n_1 = 0 \).

\((2 \cdot 53 \cdot 89)P = (539296, 488875)\), \((2 \cdot 53 \cdot 89)Q = (782288, 572333)\). A search quickly finds \( n_2 = 2 \).

\((2 \cdot 89)(Q - 2P) = \infty\), thus \( n_3 = 0 \) and \( n_2 + 53n_3 = 2 \).

This “Pohlig-Hellman method” converts an order-\( ab \) DL into an order-\( a \) DL, an order-\( b \) DL, and a few scalar multiplications.

Here \((53^2 \cdot 89)P = (1, 0)\) and \((53^2 \cdot 89)Q = \infty\), thus \( n_1 = 0 \).

\((2 \cdot 53^2)P = (877560, 947848)\) and \((2 \cdot 53^2)Q = (822491, 118220)\). Compute \( n_4 = 67 \), e.g. using BSGS.

Use Chinese Remainder Theorem to determine \( n \) modulo \( 2 \cdot 53^2 \cdot 89 \).

Pohlig-Hellman method reduces security of discrete logarithm problem in group generated by \( P \) to security of largest prime order subgroup.
This “Pohlig-Hellman method” converts an order-\(ab\) DL into an order-\(a\) DL, an order-\(b\) DL, and a few scalar multiplications. Here \((53^2 \cdot 89)P = (1, 0)\) and \((53^2 \cdot 89)Q = \infty\), thus \(n_1 = 0\).

\((2 \cdot 53 \cdot 89)P = (539296, 488875)\),
\((2 \cdot 53 \cdot 89)Q = (782288, 572333)\).
A search quickly finds \(n_2 = 2\).

\((2 \cdot 89)(Q - 2P) = \infty\), thus \(n_3 = 0\) and \(n_2 + 53n_3 = 2\).

Use Chinese Remainder Theorem
\(n \equiv 0 \mod 2\),
\(n \equiv 2 \mod 53^2\),
\(n \equiv 67 \mod 89\),
to determine \(n = 78654\).

Pohlig-Hellman method reduces security of discrete logarithm problem in group generated by \(P\) to security of largest prime order subgroup.
T = (2 · 89)(Q − n^2 P) is also a multiple of R, i.e., has order 53. Compute

\[ n^3 = \log_R T \equiv n \mod 53. \]

Now \[ n^2 + 53n^3 \equiv n \mod 53 \]

\[ R = (2 \cdot 53^2)P = (2 \cdot 53^2 \cdot 89) \]

(2 · 53^2)Q = (782288, 572333).

Compute \[ n^4 = 67, \]
e.g. using BSGS.

Use Chinese Remainder Theorem

\[ n \equiv 0 \mod 2, \]
\[ n \equiv 2 \mod 53^2, \]
\[ n \equiv 67 \mod 89, \]
to determine \( n = 78654 \).

This “Pohlig-Hellman method” converts an order-\( ab \) DL into an order-\( a \) DL, an order-\( b \) DL, and a few scalar multiplications.

Here \((53^2 \cdot 89)P = (1, 0)\) and \((53^2 \cdot 89)Q = \infty\), thus \( n_1 = 0 \).

\((2 \cdot 53 \cdot 89)P = (539296, 488875), \]
\((2 \cdot 53 \cdot 89)Q = (782288, 572333). \]

A search quickly finds \( n_2 = 2 \).

\((2 \cdot 89)(Q − 2P) = \infty\), thus \( n_3 = 0 \)
and \( n_2 + 53n_3 = 2 \).

Pohlig-Hellman method reduces security of discrete logarithm problem in group generated by \( P \) to security of largest prime order subgroup.

\((2 
\cdot 89)(Q − 2P) = \infty\), thus \( n_3 = 0 \) and \( n_2 + 53n_3 = 2 \).
This “Pohlig-Hellman method” converts an order-\(ab\) DL into an order-\(a\) DL, an order-\(b\) DL, and a few scalar multiplications.

Here \((53^2 \cdot 89)P = (1, 0)\) and \((53^2 \cdot 89)Q = \infty\), thus \(n_1 = 0\).

\((2 \cdot 53 \cdot 89)P = (539296, 488875)\), \((2 \cdot 53 \cdot 89)Q = (782288, 572333)\).

A search quickly finds \(n_2 = 2\).

\((2 \cdot 89)(Q - 2P) = \infty\), thus \(n_3 = 0\) and \(n_2 + 53n_3 = 2\).

\((2 \cdot 53^2)P = (877560, 947848)\) and \((2 \cdot 53^2)Q = (822491, 118220)\).

Compute \(n_4 = 67\), e.g. using BSGS.

Use Chinese Remainder Theorem
\(n \equiv 0 \mod 2\),
\(n \equiv 2 \mod 53^2\),
\(n \equiv 67 \mod 89\),
to determine \(n = 78654\).

Pohlig-Hellman method reduces security of discrete logarithm problem in group generated by \(P\) to security of largest \textbf{prime} order subgroup.
The "Pohlig-Hellman method" converts an order-$ab$ DL into an order-$a$ DL, an order-$b$ DL, and a few scalar multiplications.

$(2 \cdot 53^2) P = (877560, 947848)$ and $(2 \cdot 53^2) Q = (822491, 118220)$.

Compute $n_4 = 67$, e.g. using BSGS.

Use Chinese Remainder Theorem

$n \equiv 0 \mod 2$,
$n \equiv 2 \mod 53^2$,
$n \equiv 67 \mod 89$,
to determine $n = 78654$.

Pohlig-Hellman method reduces security of discrete logarithm problem in group generated by $P$ to security of largest prime order subgroup.

The rho method

Simplified, non-parallel rho:

Make a pseudo-random walk in the group $\langle P \rangle$, where the next step depends on current point:

$W_{i+1} = f(W_i)$.

Birthday paradox:

Randomly choosing from $\ell$ elements picks one element twice after about $\sqrt{\ell}$ draws.

The walk now enters a cycle.

Cycle-finding algorithm (e.g., Floyd) quickly detects this.
This “Pohlig-Hellman method” converts an order-$ab$ DL into an order-$a$ DL, an order-$b$ DL, and a few scalar multiplications.

Here $(53^2 \cdot 89)P = (1, 0)$ and $(53^2 \cdot 89)Q = \infty$, thus $n_1 = 0$.

$(2 \cdot 53^2)P = (877560, 947848)$ and $(2 \cdot 53^2)Q = (822491, 118220)$. Compute $n_4 = 67$, e.g. using BSGS.

Use Chinese Remainder Theorem $n \equiv 0 \mod 2$, $n \equiv 2 \mod 53^2$, $n \equiv 67 \mod 89$, to determine $n = 78654$.

Pohlig-Hellman method reduces security of discrete logarithm problem in group generated by $P$ to security of largest prime order subgroup.

The rho method

Simplified, non-parallel rho:

Make a pseudo-random walk in the group $\langle P \rangle$, where the next step depends on current point:

$W_{i+1} = f(W_i)$.

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

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

- $(2 \cdot 53^2)P = (877560, 947848)$ and $(2 \cdot 53^2)Q = (822491, 118220)$. Compute $n_4 = 67$, e.g. using BSGS.
- Use Chinese Remainder Theorem $n \equiv 0 \mod 2$, $n \equiv 2 \mod 53^2$, $n \equiv 67 \mod 89$, to determine $n = 78654$.
- Pohlig-Hellman method reduces security of discrete logarithm problem in group generated by $P$ to security of largest prime order subgroup.

The rho method

Simplified, non-parallel rho:

Make a pseudo-random walk in the group $\langle P \rangle$, where the next step depends on current point:

$W_{i+1} = f(W_i)$.

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

The walk now enters a cycle. Cycle-finding algorithm (e.g., Floyd) quickly detects this.
This "Pohlig-Hellman method" converts an order-$ab$ DL into an order-$a$ DL, an order-$b$ DL, and a few scalar multiplications.

Here $(53 \cdot 2 \cdot 89)P = (1; 0)$ and $(53 \cdot 2 \cdot 89)Q = \infty$, thus $n_1 = 0$.

$(2 \cdot 53^2)P = (877560; 947848)$ and $(2 \cdot 53^2)Q = (822491; 118220)$. Compute $n_4 = 67$, e.g. using BSGS.

Use Chinese Remainder Theorem $n \equiv 0 \mod 2,$
$n \equiv 2 \mod 53^2,$
$n \equiv 67 \mod 89,$
to determine $n = 78654$.

Pohlig-Hellman method reduces security of discrete logarithm problem in group generated by $P$ to security of largest prime order subgroup.

The rho method
Simplified, non-parallel rho:
Make a pseudo-random walk in the group $\langle P \rangle$, where the next step depends on current point: $W_{i+1} = f(W_i)$.

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
\((2 \cdot 53^2)P = (877560, 947848)\) and \((2 \cdot 53^2)Q = (822491, 118220)\).
Compute \(n_4 = 67\), e.g. using BSGS.

Use Chinese Remainder Theorem
\(n \equiv 0 \mod 2,\)
\(n \equiv 2 \mod 53^2,\)
\(n \equiv 67 \mod 89,\)
to determine \(n = 78654\).

Pohlig-Hellman method reduces security of discrete logarithm problem in group generated by \(P\) to security of largest prime order subgroup.

The rho method
Simplified, non-parallel rho:
Make a pseudo-random walk in the group \(\langle P \rangle\), where the next step depends on current point: \(W_{i+1} = f(W_i)\).

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
\(P = (877560, 947848)\) and \(Q = (822491, 118220)\).

Let \(n_4 = 67\),

using BSGS.

Chinese Remainder Theorem
mod 2,
mod 53^2,
mod 89,
determine \(n = 78654\).

Pohlig-Hellman method reduces
security of discrete logarithm
problem in group generated by \(P\)
to security of largest prime order
subgroup.

**The rho method**

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
\((2 \cdot 53)^2\) P = (877560; 947848) and \((2 \cdot 53)^2\) Q = (822491; 118220).
Compute \(n^4 = 67\), e.g. using BSGS.

Use Chinese Remainder Theorem
\(n \equiv 0 \mod 2\),
\(n \equiv 2 \mod 53\),
\(n \equiv 67 \mod 89\),
to determine \(n = 78654\).

Pohlig-Hellman method reduces security of discrete logarithm problem in group generated by \(P\) to security of largest prime order subgroup.

The rho method
Simplified, non-parallel rho:
Make a pseudo-random walk in the group \(\langle P \rangle\),
where the next step depends on current point: \(W_{i+1} = f(W_i)\).

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method
Simplified, non-parallel rho:
Make a pseudo-random walk
in the group $\langle P \rangle$,
where the next step depends
on current point: $W_{i+1} = f(W_i)$.

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

The walk now enters a cycle.
Cycle-finding algorithm
(e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method
Simplified, non-parallel rho:
Make a pseudo-random walk
in the group $\langle P \rangle$,
where the next step depends
on current point: $W_{i+1} = f(W_i)$.

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

The walk now enters a cycle.
Cycle-finding algorithm
(e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method
Simplified, non-parallel rho:
Make a pseudo-random walk in the group $\langle P \rangle$, where the next step depends on current point: $W_{i+1} = f(W_i)$.

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:
Make a pseudo-random walk in the group \langle P \rangle,
where the next step depends on current point: \( W_{i+1} = f(W_i) \).

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle. Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method
Simplified, non-parallel rho:
Make a pseudo-random walk in the group $\langle P \rangle$, where the next step depends on current point: $W_{i+1} = f(W_i)$.

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:
Make a pseudo-random walk in the group $\langle P \rangle$, where the next step depends on current point: $W_{i+1} = f(W_i)$.

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method
Simplified, non-parallel rho:
Make a pseudo-random walk
in the group $\langle P \rangle$, 
where the next step depends
on current point: $W_{i+1} = f(W_i)$.

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

The walk now enters a cycle.
Cycle-finding algorithm
(e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:
Make a pseudo-random walk in the group \( \langle P \rangle \), where the next step depends on current point: \( W_{i+1} = f(W_i) \).

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method
Simplified, non-parallel rho:
Make a pseudo-random walk in the group \( \langle P \rangle \), where the next step depends on current point: \( W_{i+1} = f(W_i) \).

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm
(e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

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

The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.
The rho method
Simplified, non-parallel rho:
Make a pseudo-random walk
in the group $\langle P \rangle$,
where the next step depends
on current point: $W_{i+1} = f(W_i)$.

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

The walk now enters a cycle.
Cycle-finding algorithm
(e.g., Floyd) quickly detects this.
The rho method
Simplified, non-parallel rho:
Make a pseudo-random walk
in the group $\langle P \rangle$,
where the next step depends
on current point: $W_{i+1} = f(W_i)$.

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

The walk now enters a cycle.
Cycle-finding algorithm
(e.g., Floyd) quickly detects this.
The rho method

Simplified, non-parallel rho:

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

Birthday paradox:

Randomly choosing from \( \ell \) elements picks one element twice after about \( \sqrt{\pi \ell/2} \) draws. The walk now enters a cycle. Cycle-finding algorithm (e.g., Floyd) quickly detects this.

Assume that for each point we know \( a_i, b_i \in \mathbb{Z} \) so that \( W_i = a_i P + b_i Q \).

Then \( W_i = W_j \) means that \( a_i P + b_i Q = a_j P + b_j Q \), so \( (b_i - b_j) Q = (a_j - a_i) P \).

If \( b_i \neq b_j \) the DLP is solved:

\[ n = (a_j - a_i) = (b_i - b_j). \]
Simplified, non-parallel rho:

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

Birthday paradox: Randomly choosing from $l$ elements picks one element twice after about $p\hat{1} = 2^{l/2}$ draws.

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

Assume that for each point we know $a_i, b_i \in \mathbb{Z}' = \mathbb{Z}$ so that $W_i = a_i P + b_i Q$.

Then $W_i = W_j$ means $a_i P + b_i Q = a_j P + b_j Q$ so $(b_i - b_j)Q = (a_j - a_i)P$.

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

The rho method
Simplified, non-parallel rho:
Make a pseudo-random walk
in the group $\langle P \rangle$, where the next step depends on current point:
$$W_{i+1} = f(W_i).$$

Birthday paradox:
Randomly choosing from \( \ell \) elements picks one element twice after about \( p \delta = 2^{\ell/2} \) draws.
The walk now enters a cycle.
Cycle-finding algorithm (e.g., Floyd) quickly detects this.

Assume that for each point we know \( a_i, b_i \in \mathbb{Z}/\ell\mathbb{Z} \) so that
$$W_i = a_i P + b_i Q.$$ Then \( W_i = W_j \) means that
$$a_i P + b_i Q = a_j P + b_j Q$$ so
$$(b_i - b_j)Q = (a_j - a_i)P.$$ If \( b_i \neq b_j \) the DLP is solved:
$$n = (a_j - a_i)/(b_i - b_j).$$
Assume that for each point we know \( a_i, b_i \in \mathbb{Z}/\ell \mathbb{Z} \) so that \( W_i = a_i P + b_i Q \).

Then \( W_i = W_j \) means that 
\[
 a_i P + b_i Q = a_j P + b_j Q
\]
so 
\[
 (b_i - b_j)Q = (a_j - a_i)P.
\]
If \( b_i \neq b_j \) the DLP is solved:
\[
n = (a_j - a_i)/(b_i - b_j).
\]
Assume that for each point we know \( a_i, b_i \in \mathbb{Z}/\ell \mathbb{Z} \) so that \( W_i = a_i P + b_i Q \).

Then \( W_i = W_j \) means that
\[
a_i P + b_i Q = a_j P + b_j Q
\]
so \( (b_i - b_j)Q = (a_j - a_i)P \).

If \( b_i \neq b_j \) the DLP is solved:
\[
n = (a_j - a_i)/(b_i - b_j).
\]

e.g. \( f(W_i) = a(W_i)P + b(W_i)Q \),

starting from some initial combination \( W_0 = a_0 P + b_0 Q \).

If any \( W_i \) and \( W_j \) collide then \( W_{i+1} = W_{j+1}, W_{i+2} = W_{j+2}, \) etc.
Assume that for each point we know \( a_i, b_i \in \mathbb{Z}/\ell\mathbb{Z} \) so that \( W_i = a_iP + b_iQ \).

Then \( W_i = W_j \) means that \( a_iP + b_iQ = a_jP + b_jQ \) so \( (b_i - b_j)Q = (a_j - a_i)P \).
If \( b_i \neq b_j \) the DLP is solved: \( n = (a_j - a_i)/(b_i - b_j) \).

e.g. \( f(W_i) = a(W_i)P + b(W_i)Q \), starting from some initial combination \( W_0 = a_0P + b_0Q \).
If any \( W_i \) and \( W_j \) collide then \( W_{i+1} = W_{j+1}, W_{i+2} = W_{j+2} \), etc.

If functions \( a(W) \) and \( b(W) \) are random modulo \( \ell \), iterations perform a random walk in \( \langle P \rangle \).
If \( a \) and \( b \) are chosen such that \( f(W_i) = f(-W_i) \), the walk is defined on equivalence classes under \( \pm \).
There are only \( \lceil \ell/2 \rceil \) different classes. This reduces the average number of iterations by a factor of almost \( \sqrt{2} \).

In general, Pollard's rho method can be combined with any easily computed group automorphism of small order.
Assume that for each point we know $a_i, b_i \in \mathbb{Z}/\ell\mathbb{Z}$ so that $W_i = a_i P + b_i Q$.

Then $W_i = W_j$ means that $a_i P + b_i Q = a_j P + b_j Q$ so $(b_i - b_j)Q = (a_j - a_i)P$.

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

e.g. $f(W_i) = a(W_i)P + b(W_i)Q$,

starting from some initial combination $W_0 = a_0 P + b_0 Q$.

If any $W_i$ and $W_j$ collide then $W_{i+1} = W_{j+1}, W_{i+2} = W_{j+2}$, etc.

If functions $a(W)$ and $b(W)$ are random modulo $\ell$, iterations perform a random walk in $\langle P \rangle$.

If $a$ and $b$ are chosen such that $f(W_i) = f(-W_i)$, the walk is defined on equivalence classes under $\pm$.

There are only $\lceil \ell/2 \rceil$ different classes. This reduces the number of iterations of almost exactly $\sqrt{2}$.

In general, Pollard's rho method can be combined with any easily computed group automorphism of small order.
Assume that for each point we know \( a_i, b_i \in \mathbb{Z}/\ell\mathbb{Z} \)
so that \( W_i = a_iP + b_iQ \).

Then \( W_i = W_j \) means that 
\[
a_iP + b_iQ = a_jP + b_jQ
\]
so 
\[
(b_i - b_j)Q = (a_j - a_i)P.
\]
If \( b_i \neq b_j \) the DLP is solved:
\[
n = (a_j - a_i)/(b_i - b_j).
\]

e.g. \( f(W_i) = a(W_i)P + b(W_i)Q \),
starting from some initial combination \( W_0 = a_0P + b_0Q \).
If any \( W_i \) and \( W_j \) collide then 
\[
W_{i+1} = W_{j+1}, \quad W_{i+2} = W_{j+2},
\]
etc.

If functions \( a(W) \) and \( b(W) \) are random modulo \( \ell \), iterations perform a random walk in \( \langle P \rangle \).
If \( a \) and \( b \) are chosen such that 
\[
f(W_i) = f(-W_i)
\]
then the walk is defined on equivalence classes under \( \pm \).
There are only \( \lceil \ell/2 \rceil \) different classes. This reduces the average number of iterations by a factor of almost exactly \( \sqrt{2} \).

In general, Pollard’s rho method can be combined with any easily computed group automorphism of small order.
Assume that for each point we know \( a_i, b_i \in \mathbb{Z} / \ell \mathbb{Z} \) so that \( W_i = a_i P + b_i Q \).

Then \( W_i = W_j \) means that \( a_i P + b_i Q = a_j P + b_j Q \) so \( (b_i - b_j)Q = (a_j - a_i)P \). If \( b_i \neq b_j \) the DLP is solved: \( n = (a_j - a_i)/(b_i - b_j) \).

E.g. \( f(W_i) = a(W_i)P + b(W_i)Q \), starting from some initial combination \( W_0 = a_0 P + b_0 Q \). If any \( W_i \) and \( W_j \) collide then \( W_{i+1} = W_{j+1}, W_{i+2} = W_{j+2} \), etc.

If functions \( a(W) \) and \( b(W) \) are random modulo \( \ell \), iterations perform a random walk in \( \langle P \rangle \). If \( a \) and \( b \) are chosen such that \( f(W_i) = f(-W_i) \) then the walk is defined on equivalence classes under \( \pm \).

There are only \( \lceil \ell / 2 \rceil \) different classes. This reduces the average number of iterations by a factor of almost exactly \( \sqrt{2} \).

In general, Pollard’s rho method can be combined with any easily computed group automorphism of small order.
that for each point

\( a_i, b_i \in \mathbb{Z} / \ell \mathbb{Z} \)

\( W_i = a_i P + b_i Q. \)

If \( W_i = W_j \) means that

\( a_i P + b_i Q = a_j P + b_j Q \)

so \((a_j - a_i)P. \)

\( b_j)Q = (a_j - a_i)P. \)

\( b_j Q = (a_j - a_i)P. \)

If \( b_i \neq b_j \) the DLP is solved:

\( n = (a_j - a_i)/(b_i - b_j). \)

If functions \( a(W) \) and \( b(W) \) are

random modulo \( \ell \), iterations

perform a random walk in \( \langle P \rangle \).

If \( a \) and \( b \) are chosen such that

\( f(W_i) = f(-W_i) \) then the walk

is defined on equivalence classes

under \( \pm \).

There are only \( \lfloor \ell/2 \rfloor \) different classes. This reduces the average

number of iterations by a factor of almost exactly \( \sqrt{2} \).

In general, Pollard’s rho method

can be combined with any easily

computed group automorphism of small order.

Parallel collision search

Running Pollard’s rho method on

\( N \) computers gives speedup of

\( \approx \sqrt{N} \) from increased likelihood

of finding a collision.

Want better way to spread

computation across clients. Want

to find collisions between walks

on different

machines, without

frequent synchronization!

Better method due to van

Oorschot and Wiener (1999).

Declare some subset of \( \langle P \rangle \) to

be distinguished points.
Assume that for each point \( i \) we know \( a_i, b_i \in \mathbb{Z} \) so that \( W_i = a_i P + b_i Q \). Then \( W_i = W_j \) means that 
\[
(a_j - a_i)P.
\]
If \( a \) and \( b \) are chosen such that 
\[
f(W_i) = f(-W_i)
\]
then the walk is defined on equivalence classes under \( \pm \).

There are only \( \lceil \ell/2 \rceil \) different classes. This reduces the average number of iterations by a factor of almost exactly \( \sqrt{2} \).

In general, Pollard’s rho method can be combined with any easily computed group automorphism of small order.

If functions \( a(W) \) and \( b(W) \) are random modulo \( \ell \), iterations perform a random walk in \( \langle P \rangle \).

Parallel collision search
Running Pollard’s rho method on \( N \) computers gives speedup of \( \approx \sqrt{N} \) from increased likelihood of finding collisions.

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


Declare some subset of \( \langle P \rangle \) to be distinguished points.
If functions \( a(W) \) and \( b(W) \) are random modulo \( \ell \), iterations perform a random walk in \( \langle P \rangle \). If \( a \) and \( b \) are chosen such that \( f(W_i) = f(-W_i) \) then the walk is defined on equivalence classes under \( \pm \).

There are only \( \lceil \ell/2 \rceil \) different classes. This reduces the average number of iterations by a factor of almost exactly \( \sqrt{2} \).

In general, Pollard’s rho method can be combined with any easily computed group automorphism of small order.

Parallel collision search

Running Pollard’s rho method on \( N \) computers gives speedup \( \approx \sqrt{N} \) 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!


Declare some subset of \( \langle P \rangle \) to be distinguished points.
If functions $a(W)$ and $b(W)$ are random modulo $\ell$, iterations perform a random walk in $\langle P \rangle$. If $a$ and $b$ are chosen such that $f(W_i) = f(-W_i)$ then the walk is defined on equivalence classes under $\pm$.

There are only $\ceiling{\ell/2}$ different classes. This reduces the average number of iterations by a factor of almost exactly $\sqrt{2}$.

In general, Pollard’s rho method can be combined with any easily computed group automorphism of small order.

### 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!

Better method due to van Oorschot and Wiener (1999). Declare some subset of $\langle P \rangle$ to be distinguished points.
If functions $a(W)$ and $b(W)$ are random modulo $\ell$, iterations perform a random walk in $\langle P \rangle$. If $a$ and $b$ are chosen such that $f(W_i) = f(-W_i)$ then the walk is defined on equivalence classes under $\pm$.

There are only $\lceil \ell/2 \rceil$ different classes. This reduces the average number of iterations by a factor almost exactly $\sqrt{2}$.

In general, Pollard’s rho method can be combined with any easily computed group automorphism of small order.

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!

Better method due to van Oorschot and Wiener (1999). Declare some subset of $\langle P \rangle$ to be distinguished points.

Parallel rho: Perform many walks with different starting points but same update function $f$. If two different walks find the same point, report their subsets to server.

Terminate each walk once it hits a distinguished point and report the point along with $a_i$ and $b_i$ to server. Server receives, stores, and sorts all distinguished points. Two walks reaching the same distinguished point give collision. This collision solves the DLP.
and $b(W)$ are iterations. If functions $a(W)$ and $b(W)$ are random modulo $\ell$, iterations perform a random walk in $\langle P \rangle$. If $a$ and $b$ are chosen such that $f(W_i) = f(-W_i)$ then the walk is defined on equivalence classes under $\pm$. There are only $\lceil \ell^2 \rceil$ different classes. This reduces the average number of iterations by a factor of almost exactly $\sqrt{2}$.

In general, Pollard's rho method can be combined with any easily computed group automorphism of small order.

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!

Better method due to van Oorschot and Wiener (1999). Declare some subset of $\langle P \rangle$ to be distinguished points.

Parallel rho: Perform many walks with different starting points but same update function $f$. If two different walks find the same point, their subsequent steps will match.

Terminate each walk once it hits a distinguished point and report the point along with $a_i$ and $b_i$ to the server. Server receives, stores, and sorts all distinguished points. Two walks reaching same distinguished point give collision.

This collision solves the DLP.

Parallel rho: Perform many walks with different starting points but same update function $f$. If two different walks find the same point, their subsequent steps will match.

Terminate each walk once it hits a distinguished point and report the point along with $a_i$ and $b_i$ to the server. Server receives, stores, and sorts all distinguished points. Two walks reaching same distinguished point give collision. This collision solves the DLP.
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!


Declare some subset of \( \langle P \rangle \) to be distinguished points.

Parallel rho: Perform many walks with different starting points but same update function \( f \).

If two different walks find the same point then their subsequent steps will match.

Terminate each walk once it hits a distinguished point and report the point along with \( a_i \) and \( b_i \) to server.

Server receives, stores, and sorts all distinguished points.

Two walks reaching same distinguished point give collision.

This collision solves the DLP.
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!


Declare some subset of $\langle P \rangle$ to be distinguished points.

Parallel rho: Perform many walks with different starting points but same update function $f$.

If two different walks find the same point then their subsequent steps will match.

Terminate each walk once it hits a distinguished point and report the point along with $a_i$ and $b_i$ to server.

Server receives, stores, and sorts all distinguished points.

Two walks reaching same distinguished point give collision. This collision solves the DLP.
Parallel collision search

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

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

Method due to van Oorschot and Wiener (1999). Declare some subset of $\langle P \rangle$ to be distinguished points.

Parallel rho: Perform many walks with different starting points but same update function $f$. If two different walks find the same point then their subsequent steps will match.

Terminate each walk once it hits a distinguished point and report the point along with $a_i$ and $b_i$ to server.

Server receives, stores, and sorts all distinguished points. Two walks reaching same distinguished point give collision. This collision solves the DLP.
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!


Declare some subset of $\langle P \rangle$ to be distinguished points.

Parallel rho: Perform many walks with different starting points but same update function $f$. If two different walks find the same point then their subsequent steps will match.

Terminate each walk once it hits a distinguished point and report the point along with $a_i$ and $b_i$ to server.

Server receives, stores, and sorts all distinguished points. Two walks reaching same distinguished point give collision. This collision solves the DLP.
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!


Declare some subset of $\langle P \rangle$ to be distinguished points.

Parallel rho: Perform many walks with different starting points but same update function $f$. If two different walks find the same point then their subsequent steps will match.

Terminate each walk once it hits a distinguished point and report the point along with $a_i$ and $b_i$ to server. Server receives, stores, and sorts all distinguished points. Two walks reaching same distinguished point give collision. This collision solves the DLP.
Parallel rho: Perform many walks with different starting points but same update function $f$. If two different walks find the same point then their subsequent steps will match.

