Timing attacks

1970s: TENEX operating system compares user-supplied string against secret password one character at a time, stopping at first difference:

- AAAAAA vs. SECRET: stop at 1.
- SAAAAAA vs. SECRET: stop at 2.
- SEAAAA vs. SECRET: stop at 3.

Attacker sees comparison time, deduces position of difference. A few hundred tries reveal secret password.

How typical software checks 16-byte authenticator:

```c
for (i = 0; i < 16; ++i)
    if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow from secrets to timings:

```c
uint32 diff = 0;
for (i = 0; i < 16; ++i)
    diff |= x[i] ^ y[i];
return 1 & ((diff - 1) >> 8);
```

Notice that the language makes the wrong thing simple and the right thing complex.
Timing attacks

TENEX operating system compares user-supplied string against secret password character at a time, stopping at first difference:

• AAAAAA vs. SECRET: stop at 1.
• SAAAAA vs. SECRET: stop at 2.
• SEAAAA vs. SECRET: stop at 3.

Attacker sees comparison time, deduces position of difference.
A few hundred tries reveal secret password.

How typical software checks 16-byte authenticator:

```c
for (i = 0; i < 16; ++i)
    if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow from secrets to timings:

```c
uint32 diff = 0;
for (i = 0; i < 16; ++i)
    diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language makes the wrong thing simple and the right thing complex.

Language designer's notion of "right" is too weak for security. So mistakes continue to happen.
Timing attacks
1970s: TENEX operating system compares user-supplied string against secret password one character at a time, stopping at first difference:

-AAAAAA vs. SECRET: stop at 1.
-SAAAAA vs. SECRET: stop at 2.
-SEAAAA vs. SECRET: stop at 3.

Attacker sees comparison time, deduces position of difference.
A few hundred tries reveal secret password.

How typical software checks 16-byte authenticator:

```c
for (i = 0; i < 16; ++i)
    if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow from secrets to timings:

```c
uint32 diff = 0;
for (i = 0; i < 16; ++i)
    diff |= x[i] ^ y[i];
return 1 & ((diff - 1) >> 8);
```

Notice that the language makes the wrong thing simple and the right thing complex.

Language designer's notion of "right" is too weak for security.
So mistakes continue to happen.
Timing attacks

1970s: TENEX operating system compares user-supplied string against secret password one character at a time, stopping at first difference:

- AAAAAAA vs. SECRET: stop at 1.
- SAAAAA vs. SECRET: stop at 2.
- SEAAAA vs. SECRET: stop at 3.

Attacker sees comparison time, deduces position of difference. A few hundred tries reveal secret password.

How typical software checks 16-byte authenticator:

```c
for (i = 0; i < 16; ++i)
    if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow from secrets to timings:

```c
uint32 diff = 0;
for (i = 0; i < 16; ++i)
    diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language makes the wrong thing simple and the right thing complex.

Language designer’s notion of “right” is too weak for security. So mistakes continue to happen.
How typical software checks 16-byte authenticator:

```c
for (i = 0; i < 16; ++i)
    if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow from secrets to timings:

```c
uint32 diff = 0;
for (i = 0; i < 16; ++i)
    diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language makes the wrong thing simple and the right thing complex.

Language designer’s notion of “right” is too weak for security.
So mistakes continue to happen.
How typical software checks 16-byte authenticator:

```c
for (i = 0; i < 16; ++i)
    if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow from secrets to timings:

```c
uint32 diff = 0;
for (i = 0; i < 16; ++i)
    diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language makes the wrong thing simple and the right thing complex.

Language designer’s notion of “right” is too weak for security.

So mistakes continue to happen.

One of many current examples, part of the reference software for CAESAR candidate CLOC:

```c
/* compare the tag */
int i;
for (i = 0; i < CRYPTO_ABYTES; i++)
    if (tag[i] != c[(*mlen) + i]){
        return RETURN_TAG_NO_MATCH;
    }
return RETURN_SUCCESS;
```
Typical software checks 16-byte authenticator:

```c
for (i = 0; i < 16; ++i)
    if (x[i] != y[i]) return 0;
return 1;
```

Eliminating information flow from secrets to timings:

```c
uint32 diff = 0;
for (i = 0; i < 16; ++i)
    diff |= x[i] ^ y[i];
return 1 & ((diff - 1) >> 8);
```

Notice that the language makes the wrong thing simple and the right thing complex.

Language designer’s notion of “right” is too weak for security.

So mistakes continue to happen.

One of many current examples, part of the reference software for CAESAR candidate CLOC:

```c
/* compare the tag */
int i;
for (i = 0; i < CRYPTO_ABYTES; i++)
    if (tag[i] != c[(*mlen) + i]){
        return RETURN_TAG_NO_MATCH;
    }
return RETURN_SUCCESS;
```

Do timing attacks really work?

Objection: "Timings are noisy!"
Language designer’s notion of “right” is too weak for security.

So mistakes continue to happen.