Terminate each walk once it hits a distinguished point and report the point along with $a_i$ and $b_i$ to server.

Server receives, stores, and sorts all distinguished points. Two walks reaching same distinguished point give collision. This collision solves the DLP.
Parallel rho: Perform many walks with different starting points but same update function $f$. If two different walks find the same point then their subsequent steps will match.

Terminate each walk once it hits a distinguished point and report the point along with $a_i$ and $b_i$ to server.

Server receives, stores, and sorts all distinguished points. Two walks reaching same distinguished point give collision. This collision solves the DLP.
Parallel rho: Perform many walks with different starting points but same update function $f$. If two different walks find the same point then their subsequent steps will match.

Terminate each walk once it hits a distinguished point and report the point along with $a_i$ and $b_i$ to server.

Server receives, stores, and sorts all distinguished points. Two walks reaching same distinguished point give collision. This collision solves the DLP.
Parallel rho: Perform many walks with different starting points but same update function $f$. If two different walks find the same point then their subsequent steps will match. Terminate each walk once it hits a distinguished point and report the point along with $a_i$ and $b_i$ to server. Server receives, stores, and sorts all distinguished points. Two walks reaching same distinguished point give collision. This collision solves the DLP.
Parallel rho: Perform many walks with different starting points but same update function $f$. If two different walks find the same point then their subsequent steps will match. Terminate each walk once it hits a distinguished point and report the point along with $a_i$ and $b_i$ to server. Server receives, stores, and sorts all distinguished points. Two walks reaching same distinguished point give collision. This collision solves the DLP.
Parallel rho: Perform many walks with different starting points but the same update function $f$. If two different walks find the same point then their subsequent steps will match. Terminate each walk once it hits a distinguished point and report it along with $a_i$ and $b_i$ to the server.

The server receives, stores, and sorts all distinguished points. Two walks reaching the same distinguished point give a collision. This collision solves the DLP.

Attacker chooses the frequency and definition of distinguished points. Tradeoffs are possible:

- If distinguished points are rare, a small number of very long walks will be performed. This reduces the number of distinguished points sent to the server but increases the delay before a collision is recognized.
- If distinguished points are frequent, many shorter walks will be performed.

In any case, do not wait for a cycle. Total # of iterations unchanged.
Perform many walks with different starting points but same update function $f$. If two different walks find the same point then their subsequent steps will match. Terminate each walk once it hits a distinguished point and report the point along with $a_i$ and $b_i$ to the server. 

The server receives, stores, and sorts all distinguished points. Two walks reaching the same distinguished point give a collision. This collision solves the DLP.

Attacker chooses the frequency and definition of distinguished points. Tradeoffs are possible: If distinguished points are rare, a small number of very long walks will be performed. This reduces the number of distinguished points sent to the server but increases the delay before a collision is recognized. If distinguished points are frequent, many shorter walks will be performed.

In any case do not wait for cycle. Total # of iterations unchanged.
Parallel rho: Perform many walks with different starting points but same update function $f$. If two different walks find the same point then their subsequent steps will match. Terminate each walk once it hits a distinguished point and report $a_i$ and $b_i$ to the server. Server receives, stores, and sorts all distinguished points. Two walks reaching the same distinguished point result in a collision. This collision solves the DLP.

Attacker chooses frequency and definition of distinguished points. Tradeoffs are possible:
If distinguished points are rare, a small number of very long walks will be performed. This reduces the number of distinguished points sent to the server but increases the delay before a collision is recognized. If distinguished points are frequent, many shorter walks will be performed.
In any case do not wait for cycle. Total # of iterations unchanged.
Attacker chooses frequency and definition of distinguished points. Tradeoffs are possible: If distinguished points are rare, a small number of very long walks will be performed. This reduces the number of distinguished points sent to the server but increases the delay before a collision is recognized. If distinguished points are frequent, many shorter walks will be performed. In any case do not wait for cycle. Total # of iterations unchanged.
Attacker chooses frequency and definition of distinguished points. Tradeoffs are possible:
If distinguished points are rare, a small number of very long walks will be performed. This reduces the number of distinguished points sent to the server but increases the delay before a collision is recognized.
If distinguished points are frequent, many shorter walks will be performed.
In any case do not wait for cycle. Total # of iterations unchanged.
Attacker chooses frequency and definition of distinguished points. Tradeoffs are possible: If distinguished points are rare, a small number of very long walks will be performed. This reduces the number of distinguished points sent to the server but increases the delay before a collision is recognized. If distinguished points are frequent, many shorter walks will be performed. In any case do not wait for cycle. Total # of iterations unchanged.
Attacker chooses frequency and definition of distinguished points. Tradeoffs are possible:
If distinguished points are rare, a small number of very long walks will be performed. This reduces the number of distinguished points sent to the server but increases the delay before a collision is recognized.
If distinguished points are frequent, many shorter walks will be performed.
In any case do not wait for cycle. Total # of iterations unchanged.
Attacker chooses frequency and definition of distinguished points. Tradeoffs are possible:
If distinguished points are rare, a small number of very long walks will be performed. This reduces the number of distinguished points sent to the server but increases the delay before a collision is recognized.
If distinguished points are frequent, many shorter walks will be performed.
In any case do not wait for cycle. Total # of iterations unchanged.
The attacker chooses the frequency and definition of distinguished points. Tradeoffs are possible:

- If distinguished points are rare, a small number of very long walks will be performed. This reduces the number of messages sent to the server but increases the delay before a collision is recognized.
- If distinguished points are frequent, many shorter walks will be performed.

In any case, do not wait for the cycle to occur. The total number of iterations remains unchanged.

Additive walks

Generic rho method requires two scalar multiplications for each iteration. Could replace by double-scalar multiplication; could further merge the scalar multiplications across several iterations.
Attacker chooses frequency and definition of distinguished points. Tradeoffs are possible: If distinguished points are rare, a small number of very long walks will be performed. This reduces the number of distinguished points sent to the server but increases the delay before a collision is recognized. If distinguished points are frequent, many shorter walks will be performed. In any case do not wait for cycle. Total # of iterations unchanged.

Additive walks
Generic rho method requires two scalar multiplications per iteration. Could replace by one double-scalar multiplication; could further merge the 2-scalar multiplications across several parallel iterations.
Attacker chooses frequency and definition of distinguished points. Tradeoffs are possible:
If distinguished points are rare, a small number of very long walks will be performed. This reduces the number of distinguished points sent to the server but increases the delay before a collision is recognized.
If distinguished points are frequent, many shorter walks will be performed.
In any case do not wait for cycle. Total # of iterations unchanged.

Additive walks
Generic rho method requires two scalar multiplications for each iteration. Could replace by double-scalar multiplication; could further merge the 2-scalar multiplications across several parallel iterations.
Additive walks

Generic rho method requires two scalar multiplications for each iteration. Could replace by double-scalar multiplication; could further merge the 2-scalar multiplications across several parallel iterations.
Additive walks

Generic rho method requires two scalar multiplications for each iteration. Could replace by double-scalar multiplication; could further merge the 2-scalar multiplications across several parallel iterations.

More efficient: use additive walk:
Start with $W_0 = a_0 P$ and put $f(W_i) = W_i + c_j P + d_j Q$ where $j = h(W_i)$. 
Additive walks

Generic rho method requires two scalar multiplications for each iteration.
Could replace by double-scalar multiplication; could further merge the 2-scalar multiplications across several parallel iterations.

More efficient: use additive walk:
Start with $W_0 = a_0 P$ and put $f(W_i) = W_i + c_j P + d_j Q$ where $j = h(W_i)$.

Pollard’s initial proposal:
Use $x(W_i) \mod 3$ as $h$ and update:

$$W_{i+1} = \begin{cases} W_i + P & \text{for } x(W_i) \mod 3 = 0 \\ 2W_i & \text{for } x(W_i) \mod 3 = 1 \\ W_i + Q & \text{for } x(W_i) \mod 3 = 2 \end{cases}$$

Easy to update $a_i$ and $b_i$.

$$\begin{cases} (a_{i+1}, b_i) & \text{for } x(W_i) \mod 3 = 0 \\ (2a_i, 2b_i) & \text{for } x(W_i) \mod 3 = 1 \\ (a_i, b_{i+1}) & \text{for } x(W_i) \mod 3 = 2 \end{cases}$$
Additive walks

Generic rho method requires two scalar multiplications for each iteration. Could replace by double-scalar multiplication; could further merge the 2-scalar multiplications across several parallel iterations.

More efficient: use additive walk:
Start with $W_0 = a_0 P$ and put $f(W_i) = W_i + c_j P + d_j Q$ where $j = h(W_i)$.

Pollard’s initial proposal:
Use $x(W_i) \mod 3$ as $h$ and update:

$$W_{i+1} = \begin{cases} W_i + P & \text{for } x(W_i) \mod 3 = 0 \\ 2W_i & \text{for } x(W_i) \mod 3 = 1 \\ W_i + Q & \text{for } x(W_i) \mod 3 = 2 \end{cases}$$

Easy to update $a_i$ and $b_i$:

$$ (a_{i+1}, b_{i+1}) = \begin{cases} (a_i + 1, b_i) & \text{for } x(W_i) \mod 3 = 0 \\ (2a_i, 2b_i) & \text{for } x(W_i) \mod 3 = 1 \\ (a_i, b_i + 1) & \text{for } x(W_i) \mod 3 = 2 \end{cases} $$
Additive walks

Generic rho method requires two scalar multiplications for each iteration.
Could replace by double-scalar multiplication; could further merge the 2-scalar multiplications across several parallel iterations.

More efficient: use additive walk:
Start with $W_0 = a_0 P$ and put $f(W_i) = W_i + c_j P + d_j Q$ where $j = h(W_i)$.

Pollard’s initial proposal:
Use $x(W_i) \mod 3$ as $h$ and update:

$$W_{i+1} = \begin{cases} 
W_i + P & \text{for } x(W_i) \mod 3 = 0 \\
2W_i & \text{for } x(W_i) \mod 3 = 1 \\
W_i + Q & \text{for } x(W_i) \mod 3 = 2 
\end{cases}$$

Easy to update $a_i$ and $b_i$.

$$(a_{i+1}, b_{i+1}) = \begin{cases} 
(a_i + 1, b_i) & \text{for } x(W_i) \mod 3 = 0 \\
(2a_i, 2b_i) & \text{for } x(W_i) \mod 3 = 1 \\
(a_i, b_i + 1) & \text{for } x(W_i) \mod 3 = 2 
\end{cases}$$
Additive walks

Generic rho method requires two scalar multiplications for each iteration. Could replace by double-scalar multiplication; could further merge the 2-scalar multiplications across several parallel iterations.

More efficient: use additive walk: Start with $W_0 = a_0 P$ and put $f(W_i) = W_i + c_j P + d_j Q$ where $j = h(W_i)$.

Pollard’s initial proposal:
Use $x(W_i) \mod 3$ as $h$ and update:

$$W_{i+1} = \begin{cases} W_i + P & \text{for } x(W_i) \mod 3 = 0 \\ 2W_i & \text{for } x(W_i) \mod 3 = 1 \\ W_i + Q & \text{for } x(W_i) \mod 3 = 2 \end{cases}$$

Easy to update $a_i$ and $b_i$.

$$(a_{i+1}, b_{i+1}) = \begin{cases} (a_i + 1, b_i) & \text{for } x(W_i) \mod 3 = 0 \\ (2a_i, 2b_i) & \text{for } x(W_i) \mod 3 = 1 \\ (a_i, b_{i+1}) & \text{for } x(W_i) \mod 3 = 2 \end{cases}$$
Additive walks

Pollard's initial proposal:
Use \( x(W_i) \mod 3 \) as \( h \) and update:

\[
W_{i+1} = \begin{cases} 
W_i + P & \text{for } x(W_i) \mod 3 = 0 \\
2W_i & \text{for } x(W_i) \mod 3 = 1 \\
W_i + Q & \text{for } x(W_i) \mod 3 = 2 
\end{cases}
\]

Easy to update \( a_i \) and \( b_i \):

\[
(a_{i+1}, b_{i+1}) = \begin{cases} 
(a_i + c_i, b_i) & \text{for } x(W_i) \mod 3 = 0 \\
(2a_i, 2b_i) & \text{for } x(W_i) \mod 3 = 1 \\
(a_i, b_i + d_i) & \text{for } x(W_i) \mod 3 = 2 
\end{cases}
\]

Additive walks

Generic rho method requires two scalar multiplications for each iteration. Could replace by double-scalar multiplication; could further merge the 2-scalar multiplications across several parallel iterations. More efficient: use additive walk:

Start with \( W_0 = a_0 P \) and put \( f(W_i) = W_i + c_j P + d_j Q \) where \( j = h(W_i) \).

Pollard's initial proposal:
Use \( x(W_i) \mod 3 \) as \( h \) and update:

\[
W_{i+1} = \begin{cases} 
W_i + P & \text{for } x(W_i) \mod 3 = 0 \\
2W_i & \text{for } x(W_i) \mod 3 = 1 \\
W_i + Q & \text{for } x(W_i) \mod 3 = 2 
\end{cases}
\]

Easy to update \( a_i \) and \( b_i \):

\[
(a_{i+1}, b_{i+1}) = \begin{cases} 
(a_i + c_i, b_i) & \text{for } x(W_i) \mod 3 = 0 \\
(2a_i, 2b_i) & \text{for } x(W_i) \mod 3 = 1 \\
(a_i, b_i + d_i) & \text{for } x(W_i) \mod 3 = 2 
\end{cases}
\]

Additive walk requires only one addition per iteration.

\( h \) maps from \( \langle P \rangle \) to \( \{0, 1, \ldots, r - 1\} \), and \( R_j = c_j P + d_j Q \) are precomputed for each \( j \in \{0, 1, \ldots, r - 1\} \).

Easy coefficient update:

\( W_i = a_i P + b_i Q \), where \( a_i \) and \( b_i \) are defined recursively as follows:

\[
(a_{i+1}, b_{i+1}) = \begin{cases} 
(a_i + c_i h(W_i), b_i) & \text{for } x(W_i) \mod 3 = 0 \\
(2a_i, 2b_i) & \text{for } x(W_i) \mod 3 = 1 \\
(a_i, b_i + d_i h(W_i)) & \text{for } x(W_i) \mod 3 = 2 
\end{cases}
\]
Additive walks

Generic rho method requires two scalar multiplications for each iteration. Could replace by double-scalar multiplication; could further merge the 2-scalar multiplications across several parallel iterations. More efficient: use additive walk:

Start with $W_0 = a_0 P$ and put $f(W_i) = W_i + c_j P + d_j Q$ where $j = h(W_i)$.

Pollard's initial proposal:

Use $x(W_i) \mod 3$ as $h$ and update:

$$
\begin{align*}
W_{i+1} &= W_i + P \quad \text{for } x(W_i) \mod 3 = 0 \\
2W_{i+1} &= W_i + Q \quad \text{for } x(W_i) \mod 3 = 1 \\
W_{i+1} &= W_i + Q \quad \text{for } x(W_i) \mod 3 = 2
\end{align*}
$$

Easy to update $a_i$ and $b_i$.

$$
\begin{align*}
(a_{i+1}, b_{i+1}) &= (a_i + c_{h(W_i)} P, b_i + d_{h(W_i)} Q) \\
(2a_{i+1}, 2b_{i+1}) &= (a_i + c_{h(W_i)} P, 2b_i + 2d_{h(W_i)} Q) \\
(a_{i+1}, b_{i+1}) &= (a_i, b_i + 1)
\end{align*}
$$

Addition walk requires only one addition per iteration.

$h$ maps from $P$ to $\{0, 1, \ldots, r - 1\}$, precomputed for each $j \in \{0, 1, \ldots, r - 1\}$. 

$R_j = c_j P + d_j Q$ and update:

$$
\begin{align*}
W_{i+1} &= W_i + P \quad \text{for } x(W_i) \mod 3 = 0 \\
2W_{i+1} &= W_i + Q \quad \text{for } x(W_i) \mod 3 = 1 \\
W_{i+1} &= W_i + Q \quad \text{for } x(W_i) \mod 3 = 2
\end{align*}
$$

Easy to update $a_i$ and $b_i$. 

$$
\begin{align*}
(a_{i+1}, b_{i+1}) &= (a_i + c_{h(W_i)} P, b_i + d_{h(W_i)} Q) \\
(2a_{i+1}, 2b_{i+1}) &= (a_i + c_{h(W_i)} P, 2b_i + 2d_{h(W_i)} Q) \\
(a_{i+1}, b_{i+1}) &= (a_i, b_i + 1)
\end{align*}
$$

Addition walk requires only one addition per iteration.
Pollard’s initial proposal:
Use \( x(W_i) \mod 3 \) as \( h \)
and update:
\[
W_{i+1} = \begin{cases} 
W_i + P & \text{for } x(W_i) \mod 3 = 0 \\
2W_i & \text{for } x(W_i) \mod 3 = 1 \\
W_i + Q & \text{for } x(W_i) \mod 3 = 2
\end{cases}
\]
Easy to update \( a_i \) and \( b_i \).
\[
(a_{i+1}, b_{i+1}) = \begin{cases} 
(a_i + 1, b_i) & \text{for } x(W_i) \mod 3 = 0 \\
(2a_i, 2b_i) & \text{for } x(W_i) \mod 3 = 1 \\
(a_i, b_i + 1) & \text{for } x(W_i) \mod 3 = 2
\end{cases}
\]
Additive walk requires only one addition per iteration.
\( h \) maps from \( \langle P \rangle \) to \( \{0, 1, \ldots, r - 1\} \), and \( R_j = c_j P + d_j Q \) are precomputed for each \( j \in \{0, 1, \ldots, r - 1\} \).
Easy coefficient update:
\[
W_i = a_i P + b_i Q,
\]
where \( a_i \) and \( b_i \) are defined recursively as follows:
\[
a_{i+1} = a_i + c_h(W_i) \quad \text{and} \quad b_{i+1} = b_i + d_h(W_i).
\]
Pollard’s initial proposal:
Use $x(W_i) \mod 3$ as $h$
and update:

$$W_{i+1} =
\begin{cases}
  W_i + P & \text{for } x(W_i) \mod 3 = 0 \\
  2W_i & \text{for } x(W_i) \mod 3 = 1 \\
  W_i + Q & \text{for } x(W_i) \mod 3 = 2
\end{cases}$$

Easy to update $a_i$ and $b_i$.

$$(a_{i+1}, b_{i+1}) =
\begin{cases}
  (a_i + 1, b_i) & \text{for } x(W_i) \mod 3 = 0 \\
  (2a_i, 2b_i) & \text{for } x(W_i) \mod 3 = 1 \\
  (a_i, b_i + 1) & \text{for } x(W_i) \mod 3 = 2
\end{cases}$$

Additive walk requires only one addition per iteration.

$h$ maps from $\langle P \rangle$ to $\{0, 1, \ldots, r - 1\}$, and $R_j = c_j P + d_j Q$ are precomputed for each $j \in \{0, 1, \ldots, r - 1\}$.

Easy coefficient update:

$$W_i = a_i P + b_i Q,$$
where $a_i$ and $b_i$ are defined recursively as follows:

$$a_{i+1} = a_i + c_h(W_i)$$
and

$$b_{i+1} = b_i + d_h(W_i).$$
Pollard's initial proposal:
Use $x(W_i) \mod 3$ as $h$ and update:

$$W_{i+1} = \begin{cases} 
W_i + P & \text{for } x(W_i) \mod 3 = 0 \\
2W_i & \text{for } x(W_i) \mod 3 = 1 \\
W_i + Q & \text{for } x(W_i) \mod 3 = 2 
\end{cases}$$

Update $a_i$ and $b_i$.

$$a_{i+1} = a_i + c_h(W_i)$$
$$b_{i+1} = b_i + d_h(W_i).$$

Additive walk requires only one addition per iteration.

$h$ maps from $\langle P \rangle$ to $\{0, 1, \ldots, r - 1\}$, and $R_j = c_j P + d_j Q$ are precomputed for each $j \in \{0, 1, \ldots, r - 1\}$.

Easy coefficient update:

$$W_i = a_i P + b_i Q,$$
where $a_i$ and $b_i$ are defined recursively as follows:

Additive walks have disadvantages:
The walks are noticeably nonrandom; this means they need more iterations than the generic rho method to find a collision.

This effect disappears as $r$ grows, but the precomputed table $R_0, \ldots, R_{r-1}$ does not fit into fast memory. This depends on the platform, e.g. trouble for GPUs.

More trouble with adding walks later.
Pollard's initial proposal:
Use \( x (W_i) \mod 3 \) as \( h \) and update:
\[
W_{i+1} = W_i + P \quad \text{for} \quad x (W_i) \mod 3 = 0
\]
\[
2W_i \quad \text{for} \quad x (W_i) \mod 3 = 1
\]
\[
W_i + Q \quad \text{for} \quad x (W_i) \mod 3 = 2
\]
Additive walk requires only one addition per iteration.

\( h \) maps from \( \langle P \rangle \) to \( \{0, 1, \ldots, r - 1\} \), and
\[
R_j = c_j P + d_j Q
\]
are precomputed for each \( j \in \{0, 1, \ldots, r - 1\} \).

Easy coefficient update:
\[
W_i = a_i P + b_i Q,
\]
where \( a_i \) and \( b_i \) are defined recursively as follows:
\[
a_{i+1} = a_i + c_h(W_i) \quad \text{and} \quad b_{i+1} = b_i + d_h(W_i).
\]

Additive walks have disadvantages:
The walks are noticeably nonrandom; this means they need more iterations than the generic rho method to find a collision.

This effect disappears as \( r \) grows, but then the precomputed table \( R_0, \ldots, R_{r-1} \) does not fit into fast memory. This depends on the platform, e.g. trouble for GPUs.
More trouble with adding walks later.
Pollard's initial proposal:

Use $x \ (W_i) \mod 3$ as $h$ and update:

\[
W_{i+1} = \begin{cases} 
8 \cdot W_i + P & \text{for } x \ (W_i) \mod 3 = 0 \\
2 \cdot W_i & \text{for } x \ (W_i) \mod 3 = 1 \\
W_i + Q & \text{for } x \ (W_i) \mod 3 = 2
\end{cases}
\]

Easy to update $a_i$ and $b_i$.

\[
(a_{i+1}; b_{i+1}) = \begin{cases} 
(a_i + 1; b_i + c_{h(W_i)}) & \text{for } x \ (W_i) \mod 3 = 0 \\
(2a_i; 2b_i) & \text{for } x \ (W_i) \mod 3 = 1 \\
(a_i; b_{i+1} + 1) & \text{for } x \ (W_i) \mod 3 = 2
\end{cases}
\]

Additive walk requires only one addition per iteration.

$h$ maps from $\langle P \rangle$ to $
\{0, 1, \ldots, r - 1\}$, and

\[R_j = c_j P + d_j Q\]
are precomputed for each $j \in \{0, 1, \ldots, r - 1\}$.

Easy coefficient update:

\[W_i = a_i P + b_i Q,\]
where $a_i$ and $b_i$ are defined recursively as follows:

\[a_{i+1} = a_i + c_{h(W_i)} \text{ and } b_{i+1} = b_i + d_{h(W_i)}\]

Additive walks have disadvantages:

The walks are noticeably nonrandom; this means they need more iterations than the generic rho method to find a collision.

This effect disappears as $r$ grows, but then the precomputed table $R_0, \ldots, R_{r-1}$ does not fit into fast memory. This depends on the platform, e.g. trouble for GPUs.

More trouble with adding walks later.
Additive walk requires only one addition per iteration. 

$h$ maps from $\langle P \rangle$ to $
\{0, 1, \ldots, r - 1\}$, and
$R_j = c_j P + d_j Q$ are precomputed for each $j \in \{0, 1, \ldots, r - 1\}$.

Easy coefficient update:
$W_i = a_i P + b_i Q$,
where $a_i$ and $b_i$ are defined recursively as follows:

$a_{i+1} = a_i + c_h(W_i)$ and
$b_{i+1} = b_i + d_h(W_i)$.

Additive walks have disadvantages:
The walks are noticeably nonrandom; this means they need more iterations than the generic rho method to find a collision.

This effect disappears as $r$ grows, but then the precomputed table $R_0, \ldots, R_{r-1}$ does not fit into fast memory. This depends on the platform, e.g. trouble for GPUs.

More trouble with adding walks later.
Additive walk requires only one addition per iteration. $h$ maps from $\langle P \rangle$ to $\{0, 1, \ldots, r-1\}$, and $P + d_j Q$ are precomputed for each $j \in \{0, 1, \ldots, r-1\}$.

Efficient update:
$P + b_i Q$, $a_i$ and $b_i$ are defined recursively as follows:
$a_i + c_h(W_i)$ and $b_i + d_h(W_i)$.

Additive walks have disadvantages:
The walks are noticeably nonrandom; this means they need more iterations than the generic rho method to find a collision.

This effect disappears as $r$ grows, but then the precomputed table $R_0, \ldots, R_{r-1}$ does not fit into fast memory. This depends on the platform, e.g. trouble for GPUs.

More trouble with adding walks later.

Randomness of adding walks
Let $h(W) = i$ with probability $p_i$.
Fix a point $T$, and let $W$ and $W'$ be two independent uniform random points.
Let $W \neq W'$ both map to $T$. This event occurs if...
Additive walk requires only one addition per iteration. h maps from $\langle P \rangle$ to \{0; 1; ...; r − 1\}. $R_j = c_j P + d_j Q$ are precomputed for each $j \in \{0; 1; ...; r − 1\}$.

Easy coefficient update: $W_i = a_i P + b_i Q$, where $a_i$ and $b_i$ are defined recursively as follows:

$$a_{i+1} = a_i + c_{h(W_i)}$$
$$b_{i+1} = b_i + d_{h(W_i)}$$

Additive walks have disadvantages: The walks are noticeably nonrandom; this means they need more iterations than the generic rho method to find a collision.

This effect disappears as r grows, but then the precomputed table $R_0, \ldots, R_{r-1}$ does not fit into fast memory. This depends on the platform, e.g. trouble for GPUs.

More trouble with adding walks later.

Randomness of adding walks
Let $h(W) = i$ with probability $p_i$.
Fix a point $T$, and let $W$ and $W'$ be two independent uniform random points.
Let $W \neq W'$ both map to $T$. This event occurs if

\[ \text{Randomness of adding walks} \]

Let $h(W) = i$ with probability $p_i$.
Fix a point $T$, and let $W$ and $W'$ be two independent uniform random points.
Let $W \neq W'$ both map to $T$.
This event occurs if
Additive walks have disadvantages:
The walks are noticeably nonrandom; this means they need more iterations than the generic rho method to find a collision.

This effect disappears as \( r \) grows, but then the precomputed table \( R_0, \ldots, R_{r-1} \) does not fit into fast memory. This depends on the platform, e.g. trouble for GPUs.

More trouble with adding walks later.

Randomness of adding walks

Let \( h(W) = i \) with probability \( p_i \).

Fix a point \( T \), and let \( W \) and \( W' \) be two independent uniform random points.

Let \( W \neq W' \) both map to \( T \).

This event occurs if
Additive walks have disadvantages: The walks are noticeably nonrandom; this means they need more iterations than the generic rho method to find a collision. This effect disappears as $r$ grows, but then the precomputed table $R_0, \ldots, R_{r-1}$ does not fit into fast memory. This depends on the platform, e.g. trouble for GPUs. More trouble with adding walks later.

Randomness of adding walks

Let $h(W) = i$ with probability $p_i$. Fix a point $T$, and let $W$ and $W'$ be two independent uniform random points. Let $W \neq W'$ both map to $T$. This event occurs if
Additive walks have disadvantages:
The walks are noticeably nonrandom; this means they need more iterations than the generic rho method to find a collision.

This effect disappears as \( r \) grows, but then the precomputed table \( R_0, \ldots, R_{r-1} \) does not fit into fast memory. This depends on the platform, e.g. trouble for GPUs.

More trouble with adding walks later.

Randomness of adding walks
Let \( h(W) = i \) with probability \( p_i \).

Fix a point \( T \), and let \( W \) and \( W' \) be two independent uniform random points.

Let \( W \neq W' \) both map to \( T \).

This event occurs if simultaneously for \( i \neq j \):

\[
T = W + R_i = W' + R_j;
\]

\[
h(W) = i; \ h(W') = j.
\]

These conditions have probability \( 1/\ell^2, p_i, \) and \( p_j \) respectively.
Additive walks have disadvantages:

- The walks are noticeably nonrandom; this means they need more iterations than the generic $\rho$ method to find a collision.
- This effect disappears as $r$ grows, but then the precomputed table $R_0, \ldots, R_{r-1}$ does not fit into fast memory. This depends on the platform, e.g. trouble for GPUs.
- More trouble with adding walks later.

Randomness of adding walks

Let $h(W) = i$ with probability $p_i$.

Fix a point $T$, and let $W$ and $W'$ be two independent uniform random points.

Let $W \neq W'$ both map to $T$.

This event occurs if simultaneously for $i \neq j$:

$$T = W + R_i = W' + R_j;$$

$$h(W) = i; h(W') = j.$$ 

These conditions have probability $1/\ell^2$, $p_i$, and $p_j$ respectively.

Summing over all $(i; j)$ gives the overall probability $1 = \frac{1}{\ell^2} (1 - \sum_i p_i^2)$.

This means that the probability of an immediate collision from $W$ and $W'$ is $1/\ell^2$, where we added over the choices of $T$.

In the simple case that all the $p_i$ are $1/r$, the difference from the optimal $p_i' = 2$ iterations is a factor of $1/\sqrt{1 - 1/r} \approx 1 + 1/(2r)$. 
Additive walks have disadvantages: The walks are noticeably nonrandom; this means they need more iterations than the generic rho method to find a collision. This effect disappears as \( r \) grows, but then the precomputed table \( R_0; \ldots; R_{r-1} \) does not fit into fast memory. This depends on the platform, e.g. trouble for GPUs.

More trouble with adding walks later.

Randomness of adding walks

Let \( h(W) = i \) with probability \( p_i \).

Fix a point \( T \), and let \( W \) and \( W' \) be two independent uniform random points. Let \( W \neq W' \) both map to \( T \).

This event occurs if simultaneously for \( i \neq j \):

\[
T = W + R_i = W' + R_j;
\]

\[
h(W) = i; h(W') = j.
\]

These conditions have probability \( p_i \) and \( p_j \) respectively.

Summing over all \((i; j)\) gives the overall probability

\[
\begin{align*}
\sum_{i \neq j} p_i p_j & = \sum_{i \neq j} \left( \frac{1}{r^2} \right) \\
& = \frac{1}{r^2} \left( r^2 - 1 \right) \\
& = 1 - \frac{1}{r}.
\end{align*}
\]

This means that the optimal \( \sqrt{\frac{\pi}{2}} \) it becomes \( \frac{1}{r} \), the different \( \frac{1}{\sqrt{1/r}} \sim 1 - \frac{1}{r} \), the difference.

In the simple case we added over the of an immediate collision it gives the overall probability

\[
\sum_{i \neq j} \left( \frac{1}{r^2} \right) = \frac{1}{r^2} \left( r^2 - 1 \right) = 1 - \frac{1}{r}.
\]

This gives the overall probability

\[
\frac{1}{r^2} \left( r^2 - 1 \right) = 1 - \frac{1}{r}.
\]
Randomness of adding walks

Let \( h(W) = i \) with probability \( p_i \).

Fix a point \( T \), and let \( W \) and \( W' \) be two independent uniform random points.

Let \( W \neq W' \) both map to \( T \).

This event occurs if simultaneously for \( i \neq j \):
\[
T = W + R_i = W' + R_j;
\]
\[
h(W) = i; \ h(W') = j.
\]

These conditions have probability \( 1/\ell^2 \), \( p_i \), and \( p_j \) respectively.

\[
\sum_{i \neq j} p_i p_j = \frac{1}{\ell^2}.
\]

This means that the probability of an immediate collision from \( W \) and \( W' \) is \( (1 - \sum_i p_i^2) / \ell \), where we added over the \( \ell \) choices for \( T \).

In the simple case that all the \( p_i \) are \( 1/r \), the difference from the optimal \( \sqrt{\pi \ell/2} \) iterations is a factor of \( 1/\sqrt{1 - 1/r} \approx 1 + 1/(2r) \).
Randomness of adding walks

Let $h(W) = i$ with probability $p_i$. Fix a point $T$, and let $W$ and $W'$ be two independent uniform random points.

Let $W \neq W'$ both map to $T$. This event occurs if simultaneously for $i \neq j$: $T = W + R_i = W' + R_j$; $h(W) = i; h(W') = j$.

These conditions have probability $1/\ell^2$, $p_i$, and $p_j$ respectively.

Summing over all $(i, j)$ gives the overall probability:

$$
\left( \sum_{i \neq j} p_i p_j \right) / \ell^2 = 
\left( \sum_{i,j} p_i p_j - \sum_i p_i^2 \right) / \ell^2 = 
(1 - \sum_i p_i^2) / \ell^2.
$$

This means that the probability of an immediate collision from $W$ and $W'$ is $(1 - \sum_i p_i^2) / \ell$, where we added over the $\ell$ choices of $T$. In the simple case that all the $p_i$ are $1/r$, the difference from the optimal $\sqrt{\pi \ell/2}$ iterations is a factor of

$$
1/\sqrt{1 - 1/r} \approx 1 + 1/(2r).
$$
Randomness of adding walks

Let $h(W) = i$ with probability $p_i$. Fix a point $T$, and let $W$ and $W'$ be two independent uniform random points.

If $W \neq W'$ both map to $T$. This event occurs if simultaneously for $i \neq j$:

$$T = W + R_i = W' + R_j; \quad i; h(W') = j.$$ 

These conditions have probability $p_i$ and $p_j$ respectively.

Summing over all $(i,j)$ gives the overall probability

$$\left( \sum_{i \neq j} p_i p_j \right) / \ell^2 = \left( \sum_{i,j} p_i p_j - \sum_i p_i^2 \right) / \ell^2 = \left( 1 - \sum_i p_i^2 \right) / \ell^2.$$ 

This means that the probability of an immediate collision from $W$ and $W'$ is \(1 - \sum_i p_i^2\) / \(\ell\), where we added over the \(\ell\) choices of $T$.

In the simple case that all the $p_i$ are $1/r$, the difference from the optimal $\sqrt{\pi \ell / 2}$ iterations is a factor of

$$1 / \sqrt{1 - 1/r} \approx 1 + 1 / (2r).$$ 

Various heuristics leading to standard $p_1 - 1/r$ formula in different ways:

1981 Brent–Pollard;
2001 Teske;
Randomness of adding walks

Let \( h(W) = i \) with probability \( p_i \).

Fix a point \( T \), and let \( W \) and \( W' \) be two independent uniform random points.

Let \( W \neq W' \) both map to \( T \).

This event occurs if simultaneously for \( i \neq j \):

\[
T = W + R_i = W' + R_j;
\]

\[
h(W) = i; h(W') = j.
\]

These conditions have probability \( \frac{1}{2}, p_i, \) and \( p_j \) respectively.

Summing over all \((i, j)\) gives the overall probability

\[
\left( \sum_{i \neq j} p_i p_j \right) / l^2 = \left( \sum_{i, j} p_i p_j - \sum_i p_i^2 \right) / l^2 = (1 - \sum_i p_i^2) / l^2.
\]

This means that the probability of an immediate collision from \( W \) and \( W' \) is \((1 - \sum_i p_i^2) / l\), where we added over the \( l \) choices of \( T \).

In the simple case that all the \( p_i \) are \( 1/r \), the difference from the optimal \( \sqrt{\pi l / 2} \) iterations is a factor of

\[
1 / \sqrt{1 - 1/r} \approx 1 + 1 / (2r).
\]

Various heuristics leading to standard \( \sqrt{1 - 1/r} \) formula in different ways:

1981 Brent–Pollard;
2001 Teske;
Let $h(W) = i$ with probability $p_i$. Fix a point $T$, and let $W$ and $W'$ be two independent uniform random points. Let $W \neq W'$ both map to $T$. This event occurs if simultaneously for $i \neq j$:

$$T = W + R_i = W' + R_j; h(W) = i; h(W') = j.$$  

These conditions have probability $1/l^2 = p_i p_j$ respectively. Summing over all $(i, j)$ gives the overall probability

$$P_{i \neq j} p_i p_j / l^2 = \left( \sum_{i,j} p_i p_j - \sum_i p_i^2 \right) / l^2 = \left( 1 - \sum_i p_i^2 \right) / l^2.$$  

This means that the probability of an immediate collision from $W$ and $W'$ is $(1 - \sum_i p_i^2) / l$, where we added over the $l$ choices of $T$. In the simple case that all the $p_i$ are $1/r$, the difference from the optimal $\sqrt{\pi l/2}$ iterations is a factor of

$$1/\sqrt{1 - 1/r} \approx 1 + 1/(2r).$$

Various heuristics leading to the standard $\sqrt{1 - 1/r}$ formula in different ways: 1981 Brent–Pollard; 2001 Teske; 2009 ECC2K-130 paper, eprint 2009/541.
Summing over all \((i,j)\) gives the overall probability 
\[
\frac{\left(\sum_{i \neq j} p_i p_j\right)}{l^2} = \frac{\left(\sum_{i,j} p_i p_j - \sum_i p_i^2\right)}{l^2} = \frac{1 - \sum_i p_i^2}{l^2}.
\]

This means that the probability of an immediate collision from \(W\) and \(W'\) is \(\frac{1 - \sum_i p_i^2}{l}\), where we added over the \(l\) choices of \(T\).

In the simple case that all the \(p_i\) are \(1/r\), the difference from the optimal \(\sqrt{\pi l/2}\) iterations is a factor of 
\[
\frac{1}{\sqrt{1 - 1/r}} \approx 1 + 1/(2r).
\]

Various heuristics leading to standard \(\sqrt{1 - 1/r}\) formula in different ways:
1981 Brent–Pollard;
2001 Teske;
Summing over all \((i,j)\) gives the overall probability 
\[
\frac{\sum_{i \neq j} p_i p_j}{\ell^2} = \frac{\sum_{i,j} p_i p_j - \sum_i p_i^2}{\ell^2} = \frac{1 - \sum_i p_i^2}{\ell^2}.
\]

This means that the probability of an immediate collision from \(W\) and \(W'\) is \((1 - \sum_i p_i^2) / \ell\), where we added over the \(\ell\) choices of \(T\). In the simple case that all the \(p_i\) are \(1/r\), the difference from the optimal \(\sqrt{\pi \ell/2}\) iterations is a factor of 
\[
\frac{1}{\sqrt{1 - 1/r}} \approx 1 + 1/(2r).
\]

Various heuristics leading to standard \(\sqrt{1 - 1/r}\) formula in different ways: 
1981 Brent–Pollard;
2001 Teske;

2010 Bernstein–Lange: Standard formula is wrong! There is a further slowdown from higher-order anti-collisions: e.g. \(W + R_i + R_k \neq W' + R_j + R_l\) if \(R_i + R_k = R_j + R_l\).
\[
\approx 1\% \text{ slowdown for ECC2K-130.}
\]
Summing over all \((i, j)\) gives the overall probability
\[
p_i p_j \left/ l^2 \right. = \left( p_i p_j - \sum_i p_i^2 \right) / l^2 = (p_i^2) / l^2.
\]
This means that the probability of an immediate collision from \(W\) is
\[
(1 - \sum_i p_i^2) / l,
\]
where we summed over the \(l\) choices of \(T\).

In the simple case that all the \(p_i\) are equal, the difference from the
standard \(\sqrt{1 - 1/r}\) formula in different ways:

- 1981 Brent–Pollard;
- 2001 Teske;

2010 Bernstein–Lange:
Standard formula is wrong!
There is a further slowdown from higher-order anti-collisions:
e.g. \(W + R_i + R_k \neq W' + R_j + R_l\)
if \(R_i + R_k = R_j + R_l\).
\[
\approx 1\% \text{ slowdown for ECC2K-130.}
\]

Eliminating storage
Usual description: each walk keeps track of \(a_i, b_i\) with \(W_i = a_i P + b_i Q\).
This requires each client to implement arithmetic modulo \(\ell\)
or at least keep track of how often each \(R_j\) is used.

For distinguished points these values are transmitted to the server (bandwidth)
which stores them as e.g. \((W_i, a_i, b_i)\) (space).
Summing over all \((i,j)\)
gives the overall probability

\[
P_i \neq j p_i p_j = \frac{(p_i^2)}{l^2} = \frac{1}{2r}
\]

This means that the probability of an immediate collision from \(W\) and \(W'\) is

\[
1 - \frac{1}{2r} = 2^{-2r}
\]

Various heuristics leading to standard \(\sqrt{1 - 1/r}\) formula in different ways:
1981 Brent–Pollard;
2001 Teske;
2010 Bernstein–Lange:
Standard formula is wrong!

There is a further slowdown from higher-order anti-collisions:
e.g. \(W + R_i + R_k \neq W' + R_j + R_l\)
if \(R_i + R_k = R_j + R_l\).
\(\approx 1\%\) slowdown for ECC2K-130.

Eliminating storage
Usual description: each walk keeps track of \(a_i\) and \(b_i\) with \(W_i = a_iP + b_iQ\).

This requires each client to implement arithmetics or at least keep track of how often each \(R_j\) is used.

For distinguished points these values are transmitted to server (bandwidth) which stores them as e.g. \((W_i, a_i, b_i)\) (space).
Various heuristics leading to standard $\sqrt{1/\epsilon}$ formula in different ways: 1981 Brent–Pollard; 2001 Teske; 2009 ECC2K-130 paper, eprint 2009/541.

2010 Bernstein–Lange: Standard formula is wrong!

There is a further slowdown from higher-order anti-collisions: e.g. $W + R_i + R_k \neq W' + R_j + R_l$ if $R_i + R_k = R_j + R_l$.

$\approx 1\%$ slowdown for ECC2K-130.

In the simple case that all the $p_i$ are $1/\epsilon$, the difference from the optimal $p_i' = 2$ iterations is a factor of $1/\epsilon - 1/\epsilon \approx 1 + 1/(2\epsilon)$.

Eliminating storage
Usual description: each walk keeps track of $a_i$ and $b_i$ with $W_i = a_iP + b_iQ$.

This requires each client to implement arithmetic modulo $\ell$ which stores them as transmitted to server (bandwidth)

For distinguished points these values are

For each point $W_i$, the server stores the values

The client keeps track of $a_i$ and $b_i$ with $W_i = a_iP + b_iQ$. For each client, this requires the client to implement arithmetic modulo $\ell$ with $W_i = a_iP + b_iQ$. For each point $W_i$, the server stores the values transmitted to server (bandwidth).
Various heuristics leading to standard $\sqrt{1 - 1/r}$ formula in different ways:
1981 Brent–Pollard;
2001 Teske;
2010 Bernstein–Lange: Standard formula is wrong!
There is a further slowdown from higher-order anti-collisions:
e.g. $W + R_i + R_k \neq W' + R_j + R_l$
if $R_i + R_k = R_j + R_l$.
$\approx 1\%$ slowdown for ECC2K-130.

Eliminating storage
Usual description: each walk keeps track of $a_i$ and $b_i$ with $W_i = a_iP + b_iQ$.
This requires each client to implement arithmetic modulo $\ell$
or at least keep track of how often each $R_j$ is used.
For distinguished points these values are transmitted to server (bandwidth) which stores them as
e.g. $(W_i, a_i, b_i)$ (space).
Various heuristics leading to
the \(1 - \frac{1}{r}\) formula
differ in different ways:
- Brent–Pollard,
- Teske,
- ECC2K-130 paper, eprint 2009/541.
- Bernstein–Lange: Standard formula is wrong!

There is a further slowdown
from higher-order anti-collisions:
\[ W + R_i + R_k \neq W' + R_j + R_l \]
\[ R_k = R_j + R_l. \]

\( \approx 1\% \) slowdown for ECC2K-130.

Eliminating storage

Usual description: each walk keeps track of \( a_i \) and \( b_i \)
with \( W_i = a_iP + b_iQ \).

This requires each client to implement arithmetic modulo \( \ell \)
or at least keep track of how often each \( R_j \) is used.

For distinguished points these values are transmitted to server (bandwidth)
which stores them as e.g. \((W_i, a_i, b_i)\) (space).

2009 ECC2K-130 paper: Remember where you started.
If \( W_i = W_j \) is the collision of distinguished points, can recompute these walks with \( a_i, b_i, a_j, b_j \); walk is deterministic!
Server stores \( 2^{45} \) distinguished points; only needs to know coefficients for 2 of them.

Our setup: Each walk remembers seed; server stores distinguished point and seed.
Saves time, bandwidth, space.
Various heuristics leading to standard $p^1 - 1 = r$ formula in different ways:

- 1981 Brent–Pollard;
- 2001 Teske;

2010 Bernstein–Lange: Standard formula is wrong!

There is a further slowdown from higher-order anti-collisions:

\[ W + R_i + R_k \neq W' + R_j + R_l \]

if \( R_i + R_k = R_j + R_l \).

\[ \approx 1\% \text{ slowdown for ECC2K-130.} \]

**Eliminating storage**

Usual description: each walk keeps track of \( a_i \) and \( b_i \) with \( W_i = a_i P + b_i Q \).

This requires each client to implement arithmetic modulo \( l \) or at least keep track of how often each \( R_j \) is used.

For distinguished points, these values are transmitted to server (bandwidth) which stores them as e.g. \((W_i, a_i, b_i)\) (space).

2009 ECC2K-130 paper: Remember where you started.

If \( W_i = W_j \) is the collision of distinguished points, can recompute these walks with \( a_i, b_i, a_j, \) and \( b_j \); walk is deterministic!

Server stores \( 2^{45} \) distinguished points; only needs to know coefficients for 2 of them.

Our setup: Each walk remembers seed; server stores distinguished point and seed.

Saves time, bandwidth.
Various heuristics leading to standard $p^1 - 1 = r$ formula in different ways:
1981 Brent–Pollard;
2001 Teske;
2009 ECC2K-130 paper,
eprint 2009/541.
2010 Bernstein–Lange:
Standard formula is wrong!
There is a further slowdown from higher-order anti-collisions:
E.g. $w_i + r_j + r_k \neq w'_i + r_j + r_l$ if $r_i + r_k = r_j + r_l$.
$\approx 1\%$ slowdown for ECC2K-130.

Eliminating storage
Usual description: each walk keeps track of $a_i$ and $b_i$ with $W_i = a_i P + b_i Q$.
This requires each client to implement arithmetic modulo $\ell$ or at least keep track of
how often each $R_j$ is used.
For distinguished points these values are transmitted to server (bandwidth)
which stores them as $e.g. (W_i; a_i; b_i)$ (space).

2009 ECC2K-130 paper:
Remember where you started.
If $W_i = W_j$ is the collision of distinguished points,
can recompute these walks with $a_i; b_i; a_j$, and $b_j$;
the walk is deterministic!
Server stores $2^{45}$ distinguished points; only needs to know
coefficients for 2 of them.

Our setup: Each walk remembers seed; server stores distinguished point and seed.
Saves time, bandwidth, space.

Usual description: each walk keeps track of $a_i$ and $b_i$ with $W_i = a_i P + b_i Q$.
This requires each client to keep track of $e.g. (W_i; a_i; b_i)$ (space).

\[ w_i + p' \neq p' \]

With $W_i = a'_i P + b'_i Q$ keeps track of $a'_i$ and $b'_i$.
Eliminating storage

Usual description: each walk keeps track of \(a_i\) and \(b_i\) with \(W_i = a_i P + b_i Q\).

This requires each client to implement arithmetic modulo \(\ell\) or at least keep track of how often each \(R_j\) is used.

For distinguished points these values are transmitted to server (bandwidth) which stores them as e.g. \((W_i, a_i, b_i)\) (space).

2009 ECC2K-130 paper:
Remember where you started. If \(W_i = W_j\) is the collision of distinguished points, can recompute these walks with \(a_i, b_i, a_j,\) and \(b_j\); walk is deterministic!

Server stores \(2^{45}\) distinguished points; only needs to know coefficients for 2 of them.

Our setup: Each walk remembers seed; server stores distinguished point and seed.
Saves time, bandwidth, space.
Eliminating storage

Usual description: each walk keeps track of $a_i$ and $b_i$ with $W_i = a_iP + b_iQ$.

This requires each client to implement arithmetic modulo $\ell$ and at least keep track of when each $R_j$ is used.

Distinguished points values are transmitted to server (bandwidth) which stores them as $(W_i, a_i, b_i)$ (space).

2009 ECC2K-130 paper:
Remember where you started.
If $W_i = W_j$ is the collision of distinguished points, can recompute these walks with $a_i, b_i, a_j,$ and $b_j$; walk is deterministic!
Server stores $2^{45}$ distinguished points; only needs to know coefficients for 2 of them.

Our setup: Each walk remembers seed; server stores distinguished point and seed.
Saves time, bandwidth, space.

Negation and rho

$W = (x, y)$ and $-W = (x, -y)$ have same $x$-coordinate.
Search for $x$-coordinate collision.
Search space for collisions is only $\lceil \sqrt[2]{2} \rceil$; this gives factor $\sqrt{2}$ speedup.

To ensure $f(W_i) = f(-W_i)$
Define $j = h(|W_i|)$ and $f(W_i) = |W_i| + c_jP + d_jQ$, with, e.g., $|W_i|$ the lexicographic minimum of $W_i, -W_i$.

This negation speedup is textbook material.
Each walk keeps track of $a_i$ and $b_i$ with $W_i = a_i P + b_i Q$.

This requires each client to implement arithmetic modulo $l$ or at least keep track of how often each $R_j$ is used.

For distinguished points these values are transmitted to server (bandwidth) which stores them as e.g. $(W_i; a_i; b_i)$ (space).

2009 ECC2K-130 paper:
Remember where you started. If $W_i = W_j$ is the collision of distinguished points, can recompute these walks with $a_i, b_i, a_j$, and $b_j$; walk is deterministic!

Server stores $2^{45}$ distinguished points; only needs to know coefficients for 2 of them.

Our setup: Each walk remembers seed; server stores distinguished point and seed.

Saves time, bandwidth, space.

Negation and rho

$W = (x, y)$ and $-W = (x; -y)$ have same $x$-coordinate.

Search for $x$-coordinate collision.

Search space for collisions is only $\lceil \ell/2 \rceil$; this gives speedup ... if $f(W_i) = f(-W_i)$.

To ensure $f(W_i) = f(-W_i)$
Define $j = h(|W_i|)$ with, e.g., $|W_i|$ the lexicographic minimum of $W_i, -W_i$.

This negation speedup is textbook material.
2009 ECC2K-130 paper:
Remember where you started.
If $W_i = W_j$ is the collision of distinguished points,
can recompute these walks
with $a_i, b_i, a_j, \text{ and } b_j$;
walk is deterministic!
Server stores $2^{45}$ distinguished points; only needs to know coefficients for 2 of them.

Our setup: Each walk remembers seed; server stores distinguished point and seed.
Saves time, bandwidth, space.

Negation and rho
$W = (x, y)$ and $-W = (x, -y)$ have same $x$-coordinate.
Search for $x$-coordinate collision.

Search space for collisions is only $[\ell/2]$; this gives factor speedup ... if $f(W_i) = f(-W_i)$.

To ensure $f(W_i) = f(-W_i)$:
Define $j = h(|W_i|)$ and $f(W_i) = |W_i| + c_j P + d_j Q$,
with, e.g., $|W_i|$ the lexicographic minimum of $W_i, -W_i$.
This negation speedup is textbook material.
2009 ECC2K-130 paper:
Remember where you started.
If $W_i = W_j$ is the collision of distinguished points, can recompute these walks with $a_i, b_i, a_j, \text{ and } b_j$; walk is deterministic!
Server stores $2^{45}$ distinguished points; only needs to know coefficients for 2 of them.
Our setup: Each walk remembers seed; server stores distinguished point and seed.
Saves time, bandwidth, space.

Negation and rho

$W = (x, y)$ and $-W = (x, -y)$ have same $x$-coordinate.
Search for $x$-coordinate collision.
Search space for collisions is only $[l/2]$; this gives factor $\sqrt{2}$ speedup . . . if $f(W_i) = f(-W_i)$.
To ensure $f(W_i) = f(-W_i)$:
Define $j = h(|W_i|)$ and $f(W_i) = |W_i| + cj P + dj Q$,
with, e.g., $|W_i|$ the lexicographic minimum of $W_i, -W_i$.
This negation speedup is textbook material.
EC2K-130 paper:
Remember where you started.
If \( W_i = W_j \) is the collision of distinguished points, can recompute these walks with \( a_i; b_i; a_j; b_j \); walk is deterministic!
Server stores \( 2^{45} \) distinguished points; only needs to know coefficients for 2 of them.
Our setup: Each walk remembers seed; server stores distinguished point and seed.
Saves time, bandwidth, space.

Negation and rho
\[
W = (x, y) \text{ and } -W = (x, -y)
\]
have same \( x \)-coordinate.
Search for \( x \)-coordinate collision.
Search space for collisions is only \( \lceil \ell/2 \rceil \); this gives factor \( \sqrt{2} \) speedup . . . if \( f(W_i) = f(-W_i) \).

To ensure \( f(W_i) = f(-W_i) \):
Define \( j = h(|W_i|) \) and
\[
f(W_i) = |W_i| + c_j P + d_j Q,
\]
with, e.g., \( |W_i| \) the lexicographic minimum of \( W_i, -W_i \).
This negation speedup is textbook material.

Problem: this walk can run into fruitless cycles!
Example: If \( |W_{i+1}| = -W_{i+1} \) and \( h(|W_{i+1}|) = j = h(|W_i|) \),
then \( W_{i+2} = f(W_{i+1}) = -W_{i+1} + c_j P + d_j Q = -(|W_i| + c_j P + d_j Q) + c_j P + d_j Q = -|W_i| \), so \( |W_{i+3}| = |W_i| \), so \( W_{i+3} = W_i \),
so \( W_{i+4} = W_{i+2} \), etc.

If \( h \) maps to \( r \) different values then expect this example to occur with probability \( 1 = \left(\frac{1}{2}\right)^r \) at each step.
Known issue, not quite textbook.
2009 ECC2K-130 paper:
Remember where you started.
If \( W_i = W_j \) is the collision of distinguished points, can recompute these walks with \( a_i; b_i; a_j, \) and \( b_j; \) the walk is deterministic!
Server stores \( 2^{45} \) distinguished points; only needs to know coefficients for 2 of them.
Our setup: Each walk remembers seed; server stores distinguished point and seed. Saves time, bandwidth, space.

**Negation and rho**

\[ W = (x, y) \text{ and } -W = (x, -y) \]

have same \( x \)-coordinate. Search for \( x \)-coordinate collision.

Search space for collisions is only \( \lceil \ell/2 \rceil \); this gives factor \( \sqrt{2} \) speedup . . . if \( f(W_i) = f(-W_i) \).

To ensure \( f(W_i) = f(-W_i) \):
Define \( j = h(\|W_i\|) \) and
\[ f(W_i) = \|W_i\| + c_j P + d_j Q, \]
with, e.g., \( \|W_i\| \) the lexicographic minimum of \( W_i, -W_i \).

This negation speedup is textbook material.

**Problem:** this walk can run into fruitless cycles!

Example: If \( \|W_i+1\| = -W_i+1 \) and \( h(\|W_i+1\|) = j = h(\|W_i\|) \) then \( W_{i+2} = f(W_i+1) = -W_i+1 + c_j P + d_j Q = -\|W_i\| + c_j P + d_j Q = -\|W_{i+1}\| \)

so \( W_{i+3} = W_{i+1} \)
so \( W_{i+4} = W_{i+2} \) etc.

If \( h \) maps to \( r \) different values then expect this example to occur with probability \( 1/(2^r) \) at each step.
Known issue, not quite textbook.
Negation and rho

$W = (x, y)$ and $-W = (x, -y)$ have the same $x$-coordinate. Search for $x$-coordinate collision.

Search space for collisions is only $\lceil \ell/2 \rceil$; this gives a factor $\sqrt{2}$ speedup . . . if $f(W_i) = f(-W_i)$.

To ensure $f(W_i) = f(-W_i)$:

Define $j = h(|W_i|)$ and

$f(W_i) = |W_i| + c_j P + d_j Q$,

with, e.g., $|W_i|$ the lexicographic minimum of $W_i, -W_i$.

This negation speedup is textbook material.

Problem: this walk can run into fruitless cycles!

Example: If $|W_{i+1}| = -W_{i+1}$ and $h(|W_{i+1}|) = j = h(|W_i|)$, then $W_{i+2} = f(W_{i+1}) = -W_{i+1} + c_j P + d_j Q = -(|W_i| + c_j P + d_j Q) + c_j P + d_j Q = -|W_i|$ so $|W_{i+2}| = |W_i|$ so $W_{i+3} = W_{i+1}$ so $W_{i+4} = W_{i+2}$ etc.

If $h$ maps to $r$ different values then expect this example to occur with probability $1/(2r)$ at each step.

Known issue, not quite textbook.
Negation and rho

$W = (x, y)$ and $-W = (x, -y)$ have same $x$-coordinate.
Search for $x$-coordinate collision.

Search space for collisions is only $\lceil \ell/2 \rceil$; this gives factor $\sqrt{2}$ speedup ... if $f(W_i) = f(-W_i)$.

To ensure $f(W_i) = f(-W_i)$:
Define $j = h(|W_i|)$ and

\[ f(W_i) = |W_i| + c_j P + d_j Q, \]

with, e.g., $|W_i|$ the lexicographic minimum of $W_i, -W_i$.

This negation speedup is textbook material.

Problem: this walk can run into fruitless cycles!
Example: If $|W_{i+1}| = -W_{i+1}$ and $h(|W_{i+1}|) = j = h(|W_i|)$ then $W_{i+2} = f(W_{i+1}) = -W_{i+1} + c_j P + d_j Q = -(|W_i| + c_j P + d_j Q) + c_j P + d_j Q = -|W_i|$ so $|W_{i+2}| = |W_i|$ so $W_{i+3} = W_{i+1}$ so $W_{i+4} = W_{i+2}$ etc.

If $h$ maps to $r$ different values then expect this example to occur with probability $1/(2r)$ at each step.

Known issue, not quite textbook.
Problem: this walk can run into fruitless cycles!
Example: If $|W_{i+1}| = -W_{i+1}$ and $h(|W_{i+1}|) = j = h(|W_i|)$
then $W_{i+2} = f(W_{i+1}) = -W_{i+1} + c_j P + d_j Q = -(|W_i| + c_j P + d_j Q) + c_j P + d_j Q = -|W_i|$, so $|W_{i+2}| = |W_i|$
so $W_{i+3} = W_{i+1}$
so $W_{i+4} = W_{i+2}$ etc.

If $h$ maps to $r$ different values then expect this example to occur with probability $1/(2r)$
at each step.
Known issue, not quite textbook.

Eliminating fruitless cycles
Issue of fruitless cycles is known and several fixes are proposed.
See appendix of full version ePrint 2011/003 for even more details and historical comments.
Summary: most of them got it wrong.
Negation and \( W = (x, y) \) have same \( x \)-coordinate.

Search for \( x \)-coordinate collision.

Search space for collisions is \( \lceil 2 \rceil \); this gives factor \( \sqrt{2} \) speedup.

If \( f(W_i) = f(-W_i) \):

\[
W_{i+2} = f(W_{i+1}) = -W_{i+1} + c_j P + d_j Q = -(|W_i| + c_j P + d_j Q) + c_j P + d_j Q = -|W_i| \text{ so } |W_{i+2}| = |W_i| \text{ so } W_{i+3} = W_{i+1} \text{ so } W_{i+4} = W_{i+2} \text{ etc.}
\]

If \( h \) maps to \( r \) different values then expect this example to occur with probability \( 1/(2r) \) at each step.

Known issue, not quite textbook.

Problem: this walk can run into fruitless cycles!

Example: If \( |W_{i+1}| = -W_{i+1} \) and \( h(|W_{i+1}|) = j = h(|W_i|) \) then \( W_{i+2} = f(W_{i+1}) = -W_{i+1} + c_j P + d_j Q = -(|W_i| + c_j P + d_j Q) + c_j P + d_j Q = -|W_i| \text{ so } W_{i+2} = W_i \text{ etc.}
\]

Eliminating fruitless cycles

Issue of fruitless cycles is known and several fixes are proposed. See appendix of full version ePrint 2011/003 for even more details and historical comments.

Summary: most of them got it wrong.
Problem: this walk can run into fruitless cycles!

Example: If $|W_{i+1}| = -W_{i+1}$ and $h(|W_{i+1}|) = j = h(|W_i|)$ then $W_{i+2} = f(W_{i+1}) = -W_{i+1} + c_j P + d_j Q = -(|W_i| + c_j P + d_j Q) + c_j P + d_j Q = -|W_i|$ so $|W_{i+2}| = |W_i|$ so $W_{i+3} = W_{i+1}$ so $W_{i+4} = W_{i+2}$ etc.

If $h$ maps to $r$ different values then expect this example to occur with probability $1/(2r)$ at each step.

Known issue, not quite textbook.

Eliminating fruitless cycles

Issue of fruitless cycles is known and several fixes are proposed. See appendix of full version ePrint 2011/003 for even more details and historical comments.

Summary: most of them got it wrong.
Problem: this walk can run into fruitless cycles!
Example: If $|W_{i+1}| = -W_{i+1}$ and $h(|W_{i+1}|) = j = h(|W_i|)$ then $W_{i+2} = f(W_{i+1}) = -W_{i+1} + c_j P + d_j Q = -(|W_i| + c_j P + d_j Q) + c_j P + d_j Q = -|W_i|$ so $|W_{i+2}| = |W_i|$ so $W_{i+3} = W_{i+1}$
so $W_{i+4} = W_{i+2}$ etc.

If $h$ maps to $r$ different values then expect this example to occur with probability $1/(2r)$ at each step.

Known issue, not quite textbook.

Eliminating fruitless cycles

Issue of fruitless cycles is known and several fixes are proposed. See appendix of full version ePrint 2011/003 for even more details and historical comments.

Summary: most of them got it wrong.
Problem: this walk can run into fruitless cycles!

Example: If $|W_{i+1}| = -W_{i+1}$ and $h(|W_{i+1}|) = j = h(|W_i|)$ then $W_{i+2} = f(W_{i+1}) = -W_{i+1} + c_jP + d_jQ =$

$-(|W_i| + c_jP + d_jQ) + c_jP + d_jQ = -|W_i|$ so $|W_{i+2}| = |W_i|$ so $W_{i+3} = W_{i+1}$ so $W_{i+4} = W_{i+2}$ etc.

If $h$ maps to $r$ different values then expect this example to occur with probability $1/(2r)$ at each step.

Known issue, not quite textbook.

Eliminating fruitless cycles

Issue of fruitless cycles is known and several fixes are proposed. See appendix of full version ePrint 2011/003 for even more details and historical comments.

Summary: most of them got it wrong.

So what to do?
Choose a big $r$, e.g. $r = 2048$.

$1/(2r) = 1/4096$ small; cycles infrequent.
Eliminating fruitless cycles

Issue of fruitless cycles is known and several fixes are proposed. See appendix of full version ePrint 2011/003 for even more details and historical comments.

Summary: most of them got it wrong.

So what to do?
Choose a big $r$, e.g. $r = 2048$.

$1/(2r) = 1/4096$ small;
cycles infrequent.

Precompute points $R_0, R_1, \ldots, R_{r-1}$ as known random multiples of $P$. Define $| (x; y) |$ to mean $(x; y)$ for $y \in \{0; 2; 4; \ldots; p-1\}$ or $(x; -y)$ for $y \in \{1; 3; 5; \ldots; p-2\}$.

Define $| (x; y) |$ to mean $(x; y)$ for $y \in \{0; 2; 4; \ldots; p-1\}$ or $(x; -y)$ for $y \in \{1; 3; 5; \ldots; p-2\}$.

Precompute points $R_0, R_1, \ldots, R_{r-1}$ as known random multiples of $P$. Define $| (x; y) |$ to mean $(x; y)$ for $y \in \{0; 2; 4; \ldots; p-1\}$ or $(x; -y)$ for $y \in \{1; 3; 5; \ldots; p-2\}$. Precompute points $R_0, R_1, \ldots, R_{r-1}$ as known random multiples of $P$. Define $| (x; y) |$ to mean $(x; y)$ for $y \in \{0; 2; 4; \ldots; p-1\}$ or $(x; -y)$ for $y \in \{1; 3; 5; \ldots; p-2\}$.
Eliminating fruitless cycles

Issue of fruitless cycles is known and several fixes are proposed. See appendix of full version ePrint 2011/003 for even more details and historical comments.

Summary: most of them got it wrong.

So what to do?
Choose a big $r$, e.g. $r = 2048$.

$$1/(2r) = 1/4096$$ small;
cycles infrequent.

Define $| (x, y) |$ to mean $(x; y)$ for $y \in \{0, 2, 4, \ldots, p-1\}$ or $(x; -y)$ for $y \in \{1, 3, 5, \ldots, p-2\}$.

Precompute points $R_0, R_1, \ldots, R_{r-1}$ as known random multiples of $\mathcal{P}$. 

Define $| (x, y) |$ to mean $(x; y)$ for $y \in \{0, 2, 4, \ldots, p-1\}$ or $(x; -y)$ for $y \in \{1, 3, 5, \ldots, p-2\}$.

Precompute points $R_0, R_1, \ldots, R_{r-1}$ as known random multiples of $\mathcal{P}$. 

So this walk can run into fruitless cycles!

\[
| W_{i+1} | = -W_i + 1 \quad j = h(| W_i |) \\
W_{i+1} = d_j Q + c_j \mathcal{P} + d_j \mathcal{Q} + c_j \mathcal{P} + d_j \mathcal{Q} = -| W_i | + c_j \mathcal{P} + d_j \mathcal{Q} = -| W_i | \\
| W_{i+2} | = | W_i | \\
\]

etc.

If $h$ maps to $r$ different values then expect this example to occur with probability $1 = (2r)$ at each step.

Known issue, not quite textbook.
Eliminating fruitless cycles

Issue of fruitless cycles is known and several fixes are proposed. See appendix of full version ePrint 2011/003 for even more details and historical comments.

Summary: most of them got it wrong.

So what to do?
Choose a big $r$, e.g. $r = 2048$.

$1/(2r) = 1/4096$ small;
cycles infrequent.

Define $| (x, y) |$ to mean $(x, y)$ for $y \in \{0, 2, 4, \ldots, p\}$ or $(x, -y)$ for $y \in \{1, 3, 5, \ldots, p-2\}$.

Precompute points $R_0, R_1, \ldots, R_{r-1}$ as known random multiples of $P$. 

Problem: this walk can run into fruitless cycles!

Example: If $| W_{i+1} | = -W_{i+1}$ and $h(| W_{i+1} |) = j = h(| W_i |)$ then expect this example to occur with probability $1 = (2^r)$ at each step. 

Known issue, not quite textbook.
Eliminating fruitless cycles

Issue of fruitless cycles is known and several fixes are proposed. See appendix of full version ePrint 2011/003 for even more details and historical comments.

Summary: most of them got it wrong.

So what to do?
Choose a big \( r \), e.g. \( r = 2048 \).

\[
1/(2r) = 1/4096 \text{ small; cycles infrequent.}
\]

Define \(| (x, y) |\) to mean \((x, y)\) for \( y \in \{0, 2, 4, \ldots, p - 1\} \) or \((x, -y)\) for \( y \in \{1, 3, 5, \ldots, p - 2\} \).

Precompute points \( R_0, R_1, \ldots, R_{r-1} \) as known random multiples of \( P \).
Eliminating fruitless cycles

Issue of fruitless cycles is known and several fixes are proposed. See appendix of full version ePrint 2011/003 for even more details and historical comments.

Summary: most of them got it wrong.

So what to do?

Choose a big \( r \), e.g. \( r = 2048 \).

\[
1/(2r) = 1/4096 \text{ small; cycles infrequent.}
\]

Define \((x, y)\) to mean \((x, y)\) for \( y \in \{0, 2, 4, \ldots, p - 1\} \) or \((x, -y)\) for \( y \in \{1, 3, 5, \ldots, p - 2\} \).

Precompute points \( R_0, R_1, \ldots, R_{r-1} \) as known random multiples of \( P \).

Can do full scalar multiplication in inversion-free coordinates!

Start each walk at a point \( W_0 = | b_0 Q | \), where \( b_0 \) is chosen randomly.

Compute \( W_1, W_2, \ldots \) as

\[
W_{i+1} = | W_i + R_{h(W_i)} |.
\]
Eliminating fruitless cycles

Issue of fruitless cycles is known and several fixes are proposed. See appendix of full version ePrint 2011/003 for even more details and historical comments.

Summary: most of them got it wrong.

So what to do?

Choose a big \( r \), e.g. \( r = 2048 \).

\[
1 = (2^r) = 1 = 4096 \text{ small; cycles infrequent.}
\]

Define \( |(x, y)| \) to mean \((x, y)\) for \( y \in \{0, 2, 4, \ldots, p - 1\} \) or \((x, -y)\) for \( y \in \{1, 3, 5, \ldots, p - 2\} \).

Precompute points \( R_0, R_1, \ldots, R_{r-1} \) as known random multiples of \( P \).

Can do full scalar multiplication in inversion-free coordinates!

Start each walk at a point \( W_0 = |b_0Q| \), where \( b_0 \) is chosen randomly.

Compute \( W_1, W_2, \ldots \) as

\[
W_{i+1} = |W_i + R_h(W_i)|.
\]

Occasionally, every \( w \) iterations, check for fruitless cycles of length 2. For those cases change the definition of \( W_i \) as follows:

Compute \( W_{i-1} \) and check whether

If \( W_{i-1} \neq W_{i-3} \), put \( W_i = |2 \min\{W_{i-1}; W_{i-2}\}| \), where \( \min \) means lexicographic minimum.

Doubling the point makes it escape the cycle.
Eliminating fruitless cycles

Issue of fruitless cycles is known and several fixes are proposed. See appendix of full version ePrint 2011/003 for even more details and historical comments.

Summary: most of them got it wrong.

So what to do?

Choose a big $r$, e.g. $r = 2048$.

$$1 = (2^r) = 4096$$ small; cycles infrequent.

Occasionally, every $w$ iterations, check for fruitless cycles of length 2.

For those cases change the definition of $W_i$ as follows:

Compute $W_{i-1}$ and check whether $W_{i-1} = W_{i-3}$.

If $W_{i-1} \neq W_{i-3}$, put

$$W_i = |2 \min\{W_{i-1}, W_{i-2}\}|,$$

where min means lexicographic minimum.

Doubling the point makes it escape the cycle.
Eliminating fruitless cycles

Issue of fruitless cycles is known and several fixes are proposed. See appendix of full version ePrint 2011/003 for even more details and historical comments.

Summary: most of them got it wrong.

So what to do?

Choose a big $r$, e.g. $r = 2048$.

$1 = (2^r) = 1 = 4096$ small;
cycles infrequent.

Define $| (x, y) |$ to mean $(x, y)$ for $y \in \{0, 2, 4, \ldots, p - 1\}$ or $(x, -y)$ for $y \in \{1, 3, 5, \ldots, p - 2\}$.

Precompute points $R_0, R_1, \ldots, R_{r-1}$ as known random multiples of $P$.

Can do full scalar multiplication in inversion-free coordinates!

Start each walk at a point $W_0 = | b_0 Q |$, where $b_0$ is chosen randomly.

Compute $W_1, W_2, \ldots$ as

$W_{i+1} = | W_i + R_h(W_i) |$.

Occasionally, every $w$ iterations, check for fruitless cycles of length 2.

For those cases change the definition of $W_i$ as follows:

Compute $W_{i-1}$ and check whether $W_{i-1} = W_{i-3}$.

If $W_{i-1} \neq W_{i-3}$, put $W_i = W_{i-1}$.

If $W_{i-1} = W_{i-3}$, put $W_i = | 2 \min\{W_{i-1}, W_{i-2}\} |$, where min means lexicographic minimum.

Doubling the point makes it escape the cycle.

Define $| (x, y) |$ to mean $(x, y)$ for $y \in \{0, 2, 4, \ldots, p - 1\}$ or $(x, -y)$ for $y \in \{1, 3, 5, \ldots, p - 2\}$.

Precompute points $R_0, R_1, \ldots, R_{r-1}$ as known random multiples of $P$.

Can do full scalar multiplication in inversion-free coordinates!

Start each walk at a point $W_0 = | b_0 Q |$, where $b_0$ is chosen randomly.

Compute $W_1, W_2, \ldots$ as

$W_{i+1} = | W_i + R_h(W_i) |$.
Define \( |(x, y)| \) to mean 
\((x, y)\) for \(y \in \{0, 2, 4, \ldots, p-1\}\)
or 
\((x, -y)\) for \(y \in \{1, 3, 5, \ldots, p-2\}\).

Precompute points 
\(R_0, R_1, \ldots, R_{r-1}\) as known random multiples of \(P\).

Can do full scalar multiplication in inversion-free coordinates!

Start each walk at a point 
\(W_0 = |b_0 Q|\),
where \(b_0\) is chosen randomly.

Compute \(W_1, W_2, \ldots\) as 
\(W_{i+1} = |W_i + R_h(W_i)|\).

Occasionally, every \(w\) iterations, check for fruitless cycles of length 2.
For those cases change the definition of \(W_i\) as follows:
Compute \(W_{i-1}\) and check whether \(W_{i-1} = W_{i-3}\).

If \(W_{i-1} \neq W_{i-3}\), put \(W_i = W_{i-1}\).

If \(W_{i-1} = W_{i-3}\), put \(W_i = |2 \min\{W_{i-1}, W_{i-2}\}|\),
where \(\min\) means lexicographic minimum.
Doubling the point makes it escape the cycle.
Define \( (x, y) \) to mean \((x, y)\) for \( y \in \{0, 2, 4, \ldots, p-1\}\) or \((x, -y)\) for \( y \in \{1, 3, 5, \ldots, p-2\}\).

Precompute points \( R_0, R_1, \ldots, R_{r-1} \) as known random multiples of \( P \).

Furthermore, scalar multiplication in inversion-free coordinates is possible!

Start each walk at a point \( W_0 = |b_0 Q| \), where \( b_0 \) is chosen randomly.

Compute \( W_1, W_2, \ldots \) as \( W_i = |W_i + R_h(W_i)| \).

Occasionally, every \( w \) iterations, check for fruitless cycles of length 2.
For those cases change the definition of \( W_i \) as follows:

Compute \( W_{i-1} \) and check whether \( W_{i-1} = W_{i-3} \).
If \( W_{i-1} \neq W_{i-3} \), put \( W_i = W_{i-1} \).
If \( W_{i-1} = W_{i-3} \), put \( W_i = |2 \min\{W_{i-1}, W_{i-2}\}| \), where \( \min \) means lexicographic minimum.
Doubling the point makes it escape the cycle.

Cycles of length 4, 6, or 12 occur far less frequently.
Cycles of length 4, or 6 are detected when checking for cycles of length 12; so skip individual ones.

Same way of escape: define \( W_i = |2 \min\{W_{i-1}, W_{i-2}, W_{i-3}, W_{i-4}, W_{i-5}, W_{i-6}, W_{i-7}, W_{i-8}, W_{i-9}, W_{i-10}, W_{i-11}, W_{i-12}\}| \) if trapped and \( W_i = W_{i-1} \) otherwise.
Define \((x; y)\) to mean \((x; y)\) for \(y \in \{0, 2, 4, \ldots, p - 1\}\)
or \((x; -y)\) for \(y \in \{1, 3, 5, \ldots, p - 2\}\).

Precompute points \(R_0, R_1, \ldots, R_{r - 1}\) as known random multiples of \(P\).

Can do full scalar multiplication in inversion-free coordinates!

Start each walk at a point \(W_0 = |b_0 Q|\),
where \(b_0\) is chosen randomly.

Compute \(W_1, W_2, \ldots\) as
\[W_i + 1 = |W_i + R_{h(W_i)}|\.

Occasionally, every \(w\) iterations, check for fruitless cycles
of length 2.
For those cases change the definition of \(W_i\) as follows:
Compute \(W_{i-1}\) and check whether \(W_{i-1} = W_{i-3}\).
If \(W_{i-1} \neq W_{i-3}\), put \(W_i = W_{i-1}\).
If \(W_{i-1} = W_{i-3}\), put
\[W_i = |2 \min\{W_{i-1}, W_{i-2}\}|,
\]
where \(\min\) means lexicographic minimum.
Doubling the point makes it escape the cycle.

Cycles of length 4, 6, or 12 occur far less frequently.
Cycles of length 4, or 6 are detected when checking
for cycles of length 12; so skip individual ones.

Same way of escape:
define \(W_i = \)
\[|2 \min\{W_{i-1}, W_{i-2}, W_{i-3}, W_{i-4}, W_{i-5}, W_{i-6}, W_{i-7}, W_{i-8}, W_{i-9}, W_{i-10}\}|\]
if trapped
and \(W_i = W_{i-1}\) otherwise.
Define \((x;y)\) to mean \((x;y)\) for \(y \in \{0; 2; 4; \ldots; p - 1\}\) or \((x; -y)\) for \(y \in \{1; 3; 5; \ldots; p - 2\}\).

Precompute points \(R_0; R_1; \ldots; R_{r - 1}\) as known random multiples of \(P\).

Can do full scalar multiplication in inversion-free coordinates!

Start each walk at a point \(W_0 = |b_0 Q|\), where \(b_0\) is chosen randomly.

Compute \(W_1; W_2; \ldots\) as \(W_{i + 1} = |W_i + R_{h}(W_i)|\).

Occasionally, every \(w\) iterations, check for fruitless cycles of length 2.

For those cases change the definition of \(W_i\) as follows:
Compute \(W_{i-1}\) and check whether \(W_{i-1} = W_{i-3}\).
If \(W_{i-1} \neq W_{i-3}\), put \(W_i = W_{i-1}\).
If \(W_{i-1} = W_{i-3}\), put
\(W_i = |2 \min\{W_{i-1}, W_{i-2}\}|\),
where \(\min\) means lexicographic minimum.
Doubling the point makes it escape the cycle.

Cycles of length 4, 6, or 12 occur far less frequently.

Cycles of length 4, or 6 are detected when checking for cycles of length 12; so skip individual ones.

Same way of escape:
define \(W_i =
\begin{cases} 
|2 \min\{W_{i-1}, W_{i-2}, W_{i-3}, W_{i-4}, W_{i-5}, W_{i-6}, W_{i-7}, W_{i-8}, W_{i-9}, W_{i-10}, W_{i-11}, W_{i-12}\}|, & \text{if trapped} \\
W_i-1, & \text{otherwise.}
\end{cases}
\)
Occasionally, every $w$ iterations, check for fruitless cycles of length 2. For those cases change the definition of $W_i$ as follows: Compute $W_{i-1}$ and check whether $W_{i-1} = W_{i-3}$. If $W_{i-1} \neq W_{i-3}$, put $W_i = W_{i-1}$. If $W_{i-1} = W_{i-3}$, put

$$W_i = |2 \min\{W_{i-1}, W_{i-2}\}|,$$

where min means lexicographic minimum. Doubling the point makes it escape the cycle.

Cycles of length 4, 6, or 12 occur far less frequently. Cycles of length 4, or 6 are detected when checking for cycles of length 12; so skip individual ones.

Same way of escape: define $W_i =$

$$|2 \min\{W_{i-1}, W_{i-2}, W_{i-3}, W_{i-4}, W_i-5, W_{i-6}, W_{i-7}, W_{i-8}, W_{i-9}, W_{i-10}, W_{i-11}, W_{i-12}\}|$$

if trapped and $W_i = W_{i-1}$ otherwise.
Occasionally, every $w$ iterations, check for fruitless cycles of length 2.

In those cases change the definition of $W_i$ as follows:

- If $W_{i-1} = W_{i-3}$, put $W_i = W_{i-1}$.
- If $W_{i-1} \neq W_{i-3}$, put $W_i = W_{i-1}$.

Define $W_i = \min\{W_{i-1}, W_{i-2}, W_{i-3}, W_{i-4}, W_{i-5}, W_{i-6}, W_{i-7}, W_{i-8}, W_{i-9}, W_{i-10}, W_{i-11}, W_{i-12}\}$, where $\min$ means lexicographic minimum.

Doubling the point makes it escape the cycle.

Cycles of length 4, 6, or 12 occur far less frequently.

Cycles of length 4, or 6 are detected when checking for cycles of length 12; so skip individual ones.

Same way of escape:

- If trapped and $W_i = W_{i-1}$ otherwise.
- If trapped and $W_i = W_{i-1}$ otherwise.

Do not store all these points!

When checking for cycles, store only potential entry point $W_{i-13}$ (one coordinate, for comparison) and the smallest point encountered since (to escape).

For large DLP look for larger cycles; in general, look for fruitless cycles of even lengths up to $\approx (\log r)$.
Occasionally, every $w$ iterations, check for fruitless cycles of length 2. For those cases change the definition of $W_i$ as follows: compute $W_{i-1}$ and check whether $W_{i-1} = W_{i-3}$. If $W_{i-1} \neq W_{i-3}$, put $W_i = W_{i-1}$. If $W_{i-1} = W_{i-3}$, put
$$W_i = \left| \min \{W_{i-1}, W_{i-2}, W_{i-3}, W_{i-4}, W_{i-5}, W_{i-6}, W_{i-7}, W_{i-8}, W_{i-9}, W_{i-10}, W_{i-11}, W_{i-12}\} \right|,$$
if trapped
and $W_i = W_{i-1}$ otherwise.

Cycles of length 4, 6, or 12 occur far less frequently. Cycles of length 4, or 6 are detected when checking for cycles of length 12; so skip individual ones.

Same way of escape: define $W_i =$
$$\left| \min \{W_{i-1}, W_{i-2}, W_{i-3}, W_{i-4}, W_{i-5}, W_{i-6}, W_{i-7}, W_{i-8}, W_{i-9}, W_{i-10}, W_{i-11}, W_{i-12}\} \right|,$$
if trapped
and $W_i = W_{i-1}$ otherwise.

Do not store all these points! When checking for cycle, store only potential entry point $W_{i-13}$ (one coordinate, for comparison) and the smallest point encountered since (to escape).

For large DLP look for larger cycles; in general, look for fruitless cycles of even lengths up to $\approx (\log l)/(\log r)$. 

Do not store all these points! When checking for cycle, store only potential entry point $W_{i-13}$ (one coordinate, for comparison) and the smallest point encountered since (to escape).

For large DLP look for larger cycles; in general, look for fruitless cycles of even lengths up to $\approx (\log l)/(\log r)$. 

Do not store all these points! When checking for cycle, store only potential entry point $W_{i-13}$ (one coordinate, for comparison) and the smallest point encountered since (to escape).

For large DLP look for larger cycles; in general, look for fruitless cycles of even lengths up to $\approx (\log l)/(\log r)$. 

Cycles of length 4, 6, or 12 occur far less frequently. Cycles of length 4, or 6 are detected when checking for cycles of length 12; so skip individual ones.

Cycles of length 4, 6, or 12 occur far less frequently. Cycles of length 4, or 6 are detected when checking for cycles of length 12; so skip individual ones.

Cycles of length 4, 6, or 12 occur far less frequently. Cycles of length 4, or 6 are detected when checking for cycles of length 12; so skip individual ones.
Occasionally, every \( w \) iterations, check for fruitless cycles of length 2. For those cases change the definition of \( W_i \) as follows: Compute \( W_i - 1 \) and check whether \( W_i - 1 = W_i - 3 \). If \( W_i - 1 \neq W_i - 3 \), put \( W_i = W_i - 1 \). If \( W_i - 1 = W_i - 3 \), put \( W_i = |\min \{ W_i - 1, W_i - 2, W_i - 3 \} \| \), where \( \min \) means lexicographic minimum.

Doubling the point makes it escape the cycle.

Cycles of length 4, 6, or 12 occur far less frequently. Cycles of length 4, or 6 are detected when checking for cycles of length 12; so skip individual ones.

Same way of escape: define \( W_i = \left| 2\min \{ W_i - 1, W_i - 2, W_i - 3, W_i - 4, W_i - 5, W_i - 6, W_i - 7, W_i - 8, W_i - 9, W_i - 10, W_i - 11, W_i - 12 \} \right| \) if trapped and \( W_i = W_i - 1 \) otherwise.

Do not store all these points! When checking for cycle, store only potential entry point \( W_{i-13} \) (one coordinate, for comparison) and the smallest point encountered so far (to escape).

For large DLP look for larger cycles; in general, look for fruitless cycles of even lengths up to \( \approx (\log l)/(\log r) \).
Cycles of length 4, 6, or 12 occur far less frequently. Cycles of length 4, or 6 are detected when checking for cycles of length 12; so skip individual ones.

Same way of escape:

\[
\text{define } W_i = \\
\vert 2\min\{W_{i-1}, W_{i-2}, W_{i-3}, W_{i-4}, W_{i-5}, W_{i-6}, W_{i-7}, W_{i-8}, W_{i-9}, W_{i-10}, W_{i-11}, W_{i-12}\} \vert \\
\text{if trapped}
\]

and \( W_i = W_{i-1} \) otherwise.

Do not store all these points!

When checking for cycle, store only potential entry point \( W_{i-13} \) (one coordinate, for comparison) and the smallest point encountered since (to escape).

For large DLP look for larger cycles; in general, look for fruitless cycles of even lengths up to \( \approx (\log \ell)/(\log r) \).
Cycles of length 4, 6, or 12 occur far less frequently. Cycles of length 4, or 6 are detected when checking for cycles of length 12; so skip individual ones. 

A way of escape: 

$$W_i = |W_{i-1}, W_{i-2}, W_{i-3}, W_{i-4}, W_{i-5}, W_{i-6}, W_{i-7}, W_{i-8}, W_{i-9}, W_{i-10}, W_{i-11}, W_{i-12}|$$

if trapped and $$W_i = W_{i-1}$$ otherwise.

Do not store all these points! When checking for cycle, store only potential entry point $$W_{i-13}$$ (one coordinate, for comparison) and the smallest point encountered since $$W_{i-13}$$ (one coordinate, for escape).

For large DLP look for larger cycles; in general, look for fruitless cycles of even lengths up to $$\approx (\log \ell)/(\log r)$$. If a cycle as small as $$2r$$ appears, it wastes $$\approx w = 2$$ iterations (on average) if it does appear. Do not choose $$w$$ as small as possible! If a cycle has not appeared then the check wastes an iteration.

How to choose $$w$$?

Fruitless cycles of length 2 appear with probability $$\approx 1 = (2r)$$. These cycles persist until detected. After $$w$$ iterations, the probability of cycle detection is $$\approx (2r)^w$$. These cycles with probability $$\approx w = (2r)$$, wastes $$\approx = 2$$ iterations (on average) if it does appear. Do not choose $$w$$ as small as possible! If a cycle has not appeared then the check wastes an iteration.
Cycles of length 4, 6, or 12 occur far less frequently.
Cycles of length 4, or 6 are detected when checking for cycle of length 12; so skip individual ones.

Same way of escape:

\[
W_i = \begin{cases} 
W_{i-1} & \text{if trapped} \\
W_{i-2} & W_{i-3} \ldots W_{i-12} \end{cases} 
\]

Do not store all these points!

When checking for cycle, store only potential entry point \(W_{i-13}\) (one coordinate, for comparison) and the smallest point encountered since (to escape).

For large DLP look for larger cycles; in general, look for fruitless cycles of even lengths up to \(\approx (\log \ell)/(\log r)\).

How to choose \(w\)?

Fruitless cycles of length 2 appear with probability \(\approx 1 = (2r)^{-1}\). These cycles persist until detected. After \(w\) iterations, probability of cycle appearing wastes \(\approx w/2\) iterations (on average) if it does appear.

Do not choose \(w\) as small as possible!

If a cycle has not appeared then the check wastes an iteration.
Cycles of length 4, 6, or 12 occur far less frequently.

Cycles of length 4, or 6 are detected when checking for cycles of length 12; so skip individual ones.

Same way of escape: define

$$W_i = \begin{cases} W_i - 1; & \text{if trapped} \\ W_i - 2; & \text{if not trapped} \end{cases}$$

Do not store all these points!

When checking for cycle, store only potential entry point $W_{i-13}$ (one coordinate, for comparison) and the smallest point encountered since (to escape).

For large DLP look for larger cycles; in general, look for fruitless cycles of even lengths up to $\approx (\log \ell)/(\log r)$.

How to choose $w$?

Fruitless cycles of length 2 appear with probability $\approx 1/(2r)$.

These cycles persist until detected. After $w$ iterations, probability of cycle $\approx w/(2r)$

wastes $\approx w/2$ iterations (on average) if it does appear.

Do not choose $w$ as small as possible!

If a cycle has not appeared then the check wastes an iteration.

Fruitless cycles of length 2 appear with probability $\approx 1/(2r)$.

These cycles persist until detected. After $w$ iterations, probability of cycle $\approx w/(2r)$

wastes $\approx w/2$ iterations (on average) if it does appear.

Do not choose $w$ as small as possible!

If a cycle has not appeared then the check wastes an iteration.
Do not store all these points!

When checking for cycle, store only potential entry point $W_{i-13}$ (one coordinate, for comparison) and the smallest point encountered since (to escape).

For large DLP look for larger cycles; in general, look for fruitless cycles of even lengths up to $\approx \log l / \log r$.

How to choose $w$?

Fruitless cycles of length 2 appear with probability $\approx 1 / (2r)$. These cycles persist until detected.

After $w$ iterations, probability of cycle $\approx w / (2r)$, wastes $\approx w / 2$ iterations (on average) if it does appear.

Do not choose $w$ as small as possible!

If a cycle has not appeared then the check wastes an iteration.
Do not store all these points!

When checking for cycle, store only potential entry point (one coordinate, for instance) and the smallest point encountered since (to escape).

For large DLP look for larger cycles; in general, look for fruitless cycles of even lengths up to \( \approx \log r \).

How to choose \( w \)?

Fruitless cycles of length 2 appear with probability \( \approx 1/(2r) \).
These cycles persist until detected.
After \( w \) iterations, probability of cycle \( \approx w/(2r) \), wastes \( \approx w/2 \) iterations (on average) if it does appear.

Do not choose \( w \) as small as possible!
If a cycle has not appeared then the check wastes an iteration.

The overall loss is approximately \( 1 + w^2/(4r) \).
To minimize this, choose \( w = 2/\sqrt{r} \).

Cycles of length \( c \) appear with probability \( \approx 1/r^c \), optimal checking frequency is \( \approx 1/r^c = 2 \).
Loss rapidly disappears as \( c \) increases.

Can use lcm of cycle lengths to check.
Do not store all these points!

When checking for cycle, store only potential entry point \( W_{i-13} \) (one coordinate, for comparison) and the smallest point encountered since the beginning.

For large DLP look for larger cycles; in general, look for fruitless cycles of even lengths up to \( \approx \log r \).

How to choose \( w \)?

Fruitless cycles of length 2 appear with probability \( \approx \frac{1}{2r} \).
These cycles persist until detected.
After \( w \) iterations, probability of cycle \( \approx \frac{w}{2r} \), wastes \( \approx \frac{w}{2} \) iterations (on average) if it does appear.

Do not choose \( w \) as small as possible!
If a cycle has not appeared then the check wastes an iteration.

The overall loss is \( 1 + \frac{w^2}{4r} \) iterations.
To minimize the quotient \( \frac{1}{w} + \frac{w}{4r} \) we take \( w \approx \sqrt{2r} \).

Cycles of length 2 appear with probability \( \approx \frac{1}{r^c} \), optimal checking frequency is \( \approx \frac{1}{r^{c/2}} \).
Loss rapidly disappears as \( c \) increases.
Can use lcm of cycle lengths to check.
How to choose $w$?

Fruitless cycles of length 2 appear with probability $\approx 1/(2r)$.
These cycles persist until detected.

After $w$ iterations, probability of cycle $\approx w/(2r)$, wastes $\approx w/2$ iterations (on average) if it does appear.

Do not choose $w$ as small as possible!
If a cycle has not appeared then the check wastes an iteration.

The overall loss is approximately $1 + w^2/(4r)$ iterations out of $w$.
To minimize the quotient $1/w + w/(4r)$ we take $w \approx 2 \sqrt{r}$.

Cycles of length $2c$ appear with probability $\approx 1/r^c$, optimal checking frequency $\approx 1/r^c/2$.
Loss rapidly disappears as $c$ increases.
Can use lcm of cycle lengths to check.
How to choose $w$?

Fruitless cycles of length 2 appear with probability $\approx 1/(2r)$. These cycles persist until detected. After $w$ iterations, probability of cycle $\approx w/(2r)$, wastes $\approx w/2$ iterations (on average) if it does appear.

Do not choose $w$ as small as possible! If a cycle has not appeared then the check wastes an iteration.

The overall loss is approximately $1 + w^2/(4r)$ iterations out of $w$. To minimize the quotient $1/w + w/(4r)$ we take $w \approx 2\sqrt{r}$.

Cycles of length $2c$ appear with probability $\approx 1/r^c$, optimal checking frequency is $\approx 1/r^{c/2}$. Loss rapidly disappears as $c$ increases. Can use lcm of cycle lengths to check.
How to choose $w$?

Cycles of length 2 appear with probability $\approx 1/(2r)$.

Cycles persist until detected.

After $w$ iterations, probability of cycle appears $\approx w/(2r)$, wastes $\approx w/2$ iterations (on average) if it does appear.

Do not choose $w$ as small as possible!

If a cycle has not appeared then the check wastes an iteration.

The overall loss is approximately $1 + w^2/(4r)$ iterations out of $w$.

To minimize the quotient $1/w + w/(4r)$ we take $w \approx 2\sqrt{r}$.

Cycles of length $2c$ appear with probability $\approx 1/r^c$, optimal checking frequency is $\approx 1/r^{c/2}$.

Loss rapidly disappears as $c$ increases.

Can use lcm of cycle lengths to check.

Concrete example:

Use $r = 2048$. Check for 2-cycles every 48 iterations.

Check for larger cycles much less frequently.

Unify the checks for 4-cycles and 6-cycles into a check for 12-cycles every 49152 iterations.

Choice of $r$ has big impact!

Use $r = 512$ calls for checking for 2-cycles every 24 iterations.

In general, negation overhead $\approx$ doubles when table size is reduced by factor of 4.
How to choose $w$?

Fruitless cycles of length 2 appear with probability $\approx \frac{1}{2r}$.

These cycles persist until detected. After $w$ iterations, probability of cycle $\approx \frac{w}{2r}$, wastes $\approx \frac{w}{2}$ iterations (on average) if it does appear.

Do not choose $w$ as small as possible!

If a cycle has not appeared then the check wastes an iteration.

The overall loss is approximately $1 + \frac{w^2}{(4r)}$ iterations out of $w$.

To minimize the quotient $1/w + \frac{w}{(4r)}$ we take $w \approx 2\sqrt{r}$.

Cycles of length $2c$ appear with probability $\approx \frac{1}{r^c}$, optimal checking frequency is $\approx \frac{1}{r^{c/2}}$.

Loss rapidly disappears as $c$ increases.

Can use lcm of cycle lengths to check.

Concrete example: 112-bit DLP

Use $r = 2048$. Check for 2-cycles every 48 iterations.

Check for larger cycles much less frequently.

Unify the checks for 4-cycles and 6-cycles into a check for 12-cycles every 49152 iterations.

Choice of $r$ has big impact! $r = 512$ calls for checking for 2-cycles every 24 iterations.

In general, negation overhead $\approx$ doubles when table size is reduced by factor of 4.
How to choose \( w \)?

Fruitless cycles of length 2 appear with probability \( \approx \frac{1}{2^r} \).
These cycles persist until detected.
After \( w \) iterations, probability of cycle \( \approx \frac{w}{2^r} \),
wastes \( \approx \frac{w}{2^r} \) iterations (on average) if it does appear.

Do not choose \( w \) as small as possible!
If a cycle has not appeared then the check wastes an iteration.

The overall loss is approximately \( 1 + \frac{w^2}{(4r)} \) iterations out of \( w \).
To minimize the quotient \( \frac{1}{w} + \frac{w}{(4r)} \) we take \( w \approx 2\sqrt{r} \).

Cycles of length \( 2c \) appear with probability \( \approx \frac{1}{r^c} \),
optimal checking frequency is \( \approx \frac{1}{r^{c/2}} \).
Loss rapidly disappears as \( c \) increases.
Can use lcm of cycle lengths to check.

Concrete example: 112-bit DLP

Use \( r = 2048 \). Check for 2-cycles every 48 iterations.
Check for larger cycles much less frequently.
Unify the checks for 4-cycles and 6-cycles into a check for 12-cycles every 49152 iterations.

Choice of \( r \) has big impact!
\( r = 512 \) calls for checking for 2-cycles every 24 iterations.
In general, negation overhead \( \approx \) doubles when table size is reduced by factor of 4.
The overall loss is approximately 
\[ 1 + \frac{w^2}{(4r)} \] iterations out of \( w \).
To minimize the quotient 
\[ \frac{1}{w} + \frac{w}{(4r)} \]
we take 
\( w \approx 2\sqrt{r} \).

Cycles of length \( 2c \) appear with probability \( \approx \frac{1}{r^c} \),
optimal checking frequency is 
\( \approx \frac{1}{r^{c/2}} \).

Loss rapidly disappears as \( c \) increases.
Can use lcm of cycle lengths to check.

Concrete example: 112-bit DLP
Use \( r = 2048 \). Check for 2-cycles every 48 iterations.
Check for larger cycles much less frequently.
Unify the checks for 4-cycles and 6-cycles into a check for 12-cycles every 49152 iterations.

Choice of \( r \) has big impact!
\( r = 512 \) calls for checking for 2-cycles every 24 iterations.
In general, negation overhead \( \approx \) doubles when table size is reduced by factor of 4.
The overall loss is approximately \(1 + \frac{w}{(4r)}\) iterations out of \(w\). To minimize the quotient \(\frac{1}{w} = \frac{1}{(4r)}\) we take \(w \approx 2\sqrt{r}\).

Cycles of length \(2c\) appear with probability \(\approx \frac{1}{r^c}\),
so the checking frequency is \(2\).
Loss rapidly disappears as \(c\) increases.
Can use lcm of cycle lengths to check.

Concrete example: 112-bit DLP
Use \(r = 2048\). Check for 2-cycles every 48 iterations.
Check for larger cycles much less frequently.
Unify the checks for 4-cycles and 6-cycles into a check for 12-cycles every 49152 iterations.
Choice of \(r\) has big impact!
\(r = 512\) calls for checking for 2-cycles every 24 iterations.
In general, negation overhead \(\approx\) doubles when table size is reduced by factor of 4.

Bernstein, Lange, Schwabe (PKC 2011):
Our software solves random ECDL on the same curve (with no precomputation) in 35.6 PS3 years on average.
For comparison: Bos–Kaihara–Kleinjung–Lenstra–Montgomery software uses 65
The overall loss is approximately \( 1 + \frac{1}{w} \) iterations out of \( w \). To minimize the quotient \( \frac{1}{w} \), take \( w \approx 2\sqrt{r} \).

Cycles of length 2 appear with frequency \( \approx \frac{1}{r} \), so optimal checking frequency is \( \approx \frac{1}{r} c = 2 \).

Loss rapidly disappears as \( c \) increases. Can use lcm of cycle lengths to check.

Concrete example: 112-bit DLP

Use \( r = 2048 \). Check for 2-cycles every 48 iterations. Check for larger cycles much less frequently. Unify the checks for 4-cycles and 6-cycles into a check for 12-cycles every 49152 iterations.

Choice of \( r \) has big impact!

- \( r = 512 \) calls for checking for 2-cycles every 24 iterations.
- In general, negation overhead \( \approx \) doubles when table size is reduced by factor of 4.

Bernstein, Lange, Schwabe (PKC 2011):

Our software solves random ECDL on the same curve (with no precomputation) in 35.6 PS3 years on average.

For comparison: Bos–Kaihara–Kleinjung–Lenstra–Montgomery software uses 65 PS3 years on average.
The overall loss is approximately 
\[ 1 + w^2 = (4 \sqrt{r}) \] iterations out of \( w \).

To minimize the quotient 
\[ \frac{1}{w + \sqrt{w}} = (4 \sqrt{r}) \]
we take 
\[ w \approx 2 \sqrt{r} \].

Cycles of length 2 appear with probability
\[ \approx \frac{1}{r c} \],
optimal checking frequency is
\[ \approx \frac{1}{r c} = 2 \].
Loss rapidly disappears as \( c \) increases.
Can use lcm of cycle lengths to check.

Concrete example: 112-bit DLP
Use \( r = 2048 \). Check for 2-cycles every 48 iterations.
Check for larger cycles much less frequently.
Unify the checks for 4-cycles and 6-cycles into a check for 12-cycles every 49152 iterations.
Choice of \( r \) has big impact!
\( r = 512 \) calls for checking for 2-cycles every 24 iterations.
In general, negation overhead \( \approx \) doubles when table size is reduced by factor of 4.

Bernstein, Lange, Schwabe (PKC 2011):
Our software solves random ECDL on the same curve (with no precomputation) in 35.6 PS3 years on average.
For comparison:
Bos–Kaihara–Kleinjung–Lenstra–Montgomery software uses 65 PS3 years on average.
Concrete example: 112-bit DLP

Use $r = 2048$. Check for 2-cycles every 48 iterations.
Check for larger cycles much less frequently.
Unify the checks for 4-cycles and 6-cycles into a check for 12-cycles every 49152 iterations.

Choice of $r$ has big impact!

$r = 512$ calls for checking for 2-cycles every 24 iterations.
In general, negation overhead $\approx$ doubles when table size is reduced by factor of 4.

Bernstein, Lange, Schwabe (PKC 2011):

Our software solves random ECDL on the same curve (with no precomputation) in 35.6 PS3 years on average.

For comparison:
Bos–Kaihara–Kleinjung–Lenstra–Montgomery software uses 65 PS3 years on average.
Concrete example: 112-bit DLP

Use $r = 2048$. Check for 2-cycles every 48 iterations.
Check for larger cycles much less frequently.
Unify the checks for 4-cycles and 6-cycles into a check for 12-cycles every 49152 iterations.

Choice of $r$ has big impact!

$r = 512$ calls for checking for 2-cycles every 24 iterations.
In general, negation overhead $\approx$ doubles when table size is reduced by factor of 4.

Bernstein, Lange, Schwabe (PKC 2011):

Our software solves random ECDL on the same curve (with no precomputation) in 35.6 PS3 years on average.

For comparison:
Bos–Kaihara–Kleinjung–Lenstra–Montgomery software uses 65 PS3 years on average.

First big speedup:
We use the negation map.
Second speedup: Fast arithmetic.
Concrete example: 112-bit DLP
Use $r = 2048$. Check for 2-cycles every 48 iterations.
Check for larger cycles much less frequently.
Unify the checks for 4-cycles and 6-cycles into a check for 12-cycles every 49152 iterations.

Choice of $r$ has big impact!

$2^10$ calls for checking for 2-cycles every 24 iterations.

Negation overhead approximately doubles when table size is reduced by factor of 4.

Bernstein, Lange, Schwabe (PKC 2011):
Our software solves random ECDL on the same curve (with no precomputation) in 35.6 PS3 years on average.

For comparison:
Bos–Kaihara–Kleinjung–Lenstra–Montgomery software uses 65 PS3 years on average.

First big speedup:
We use the negation map.
Second speedup: Fast arithmetic.

Why are we confident this works?
We only have 1 PlayStation-3, not 200 used in their record.
Don’t want to wait for 36 years to show that we actually compute the right thing.
Concrete example: 112-bit DLP

Use $r = 2048$. Check for 2-cycles every 48 iterations.
Check for larger cycles much less frequently.

Unify the checks for 4-cycles and 6-cycles into a check for 12-cycles every 49152 iterations.

Choice of $r$ has big impact!

$r = 512$ calls for checking for 2-cycles every 24 iterations.

In general, negation overhead $\approx$ doubles when table size is reduced by factor of 4.

Bernstein, Lange, Schwabe (PKC 2011):

Our software solves random ECDL on the same curve (with no precomputation) in 35.6 PS3 years on average.

For comparison:
Bos–Kaihara–Kleinjung–Lenstra–Montgomery software uses 65 PS3 years on average.

First big speedup:
We use the negation map.

Second speedup: Fast arithmetic.

Why are we confident this works?
We only have 1 PlayStation-3, not 200 used in their record.
Don’t want to wait for 36 years to show that we actually compute the right thing.
Concrete example: 112-bit DLP

Use $r = 2048$. Check for 2-cycles every 48 iterations.
Check for larger cycles much less frequently.
Unify the checks for 4-cycles and 6-cycles into a check for 12-cycles every 49152 iterations.

Choice of $r$ has big impact!

$r = 512$ calls for checking for 2-cycles every 24 iterations.

In general, negation overhead $\approx$ doubles when table size is reduced by factor of 4.

Bernstein, Lange, Schwabe (PKC 2011):
Our software solves random ECDL on the same curve (with no precomputation) in 35.6 PS3 years on average.

For comparison:
Bos–Kaihara–Kleinjung–Lenstra–Montgomery software uses 65 PS3 years on average.

First big speedup:
We use the negation map.
Second speedup: Fast arithmetic.

Why are we confident this works?
We only have 1 PlayStation-3, not 200 used in their record.
Don't want to wait for 36 years to show that we actually compute the right thing.
Bernstein, Lange, Schwabe (PKC 2011):
Our software solves random ECDL on the same curve (with no precomputation) in 35.6 PS3 years on average.

For comparison:
Bos–Kaihara–Kleinjung–Lenstra–Montgomery software uses 65 PS3 years on average.

First big speedup:
We use the negation map.

Second speedup: Fast arithmetic.

Why are we confident this works?
We only have 1 PlayStation-3, not 200 used in their record. Don’t want to wait for 36 years to show that we actually compute the right thing.
Bernstein, Lange, Schwabe (PKC 2011):

Our software solves random ECDL on the same curve (with no precomputation) in 35.6 PS3 years on average.

For comparison:
Bos–Kaihara–Kleinjung–Lenstra–Montgomery software uses 65 PS3 years on average.

First big speedup:
We use the negation map.

Second speedup: Fast arithmetic.

Why are we confident this works?
We only have 1 PlayStation-3, not 200 used in their record. Don’t want to wait for 36 years to show that we actually compute the right thing.

Can produced scaled versions: Use same prime field (so that we can compare the field arithmetic) and same curve shape $y^2 = x^3 - 3x + b$ but vary $b$ to get curves with small subgroups.
Our software solves random ECDL on the same curve (with no precomputation) in 35.6 PS3 years on average. For comparison: Bos–Kaihara–Kleinjung–Lenstra–Montgomery software uses 65 PS3 years on average.

First big speedup: We use the negation map.
Second speedup: Fast arithmetic.

Why are we confident this works?

We only have 1 PlayStation-3, not 200 used in their record. Don't want to wait for 36 years to show that we actually compute the right thing.

Can produced scaled versions: Use same prime field (so that we can compare the field arithmetic) and same curve shape \( y^2 = x^3 - 3x + b \) but vary \( b \) to get curves with small subgroups.

This produces other curves, and many of those have smaller order subgroups. Specify DLP in subgroup of size \( 2^{50} \), or \( 2^{55} \), or \( 2^{60} \) and show that the actual running time matches the expectation. And that DLP is correct.

We used same property for a point to be distinguished as in big attack; probability is \( 2^{-20} \).

Need to watch out that walks do not run into rho-type cycles (artefact of small group order). We aborted overlong walks.
Our software solves random ECDL on the same curve (with no precomputation) in 35.6 PS3 years on average. For comparison: Bos–Kaihara–Kleinjung–Lenstra–Montgomery software uses 65 PS3 years on average.

First big speedup: We use the negation map.
Second speedup: Fast arithmetic.

Why are we confident this works? We only have 1 PlayStation-3, not 200 used in their record. Don’t want to wait for 36 years to show that we actually compute the right thing.

Can produced scaled versions: Use same prime field (so that we can compare the field arithmetic) and same curve shape $y^2 = x^3 - 3x + b$ but vary $b$ to get curves with small subgroups.

This produces other curves, and many of those have smaller order subgroups. Specify DLP in subgroup of size $2^{50}$, or $2^{55}$, or $2^{60}$ and show that the actual running time matches the expectation. And that DLP is correct.

We used same property for a point to be distinguished as in big attack; probability is $2^{-20}$. Need to watch out that walks do not run into rho-type cycles (artefact of small group order). We aborted overlong walks.
Bernstein, Lange, Schwabe (PKC 2011):
Our software solves random ECDL on the same curve (with no precomputation) in 35.6 PS3 years on average. For comparison: Bos–Kaihara–Kleinjung–Lenstra–Montgomery software uses 65 PS3 years on average. First big speedup: We use the negation map. Second speedup: Fast arithmetic.

Why are we confident this works? We only have 1 PlayStation-3, not 200 used in their record. Don’t want to wait for 36 years to show that we actually compute the right thing. Can produced scaled versions: Use same prime field (so that we can compare the field arithmetic) and same curve shape \( y^2 = x^3 - 3x + b \) but vary \( b \) to get curves with small subgroups.

This produces other curves, and many of those have smaller subgroups. Specify DLP in subgroup of size \( 2^{50} \), or \( 2^{55} \), or \( 2^{60} \) and show that the actual running time matches the expectation. And that DLP is correct.

We used same property for a point to be distinguished as in big attack; probability is \( 2^{-20} \). Need to watch out that walks do not run into rho-type cycles (artefact of small group order). We aborted overlong walks.
Why are we confident this works?
We only have 1 PlayStation-3, not 200 used in their record.
Don’t want to wait for 36 years to show that we actually compute the right thing.

Can produced scaled versions: Use same prime field (so that we can compare the field arithmetic) and same curve shape $y^2 = x^3 - 3x + b$ but vary $b$ to get curves with small subgroups.

This produces other curves, and many of those have smaller order subgroups.
Specify DLP in subgroup of size $2^{50}$, $2^{55}$, or $2^{60}$ and show that the actual running time matches the expectation.
And that DLP is correct.

We used same property for a point to be distinguished as in big attack; probability is $2^{-20}$. Need to watch out that walks do not run into rho-type cycles (artefact of small group order). We aborted overlong walks.
**Why are we confident this works?**

We only have 1 PlayStation-3, not 200 used in their record. Don't want to wait for 36 years to show that we actually compute the right thing.

Produced scaled versions:
- Same prime field (so that we can compare the field arithmetic) and same curve shape $y^2 = x^3 - 3x + b$
- Vary $b$ to get curves with small subgroups.

This produces other curves, and many of those have smaller order subgroups.

Specify DLP in subgroup of size $2^{50}$, or $2^{55}$, or $2^{60}$ and show that the actual running time matches the expectation.

And that DLP is correct.

We used same property for a point to be distinguished as in big attack; probability is $2^{-20}$.

Need to watch out that walks do not run into rho-type cycles (artefact of small group order). We aborted overlong walks.

**New record**

Announced 29 Nov 2016, most work by Ruben Niederhagen (@cryptocephaly on twitter).

Elliptic curve over $\mathbb{F}_{2^{127}}$, DLP in subgroup of order $2^{117}$: $35$.

Used parallel Pollard rho, DP criterion: 30 top bits equal 0.

Expected $2^{\pi 2^{117}/2^{35}} \approx 2^{379821956}$ DPs, but ended up needing 968 531 433.

Computations ran on 64 to 576 FPGAs in parallel.
We only have 1 PlayStation-3, not 200 used in their record. Don't want to wait for 36 years to show that we actually compute the right thing.

Can produce scaled versions: use the same prime field (so that we can compare the field arithmetic) and same curve shape $y^2 = x^3 - 3x + b$ but vary $b$ to get curves with small subgroups.

This produces other curves, and many of those have smaller order subgroups.
Specify DLP in subgroup of size $2^{50}$, or $2^{55}$, or $2^{60}$ and show that the actual running time matches the expectation.
And that DLP is correct.

We used same property for a point to be distinguished as in big attack; probability is $2^{-20}$. Need to watch out that walks do not run into rho-type cycles (artefact of small group order). We aborted overlong walks.

New record
Announced 29 Nov 2016, most work by Ruben Niederhagen (@cryptocephaly on twitter).

Elliptic curve over $\mathbb{F}_{2^{127}}$, DLP in subgroup of order $2^{117}$: 35.
Used parallel Pollard rho, DP criterion: 30 top bits equal 0.

Expected $\sqrt{\pi 2^{117.35}/4}/2^{30} = 971,931,461$ DPs, but ended up needing 968,531,433.
Computations ran on 64 to 576 FPGAs in parallel.
Why are we confident this works?
We only have 1 PlayStation-3, not 200 used in their record.
Don't want to wait for 36 years to show that we actually compute the right thing.
Can produce scaled versions:
Use the same prime field (so that we can compare the field arithmetic) and same curve shape
\[ y^2 = x^3 - 3x + b \]
but vary \( b \) to get curves with small subgroups.
This produces other curves, and many of those have smaller order subgroups.
Specify DLP in subgroup of size \( 2^{50} \), or \( 2^{55} \), or \( 2^{60} \) and show that the actual running time matches the expectation.
And that DLP is correct.
We used same property for a point to be distinguished as in big attack; probability is \( 2^{-20} \).
Need to watch out that walks do not run into rho-type cycles (artefact of small group order). We aborted overlong walks.

New record
Announced 29 Nov 2016, most work by Ruben Niederhagen (@cryptocephaly on twitter).
Elliptic curve over \( \mathbb{F}_2^{127} \),
DLP in subgroup of order \( 2^{117} \):
Used parallel Pollard rho, DP criterion: 30 top bits equal 0.
Expected
\[ \sqrt{\pi 2^{117.35} / 4 / 2^{30}} \sim 379\,821\,956 \]
DPs, but ended up needing 968\,531\,433.
Computations ran on 64 to 576 FPGAs in parallel.
This produces other curves, and many of those have smaller order subgroups. Specify DLP in subgroup of size $2^{50}$, or $2^{55}$, or $2^{60}$ and show that the actual running time matches the expectation. And that DLP is correct.

We used same property for a point to be distinguished as in big attack; probability is $2^{-20}$. Need to watch out that walks do not run into rho-type cycles (artefact of small group order). We aborted overlong walks.

---

New record
Announced 29 Nov 2016, most work by Ruben Niederhagen (@cryptocephaly on twitter).

Elliptic curve over $\mathbb{F}_{2^{127}}$, DLP in subgroup of order $2^{117.35}$. Used parallel Pollard rho, DP criterion: 30 top bits equal 0.

Expected
\[ \sqrt{\pi 2^{117.35}/4/2^{30}} \sim 379\,821\,956 \]
DPs, but ended up needing 968 531 433.
Computations ran on 64 to 576 FPGAs in parallel.
This produces other curves, and those have smaller order subgroups.

Specify DLP in subgroup of size $2^{50}$, or $2^{55}$, or $2^{60}$ and show that actual running time matches expectation.

But DLP is correct.

We used same property for a point to be distinguished as in big attack; probability is $2^{-20}$.

We can use this in baby-step giant-step algorithm.

How to use this in a memory-less algorithm?

New record

Announced 29 Nov 2016, most work by Ruben Niederhagen (@cryptocephaly on twitter).

Elliptic curve over $F_{2^{127}}$, DLP in subgroup of order $2^{117.35}$.

Used parallel Pollard rho, DP criterion: 30 top bits equal 0.

Expected

$$\sqrt{\pi 2^{117.35}/4/2^{30}} \sim 379\,821\,956$$

DPs, but ended up needing 968\,531\,433.

Computations ran on 64 to 576 FPGAs in parallel.

DLs in intervals

Want to use knowledge that DLP is in a small interval $[a; b]$, much smaller than $\epsilon$.

We can use this in baby-step giant-step algorithm.

How to use this in a memory-less algorithm?
This produces other curves, and many of those have smaller order subgroups. Specify DLP in subgroup of size $2^{50}$, $2^{55}$, or $2^{60}$, and show that the actual running time matches the expectation. And that DLP is correct.

We used same property for a point to be distinguished as in big attack; probability is $2^{-20}$. Need to watch out that walks do not run into rho-type cycles (artefact of small group order). We aborted overlong walks.

**New record**

Announced 29 Nov 2016, most work by Ruben Niederhagen (@cryptocephaly on twitter).

Elliptic curve over $\mathbb{F}_{2^{127}}$, DLP in subgroup of order $2^{117.35}$. Used parallel Pollard rho, DP criterion: 30 top bits equal 0.

Expected

$$\sqrt{\pi 2^{117.35} / 4 / 2^{30}} \sim 379\,821\,956$$

DPs, but ended up needing 968,531,433. Computations ran on 64 to 576 FPGAs in parallel.

**DLs in intervals**

Want to use knowledge that DLP is in a small interval $[a, b]$, much smaller than $\pi$. We can use this in a baby-step giant-step algorithm. How to use this in a memory-less algorithm?
This produces other curves, and many of those have smaller order subgroups. Specify DLP in subgroup of size $2^{50}$, or $2^{55}$, or $2^{60}$ and show that the actual running time matches the expectation. And that DLP is correct.

We used same property for a point to be distinguished as in big attack; probability is $2^{-20}$. Need to watch out that walks do not run into rho-type cycles (artefact of small group order). We aborted overlong walks.

New record

Announced 29 Nov 2016, most work by Ruben Niederhagen (@cryptocephaly on twitter).

Elliptic curve over $F_{2^{127}}$, DLP in subgroup of order $2^{117.35}$. Used parallel Pollard rho, DP criterion: 30 top bits equal 0.

Expected

$$\sqrt{\pi 2^{117.35}/4}/2^{30} \sim 379\,821\,956$$

DPs, but ended up needing 968 531 433.

Computations ran on 64 to 576 FPGAs in parallel.

DLs in intervals

Want to use knowledge that DL is in a small interval $[a, b]$, much smaller than $\ell$. We can use this in baby-step giant-step algorithm.

How to use this in a memory-less algorithm?
New record
Announced 29 Nov 2016,
most work by Ruben Niederhagen (@cryptocephaly on twitter).
Elliptic curve over $\mathbb{F}_{2^{127}}$,
DLP in subgroup of order $2^{117.35}$. Used parallel Pollard rho,
DP criterion: 30 top bits equal 0.
Expected
$$\sqrt{\pi 2^{117.35}/4/2^{30}} \sim 379\,821\,956$$
DPs, but ended up needing
968\,531\,433.
Computations ran on 64 to 576
FPGAs in parallel.

DLs in intervals
Want to use knowledge
that DL is in a
small interval $[a, b]$,
much smaller than $\ell$.
We can use this in baby-step
giant-step algorithm.
How to use this in a
memory-less algorithm?
Announced 29 Nov 2016, most work by Ruben Niederhagen (@cryptocephaly on twitter).

Elliptic curve over $\mathbb{F}_{2^{127}}$, subgroup of order $2^{117.35}$. Parallel Pollard rho, criterion: 30 top bits equal 0.

Expected $p = 2^{117.35}/4/2^{30} \sim 379,821,956$ DLPs, but ended up needing 968,531,433. Computations ran on 64 to 576 FPGAs in parallel.

DLs in intervals

Want to use knowledge that DL is in a small interval $[a, b]$, much smaller than $\ell$.

We can use this in baby-step giant-step algorithm.

How to use this in a memory-less algorithm?
New record announced 29 Nov 2016, most work by Ruben Niederhagen (@cryptocephaly on twitter).

Elliptic curve over $\mathbb{F}_{2^{127}}$, DLP in subgroup of order $2^{117.35}$.

Used parallel Pollard rho, DP criterion: 30 top bits equal 0.

Expected $p \approx 379\,821\,956$

but ended up needing 968,531,433.

Computations ran on 64 to 576 FPGAs in parallel.

**DLs in intervals**

Want to use knowledge that DL is in a small interval $[a, b]$, much smaller than $\ell$.

We can use this in baby-step giant-step algorithm.

How to use this in a memory-less algorithm?

Standard interval method: Pollard’s kangaroo method.

Pollard’s kangaroos do small jumps around the interval.
New record announced 29 Nov 2016, most work by Ruben Niederhagen (@cryptocephaly on twitter).

Elliptic curve over $F_{2^{117.35}}$, DLP in subgroup of order $2^{117.35} - 35$.

Used parallel Pollard rho, DP criterion: 30 top bits equal 0.

Expected $p = 2^{117.35} - 35 = 4 = 2^{30} \sim 379\,821\,956$ DPs, but ended up needing 968,531,433.

Computations ran on 64 to 576 FPGAs in parallel.

DLs in intervals

Want to use knowledge that DL is in a small interval $[a, b]$, much smaller than $\ell$.

We can use this in baby-step giant-step algorithm.

How to use this in a memory-less algorithm?

Standard interval method: Pollard’s kangaroo method.

Pollard’s kangaroos do small jumps around the interval.
DLs in intervals

Want to use knowledge that DL is in a small interval \([a, b]\), much smaller than \(l\).

We can use this in baby-step giant-step algorithm.

How to use this in a memory-less algorithm?

Standard interval method: Pollard’s kangaroo method.

Pollard’s kangaroos do small jumps around the interval.
Want to use knowledge that DL is in a small interval $[a, b]$, much smaller than $\ell$.

We can use this in baby-step giant-step algorithm.

How to use this in a memory-less algorithm?

Standard interval method: Pollard’s kangaroo method.

Pollard’s kangaroos do small jumps around the interval.

Real kangaroos sleep
DLs in intervals

Want to use knowledge that DL is in a small interval $[a, b]$, much smaller than $\ell$.

We can use this in baby-step giant-step algorithm.

How to use this in a memory-less algorithm?

Standard interval method: Pollard’s kangaroo method.

Pollard’s kangaroos do small jumps around the interval.

Real kangaroos sleep

(at least outside Australia).
Intervals

Want to use knowledge that DL is in a small interval $(a, b)$, much smaller than $\ell$.

We can use this in baby-step giant-step algorithm.

How to use this in a memory-less algorithm?

Standard interval method:
Pollard’s kangaroo method.

Pollard’s kangaroos do small jumps around the interval.

Real kangaroos sleep (at least outside Australia).
Want to use knowledge that DL is in a small interval \([a;b]\), much smaller than \(\ell\).
We can use this in baby-step giant-step algorithm.

How to use this in a memory-less algorithm?

Standard interval method: Pollard’s kangaroo method.
Pollard’s kangaroos do small jumps around the interval.

Real kangaroos sleep (at least outside Australia).
Standard interval method: Pollard’s kangaroo method.

Pollard’s kangaroos do small jumps around the interval.

Real kangaroos sleep (at least outside Australia).

Kangaroo method in Australia

Main actor:
Standard interval method:
Pollard’s kangaroo method.

Pollard’s kangaroos do small jumps around the interval.

Real kangaroos sleep (at least outside Australia).

Kangaroo method in Australia
Main actor:
Standard interval method: Pollard’s kangaroo method.

Pollard’s kangaroos do small jumps around the interval.

Real kangaroos sleep

The tame kangaroo starts at a known multiple of $P$, e.g. $bP$.

(at least outside Australia).
Standard interval method: Pollard’s kangaroo method.
Pollard’s kangaroos do small jumps around the interval.
Real kangaroos sleep (at least outside Australia).

The tame kangaroo jumps. Jumps are determined by current position.
Standard interval method: Pollard’s kangaroo method.

Pollard’s kangaroos do small jumps around the interval.

Real kangaroos sleep (at least outside Australia).

The tame kangaroo jumps.

Jumps are determined by current position. Average jump distance is $\sqrt{b - a}$. 
Standard interval method: Pollard’s kangaroo method.

Pollard’s kangaroos do small jumps around the interval.

Real kangaroos sleep (at least outside Australia).

The tame kangaroo jumps.

Jumps are determined by current position. Average jump distance is $\sqrt{b - a}$. 
Standard interval method: Pollard’s kangaroo method.

Pollard’s kangaroos do small jumps around the interval.

Real kangaroos sleep (at least outside Australia).

The tame kangaroo jumps.

Jumps are determined by current position. Average jump distance is $\sqrt{b-a}$. 
Standard interval method: Pollard’s kangaroo method.

Pollard’s kangaroos do small jumps around the interval.

Real kangaroos sleep (at least outside Australia).

The tame kangaroo jumps.

Jumps are determined by current position. Average jump distance is $\sqrt{b-a}$.
Standard interval method: Pollard's kangaroo method. Pollard's kangaroos do small jumps around the interval. Real kangaroos sleep (at least outside Australia).

The tame kangaroo jumps. Jumps are determined by current position. Average jump distance is $\sqrt{b-a}$.

The tame kangaroo stops after a fixed number of jumps (about $\sqrt{b-a}$ many). The tame kangaroo installs a trap and waits.
Standard interval method:
Pollard’s kangaroo method.
Pollard’s kangaroos do small
jumps around the interval.
Real kangaroos sleep
(at least outside Australia).

The tame kangaroo jumps.
Jumps are determined
by current position.
Average jump distance
is $\sqrt{b-a}$.

The tame kangaroo stops
after a fixed number of jumps
(about $\sqrt{b-a}$ many).
The tame kangaroo installs a trap
and waits.
The tame kangaroo jumps.

Jumps are determined by current position.
Average jump distance is $\sqrt{b-a}$.

The tame kangaroo stops after a fixed number of jumps (about $\sqrt{b-a}$ many).

The tame kangaroo installs a trap and waits.
The tame kangaroo jumps.

Jumps are determined by current position.

Average jump distance is $\sqrt{b-a}$.

The tame kangaroo stops after a fixed number of jumps (about $\sqrt{b-a}$ many).

The tame kangaroo installs a trap and waits.

The wild kangaroo starts at point $Q$.

Follows the same instructions for jumps.

The wild kangaroo starts at $Q$. Follows the same instructions for jumps.
The tame kangaroo jumps. Jumps are determined by current position. Average jump distance is $\sqrt{b - a}$.

The tame kangaroo stops after a fixed number of jumps (about $\sqrt{b - a}$ many). The tame kangaroo installs a trap and waits.

The wild kangaroo starts at point $Q$. Follows the same instructions for jumps.
The tame kangaroo jumps. Jumps are determined by current position. Average jump distance is $\sqrt{b - a}$.

The tame kangaroo stops after a fixed number of jumps (about $\sqrt{b - a}$ many). The tame kangaroo installs a trap and waits.

The wild kangaroo starts at point $Q$. Follows the same instructions for jumps.
The tame kangaroo stops after a fixed number of jumps (about $\sqrt{b-a}$ many).

The tame kangaroo installs a trap and waits.

The wild kangaroo starts at point $Q$. Follows the same instructions for jumps.
The tame kangaroo stops after a fixed number of jumps (about $\sqrt{b - a}$ many).
The tame kangaroo installs a trap and waits.

The wild kangaroo starts at point $Q$.
Follows the same instructions for jumps.

But we don't know where the starting point $Q$ is.

Know $Q = nP$ with $n \in [a; b]$.

Hope that the paths of the tame and wild kangaroos intersect.

Similar to the rho method the kangaroos will hop on the same path from that point onwards.

Eventually the wild kangaroo falls into the trap.
(Or disappears in the distance if paths have not intersected.
Start a fresh one from $Q + P; Q + 2P; \ldots$.)

But we don't know where the starting point $Q$ is.

Know $Q = nP$ with $n \in [a; b]$.

Hope that the paths of the tame and wild kangaroos intersect.

Similar to the rho method the kangaroos will hop on the same path from that point onwards.

Eventually the wild kangaroo falls into the trap.
(Or disappears in the distance if paths have not intersected.
Start a fresh one from $Q + P; Q + 2P; \ldots$.)
The tame kangaroo stops after a fixed number of jumps (√b − a many).
The tame kangaroo installs a trap and waits.

But we don’t know where the starting point Q is.
Know Q = nP with n ∈ [a; b].
Hope that the paths of the tame and wild kangaroos intersect.

Similar to the rho method the kangaroos will hop on the same path from that point onwards.
Eventually the wild kangaroo falls into the trap. (Or disappears in the distance if paths have not intersected.
Start a fresh one from Q + P, Q + 2P, . . . .)
The wild kangaroo

starts at point \( Q \).
Follows the same instructions for jumps.

But we don’t know where the starting point \( Q \) is.
Know \( Q = nP \) with \( n \in [a, b] \).

Hope that the paths of the tame and wild kangaroo intersect.

Similar to the rho method the kangaroos will hop on the same path from that point onwards.

Eventually the wild kangaroo falls into the trap.
(Or disappears in the distance if paths have not intersected.
Start a fresh one from \( Q + P, Q + 2P, \ldots \)).
The wild kangaroo starts at point $Q$. Follows the same instructions for jumps.

But we don’t know where the starting point $Q$ is. Know $Q = nP$ with $n \in [a, b]$.

Hope that the paths of the tame and wild kangaroo intersect.

Similar to the rho method the kangaroos will hop on the same path from that point onwards.

Eventually the wild kangaroo falls into the trap.
(Or disappears in the distance if paths have not intersected.
Start a fresh one from $Q + P, Q + 2P, \ldots$)
But we don’t know where the starting point $Q$ is. Know $Q = nP$ with $n \in [a, b]$. Hope that the paths of the tame and wild kangaroo intersect.

Similar to the rho method the kangaroos will hop on the same path from that point onwards.

Eventually the wild kangaroo falls into the trap. (Or disappears in the distance if paths have not intersected.
Start a fresh one from $Q + P, Q + 2P, \ldots$)
But we don’t know where the starting point $Q$ is. Know $Q = nP$ with $n \in [a, b]$.

Hope that the paths of the tame and wild kangaroo intersect.

Similar to the rho method the kangaroos will hop on the same path from that point onwards.

Eventually the wild kangaroo falls into the trap. (Or disappears in the distance if paths have not intersected. Start a fresh one from $Q + P, Q + 2P, \ldots$)

Same story in math

Kangaroo = sequence $X_i \in \langle P \rangle$.

Starting point $X_0 = s_0 P$.

Distance $d_0 = 0$.

Step set: $S = \{s_1 P, \ldots, s_L P\}$ with $s_i$ on average $s = \beta \sqrt{b - a}$.

Hash function $H : \langle P \rangle \rightarrow \{1, 2, \ldots\}$.

Update function

$$d_{i+1} = d_i + s H(X_i)$$

$$X_{i+1} = X_i + s H(X_i)$$
But we don’t know where the starting point $Q$ is.
Know $Q = nP$ with $n \in [a, b]$.

Hope that the paths of the tame and wild kangaroo intersect.

Similar to the rho method the kangaroos will hop on the same path from that point onwards.

Eventually the wild kangaroo falls into the trap.
(Or disappears in the distance if paths have not intersected.
Start a fresh one from $Q + P, Q + 2P, \ldots$)

Same story in math

Kangaroo = sequence $X_i \in \langle P \rangle$.
Starting point $X_0 = s_0P$.
Distance $d_0 = 0$.
Step set: $S = \{s_1P, \ldots, s_L P\}$ with $s_i$ on average $s = \beta \sqrt{b-a}$.
Hash function $H : \langle P \rangle \rightarrow \{1, 2, \ldots, L\}$.
Update function $d_{i+1} = d_i + s_{H(X_i)}, \quad i = 0, 1, 2, \ldots$,
$X_{i+1} = X_i + s_{H(X_i)}P, \quad i = 0, 1, 2, \ldots$.
But we don’t know where the starting point $Q$ is. Know $Q = nP$ with $n \in [a, b]$.

Hope that the paths of the tame and wild kangaroo intersect.

Similar to the rho method the kangaroos will hop on the same path from that point onwards.

Eventually the wild kangaroo falls into the trap. (Or disappears in the distance if paths have not intersected. Start a fresh one from $Q + P, Q + 2P, \ldots$)

Same story in math

Kangaroo = sequence $X_i \in \langle P \rangle$.
Starting point $X_0 = s_0P$.
Distance $d_0 = 0$.
Step set: $S = \{s_1P, \ldots, s_LP\}$, with $s_i$ on average $s = \beta\sqrt{b-a}$.
Hash function $H : \langle P \rangle \to \{1, 2, \ldots, L\}$.
Update function

\[
d_{i+1} = d_i + s_{H(X_i)}, \quad i = 0, 1, 2, \ldots,
\]

\[
X_{i+1} = X_i + s_{H(X_i)}P, \quad i = 0, 1, 2, \ldots.
\]
But we don’t know where the starting point $Q$ is. Know $Q = nP$ with $n \in [a, b]$. Hope that the paths of the tame and wild kangaroo intersect. Similar to the rho method the kangaroos will hop on the same path from that point onwards. Eventually the wild kangaroo falls into the trap. (Or disappears in the distance if paths have not intersected. Start a fresh one $Q + P, Q + 2P, \ldots$)

Same story in math

Kangaroo = sequence $X_i \in \langle P \rangle$.
Starting point $X_0 = s_0P$.
Distance $d_0 = 0$.
Step set: $S = \{s_1P, \ldots, s_LP\}$, with $s_i$ on average $s = \beta \sqrt{b-a}$.
Hash function $H : \langle P \rangle \to \{1, 2, \ldots, L\}$.
Update function

$X'_0 = Q = nP$.
Trap: distance $d_N$.
Endpoint $X_N = (b + d_N)P$.

Picture credit: Christine van Vredendaal.
But we don't know where
the starting point Q is.

Know \( Q = nP \) with \( n \in [a, b] \).

Hope that the paths of the tame
and wild kangaroo intersect.

Similar to the rho method the
kangaroos will hop on the same
path from that point onwards.

Eventually the wild kangaroo falls
into the trap.

(Or disappears in the distance if
paths have not intersected.
Start a fresh one
from \( Q + P; Q + 2P; \ldots \).)

Same story in math

Kangaroo = sequence \( X_i \in \langle P \rangle \).
Starting point \( X_0 = s_0 P \).
Distance \( d_0 = 0 \).
Step set: \( S = \{s_1 P, \ldots, s_L P\} \),
with \( s_i \) on average
\( s = \beta \sqrt{b - a} \).
Hash function
\( H : \langle P \rangle \rightarrow \{1, 2, \ldots, L\} \).
Update function
\( d_{i+1} = d_i + s_H(X_i), \quad i = 0, 1, 2, \ldots \),
\( X_{i+1} = X_i + s_H(X_i) P, \quad i = 0, 1, 2, \ldots \).

Tame kangaroo starts at
\( X_0 = bP \),
wild kangaroo starts at
\( X'_0 = Q = nP \).
Trap: distance \( d_N \),
endpoint \( X_N = (b + d_N) P \).

Picture credit:
Christine van Vredendaal.
But we don't know where the starting point is.

Know \( Q = nP \) with \( n \in \left[ a; b \right] \).

Hope that the paths of the tame and wild kangaroo intersect. Similar to the rho method the kangaroos will hop on the same path from that point onwards. Eventually the wild kangaroo falls into the trap. (Or disappears in the distance if paths have not intersected. Start a fresh one from \( Q + P; Q + 2 P; \ldots \)).

Same story in math

Kangaroo = sequence \( X_i \in \langle P \rangle \).
Starting point \( X_0 = s_0P \).
Distance \( d_0 = 0 \).
Step set: \( S = \{s_1P, \ldots, s_LP\} \), with \( s_i \) on average
\( s = \beta \sqrt{b - a} \).
Hash function
\( H : \langle P \rangle \to \{1, 2, \ldots, L\} \).
Update function
\( d_{i+1} = d_i + s H(X_i), \quad i = 0, 1, 2, \ldots \),
\( X_{i+1} = X_i + s H(X_i)P, \quad i = 0, 1, 2, \ldots \).

Tame kangaroo starts at \( X_0 = bP \),
wild kangaroo starts at \( X'_0 = Q = nP \).
Trap: distance \( d_N \),
endpoint \( X_N = (b + d_N)P \).

Picture credit:
Christine van Vredendaal.
Same story in math

Kangaroo = sequence $X_i \in \langle P \rangle$.
Starting point $X_0 = s_0 P$.
Distance $d_0 = 0$.
Step set: $S = \{s_1 P, \ldots, s_L P\}$, with $s_i$ on average $s = \beta \sqrt{b - a}$.

Hash function $H : \langle P \rangle \to \{1, 2, \ldots, L\}$.

Update function $d_{i+1} = d_i + s_H(X_i), \quad i = 0, 1, 2, \ldots$,
$X_{i+1} = X_i + s_H(X_i) P, \quad i = 0, 1, 2, \ldots$.

Tame kangaroo starts at $X_0 = bP$,
wild kangaroo starts at $X'_0 = Q = nP$.
Trap: distance $d_N$,
endpoint $X_N = (b + d_N) P$.

Picture credit: Christine van Vredendaal.
Tame kangaroo starts at $X_0 = bP$, wild kangaroo starts at $X'_0 = Q = nP$. Trap: distance $d_N$, endpoint $X_N = (b + d_N)P$.

Parallel kangaroo method
Use an entire herd of tame kangaroos, all starting around $((b - a) = 2)P$.

Picture credit: Christine van Vredendaal.
Same story in math
Kangaroo = sequence $X_i \in \langle P \rangle$.
Starting point $X_0 = s_0P$.
Distance $d_0 = 0$.
Step set: $S = \{ s_1P ; \ldots ; s_LP \}$,
with $s_i$ on average 
$s = \frac{1}{\sqrt{b-a}}$.
Hash function $H$: $\langle P \rangle \rightarrow \{ 1 ; \ldots ; L \}$.
Update function $d_{i+1} = d_i + s\ H(X_i)$; $i = 0 ; 1 ; 2 ; \ldots$,
$X_{i+1} = X_i + s\ H(X_i)P$; $i = 0 ; 1 ; 2 ; \ldots$.

Tame kangaroo starts at $X_0 = bP$,
wild kangaroo starts at $X_0' = Q = nP$.
Trap: distance $d_N$,
endpoint $X_N = (b + d_N)P$.

Parallel kangaroo method
Use an entire herd
of tame kangaroos, all starting around $((b-a)/2)P$.
Tame kangaroo starts at
\[ X_0 = bP, \]
wild kangaroo starts at
\[ X'_0 = Q = nP. \]
Trap: distance \( d_N \),
endpoint \( X_N = (b + d_N)P \).

Parallel kangaroo method
Use an entire herd
of tame kangaroos,
all starting
around \(((b - a)/2)P \ldots\)

Picture credit: Christine van Vredendaal.
Tame kangaroo starts at 
\[ X_0 = bP, \]
wild kangaroo starts at 
\[ X'_0 = Q = nP. \]
Trap: distance \( d_N \),
endpoint \( X_N = (b + d_N)P. \)

Picture credit: Christine van Vredendaal.

Parallel kangaroo method

Use an entire herd

of tame kangaroos,
all starting
around \( ((b - a)/2)P \)
Parallel kangaroo method

Use an entire herd of tame kangaroos, all starting around \( ((b - a)/2)P \). ...

and define certain spots as distinguished points.

Also start a herd of wild kangaroos around \( Q \).

Hope that one wild and one tame kangaroo meet at one distinguished point.
Parallel kangaroo method

Use an entire herd of tame kangaroos, all starting around \((b-a)/2)P\) ... and define certain spots as distinguished points. Also start a herd of wild kangaroos around \(Q\). Hope that one wild and one tame kangaroo meet at one distinguished point.

Tame kangaroo starts at \(X_0 = bP\), wild kangaroo starts at \(X'_0 = Q = nP\). Trap: distance \(d_N\), endpoint \(X_N = (b + d_N)P\).

Picture credit: Christine van Vredendaal.
Parallel kangaroo method
Use an entire herd of tame kangaroos, all starting around \((b - a)/2\)P ...

... and define certain spots as distinguished points...

Also start a herd of wild kangaroos around Q. Hope that one wild and one tame kangaroo meet at one distinguished point.
Parallel kangaroo method

Use an entire herd of tame kangaroos, all starting around \((b - a) = 2\).

...and define certain spots as distinguished points.

Also start a herd of wild kangaroos around \(Q\). Hope that one wild and one tame kangaroo meet at one distinguished point.
Parallel kangaroo method

Use an entire herd of tame kangaroos, all starting around $(b - a) = 2$.

... and define certain spots as distinguished points.

Also start a herd of wild kangaroos around $Q$.
Hope that one wild and one tame kangaroo meet at one distinguished point.

Pairings

Let $(G_1, +)$, $(G_2, +)$, and $(G_T, \cdot)$ be groups of prime order $\prime$ and let $e : G_1 \times G_2 \to G_T$ be a map satisfying $e(P + Q, R) = e(P, R)e(Q, R)$.

Request further that $e$ is non-degenerate in the first argument, i.e., if for some $P$ $e(P, R') = 1$ for all $R' \in G_2$, then $P$ is the identity in $G_1$.

Such an $e$ is called a bilinear map or pairing.
Parallel kangaroo method

Use an entire herd of tame kangaroos, all starting around 

\[ (b - a) = 2 \]

... and define certain spots as distinguished points

Also start a herd of wild kangaroos around \( Q \). Hope that one wild and one tame kangaroo meet at one distinguished point.

Pairings

Let \((G_1, +), (G_2, +), (G_T, \cdot)\) be groups of prime order \( ' \) and let \( e : G_1 \times G_2 \rightarrow G_T \) be a map satisfying:

\[
e(P + Q, R') = e(P, R') e(Q, R')
\]

\[
e(P, R' + S') = e(P, R') e(P, S')
\]

Request further that \( e \) is non-degenerate in the first argument, i.e., if for some \( P \) \( e(P, R') = 1 \) for all \( R' \in G_2 \), then \( P \) is the identity in \( G_1 \).

Such an \( e \) is called a bilinear map or pairing.
Parallel kangaroo method

Use an entire herd of tame kangaroos, all starting around \((b - a) = 2\).

... and define certain spots as distinguished points.

Also start a herd of wild kangaroos around \(Q\).
Hope that one wild and one tame kangaroo meet at one distinguished point.

Pairings

Let \((G_1, +), (G_2, +)\) and \((G_T, \cdot)\) be groups of prime order \(\ell\) and let \(e: G_1 \times G_2 \to G_T\) be a map satisfying

\[
e(P + Q, R') = e(P, R')e(Q, R')
\]
\[
e(P, R' + S') = e(P, R')e(P, S')
\]

Request further that \(e\) is non-degenerate in the first argument, i.e., if for some \(P\),
\[
e(P, R') = 1 \text{ for all } R' \in G_2,
\]
then \(P\) is the identity in \(G_1\).

Such an \(e\) is called a bilinear map or pairing.
...and define certain spots as distinguished points

Also start a herd of wild kangaroos around $Q$. Hope that one wild and one tame kangaroo meet at one distinguished point.

**Pairings**

Let $(G_1, +), (G_2, +)$ and $(G_T, \cdot)$ be groups of prime order $\ell$ and let $e : G_1 \times G_2 \to G_T$ be a map satisfying

\[
e(P + Q, R') = e(P, R')e(Q, R'),\\
e(P, R' + S') = e(P, R')e(P, S').
\]

Request further that $e$ is non-degenerate in the first argument, i.e., if for some $P$ \( e(P, R') = 1 \) for all $R' \in G_2$, then $P$ is the identity in $G_1$

Such an $e$ is called a *bilinear map* or *pairing*. 
Define certain spots as distinguished points.

Also start a herd of wild kangaroos around $Q$. Hope that one wild and one tame kangaroo meet at one distinguished point.

Pairings

Let $(G_1, +), (G_2, +)$ and $(G_T, \cdot)$ be groups of prime order $\ell$ and let $e : G_1 \times G_2 \to G_T$ be a map satisfying
\[ e(P + Q, R') = e(P, R')e(Q, R'), \]
\[ e(P, R' + S') = e(P, R')e(P, S'). \]
Request further that $e$ is non-degenerate in the first argument, i.e., if for some $P$ $e(P, R') = 1$ for all $R' \in G_2$, then $P$ is the identity in $G_1$.

Such an $e$ is called a bilinear map or pairing.

Consequences of pairings

Assume that $G_1 = G_2$, in particular $e(P, P) \neq 1$.
Then for all triples $(P_1, P_2, P_3)$ one can decide in time polynomial in $\log \ell$ whether
\[ \log_{P_1}(P_3) = \log_{P_2}(P_3) \]
by comparing $e(P_1, P_2)$ and $e(P_1, P_3)$. This means that the decisional Diffie-Hellman problem is easy.
Pairings

Let \((G_1, +), (G_2, +)\) and \((G_T, \cdot)\) be groups of prime order \(\ell\) and let \(e: G_1 \times G_2 \to G_T\) be a map satisfying
\[
e(P + Q, R') = e(P, R')e(Q, R'),
\]
\[
e(P, R' + S') = e(P, R')e(P, S').
\]
Request further that \(e\) is non-degenerate in the first argument, i.e., if for some \(P\)
\[
e(P, R') = 1\] for all \(R' \in G_2\), then \(P\) is the identity in \(G_1\).

Such an \(e\) is called a \textit{bilinear map} or \textit{pairing}.

Consequences of pairings

Assume that \(G_1 = G_2\), in particular \(e(P, P) \neq 1\):

Then for all triples \((P_1, P_2, P_3) \in \langle P \rangle^3\), one can decide in time polynomial in \(\log \ell\) whether
\[
\log_P(P_3) = \log_P(P_1) \cdot \log_P(P_2)
\] by comparing \(e(P_1, P_2)\) and \(e(P, P_3)\). This means that the Decisional Diffie-Hellman problem is easy.
Pairings

Let $(G_1, +), (G_2, +)$ and $(G_T, \cdot)$ be groups of prime order $\ell$ and let $e : G_1 \times G_2 \to G_T$ be a map satisfying

\[ e(P + Q, R') = e(P, R')e(Q, R'), \]
\[ e(P, R' + S') = e(P, R')e(P, S'). \]

Request further that $e$ is non-degenerate in the first argument, i.e., if for some $P$ $e(P, R') = 1$ for all $R' \in G_2$, then $P$ is the identity in $G_1$.

Such an $e$ is called a bilinear map or pairing.

Consequences of pairings

Assume that $G_1 = G_2$, in particular $e(P, P) \neq 1$.

Then for all triples $(P_1, P_2, P_3) \in \langle P \rangle^3$ one can decide in time polynomial in $\log \ell$ whether $\log_P(P_3) = \log_P(P_1) \log_P(P_2)$ by comparing $e(P_1, P_2)$ and $e(P, P_3)$.

This means that the decisional Diffie-Hellman problem is easy.
Pairings

Let \((G_1, +), (G_2, +)\) and \((G_T, \cdot)\) be groups of prime order \(\ell\) and let \(e : G_1 \times G_2 \to G_T\) be a map satisfying
\[
e(P + Q, R') = e(P, R')e(Q, R'),
\]
\[
e(P, R' + S') = e(P, R')e(P, S').
\]
Request further that \(e\) is non-degenerate in the first argument, i.e., if for some \(P\)
\[
e(P, R') = 1 \text{ for all } R' \in G_2,
\]
then \(P\) is the identity in \(G_1\).

Such an \(e\) is called a \textit{bilinear map} or \textit{pairing}.

Consequences of pairings

Assume that \(G_1 = G_2\), in particular \(e(P, P) \neq 1\).

Then for all triples \((P_1, P_2, P_3) \in \langle P \rangle^3\)
one can decide in time polynomial in \(\log \ell\) whether
\[
\log_P(P_3) = \log_P(P_1) \log_P(P_2)
\]
by comparing
\[
e(P_1, P_2) \text{ and } e(P, P_3).
\]
This means that the decisional Diffie-Hellman problem is easy.
Pairings

Let \((G_1, +)\), \((G_2, +)\) and \((G_T, \cdot)\) be groups of prime order \(\ell\) and let \(e : G_1 \times G_2 \to G_T\) be a map satisfying

\[
e(P + Q, R') = e(P, R') e(Q, R') ,
\]

\[
e(P, R' + S') = e(P, R') e(P, S') .
\]

Suppose further that \(e\) is non-degenerate in the first argument, i.e., if for some \(P\)

\[
e(P, R') = 1 \quad \text{for all } R' \in G_2 ,
\]

then \(P\) is the identity in \(G_1\).

Such an \(e\) is called a bilinear map or pairing.

Consequences of pairings

Assume that \(G_1 = G_2\), in particular \(e(P, P) \neq 1\).

Then for all triples \((P_1, P_2, P_3) \in \langle P \rangle^3\)

one can decide in time polynomial in \(\log \ell\) whether

\[
\log_P(P_3) = \log_P(P_1) \log_P(P_2)
\]

by comparing

\[e(P_1, P_2)\] and \(e(P, P_3)\).

This means that the decisional Diffie-Hellman problem is easy.

The DL system \(G_1\) is at most as secure as the system \(G_T\).

Even if \(G_1 \neq G_2\) one can transfer the DLP in \(G_1\) to a DLP in \(G_T\), provided one can find an element \(P' \in G_2\) such that the map

\[P \mapsto e(P, P')\]

are interesting attack tool if DLP in \(G_T\) is easier to solve; e.g. if \(G_T\) has index calculus attacks.

Pairings

Assume that \(G_1 = G_2\), in particular \(e(P, P) \neq 1\).

Then for all triples \((P_1, P_2, P_3) \in \langle P \rangle^3\)

one can decide in time polynomial in \(\log \ell\) whether

\[
\log_P(P_3) = \log_P(P_1) \log_P(P_2)
\]

by comparing

\[e(P_1, P_2)\] and \(e(P, P_3)\).

This means that the decisional Diffie-Hellman problem is easy.

The DL system \(G_1\) is at most as secure as the system \(G_T\).

Even if \(G_1 \neq G_2\) one can transfer the DLP in \(G_1\) to a DLP in \(G_T\), provided one can find an element \(P' \in G_2\) such that the map

\[P \mapsto e(P, P')\]

are interesting attack tool if DLP in \(G_T\) is easier to solve; e.g. if \(G_T\) has index calculus attacks.
Pairings

Let \((G_1, +)\) and \((G_2, +)\) be groups of prime order \(l\) and let \(e : G_1 \times G_2 \rightarrow G_T\) be a map satisfying
\[
e(P + Q, R') = e(P, R') e(Q, R'), \quad \text{and} \quad e(P, R' + S') = e(P, R') e(P, S').
\]

Request further that \(e\) is non-degenerate in the first argument, i.e., if for some \(P\)
\[
e(P, R') = 1 \quad \text{for all } R' \in G_2,
\]
then \(P\) is the identity in \(G_1\). Such an \(e\) is called a bilinear map or pairing.

Consequences of pairings

Assume that \(G_1 = G_2\), in particular \(e(P, P) \neq 1\).

Then for all triples \((P_1, P_2, P_3) \in \langle P \rangle^3\)
\[
one can decide in time polynomial in \(\log l\) whether
\[
\log_P (P_3) = \log_P (P_1) \log_P (P_2)
\]
by comparing
\[
e(P_1, P_2) \text{ and } e(P, P_3).
\]
This means that the decisional Diffie-Hellman problem is easy.

The DL system \(G_1\) is at most as secure as the system \(G_T\).

Even if \(G_1 \neq G_2\) one can transfer the DLP in \(G_1\) to a DLP in \(G_T\), provided one can find an element \(P' \in G_2\) such that the map \(P \rightarrow e(P, P')\) is injective.

Pairings are interesting attack tool if DLP in \(G_T\) is easier to solve; e.g. if \(G_T\) has index calculus attacks.
Consequences of pairings

Assume that $G_1 = G_2$, in particular $e(P, P) \neq 1$.

Then for all triples $(P_1, P_2, P_3) \in \langle P \rangle^3$ one can decide in time polynomial in $\log \ell$ whether

$$\log_P(P_3) = \log_P(P_1) \log_P(P_2)$$

by comparing $e(P_1, P_2)$ and $e(P, P_3)$.

This means that the decisional Diffie-Hellman problem is easy.

The DL system $G_1$ is at most as secure as the system $G_T$.

Even if $G_1 \neq G_2$ one can transfer the DLP in $G_1$ to a DLP in $G_T$, provided one can find an element $P' \in G_2$ such that the map $P \rightarrow e(P, P')$ is injective.

Pairings are interesting attack tool if DLP in $G_T$ is easier to solve; e.g. if $G_T$ has index calculus attacks.

The DL system $G_1$ is at most as secure as the system $G_T$.
Consequences of pairings

Assume that $G_1 = G_2$, in particular $e(P, P) \neq 1$.

Then for all triples $(P_1, P_2, P_3) \in \langle P \rangle^3$ one can decide in time polynomial in $\log \ell$ whether

$$ \log_P(P_3) = \log_P(P_1) \log_P(P_2) $$

by comparing $e(P_1, P_2)$ and $e(P, P_3)$.

This means that the decisional Diffie-Hellman problem is easy.

The DL system $G_1$ is at most as secure as the system $G_T$.

Even if $G_1 \neq G_2$ one can transfer the DLP in $G_1$ to a DLP in $G_T$, provided one can find an element $P' \in G_2$ such that the map $P \mapsto e(P, P')$ is injective.

Pairings are interesting attack tool if DLP in $G_T$ is easier to solve; e.g. if $G_T$ has index calculus attacks.
Consequences of pairings

Assume that \( G_1 = G_2 \), in particular \( e(P, P) \neq 1 \).

Then for all triples \((P_1; P_2; P_3) \in \langle P \rangle^3\) one can decide in time polynomial whether

\[ \log_P(P_1) \log_P(P_2) = \log_P(P_3) \]

by comparing \( e(P_1; P_2) \) and \( e(P; P_3) \). This means that the decisional Diffie-Hellman problem is easy.

The DL system \( G_1 \) is at most as secure as the system \( G_T \).

Even if \( G_1 \neq G_2 \) one can transfer the DLP in \( G_1 \) to a DLP in \( G_T \), provided one can find an element \( P' \in G_2 \) such that the map \( P \to e(P, P') \) is injective.

Pairings are interesting attack tool if DLP in \( G_T \) is easier to solve; e.g. if \( G_T \) has index calculus attacks.

We want to define pairings \( G_1 \times G_2 \to G_T \) preserving the group structure.

The pairings we will use map to the multiplicative group of a finite extension field \( \mathbb{F}_{q^k} \).

More precisely, \( G_T \subset \mathbb{F}_{q^k} \), order \( \epsilon \).

To embed the points of order \( \epsilon \) into \( \mathbb{F}_{q^k} \) there need to be \( \epsilon \)-th roots of unity are in \( \mathbb{F}^*_q \).

The embedding degree \( k \) satisfies \( k \) is minimal with \( \epsilon \mid q^k - 1 \).
We want to define pairings $G_1 \times G_2 \rightarrow G_T$ preserving the group structure. The pairings we will use map to the multiplicative group of a finite extension field $F_{q^k}$. More precisely, $G_T \subset F_{q^k}$, order $e$.

To embed the points of order $e$ into $F_{q^k}$ there need to be $e$-th roots of unity in $F_{q^k}^*$. The embedding degree $k$ satisfies $k$ is minimal with $e \mid q^k - 1$.

The decisional Diffie-Hellman problem is easy. Even if $G_1 \neq G_2$ one can transfer the DLP in $G_1$ to a DLP in $G_T$, provided one can find an element $P' \in G_2$ such that the map $P \rightarrow e(P, P')$ is injective.

Pairings are interesting attack tool if DLP in $G_T$ is easier to solve; e.g. if $G_T$ has index calculus attacks.

The DL system $G_1$ is at most as secure as the system $G_T$. Even if $G_1 \neq G_2$ one can decide in time polynomial in $\log' \text{log} P_1 \text{log} P_2 \text{log} P_3$ whether $e(P_1, P_2) = e(P, P_3)$ by comparing $e(P_1, P_2)$ and $e(P, P_3)$. This means that the decisional Diffie-Hellman problem is easy.
The DL system $G_1$ is at most as secure as the system $G_T$.

Even if $G_1 \neq G_2$ one can transfer the DLP in $G_1$ to a DLP in $G_T$, provided one can find an element $P' \in G_2$ such that the map $P \rightarrow e(P, P')$ is injective.

Pairings are interesting attack tool if DLP in $G_T$ is easier to solve; e.g. if $G_T$ has index calculus attacks.

We want to define pairings $G_1 \times G_2 \rightarrow G_T$ preserving the group structure.

The pairings we will use map to the multiplicative group of a finite extension field $\mathbb{F}_{q^k}$. More precisely, $G_T \subset \mathbb{F}_{q^k}$.

To embed the points of order $\ell$ into $\mathbb{F}_{q^k}$ there need to be $\ell$-th roots of unity are in $\mathbb{F}_{q^k}^*$.

The embedding degree $k$ satisfies $k$ is minimal with $\ell \mid q^k - 1$. 

- Polynomial
- Theorem
- Consequences
- Security
The DL system $G_1$ is at most as secure as the system $G_T$.

Even if $G_1 \neq G_2$ one can transfer the DLP in $G_1$ to a DLP in $G_T$, provided one can find an element $P' \in G_2$ such that the map $P \rightarrow e(P, P')$ is injective.

Pairings are interesting attack tool if DLP in $G_T$ is easier to solve; e.g. if $G_T$ has index calculus attacks.

We want to define pairings $G_1 \times G_2 \rightarrow G_T$ preserving the group structure.

The pairings we will use map to the multiplicative group of a finite extension field $\mathbf{F}_{q^k}$.
More precisely, $G_T \subset \mathbf{F}_{q^k}$, order $\ell$.

To embed the points of order $\ell$ into $\mathbf{F}_{q^k}$ there need to be $\ell$-th roots of unity are in $\mathbf{F}_{q^k}^\ast$.

The embedding degree $k$ satisfies $k$ is minimal with $\ell \mid q^k - 1$. 
system $G_1$ is at most as secure as the system $G_T$. 

If $G_1 \neq G_2$ one can transfer the DLP in $G_1$ to a DLP in $G_T$, provided one can find an element $P' \in G_2$ such that the map $P \mapsto e(P;P')$ is injective.

Pairings are interesting attack tools if DLP in $G_T$ is easier to solve; e.g. if $G_T$ has index calculus attacks.

We want to define pairings $G_1 \times G_2 \to G_T$ preserving the group structure.

The pairings we will use map to the multiplicative group of a finite extension field $\mathbb{F}_{q^k}$. More precisely, $G_T \subset \mathbb{F}_{q^k}$, order $\ell$.

To embed the points of order $\ell$ into $\mathbb{F}_{q^k}$ there need to be $\ell$-th roots of unity are in $\mathbb{F}_{q^k}^*$. The embedding degree $k$ satisfies $k$ is minimal with $\ell \mid q^k - 1$.

$E$ is supersingular if

\[ |E(\mathbb{F}_q)| = q + 1 - t, \]

\[ q = p^r, \]

it holds that $t \equiv 0 \mod p$.

Otherwise it is ordinary.

Example:

\[ y^2 + y = x^3 + a_4 x + a_6 \text{ over } \mathbb{F}_2. \]

It is supersingular: each $(x;y)$ point also gives $(x;y + 1) \neq (x;y)$. All points come in pairs, except for $\infty$, so $|E(\mathbb{F}_{2^r})| = 1 + \text{even}$, so $t \equiv 0 \mod 2$. 

\[ E \text{ is supersingular if } |E(\mathbb{F}_q)| = q + 1 - t, q = p^r, \]

it holds that $t \equiv 0 \mod p$. Otherwise it is ordinary.

Example:

\[ y^2 + y = x^3 + a_4 x + a_6 \text{ over } \mathbb{F}_2. \]

It is supersingular: each $(x;y)$ point also gives $(x;y + 1) \neq (x;y)$. All points come in pairs, except for $\infty$, so $|E(\mathbb{F}_{2^r})| = 1 + \text{even}$, so $t \equiv 0 \mod 2$. 


We want to define pairings $G_1 \times G_2 \to G_T$ preserving the group structure.

The pairings we will use map to the multiplicative group of a finite extension field $\mathbb{F}_{q^k}$.

More precisely, $G_T \subset \mathbb{F}_{q^k}$, order $\ell$.

To embed the points of order $\ell$ into $\mathbb{F}_{q^k}$ there need to be $\ell$-th roots of unity are in $\mathbb{F}_{q^k}^*$.

The embedding degree $k$ satisfies $k$ is minimal with $\ell \mid q^k - 1$.

$E$ is supersingular if for $|E(\mathbb{F}_q)| = q + 1 - t$, $q = p^r$, it holds that $t \equiv 0 \mod p$.

Otherwise it is ordinary.

Example:

$y^2 + y = x^3 + a_4 x + a_6$ over $\mathbb{F}_{2^r}$ is supersingular:

Each $(x, y)$ point $(x, y + 1) \neq (x, y)$.

All points come in pairs, except for $\infty$,

so $|E(\mathbb{F}_{2^r})| = 1 + \text{even}$,

so $t \equiv 0 \mod 2$. 

The DL system $G_1$ is at most as secure as the system $G_T$.

Even if $G_1 \neq G_2$ one can transfer the DLP in $G_1$ to a DLP in $G_T$, provided one can find an element $P' \in G_2$ such that the map $P \mapsto e(P; P')$ is injective.

Pairings are interesting attack tool if DLP in $G_T$ is easier to solve; e.g. if $G_T$ has index calculus attacks.

We want to define pairings $G_1 \times G_2 \to G_T$ preserving the group structure.

The pairings we will use map to the multiplicative group of a finite extension field $\mathbf{F}_{q^k}$.

More precisely, $G_T \subset \mathbf{F}_{q^k}$, order $\ell$.

To embed the points of order $\ell$ into $\mathbf{F}_{q^k}$ there need to be $\ell$-th roots of unity are in $\mathbf{F}_{q^k}^*$.

The embedding degree $k$ satisfies $k$ is minimal with $\ell \mid q^k - 1$.

$E$ is supersingular if for $|E(\mathbf{F}_q)| = q + 1 - t$, $q = p^r$, it holds that $t \equiv 0 \bmod p$.

Otherwise it is ordinary.

Example: $y^2 + y = x^3 + a_4x + a_6$ over $\mathbf{F}_{2^r}$ is supersingular:

Each $(x, y)$ point also gives $(x, y + 1) \neq (x, y)$.

All points come in pairs, except for $\infty$, so $|E(\mathbf{F}_{2^r})| = 1 + \text{even}$, so $t \equiv 0 \bmod 2$. 
We want to define pairings $G_1 \times G_2 \rightarrow G_T$ preserving the group structure.

The pairings we will use map to the multiplicative group of a finite extension field $\mathbf{F}_{q^k}$.

More precisely, $G_T \subset \mathbf{F}_{q^k}$, order $\ell$.

To embed the points of order $\ell$ into $\mathbf{F}_{q^k}$ there need to be $\ell$-th roots of unity are in $\mathbf{F}_{q^k}^*$.

The embedding degree $k$ satisfies $k$ is minimal with $\ell \mid q^k - 1$.

$E$ is supersingular if for $|E(\mathbf{F}_q)| = q + 1 - t$, $q = p^r$, it holds that $t \equiv 0 \mod p$.

Otherwise it is ordinary.

Example:
$y^2 + y = x^3 + a_4 x + a_6$ over $\mathbf{F}_{2^r}$ is supersingular:
Each $(x, y)$ point also gives $(x, y + 1) \neq (x, y)$.

All points come in pairs, except for $\infty$, so $|E(\mathbf{F}_{2^r})| = 1 + \text{even}$, so $t \equiv 0 \mod 2$. 

$E$ is supersingular if for $|E(\mathbf{F}_q)| = q + 1 - t$, $q = p^r$, it holds that $t \equiv 0 \mod p$.

Otherwise it is ordinary.

Example:
$y^2 + y = x^3 + a_4 x + a_6$ over $\mathbf{F}_{2^r}$ is supersingular:
Each $(x, y)$ point also gives $(x, y + 1) \neq (x, y)$.

All points come in pairs, except for $\infty$, so $|E(\mathbf{F}_{2^r})| = 1 + \text{even}$, so $t \equiv 0 \mod 2$. 

We want to define pairings $G_1 \times G_2 \to G_T$ preserving the group structure. The pairings we will use map to the multiplicative group of a finite extension field $\mathbb{F}_{q^k}$. More precisely, $G_T \subset \mathbb{F}_{q^k}$, order $\ell$.

To embed the points of order $\ell$ into $\mathbb{F}_{q^k}$ there need to be $\ell$-th roots of unity are in $\mathbb{F}_{q^k}^*$. The embedding degree $k$ satisfies $k$ is minimal with $\ell \mid q^k - 1$.

$E$ is supersingular if for $| E(\mathbb{F}_q) | = q + 1 - t$, $q = p^r$, it holds that $t \equiv 0 \mod p$.

Otherwise it is ordinary.

Example:
$y^2 + y = x^3 + a_4 x + a_6$ over $\mathbb{F}_{2^r}$ is supersingular:
Each $(x, y)$ point also gives $(x, y + 1) \neq (x, y)$.
All points come in pairs, except for $\infty$, so $| E(\mathbb{F}_{2^r}) | = 1 + \text{even}$, so $t \equiv 0 \mod 2$.

Embedding degrees
Let $E$ be supersingular and $q = p \geq 5$, i.e. $p > 2 \sqrt{p}$.
Hasse’s Theorem states $| t | \leq 2 \sqrt{p}$.
$E$ supersingular implies $t \equiv 0 \mod p$, so $t = 0$ and $| E(\mathbb{F}_p) | = p + 1$.

Obviously $(p + 1) \mid p^2 - 1 = (p + 1)(p - 1)$ so $k \leq 2$ for supersingular curves over prime fields.
We want to define pairings \( G_1 \times G_2 \rightarrow G_T \) preserving the group structure.

The pairings we will use map to the multiplicative group of a finite extension field \( \mathbb{F}_{q^k} \).

More precisely, \( G_T \subset \mathbb{F}_{q^k} \), order \( \ell \).

To embed the points of order \( \ell \) into \( \mathbb{F}_{q^k} \) there need to be \( \ell \)-th roots of unity are in \( \mathbb{F}^*_{q^k} \).

The embedding degree \( k \) satisfies \( k \) is minimal with \( \ell \mid q^k - 1 \).

\[ E \text{ is supersingular if} \]
for \( |E(\mathbb{F}_q)| = q + 1 - t \), \( q = p^r \),
it holds that \( t \equiv 0 \mod p \).

Otherwise it is ordinary.

Example:
\( y^2 + y = x^3 + a_4 x + a_6 \) over \( \mathbb{F}_{2^r} \)
is supersingular:
Each \((x, y)\) point also gives \((x, y + 1) \neq (x, y)\).
All points come in pairs, except for \( \infty \),
so \( |E(\mathbb{F}_{p^r})| = 1 + \text{even} \),
so \( t \equiv 0 \mod 2 \).

Embedding degrees

Let \( E \) be supersingular, \( q = p \geq 5 \), i.e \( p > 2 \sqrt{p} \).

Hasse’s Theorem states
\( |t| \leq 2 \sqrt{p} \).

\( E \) supersingular implies \( t \equiv 0 \mod p \), so \( t \equiv 0 \mod p \).

Obviously
\( (p + 1) \mid p^2 - 1 = (p + 1)(p - 1) \),
so \( k \leq 2 \) for supersingular curves over prime fields.
We want to define pairings $G_1 \times G_2 \to G_T$ preserving the group structure.

The pairings we will use map to the multiplicative group of a finite extension field $F_{q^k}$.

More precisely, $G_T \subset F_{q^k}$, order $'$. To embed the points of order $'$ into $F_{q^k}$ there need to be $'$-th roots of unity are in $F^*_{q^k}$.

The embedding degree $k$ satisfies $k$ is minimal with $' | q^k - 1$.

---

$E$ is supersingular if
for $| E(F_q) | = q + 1 - t$, $q = p^r$, it holds that $t \equiv 0 \text{ mod } p$.

Otherwise it is ordinary.

Example:
$y^2 + y = x^3 + a_4 x + a_6$ over $\mathbb{F}_{2^r}$ is supersingular:
Each $(x, y)$ point also gives $(x, y + 1) \neq (x, y)$.

All points come in pairs, except for $\infty$,
so $| E(\mathbb{F}_{2^r}) | = 1 + \text{even}$, so $t \equiv 0 \text{ mod } 2$.

---

Embedding degrees
Let $E$ be supersingular and $q = p \geq 5$, i.e $p > 2\sqrt{p}$.

Hasse's Theorem states $| t | \leq 2\sqrt{p}$.

$E$ supersingular implies $t \equiv 0 \text{ mod } p$, so $t = 0$ and $| E(\mathbb{F}_p) | = p + 1$.

Obviously $(p + 1) | p^2 - 1 = (p + 1)(p - 1)$, so $k \leq 2$ for supersingular curves over prime fields.
$E$ is supersingular if
for $|E(F_q)| = q + 1 - t, q = p^r$, it holds that $t \equiv 0 \mod p$.
Otherwise it is ordinary.

Example:
$y^2 + y = x^3 + a_4 x + a_6$ over $F_{2^r}$ is supersingular:
Each $(x, y)$ point also gives $(x, y + 1) \neq (x, y)$.
All points come in pairs, except for $\infty$,
so $|E(F_{2^r})| = 1 + \text{even}$, so $t \equiv 0 \mod 2$.

Embedding degrees
Let $E$ be supersingular and $q = p \geq 5$, i.e $p > 2\sqrt{p}$.
Hasse’s Theorem states $|t| \leq 2\sqrt{p}$.
$E$ supersingular implies $t \equiv 0 \mod p$, so $t = 0$ and
$|E(F_p)| = p + 1$.
Obviously $(p + 1) | p^2 - 1 = (p + 1)(p - 1)$
so $k \leq 2$ for supersingular curves over prime fields.
A curve $E$ is supersingular if
$$|E(F_q)| = q + 1 - t, \quad q = p^r,$$
that $t \equiv 0 \mod p$.

Otherwise it is ordinary.

Example:
$$y^2 + y = x^3 + a_4 x + a_6 \text{ over } F_{2^r}$$
is supersingular:
A $(x, y) \text{ point also gives } (x, y + 1) \neq (x, y)$.

All points come in pairs, except for $\infty$, so $|E(F_{2^r})| = 1 + \text{even}$, mod 2.

Embedding degrees
Let $E$ be supersingular and $q = p \geq 5$, i.e $p > 2\sqrt{p}$.

Hasse's Theorem states
$$|t| \leq 2\sqrt{p}.$$

$E$ supersingular implies $t \equiv 0 \mod p$, so $t = 0$ and
$$|E(F_p)| = p + 1.$$

Obviously
$$(p + 1) | p^2 - 1 = (p + 1)(p - 1),$$
so $k \leq 2$ for supersingular curves over prime fields.

Distortion maps
For supersingular curves there exist maps
$$\phi : E(F_q) \to E(F_{q^k})$$
i.e. maps $G_1 \to G_2$, giving
$$\tilde{e}(P, P) \neq 1 \text{ for } \tilde{e}(P, P) = e(P, \phi(P)).$$
Such a map is called a distortion map.

These maps are important since the only pairings we know how to compute are variants of
Weil pairing and Tate pairing which have $e(P, P) = 1$. 

**Embedding degrees**

Let $E$ be supersingular and $q = p \geq 5$, i.e. $p > 2\sqrt{p}$.

Hasse’s Theorem states $|t| \leq 2\sqrt{p}$.

$E$ supersingular implies $t \equiv 0 \mod p$, so $t = 0$ and $|E(F_p)| = p + 1$.

Obviously $(p + 1) | p^2 - 1 = (p + 1)(p - 1)$ so $k \leq 2$ for supersingular curves over prime fields.

**Distortion maps**

For supersingular curves there exist maps $\phi : E(F_q) \to E(F_{q^k})$, i.e. maps $G_1 \to G_2$, giving $\tilde{e}(P, P) \neq 1$ for all $\tilde{e}(P, \phi(P))$.

Such a map is called a *distortion map*.

These maps are important since the only pairings we know how to compute are variants of the *Weil pairing* and *Tate pairing*, which have $e(P, P) = 1$.
Embedding degrees

Let $E$ be supersingular and $q = p \geq 5$, i.e. $p > 2\sqrt{p}$.

Hasse's Theorem states $|t| \leq 2\sqrt{p}$.

$E$ supersingular implies $t \equiv 0 \mod p$, so $t = 0$ and $|E(F_p)| = p + 1$.

Obviously $(p + 1) | p^2 - 1 = (p + 1)(p - 1)$ so $k \leq 2$ for supersingular curves over prime fields.

Distortion maps

For supersingular curves there exist maps $\phi : E(F_q) \rightarrow E(F_{q^k})$

i.e. maps $G_1 \rightarrow G_2$, giving $\tilde{e}(P, P) \neq 1$ for $\tilde{e}(P, P) = e(P, \phi(P))$.

Such a map is called a distortion map.

These maps are important since the only pairings we know how to compute are variants of Weil pairing and Tate pairing which have $e(P, P) = 1$. 
Embedding degrees

Let $E$ be supersingular and $q = p \geq 5$, i.e. $p > 2\sqrt{p}$.

Hasse’s Theorem states $|t| \leq 2\sqrt{p}$.

$E$ supersingular implies $t \equiv 0 \text{ mod } p$, so $t = 0$ and $|E(F_p)| = p + 1$.

Obviously $(p + 1) | p^2 - 1 = (p + 1)(p - 1)$ so $k \leq 2$ for supersingular curves over prime fields.

Distortion maps

For supersingular curves there exist maps

$\phi : E(F_q) \rightarrow E(F_{q^k})$

i.e. maps $G_1 \rightarrow G_2$, giving $\tilde{e}(P, P) \neq 1$ for $\tilde{e}(P, P) = e(P, \phi(P))$.

Such a map is called a distortion map.

These maps are important since the only pairings we know how to compute are variants of Weil pairing and Tate pairing which have $e(P, P) = 1$. 
Embedding degrees

Let $E$ be supersingular and $q = p \geq 5$, i.e. $p > 2\sqrt{p}$.

Hasse's Theorem states $|t| \leq 2\sqrt{p}$.

$E$ supersingular implies $t \equiv 0 \mod p$, so $t = 0$ and $|E(F_p)| = p + 1$.

Obviously $(p + 1) \mid p^2 - 1 = (p + 1)(p - 1)$ so $k = 2$ for supersingular curves over prime fields.

Distortion maps

For supersingular curves there exist maps

$\phi : E(F_q) \to E(F_{q^k})$

i.e. maps $G_1 \to G_2$, giving

$\tilde{e}(P, P) \neq 1$ for $\tilde{e}(P, P) = e(P, \phi(P))$.

Such a map is called a distortion map.

These maps are important since the only pairings we know how to compute are variants of Weil pairing and Tate pairing which have $e(P, P) = 1$.

Examples:

$y^2 = x^3 + a_4 x$, for $p \equiv 3 \pmod 4$.

Distortion map $(x, y) \mapsto (\sqrt{-1} y)$.

$y^2 = x^3 + a_6$, for $p \equiv 2 \pmod 3$.

Distortion map $(x, y) \mapsto (jx, y)$ with $j^3 = 1; j \neq 1$.

In both cases, $|E(F_p)| = p + 1$, so $k = 2$. 

Embedding degrees

Let $E$ be supersingular and $q = p \geq 5$, i.e. $p > 2 \sqrt{p}$.

Hasse's Theorem states $|t| \leq 2 \sqrt{p}$.

$E$ supersingular implies $t \equiv 0 \mod p$, so $t = 0$ and $|E(F_p)| = p + 1$.

Obviously $(p + 1) \mid p^2 - 1 = (p + 1)(p - 1)$ so $k \leq 2$ for supersingular curves over prime fields.

Distortion maps

For supersingular curves there exist maps

$\phi : E(F_q) \to E(F_{q^k})$

i.e. maps $G_1 \to G_2$, giving

$\tilde{e}(P, P) \neq 1$ for $\tilde{e}(P, P) = e(P, \phi(P))$.

Such a map is called a distortion map.

These maps are important since the only pairings we know how to compute are variants of

Weil pairing and Tate pairing

which have $e(P, P) = 1$.

Examples:

$y^2 = x^3 + a_4 x$, for $p \equiv 3 \pmod{4}$.

Distortion map $(x, y) \mapsto (-x, \sqrt{-1} y)$.

$y^2 = x^3 + a_6$, for $p \equiv 2 \pmod{3}$.

Distortion map $(x, y) \mapsto (j x, y)$ with $j^3 = 1, j \neq 1$.

In both cases, $\#E(F_p) = p + 1$, so $k = 2$. 
Distortion maps

For supersingular curves there exist maps
\( \phi : E(\mathbb{F}_q) \to E(\mathbb{F}_{q^k}) \)
i.e. maps \( G_1 \to G_2 \), giving
\( \tilde{e}(P, P) \neq 1 \) for \( \tilde{e}(P, P) = e(P, \phi(P)) \).
Such a map is called a distortion map.

These maps are important since the only pairings we know how to compute are variants of Weil pairing and Tate pairing which have \( e(P, P) = 1 \).

Examples:
\( y^2 = x^3 + a_4 x \), for \( p \equiv 3 \pmod{4} \),
Distortion map \( (x, y) \mapsto (-x, \sqrt{-1}y) \).
\( y^2 = x^3 + a_6 \), for \( p \equiv 2 \pmod{3} \),
Distortion map \( (x, y) \mapsto (jx, y) \) with \( j^3 = 1 \), \( j \neq 1 \).
In both cases, \( \#E(\mathbb{F}_p) = p + 1 \), so \( k = 2 \).
Distortion maps

For supersingular curves there exist maps
\( \phi : E(F_q) \rightarrow E(F_{q^k}) \)
i.e. maps \( G_1 \rightarrow G_2 \), giving
\( \tilde{e}(P, P) \neq 1 \) for \( \tilde{e}(P, P) = e(P, \phi(P)) \).
Such a map is called a distortion map.

These maps are important since the only pairings we know how to compute are variants of Weil pairing and Tate pairing which have \( e(P, P) = 1 \).

Examples:
\( y^2 = x^3 + a_4 x \), for \( p \equiv 3 \pmod{4} \).
Distortion map
\( (x, y) \mapsto (-x, \sqrt{-1}y) \).
\( y^2 = x^3 + a_6 \), for \( p \equiv 2 \pmod{3} \).
Distortion map \( (x, y) \mapsto (jx, y) \)
with \( j^3 = 1, j \neq 1 \).

In both cases, \( \# E(F_p) = p + 1 \), so \( k = 2 \).
Distortion maps

For supersingular curves there exist maps

\[ \phi : E(\mathbb{F}_{q^k}) \rightarrow E(\mathbb{F}_q) \]

i.e. maps \( G_1 \rightarrow G_2 \), giving \( \tilde{e}(P;P) \neq 1 \) for \( \tilde{e}(P;P) = e(P;\phi(P)) \).

Such a map is called a distortion map.

These maps are important since the only pairings we know how to compute are variants of Weil pairing and Tate pairing which have \( e(P;P) = 1 \).

Examples:

\[ y^2 = x^3 + a_4 x, \]

for \( p \equiv 3 \pmod{4} \).

Distortion map \( (x, y) \mapsto (-x, \sqrt{-1}y) \).

\[ y^2 = x^3 + a_6, \] for \( p \equiv 2 \pmod{3} \).

Distortion map \( (x, y) \mapsto (jx, y) \) with \( j^3 = 1, j \neq 1 \).

In both cases, \( \#E(\mathbb{F}_p) = p + 1 \), so \( k = 2 \).

Example from Tuesday:

\[ p = 1000003 \equiv 3 \mod 4 \]

\[ y^2 = x^3 - x \] over \( \mathbb{F}_p \).

Has 1000004 = \( p + 1 \) points.

\( P = (101384; 614510) \) is a point of order 500002.

\( nP = (670366; 740819) \).

Construct \( \mathbb{F}_{p^2} = \mathbb{F}_p(i) \).

\( \phi(P) = (898619; 614510i) \).

Invoke magma and compute \( e(P, \phi(P)) = 387265 + 276048i \);
\( e(Q, \phi(P)) = 609466 + 807033i \).

Solve with index calculus to get \( n = 78654 \).

(Btw. this is the clock).
For supersingular curves there exist maps $\phi: E(F_q) \to E(F_{q^k})$, giving $\tilde{e}(P, P) \neq 1$ for $\tilde{e}(P, P) = e(P, \phi(P))$:

Such a map is called a distortion map.

These maps are important since the only pairings we know how to compute are variants of Weil pairing and Tate pairing which have $e(P, P) = 1$.

Examples:

$y^2 = x^3 + a_4x$, for $p \equiv 3 \pmod{4}$.
Distortion map $(x, y) \mapsto (-x, \sqrt{-1}y)$.

$y^2 = x^3 + a_6$, for $p \equiv 2 \pmod{3}$.
Distortion map $(x, y) \mapsto (jx, y)$ with $j^3 = 1, j \neq 1$.

In both cases, $\#E(F_p) = p + 1$, so $k = 2$.

Example from Tuesday:

$p = 1000003 \equiv 3 \pmod{4}$ and $y^2 = x^3 - x$ over $F_p$.

Has $1000004 = p + 1$ points.

$P = (101384, 614510)$ is a point of order 500002.

$nP = (670366, 740819)$.

Construct $F_{p^2}$ as $F_p(i)$.

$\phi(P) = (898619, 614510i)$.

Invoke magma and compute $e(P, \phi(P)) = 387265 + 276048i$;

$e(Q, \phi(P)) = 609466 + 807033i$.

Solve with index calculus to get $n = 78654$.

(Btw. this is the clock).
Distortion maps

For supersingular curves there exist maps
\( \phi : E(F_q) \to E(F_{q^k}) \)
i.e. maps \( G_1 \to G_2 \), giving
\( \tilde{e}(P;P) \neq 1 \) for \( \tilde{e}(P;P) = e(P;\phi(P)) \):

Such a map is called a distortion map.

These maps are important since the only pairings we know how to compute are variants of Weil pairing and Tate pairing which have \( e(P;P) = 1 \).

Example: \( y^2 = x^3 + a_4 x \), for \( p \equiv 3 \pmod{4} \).
Distortion map \((x, y) \mapsto (-x, \sqrt{-1}y)\).

\( y^2 = x^3 + a_6 \), for \( p \equiv 2 \pmod{3} \).
Distortion map \((x, y) \mapsto (jx, y)\) with \( j^3 = 1; j \neq 1 \).

In both cases, \( \#E(F_p) = p + 1 \), so \( k = 2 \).

Example from Tuesday:
\( p = 1000003 \equiv 3 \pmod{4} \) and \( y^2 = x^3 - x \) over \( F_p \).
Has \( 1000004 = p + 1 \) points.
\( P = (101384, 614510) \) is a point of order 500002.
\( nP = (670366, 740819) \).
Construct \( F_{p^2} \) as \( F_p(i) \).
\( \phi(P) = (898619, 614510i) \).

Invoke magma and compute \( e(P, \phi(P)) = 387265 + 276048i \);
\( e(Q, \phi(P)) = 609466 + 807033i \).
Solve with index calculus to get \( n = 78654 \).

(Btw. this is the clock).
Examples:

\[ y^2 = x^3 + a_4x, \]
for \( p \equiv 3 \pmod{4} \).

Distortion map

\[(x, y) \mapsto (-x, \sqrt{-1}y).\]

\[ y^2 = x^3 + a_6, \] for \( p \equiv 2 \pmod{3} \).

Distortion map \((x, y) \mapsto (jx, y)\)
with \( j^3 = 1, j \neq 1 \).

In both cases, \( \#E(\mathbb{F}_p) = p + 1 \),
so \( k = 2 \).

Example from Tuesday:

\( p = 1000003 \equiv 3 \pmod{4} \) and
\[ y^2 = x^3 - x \] over \( \mathbb{F}_p \).

Has \( 1000004 = p + 1 \) points.

\( P = (101384, 614510) \) is a point
of order 500002.

\( nP = (670366, 740819) \).

Construct \( \mathbb{F}_{p^2} \) as \( \mathbb{F}_p(i) \).

\( \phi(P) = (898619, 614510i) \).

Invoke magma and compute

\( e(P, \phi(P)) = 387265 + 276048i; \)
\( e(Q, \phi(P)) = 609466 + 807033i \).

Solve with index calculus to get
\( n = 78654 \).

(Btw. this is the clock).
Example from Tuesday:

\[
p = 1000003 \equiv 3 \pmod{4}
\]

and

\[
y^2 = x^3 - x \quad \text{over } \mathbb{F}_p.
\]

Has 1000004 = \(p + 1\) points.

\[
P = (101384, 614510)
\]

is a point of order 500002.

\[
nP = (670366, 740819).
\]

Construct \(\mathbb{F}_{p^2}\) as \(\mathbb{F}_p(i)\).

\[
\phi(P) = (898619, 614510i).
\]

Invoke magma and compute

\[
e(P, \phi(P)) = 387265 + 276048i;
\]

\[
e(Q, \phi(P)) = 609466 + 807033i.
\]

Solve with index calculus to get

\[
n = 78654.
\]

(Btw. this is the clock).

Summary of pairings

Menezes, Okamoto, and Vanstone
for \(E\) supersingular:

For \(p = 2\) have \(k \leq 4\).

For \(p = 3\) we \(k \leq 6\).

Over \(\mathbb{F}_p\), \(p \geq 5\) have \(k \leq 2\).

These bounds are attained.

Not only supersingular curves: MNT curves are non-supersingular curves with small \(k\).

Other examples constructed for pairing-based cryptography— but small \(k\) unlikely to occur for random curves.
Example from Tuesday:
\[ p = 1000003 \equiv 3 \text{ mod } 4 \] and
\[ y^2 = x^3 - x \] over \( \mathbb{F}_p \).
Has 1000004 = \( p + 1 \) points.
\( P = (101384, 614510) \) is a point of order 500002.
\( nP = (670366, 740819) \).
Construct \( \mathbb{F}_{p^2} \) as \( \mathbb{F}_p(i) \).
\( \phi(P) = (898619, 614510i) \).
Invoke magma and compute
\[ e(P, \phi(P)) = 387265 + 276048i; \]
\[ e(Q, \phi(P)) = 609466 + 807033i. \]
Solve with index calculus to get \( n = 78654 \).
(Btw. this is the clock).

Summary of pairings
Menezes, Okamoto, and Vanstone
for \( E \) supersingular:
For \( p = 2 \) have \( k \leq 4 \).
For \( p = 3 \) we \( k \leq 6 \).
Over \( \mathbb{F}_p, p \geq 5 \) have \( k \leq 2 \).
These bounds are attained.
Not only supersingular curves:
MNT curves are non-supersingular curves with small \( k \).
Other examples constructed for pairing-based cryptography—but small \( k \) unlikely to occur for random curve.
Example from Tuesday:

\( p = 1000003 \equiv 3 \mod 4 \) and
\( y^2 = x^3 - x \) over \( \mathbb{F}_p \).
Has 1000004 = \( p + 1 \) points.
\( P = (101384, 614510) \) is a point of order 500002.
\( nP = (670366, 740819) \).

Construct \( \mathbb{F}_{p^2} \) as \( \mathbb{F}_p(i) \).
\( \phi(P) = (898619, 614510i) \).
Invoke magma and compute
\( e(P, \phi(P)) = 387265 + 276048i \);
\( e(Q, \phi(P)) = 609466 + 807033i \).
Solve with index calculus to get
\( n = 78654 \).
(Btw. this is the clock).

Summary of pairings

Menezes, Okamoto, and Vanstone for \( E \) supersingular:
For \( p = 2 \) have \( k \leq 4 \).
For \( p = 3 \) we \( k \leq 6 \).
Over \( \mathbb{F}_p \), \( p \geq 5 \) have \( k \leq 2 \).
These bounds are attained.

Not only supersingular curves:
MNT curves are non-supersingular curves with small \( k \).
Other examples constructed for pairing-based cryptography – but small \( k \) unlikely to occur for random curve.
Example from Tuesday:

\( p = 1000003 \equiv 3 \mod 4 \) and \( y^2 = x^3 - x \) over \( \mathbb{F}_p \).

Has 1000004 = \( p + 1 \) points.

\( P = (101384, 614510) \) is a point of order 500002.

\( nP = (670366, 740819) \).

Construct \( \mathbb{F}_{p^2} \) as \( \mathbb{F}_p(i) \).

\( \phi(P) = (898619, 614510i) \).

Invoke magma and compute

\( e(P, \phi(P)) = 387265 + 276048i \);
\( e(Q, \phi(P)) = 609466 + 807033i \).

Solve with index calculus to get \( n = 78654 \).

(Btw. this is the clock).

---

Summary of pairings

Menezes, Okamoto, and Vanstone for \( E \) supersingular:

For \( p = 2 \) have \( k \leq 4 \).

For \( p = 3 \) we \( k \leq 6 \).

Over \( \mathbb{F}_p \), \( p \geq 5 \) have \( k \leq 2 \).

These bounds are attained.

Not only supersingular curves:

MNT curves are non-supersingular curves with small \( k \).

Other examples constructed for pairing-based cryptography –
but small \( k \) unlikely to occur for random curve.
Example from Tuesday:
$p = 1000003 \equiv 3 \mod 4$ and

$y^2 = x^3 - x$ over $\mathbb{F}_p$.

Has $1000004 = p + 1$ points.

$P = (101384, 614510)$ is a point of order $500002$.

$nP = (670366, 740819)$.

Construct $\mathbb{F}_{p^2}$ as $\mathbb{F}_p(i)$.

$\text{ff}(P) = (898619, 614510i)$.

Invoke magma and compute $e(P; \text{ff}(P)) = 387265 + 276048i$; $e(Q; \text{ff}(P)) = 609466 + 807033i$.

With index calculus to get $n = 78654$.

This is the clock).

Summary of pairings

Menezes, Okamoto, and Vanstone for $E$ supersingular:
For $p = 2$ have $k \leq 4$.
For $p = 3$ we $k \leq 6$
Over $\mathbb{F}_p$, $p \geq 5$ have $k \leq 2$.
These bounds are attained.

Not only supersingular curves:
MNT curves are non-supersingular curves with small $k$.

Other examples constructed for pairing-based cryptography –
but small $k$ unlikely to occur for random curve.

Summary of other attacks

Definition of embedding degree does not cover all attacks.
For $\mathbb{F}_{p^n}$ watch out that pairing can map to $\mathbb{F}_{p^k}$ with $m < n$.
Watch out for this when selecting curves over $\mathbb{F}_{p^n}$.

Anomalous curves:
If $E = \mathbb{F}_p$ has $#E(\mathbb{F}_p) = p$ then transfer $E(\mathbb{F}_p)$ to $(\mathbb{F}_p; +)$.
Very easy DLP.
Not a problem for Koblitz curves, attack applies to order-$p$ subgroup.
Example from Tuesday:

\[ p = 1000003 \equiv 3 \mod 4 \]
\[ y^2 = x^3 - x \text{ over } F_p. \]

Has 1000004 = \( p + 1 \) points.

\( P = (101384; 614510) \) is a point of order 500002.

\( nP = (670366; 740819). \)

Construct \( F_p^2 \) as \( F_p(\alpha) \).

\( \alpha P = (898619; 614510 i). \)

Invoke magma and compute
\( e(P; \alpha P) = 387265 + 276048 i \);
\( e(Q; \alpha P) = 609466 + 807033 i. \)

Solve with index calculus to get \( n = 78654. \)

(Btw. this is the clock).

Summary of pairings

Menezes, Okamoto, and Vanstone
for \( E \) supersingular:
For \( p = 2 \) have \( k \leq 4. \)
For \( p = 3 \) we \( k \leq 6. \)

Over \( F_p, p \geq 5 \) have \( k \leq 2. \)

These bounds are attained.

Not only supersingular curves:
MNT curves are non-supersingular curves with small \( k. \)
Other examples constructed for pairing-based cryptography –
but small \( k \) unlikely to occur for random curve.

Summary of other attacks

Definition of embedding degree does not cover all attacks.
For \( F_p^n \) watch out that pairing can map to \( F_p^{km} \).
Watch out for this when selecting curves over \( F_p^n \).

Anomalous curves:
If \( E/F_p \) has \( \#E(F_p) = p \),
then transfer \( E(F_p) \) to \( (F_p; +) \).
Very easy DLP.
Not a problem for Koblitz curves,
attack applies to order-\( p \) subgroup.
Summary of pairings

Menezes, Okamoto, and Vanstone for $E$ supersingular:
For $p = 2$ have $k \leq 4$.
For $p = 3$ we $k \leq 6$
Over $\mathbb{F}_p$, $p \geq 5$ have $k \leq 2$.
These bounds are attained.

Not only supersingular curves:
MNT curves are non-supersingular curves with small $k$.
Other examples constructed for pairing-based cryptography – but small $k$ unlikely to occur for random curve.

Summary of other attacks

Definition of embedding degree does not cover all attacks.
For $\mathbb{F}_{p^n}$ watch out that pairing can map to $\mathbb{F}_{p^{km}}$ with $m < n$.
Watch out for this when selecting curves over $\mathbb{F}_{p^n}$.

Anomalous curves:
If $E/\mathbb{F}_p$ has $\#E(\mathbb{F}_p) = p$ then transfer $E(\mathbb{F}_p)$ to $(\mathbb{F}_p, +)$.
Very easy DLP.
Not a problem for Koblitz curves, attack applies to order-$p$ subgroup.
Summary of pairings

Menezes, Okamoto, and Vanstone
for $E$ supersingular:
For $p = 2$ have $k \leq 4$.
For $p = 3$ we $k \leq 6$
Over $\mathbb{F}_p$, $p \geq 5$ have $k \leq 2$.
These bounds are attained.

Not only supersingular curves:
MNT curves are non-supersingular curves with small $k$.
Other examples constructed for pairing-based cryptography – but small $k$ unlikely to occur for random curve.

Summary of other attacks

Definition of embedding degree does not cover all attacks.
For $\mathbb{F}_{p^n}$ watch out that pairing can map to $\mathbb{F}_{p^km}$ with $m < n$.
Watch out for this when selecting curves over $\mathbb{F}_{p^n}$.

Anomalous curves:
If $E/\mathbb{F}_p$ has $\#E(\mathbb{F}_p) = p$ then transfer $E(\mathbb{F}_p)$ to $(\mathbb{F}_p, +)$.
Very easy DLP.
Not a problem for Koblitz curves, attack applies to order-$p$ subgroup.
Summary of pairings
Menezes, Okamoto, and Vanstone
for E supersingular:
For $p = 2$ have $k \leq 4$.
For $p = 3$ we $k \leq 6$.
For $p \geq 5$ have $k \leq 2$.
These bounds are attained.

Not only supersingular curves:
MNT curves are non-supersingular curves with small $k$.
Examples constructed for pairing-based cryptography –
but small $k$ unlikely to occur for random curve.

Summary of other attacks
Definition of embedding degree
does not cover all attacks.
For $\mathbb{F}_{p^n}$ watch out that pairing
can map to $\mathbb{F}_{p^{km}}$ with $m < n$.
Watch out for this when selecting curves over $\mathbb{F}_{p^n}$.

Anomalous curves:
If $E/\mathbb{F}_p$ has $\#E(\mathbb{F}_p) = p$
then transfer $E(\mathbb{F}_p)$ to $(\mathbb{F}_p, +)$.
Very easy DLP.
Not a problem for Koblitz curves, attack applies to order-$p$ subgroup.

Weil descent:
Maps DLP in $E$ over $\mathbb{F}_{p^m}$ to DLP on $J$ over $\mathbb{F}_{p^n}$.
$J$ has larger dimension; elements represented as polynomials of low degree.
This is efficient if dimension of $J$ is not too big.
Particularly nice to compute with $J$ if it is the Jacobian of a hyperelliptic curve $C$.
For genus $g$ get complexity $\tilde{O}(p^{2 - \frac{g}{2}} + 1)$ with the factor base described before, since polynomials have degree $\leq g$. 
Summary of pairings

Menezes, Okamoto, and Vanstone
for \( E \) supersingular:
For \( p = 2 \) have \( k \leq 4 \).
For \( p = 3 \) we have \( k \leq 6 \).
Over \( F_p \), \( p \geq 5 \) have \( k \leq 2 \).
These bounds are attained.

Not only supersingular curves:
MNT curves are non-supersingular curves with small \( k \).
Other examples constructed for pairing-based cryptography –
but small \( k \) unlikely to occur for random curve.

Summary of other attacks

Definition of embedding degree does not cover all attacks.
For \( F_p^n \) watch out that pairing can map to \( F_{p^km} \) with \( m < n \).
Watch out for this when selecting curves over \( F_p^n \).

Anomalous curves:
If \( E/F_p \) has \( \#E(F_p) = p \) then transfer \( E(F_p) \) to \( (F_p, +) \).
Very easy DLP.
Not a problem for Koblitz curves, attack applies to order-\( p \) subgroup.

Weil descent:
Maps DLP in \( E \) over \( F_{p^km} \) to DLP on variety \( J \) over \( F_p^n \).
\( J \) has larger dimension; elements represented as polynomials of low degree. \( \Rightarrow \) index calculus.
This is efficient if dimension of \( J \) is not too big.
Particularly nice to compute with \( J \) if it is the Jacobian of a hyperelliptic curve \( C \).
For genus \( g \) get complexity \( \tilde{O}(p^{2-g+1}) \) with the factor base described before, since polynomials have degree < \( g \).
Summary of pairings
Menezes, Okamoto, and Vanstone
for E supersingular:
For \( p = 2 \) have \( k \leq 4 \).
For \( p = 3 \) we \( k \leq 6 \). Over \( \mathbb{F}_p \), \( p \geq 5 \) have \( k \leq 2 \).
These bounds are attained.

Not only supersingular curves:
MNT curves are non-supersingular curves with small \( k \).
Other examples constructed for pairing-based cryptography – but small \( k \) unlikely to occur for random curve.

Summary of other attacks
Definition of embedding degree does not cover all attacks.
For \( \mathbb{F}_p^n \) watch out that pairing can map to \( \mathbb{F}_{p^km} \) with \( m < n \). Watch out for this when selecting curves over \( \mathbb{F}_p^n \).

Anomalous curves:
If \( E/\mathbb{F}_p \) has \( \#E(\mathbb{F}_p) = p \) then transfer \( E(\mathbb{F}_p) \) to \( (\mathbb{F}_p, +) \). Very easy DLP.
Not a problem for Koblitz curves, attack applies to order-\( p \) subgroup.

Weil descent:
Maps DLP in \( E \) over \( \mathbb{F}_{p^mn} \) to DLP on variety \( J \) over \( \mathbb{F}_{p^n} \). \( J \) has larger dimension; elements represented as polynomials of low degree. \( \Rightarrow \) index calculus.
This is efficient if dimension of \( J \) is not too big.
Particularly nice to compute with \( J \) if it is the Jacobian of a hyperelliptic curve \( C \).
For genus \( g \) get complexity \( \tilde{O}(p^{2-\frac{2}{g+1}}) \) with the factor base described before, since polynomials have degree \( \leq g \).
Summary of other attacks
Definition of embedding degree does not cover all attacks. For \( F_p^n \) watch out that pairing can map to \( F_{p^{km}} \) with \( m < n \). Watch out for this when selecting curves over \( F_p^n \).

Anomalous curves:
If \( E/F_p \) has \( \#E(F_p) = p \) then transfer \( E(F_p) \) to \( (F_p, +) \). Very easy DLP. Not a problem for Koblitz curves, attack applies to order-\( p \) subgroup.

Weil descent:
Maps DLP in \( E \) over \( F_{p^{mn}} \) to DLP on variety \( J \) over \( F_p^n \). \( J \) has larger dimension; elements represented as polynomials of low degree. \( \Rightarrow \) index calculus.
This is efficient if dimension of \( J \) is not too big.
Particularly nice to compute with \( J \) if it is the Jacobian of a hyperelliptic curve \( C \).
For genus \( g \) get complexity \( \tilde{O}(p^{2 - \frac{2}{g+1}}) \) with the factor base described before, since polynomials have degree \( \leq g \).