One of many current examples, part of the reference software for CAESAR candidate CLOC:

```c
/* compare the tag */
int i;
for (i = 0; i < CRYPTO_ABYTES; i++)
    if (tag[i] != c[(*mlen) + i]){
        return RETURN_TAG_NO_MATCH;
    }
return RETURN_SUCCESS;
```

Objection: “Timings are noisy!”
How typical software checks 16-byte authenticator:

```c
for (i = 0; i < 16; ++i)
    if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow from secrets to timings:

```c
uint32 diff = 0;
for (i = 0; i < 16; ++i)
    diff |= x[i] ^ y[i];
return 1 & ((diff - 1) >> 8);
```

Notice that the language makes the wrong thing simple and the right thing complex.

Language designer’s notion of “right” is too weak for security. So mistakes continue to happen.

One of many current examples, part of the reference software for CAESAR candidate CLOC:

```c
/* compare the tag */
int i;
for (i = 0; i < CRYPTO_ABYTES; i++)
    if (tag[i] != c[(*mlen) + i]){
        return RETURN_TAG_NO_MATCH;
    }
return RETURN_SUCCESS;
```

Do timing attacks really work?

Objection: “Timings are noisy!”
Language designer’s notion of “right” is too weak for security.
So mistakes continue to happen.
One of many current examples, part of the reference software for CAESAR candidate CLOC:

```c
/* compare the tag */
int i;
for(i = 0; i < CRYPTO_ABYTES; i++)
    if(tag[i] != c[(*mlen) + i]){
        return RETURN_TAG_NO_MATCH;
    }
return RETURN_SUCCESS;
```

Do timing attacks really work?
Objection: “Timings are noisy!”
Language designer’s notion of “right” is too weak for security.
So mistakes continue to happen.
One of many current examples, part of the reference software for CAESAR candidate CLOC:

```c
/* compare the tag */
int i;
for(i = 0; i < CRYPTO_ABYTES; i++)
    if(tag[i] != c[(*mlen) + i]){
        return RETURN_TAG_NO_MATCH;
    }
return RETURN_SUCCESS;
```

Do timing attacks really work?
Objection: “Timings are noisy!”
Answer #1:
Does noise stop all attacks?
To guarantee security, defender must block all information flow.
Language designer’s notion of “right” is too weak for security. So mistakes continue to happen. One of many current examples, part of the reference software for CAESAR candidate CLOC:

```c
/* compare the tag */
int i;
for(i = 0; i < CRYPTO_ABYTES; i++)
    if(tag[i] != c[(*mlen) + i]){
        return RETURN_TAG_NO_MATCH;
    }
return RETURN_SUCCESS;
```

Do timing attacks really work?
Objection: “Timings are noisy!”
Answer #1: Does noise stop all attacks? To guarantee security, defender must block all information flow.
Answer #2: Attacker uses statistics to eliminate noise.
Language designer’s notion of “right” is too weak for security.
So mistakes continue to happen.
One of many current examples, part of the reference software for CAESAR candidate CLOC:

```c
/* compare the tag */
int i;
for(i = 0; i < CRYPTO_ABYTES; i++)
    if(tag[i] != c[(*mlen) + i]){
        return RETURN_TAG_NO_MATCH;
    }
return RETURN_SUCCESS;
```

Do timing attacks really work?
Objection: “Timings are noisy!”
Answer #1: Does noise stop all attacks?
To guarantee security, defender must block all information flow.
Answer #2: Attacker uses statistics to eliminate noise.
Answer #3, what the 1970s attackers actually did: Cross page boundary, inducing page faults, to amplify timing signal.
Language designer’s notion of "right" is too weak for security. So mistakes continue to happen. One of many current examples, part of the reference software for CAESAR candidate CLOC:

```c
/* compare the tag */

int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
    if(tag[i] != c[(*mlen) + i]){
        return RETURN_TAG_NO_MATCH;
    }
return RETURN_SUCCESS;
```

Do timing attacks really work?
Objection: “Timings are noisy!”

Answer #1:
Does noise stop all attacks?
To guarantee security, defender must block all information flow.

Answer #2: Attacker uses statistics to eliminate noise.

Answer #3, what the 1970s attackers actually did:
Cross page boundary, inducing page faults, to amplify timing signal.

Examples of successful attacks:

- 2013 AlFardan–Paterson "Lucky Thirteen: breaking the TLS and DTLS record protocols" steals plaintext using decryption timings.
- 2014 van de Pol–Smart–Yarom steals Bitcoin key from timings of 25 OpenSSL signatures.
- 2016 Yarom–Genkin–Heninger "CacheBleed" steals RSA secret key via timings of OpenSSL.
Language designer's notion of “right” is too weak for security. So mistakes continue to happen. One of many current examples, part of the reference software for CAESAR candidate CLOC:

```c
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
    if(tag[i] != c[(*mlen) + i])
        return RETURN_TAG_NO_MATCH;
return RETURN_SUCCESS;
```

Do timing attacks really work?
Objection: “Timings are noisy!”
Answer #1: Does noise stop all attacks? To guarantee security, defender must block all information flow.
Answer #2: Attacker uses statistics to eliminate noise.
Answer #3, what the 1970s attackers actually did: Cross page boundary, inducing page faults, to amplify timing signal.

Examples of successful attacks:
2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.
2013 AlFardan–Paterson “Lucky Thirteen: breaking the TLS and DTLS record protocols” steals plaintext using decryption timings.
2014 van de Pol–Smart–Yarom steals Bitcoin key from timings of 25 OpenSSL signatures.
2016 Yarom–Genkin–Heninger “CacheBleed” steals RSA secret key via timings of OpenSSL.
Language designer's notion of "right" is too weak for security. So mistakes continue to happen. One of many current examples, part of the reference software for CAESAR candidate CLOC:

```c
/* compare the tag */
int i;
for(i = 0; i < CRYPTO_ABYTES; i++)
  if(tag[i] != c[(*mlen) + i])
    return RETURN_TAG_NO_MATCH;
return RETURN_SUCCESS;
```

Do timing attacks really work? Objection: "Timings are noisy!"

Answer #1: Does noise stop all attacks? To guarantee security, defender must block all information flow.

Answer #2: Attacker uses statistics to eliminate noise.

Answer #3, what the 1970s attackers actually did: Cross page boundary, inducing page faults, to amplify timing signal.

Examples of successful attacks:

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.

2013 AlFardan–Paterson “Lucky Thirteen: breaking the TLS and DTLS record protocols” steals plaintext using decryption timings.

2014 van de Pol–Smart–Yarom steals Bitcoin key from timings of 25 OpenSSL signatures.

2016 Yarom–Genkin–Heninger “CacheBleed” steals RSA secret key via timings of OpenSSL.
Do timing attacks really work?

Objection: “Timings are noisy!”

Answer #1: Does noise stop all attacks? To guarantee security, defender must block all information flow.

Answer #2: Attacker uses statistics to eliminate noise.

Answer #3, what the 1970s attackers actually did: Cross page boundary, inducing page faults, to amplify timing signal.

Examples of successful attacks:

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.

2013 AlFardan–Paterson “Lucky Thirteen: breaking the TLS and DTLS record protocols” steals plaintext using decryption timings.

2014 van de Pol–Smart–Yarom steals Bitcoin key from timings of 25 OpenSSL signatures.

2016 Yarom–Genkin–Heninger “CacheBleed” steals RSA secret key via timings of OpenSSL.
Timing attacks really work?

Objection: “Timings are noisy!”

#1: Does noise stop all attacks?
To guarantee security, defender must block all information flow.

#2: Attacker uses statistics to eliminate noise.

#3, what the 1970s attackers actually did:
Cross page boundary, inducing page faults, to amplify timing signal.

Examples of successful attacks:

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.

2013 AlFardan–Paterson “Lucky Thirteen: breaking the TLS and DTLS record protocols” steals plaintext using decryption timings.

2014 van de Pol–Smart–Yarom steals Bitcoin key from timings of 25 OpenSSL signatures.

2016 Yarom–Genkin–Heninger “CacheBleed” steals RSA secret key via timings of OpenSSL.

Constant-time ECC

ECDH computation: $a; P \mapsto aP$ where $a$ is your secret key.
Key generation: $a \mapsto aB$.
Signing: $r \mapsto rB$.

All of these use secret data.

Does timing leak this data?

Are there any branches in ECC ops? Point ops? Field ops?
Do the underlying machine insns take variable time?
Do timing attacks really work?

Objection: “Timings are noisy!”

Answer #1: Does noise stop all attacks?

To guarantee security, defender must block all information flow.

Answer #2: Attacker uses statistics to eliminate noise.

Answer #3, what the 1970s attackers actually did:

Cross page boundary, inducing page faults, to amplify timing signal.

Examples of successful attacks:

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.

2013 AlFardan–Paterson “Lucky Thirteen: breaking the TLS and DTLS record protocols” steals plaintext using decryption timings.

2014 van de Pol–Smart–Yarom steals Bitcoin key from timings of 25 OpenSSL signatures.

2016 Yarom–Genkin–Heninger “CacheBleed” steals RSA secret key via timings of OpenSSL.

Constant-time ECC ECDH computation:

\[ a \cdot P \rightarrow aP \]

where \( a \) is your secret key.

Key generation:

\[ a \rightarrow aB \]

Signing:

\[ r \rightarrow rB \]

All of these use secret data.

Does timing leak this data?

Are there any branches in ECC ops? Point ops? Field ops?

Do the underlying machine insns take variable time?
Examples of successful attacks:

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.

2013 AlFardan–Paterson “Lucky Thirteen: breaking the TLS and DTLS record protocols” steals plaintext using decryption timings.

2014 van de Pol–Smart–Yarom steals Bitcoin key from timings of 25 OpenSSL signatures.

2016 Yarom–Genkin–Heninger “CacheBleed” steals RSA secret key via timings of OpenSSL.

Constant-time ECC

ECDH computation: $a, P \mapsto aP$ where $a$ is your secret key.

Key generation: $a \mapsto aB$.

Signing: $r \mapsto rB$.

All of these use secret data. Does timing leak this data?

Are there any branches in ECC ops? Point ops? Field ops? Do the underlying machine insns take variable time?
Examples of successful attacks:

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.

2013 AlFardan–Paterson “Lucky Thirteen: breaking the TLS and DTLS record protocols” steals plaintext using decryption timings.

2014 van de Pol–Smart–Yarom steals Bitcoin key from timings of 25 OpenSSL signatures.

2016 Yarom–Genkin–Heninger “CacheBleed” steals RSA secret key via timings of OpenSSL.

Constant-time ECC

ECDH computation: \( a, P \mapsto aP \)

where \( a \) is your secret key.

Key generation: \( a \mapsto aB \).

Signing: \( r \mapsto rB \).

All of these use secret data. Does timing leak this data?

Are there any branches in ECC ops? Point ops? Field ops?

Do the underlying machine insns take variable time?
Examples of successful attacks:

- 2013 AlFardan–Paterson “Lucky Thirteen: breaking the TLS and DTLS record protocols” steals plaintext using decryption timings.
- 2014 van de Pol–Smart–Yarom steals Bitcoin key from timings of 25 OpenSSL signatures.
- 2016 Yarom–Genkin–Heninger “CacheBleed” steals RSA secret key via timings of OpenSSL.

Constant-time ECC

ECDH computation: \( a, P \rightarrow aP \) where \( a \) is your secret key.

Key generation: \( a \rightarrow aB \).

Signing: \( r \rightarrow rB \).

All of these use secret data. Does timing leak this data?

Are there any branches in ECC ops? Point ops? Field ops? Do the underlying machine insns take variable time?

Recall left-to-right binary method to compute \( n; P \rightarrow nP \) using point addition:

```python
def scalarmult(n, P):
    if n == 0:
        return 0
    if n == 1:
        return P
    R = scalarmult(n // 2, P)
    R = R + R
    if n % 2:
        R = R + P
    return R
```

Many branches here. NAF etc. also use many branches.
Examples of successful attacks:

2005 Tromer–Osvik–Shamir:
65ms to steal Linux AES key
used for hard-disk encryption.

2013 AlFardan–Paterson “Lucky
Thirteen: breaking the TLS and
DTLS record protocols” steals
plaintext using decryption timings.

2014 van de Pol–Smart–Yarom
steals Bitcoin key from timings
of 25 OpenSSL signatures.

2016 Yarom–Genkin–Heninger
“CacheBleed” steals RSA secret
key via timings of OpenSSL.

Constant-time ECC
ECDH computation: \( a, P \mapsto aP \)
where \( a \) is your secret key.

Key generation: \( a \mapsto aB \).

Signing: \( r \mapsto rB \).

All of these use secret data.
Does timing leak this data?
Are there any branches in
ECC ops? Point ops? Field ops?
Do the underlying machine insns
take variable time?

Recall left-to-right binary method
to compute \( n, P \mapsto nP \)
using point addition:

\[
def \text{scalarmult}(n, P):
    \text{if } n == 0: \text{return } 0
    \text{if } n == 1: \text{return } P
    R = \text{scalarmult}(n//2, P)
    R = R + R
    \text{if } n \% 2: R = R + P
    \text{return } R
\]

Many branches here.
NAF etc. also use many branches.
Examples of successful attacks:

2005 Tromer–Osvik–Shamir:
65ms to steal Linux AES key used for hard-disk encryption.

2013 AlFardan–Paterson "Lucky Thirteen: breaking the TLS and DTLS record protocols" steals plaintext using decryption timings.

2014 van de Pol–Smart–Yarom steals Bitcoin key from timings of 25 OpenSSL signatures.

2016 Yarom–Genkin–Heninger "CacheBleed" steals RSA secret key via timings of OpenSSL.

Constant-time ECC

ECDH computation: \( a, P \mapsto aP \) where \( a \) is your secret key.

Key generation: \( a \mapsto aB \).

Signing: \( r \mapsto rB \).

All of these use secret data.

Does timing leak this data?

Are there any branches in ECC ops? Point ops? Field ops?

Do the underlying machine insns take variable time?

Recall left-to-right binary method to compute \( n, P \mapsto nP \) using point addition:

```python
def scalarmult(n,P):
    if n == 0: return 0
    if n == 1: return P
    R = scalarmult(n//2,P)
    R = R + R
    if n % 2: R = R + P
    return R
```

Many branches here.

NAF etc. also use many branches.
Constant-time ECC

ECDH computation: \( a, P \mapsto aP \) where \( a \) is your secret key.

Key generation: \( a \mapsto aB \).

Signing: \( r \mapsto rB \).

All of these use secret data. Does timing leak this data?

Are there any branches in ECC ops? Point ops? Field ops? Do the underlying machine insns take variable time?

Recall left-to-right binary method

to compute \( n, P \mapsto nP \) using point addition:

```python
def scalarmult(n,P):
    if n == 0: return 0
    if n == 1: return P
    R = scalarmult(n//2,P)
    R = R + R
    if n % 2: R = R + P
    return R
```

Many branches here. NAF etc. also use many branches.
Constant-time ECC
ECDH computation: $a, P \mapsto aP$ where $a$ is your secret key.
Key generation: $a \mapsto aB$.
Signing: $r \mapsto rB$.
These use secret data.
Does timing leak this data?
Are there any branches in ECC ops? Point ops? Field ops?
Do the underlying machine insns take variable time?
Recall left-to-right binary method to compute $n, P \mapsto nP$ using point addition:
```python
def scalarmult(n, P):
    if n == 0: return 0
    if n == 1: return P
    R = scalarmult(n//2, P)
    R = R + R
    if n % 2: R = R + P
    return R
```
Many branches here.
NAF etc. also use many branches.
Even if each point addition takes the same amount of time (certainly not true in Python), total time depends on $n$.
If $2^{e-1} \leq n < 2^e$ and $n$ has exactly $w$ bits set:
number of additions is $e + w - 2$.
Particularly fast total time usually indicates very small $n$.
"Lattice attacks" on signatures compute the secret key given positions of very small nonces $r$. 
Constant-time ECC

ECDH computation: \(a; P \mapsto aP\) where \(a\) is your secret key.

Key generation: \(a \mapsto aB\).

Signing: \(r \mapsto rB\).

All of these use secret data. Does timing leak this data?

Are there any branches in ECC ops? Point ops? Field ops?

Do the underlying machine insns take variable time?

Recall left-to-right binary method to compute \(n; P \mapsto nP\) using point addition:

\[
\text{def scalarmult}(n,P):
\]

\[
\begin{align*}
&\text{if } n == 0: \text{ return } 0 \\
&\text{if } n == 1: \text{ return } P \\
&R = \text{scalarmult}(n//2,P) \\
&R = R + R \\
&\text{if } n \% 2: R = R + P \\
&\text{return } R
\end{align*}
\]

Many branches here.
NAF etc. also use many branches.

Even if each point addition takes the same amount of time (certainly not true in Python), total time depends on \(n\).

If \(2^{e-1} \leq n < 2^e\) and \(n\) has exactly \(w\) bits set:

\[
\text{number of additions is } e + w - 2.
\]

Particularly fast total time usually indicates very small \(n\).

“Lattice attacks” on signatures compute the secret key given positions of very small nonces \(r\).
Constant-time ECC

ECDH computation: \( a; P \mapsto aP \) where \( a \) is your secret key.

Key generation: \( a \mapsto aB \).

Signing: \( r \mapsto rB \).

All of these use secret data. Does timing leak this data?

Are there any branches in ECC ops? Point ops? Field ops?

Do the underlying machine insns take variable time?

Recall left-to-right binary method to compute \( n; P \mapsto nP \) using point addition:

```python
def scalarmult(n,P):
    if n == 0: return 0
    if n == 1: return P
    R = scalarmult(n//2,P)
    R = R + R
    if n % 2: R = R + P
    return R
```

Many branches here.

NAF etc. also use many branches.

Even if each point addition takes the same amount of time (certainly not true in Python), total time depends on \( n \).

If \( 2^{e-1} \leq n < 2^e \) and \( n \) has exactly \( w \) bits set:

- number of additions is \( e + w - 2 \).

Particularly fast total time usually indicates very small \( n \).

“Lattice attacks” on signatures compute the secret key given positions of very small nonces.
Recall left-to-right binary method to compute $n, P \mapsto nP$ using point addition:

```python
def scalarmult(n,P):
    if n == 0: return 0
    if n == 1: return P
    R = scalarmult(n//2,P)
    R = R + R
    if n % 2: R = R + P
    return R
```

Many branches here.
NAF etc. also use many branches.

Even if each point addition takes the same amount of time (certainly not true in Python), total time depends on $n$. If $2^{e-1} \leq n < 2^e$ and $n$ has exactly $w$ bits set: number of additions is $e + w - 2$.

Particularly fast total time usually indicates very small $n$.

“Lattice attacks” on signatures compute the secret key given positions of very small nonces $r$. 
Recall left-to-right binary method to compute \( n, P \mapsto nP \) using point addition:

```python
def scalarmult(n, P):
    if n == 0: return 0
    if n == 1: return P
    R = scalarmult(n//2, P)
    R = R + R
    if n % 2: R = R + P
    return R
```

Many branches here. NAF etc. also use many branches.

Even if each point addition takes the same amount of time (certainly not true in Python), total time depends on \( n \).

If \( 2^{e-1} \leq n < 2^e \) and \( n \) has exactly \( w \) bits set:

- number of additions is \( e + w - 2 \).
- Particularly fast total time usually indicates very small \( n \).
- “Lattice attacks” on signatures compute the secret key given positions of very small nonces \( r \).

Even worse:

- CPUs do not try to protect metadata regarding branches.
- Actual time for a branch affects, and is affected by, detailed state of code cache, branch predictor, etc.
- Attacker interacts with this state, often sees pattern of branches.
- Exploited in, e.g., Bitcoin attack.
Recall left-to-right binary method to compute $n \cdot P$ using point addition:

```python
def scalarmult(n, P):
    if n == 0: return 0
    if n == 1: return P
    R = scalarmult(n // 2, P)
    R = R + R
    if n % 2: R = R + P
    return R
```

Many branches here. NAF etc. also use many branches.

Even if each point addition takes the same amount of time (certainly not true in Python), total time depends on $n$.

If $2^{e-1} \leq n < 2^e$ and $n$ has exactly $w$ bits set: number of additions is $e + w - 2$.

Particularly fast total time usually indicates very small $n$.

“Lattice attacks” on signatures compute the secret key given positions of very small nonces $r$.

Even worse: CPUs do not try to protect metadata regarding branches.

Actual time for a branch affects, and is affected by, detailed state of code cache, branch predictor, etc.

Attacker interacts with this state, often sees pattern of branches.

Exploited in, e.g., Bitcoin attack.
Recall left-to-right binary method to compute \( n \cdot P \) using point addition:

\[
def \text{scalarmult}(n, P):
    \text{if } n == 0: \text{return } 0
    \text{if } n == 1: \text{return } P
    R = \text{scalarmult}(n//2, P)
    R = R + R
    \text{if } n \text{ % 2: } R = R + P
    \text{return } R
\]

Many branches here.
NAF etc. also use many branches.

Even if each point addition takes the same amount of time (certainly not true in Python), total time depends on \( n \).

If \( 2^{e-1} \leq n < 2^e \) and \( n \) has exactly \( w \) bits set:
number of additions is \( e + w - 2 \).

Particularly fast total time usually indicates very small \( n \).

“Lattice attacks” on signatures compute the secret key given positions of very small nonces \( r \).

Even worse: CPUs do not try to protect metadata regarding branches.

Actual time for a branch affects, and is affected by, detailed state of code cache, branch predictor, etc.

Attacker interacts with this state, often sees pattern of branches.
Exploited in, e.g., Bitcoin attack.
Even if each point addition takes the same amount of time (certainly not true in Python), total time depends on $n$.

If $2^{e-1} \leq n < 2^e$ and $n$ has exactly $w$ bits set: number of additions is $e + w - 2$.

Particularly fast total time usually indicates very small $n$.

“Lattice attacks” on signatures compute the secret key given positions of very small nonces $r$.

Even worse: CPUs do not try to protect metadata regarding branches.

Actual time for a branch affects, and is affected by, detailed state of code cache, branch predictor, etc.

Attacker interacts with this state, often sees pattern of branches. Exploited in, e.g., Bitcoin attack.
Even if each point addition takes the same amount of time (certainly not true in Python), total time depends on $n$.

If $2^{e-1} \leq n < 2^e$ and $n$ has exactly $w$ bits set: number of additions is $e + w - 2$.

Particularly fast total time usually indicates very small $n$.

“Lattice attacks” on signatures compute the secret key given positions of very small nonces $r$.

Even worse:

CPUs do not try to protect metadata regarding branches.

Actual time for a branch affects, and is affected by, detailed state of code cache, branch predictor, etc.

Attacker interacts with this state, often sees pattern of branches.

Exploited in, e.g., Bitcoin attack.

Confidence-inspiring solution: Avoid all data flow from secrets to branch conditions.
Even if each point addition takes the same amount of time (certainly not true in Python), total time depends on $n$. If $2e^{-1} \leq n < 2e$ and $n$ has exactly $w$ bits set: the number of additions is $e + w - 2$.

Particularly fast total time indicates very small $n$.

"Lattice attacks" on signatures compute the secret key given positions of very small nonces $r$.

Even worse: CPUs do not try to protect metadata regarding branches.

Actual time for a branch affects, and is affected by, detailed state of code cache, branch predictor, etc.

Attacker interacts with this state, often sees pattern of branches. Exploited in, e.g., Bitcoin attack.

Confidence-inspiring solution: Avoid all data flow from secrets to branch conditions.

Double-and-add-always

Eliminate branches by always computing both results:

```python
def scalarmult(n, b, P):
    if b == 0:
        return 0
    R = scalarmult(n // 2, b - 1, P)
    R2 = R + R
    S = [R2, R2 + P]
    return S[n % 2]
```

Works for $0 \leq n < 2^b$.

Always takes $2^b$ additions (including $b$ doublings).

Use public $b$: bits allowed in $n$. 

Confidence-inspiring solution: Avoid all data flow from secrets to branch conditions.
Even if each point addition takes the same amount of time (certainly not true in Python), total time depends on $n$.

If $2e - 1 \leq n < 2e$ and $n$ has exactly $w$ bits set:

- the number of additions is $e + w - 2$.
- Particularly fast total time usually indicates very small $n$.

"Lattice attacks" on signatures compute the secret key given positions of very small nonces $r$.

Even worse:

- CPUs do not try to protect metadata regarding branches.
- Actual time for a branch affects, and is affected by, detailed state of code cache, branch predictor, etc.
- Attacker interacts with this state, often sees pattern of branches.
- Exploited in, e.g., Bitcoin attack.

Confidence-inspiring solution:

**Avoid all data flow from secrets to branch conditions.**

Double-and-add-always

Eliminate branches by always computing both results:

```python
def scalarmult(n,b,P):
    if b == 0: return 0
    R = scalarmult(n//2,b-1,P)
    R2 = R + R
    S = [R2,R2 + P]
    return S[n % 2]
```

Works for $0 \leq n < 2^b$.
- Always takes $2b$ additions (including $b$ doublings).
- Use public $b$: bits allowed in $n$. 
Even if each point addition takes the same amount of time (certainly not true in Python), total time depends on $n$.

If $2^{e-1} \leq n < 2^e$ and $n$ has exactly $w$ bits set: number of additions is $e + w - 2$.

Particularly fast total time usually indicates very small $n$.

"Lattice attacks" on signatures compute the secret key given positions of very small nonces $r$.

Even worse:
CPUs do not try to protect metadata regarding branches.

Actual time for a branch affects, and is affected by, detailed state of code cache, branch predictor, etc.

Attacker interacts with this state, often sees pattern of branches.
Exploited in, e.g., Bitcoin attack.

Confidence-inspiring solution: Avoid all data flow from secrets to branch conditions.

Double-and-add-always
Eliminate branches by always computing both results:

```python
def scalarmult(n, b, P):
    if b == 0: return 0
    R = scalarmult(n//2, b-1, P)
    R2 = R + R
    S = [R2, R2 + P]
    return S[n % 2]
```

Works for $0 \leq n < 2^b$.
Always takes $2b$ additions (including $b$ doublings).
Use public $b$: bits allowed in $n$. 

Even worse: CPUs do not try to protect metadata regarding branches. Actual time for a branch affects, and is affected by, detailed state of code cache, branch predictor, etc. Attacker interacts with this state, often sees pattern of branches. Exploited in, e.g., Bitcoin attack. Confidence-inspiring solution: Avoid all data flow from secrets to branch conditions.

Double-and-add-always
Eliminate branches by always computing both results:

```python
def scalarmult(n,b,P):
    if b == 0: return 0
    R = scalarmult(n//2,b-1,P)
    R2 = R + R
    S = [R2,R2 + P]
    return S[n % 2]
```

Works for $0 \leq n < 2^b$. Always takes $2b$ additions (including $b$ doublings). Use public $b$: bits allowed in $n$. 
**Even worse:** CPUs do not try to protect metadata regarding branches. Actual time for a branch affects, and is affected by, detailed state of code cache, branch predictor, etc. Attacker interacts with this state, often sees pattern of branches. Exploited in, e.g., Bitcoin attack.

Confidence-inspiring solution: Avoid all data flow from secrets to branch conditions.

---

**Double-and-add-always**

Eliminate branches by always computing both results:

```python
def scalarmult(n,b,P):
    if b == 0: return 0
    R = scalarmult(n//2,b-1,P)
    R2 = R + R
    S = [R2,R2 + P]
    return S[n % 2]
```

Works for $0 \leq n < 2^b$. Always takes $2^b$ additions (including $b$ doublings). Use public $b$: bits *allowed* in $n$.

---

**Another big problem:** CPUs do not try to protect metadata regarding array indices. Actual time for $x[i]$ affects, and is affected by, detailed state of data cache, store-to-load forwarder, etc. Exploited in, e.g., CacheBleed, despite Intel and OpenSSL claiming their code was safe.
Even worse: CPUs do not try to protect metadata regarding branches. Actual time for a branch affects, and is affected by, detailed state of code cache, branch predictor, etc. Attacker interacts with this state, often sees pattern of branches. Exploited in, e.g., Bitcoin attack.

Confidence-inspiring solution: Avoid all data flow from secrets to branch conditions.

Double-and-add-always

Eliminate branches by always computing both results:

```python
def scalarmult(n, b, P):
    if b == 0: return 0
    R = scalarmult(n//2, b-1, P)
    R2 = R + R
    S = [R2, R2 + P]
    return S[n % 2]
```

Works for $0 \leq n < 2^b$.
Always takes $2^b$ additions (including $b$ doublings).
Use public $b$: bits allowed in $n$.

Another big problem: CPUs do not try to protect metadata regarding array indices. Actual time for $x[i]$ affects, and is affected by, detailed state of data cache, store-to-load forwarder, etc. Exploited in, e.g., CacheBleed, despite Intel and OpenSSL claiming their code was safe.
Double-and-add-always

Eliminate branches by always computing both results:

```python
def scalarmult(n,b,P):
    if b == 0: return 0
    R = scalarmult(n//2,b-1,P)
    R2 = R + R
    S = [R2,R2 + P]
    return S[n % 2]
```

Works for $0 \leq n < 2^b$.
Always takes $2b$ additions (including $b$ doublings).
Use public $b$: bits allowed in $n$.

Another big problem:
CPUs do not try to protect metadata regarding *array indices*.

Actual time for $x[i]$ affects, and is affected by, detailed state of data cache, store-to-load forwarder, etc.

Exploited in, e.g., CacheBleed, despite Intel and OpenSSL claiming their code was safe.
Double-and-add-always

Eliminate branches by always computing both results:

def scalarmult(n,b,P):
    if b == 0: return 0
    R = scalarmult(n//2,b-1,P)
    R2 = R + R
    S = [R2,R2 + P]
    return S[n % 2]

Works for $0 \leq n < 2^b$.
Always takes $2^b$ additions (including $b$ doublings).
Use public $b$: bits allowed in $n$.

Another big problem:
CPUs do not try to protect metadata regarding *array indices*.

Actual time for $x[i]$ affects, and is affected by, detailed state of data cache, store-to-load forwarder, etc.

Exploited in, e.g., CacheBleed, despite Intel and OpenSSL claiming their code was safe.
Double-and-add-always

Eliminate branches by always computing both results:

```python
def scalarmult(n,b,P):
    if b == 0: return 0
    R = scalarmult(n//2,b-1,P)
    R2 = R + R
    S = [R2,R2 + P]
    return S[n % 2]
```

Works for $0 \leq n < 2^b$.
Always takes $2^b$ additions (including $b$ doublings).
Use public $b$: bits allowed in $n$.

Another big problem:
CPUs do not try to protect metadata regarding array indices.

Actual time for $x[i]$ affects, and is affected by, detailed state of data cache, store-to-load forwarder, etc.

Exploited in, e.g., CacheBleed, despite Intel and OpenSSL claiming their code was safe.

Confidence-inspiring solution:
Avoid all data flow from secrets to memory addresses.
Double-and-add-always

Eliminate branches by always computing both results:

```python
def scalarmult(n,b,P):
    if b == 0: return 0
    R = scalarmult(n//2,b-1,P)
    R2 = R + R
    S = [R2, R2 + P]
    return S[n % 2]
```

Works for $0 \leq n < 2^b$.

Always takes $2^b$ additions (including $b$ doublings).

Public $b$: bits allowed in $n$.

Another big problem:

CPUs do not try to protect metadata regarding *array indices*.

Actual time for $x[i]$ affects, and is affected by, detailed state of data cache, store-to-load forwarder, etc.

Exploited in, e.g., CacheBleed, despite Intel and OpenSSL claiming their code was safe.

Confidence-inspiring solution:

*Avoid all data flow from secrets to memory addresses.*

Table lookups via arithmetic

Always read all table entries.

Use bit operations to select the desired table entry:

```python
def scalarmult(n,b,P):
    if b == 0: return 0
    R = scalarmult(n//2,b-1,P)
    R2 = R + R
    S = [R2, R2 + P]
    mask = -(n % 2)
    return S[0]^(mask&(S[1]^S[0]))
```
Double-and-add-always

Eliminate branches by always computing both results:

```python
def scalarmult(n,b,P):
    if b == 0: return 0
    R = scalarmult(n//2,b-1,P)
    R2 = R + R
    S = [R2,R2 + P]
    return S[n % 2]
```

Works for $0 \leq n < 2^b$.

Always takes $2^b$ additions (including $b$ doublings).

Use public $b$: bits allowed in $n$.

Another big problem:

CPUs do not try to protect metadata regarding *array indices*.

Actual time for $x[i]$ affects, and is affected by, detailed state of data cache, store-to-load forwarder, etc.

Exploited in, e.g., CacheBleed, despite Intel and OpenSSL claiming their code was safe.

Confidence-inspiring solution:

Avoid all data flow from secrets to memory addresses.

Table lookups via arithmetic

Always read all table entries.

Use bit operations to select the desired table entry:

```python
def scalarmult(n,b,P):
    if b == 0: return 0
    R = scalarmult(n//2,b-1,P)
    R2 = R + R
    S = [R2,R2 + P]
    mask = -(n % 2)
    return S[0]^(mask&(S[1]^S[0]))
```
Another big problem: CPUs do not try to protect metadata regarding *array indices*. Actual time for $x[i]$ affects, and is affected by, detailed state of data cache, store-to-load forwarder, etc. Exploited in, e.g., CacheBleed, despite Intel and OpenSSL claiming their code was safe. Confidence-inspiring solution: 

**Avoid all data flow from secrets to memory addresses.**

---

### Table lookups via arithmetic

Always read all table entries. Use bit operations to select the desired table entry:

```python
def scalarmult(n,b,P):
    if b == 0: return 0
    R = scalarmult(n//2,b-1,P)
    R2 = R + R
    S = [R2,R2 + P]
    mask = -(n % 2)
    return S[0]^(mask&(S[1]^S[0]))
```

---
Another big problem: CPUs do not try to protect metadata regarding *array indices*.

Actual time for $x[i]$ affects, and is affected by, detailed state of data cache, store-to-load forwarder, etc.

Exploited in, e.g., CacheBleed, despite Intel and OpenSSL claiming their code was safe.

Confidence-inspiring solution: **Avoid all data flow from secrets to memory addresses.**

---

**Table lookups via arithmetic**

Always read all table entries. Use bit operations to select the desired table entry:

```
def scalarmult(n,b,P):
    if b == 0: return 0
    R = scalarmult(n//2,b-1,P)
    R2 = R + R
    S = [R2,R2 + P]
    mask = -(n % 2)
    return S[0]^(mask&(S[1]^S[0]))
```
Another big problem: CPUs do not try to protect metadata regarding array indices. Actual time for $x[i]$ and is affected by, state of data cache, store-to-load forwarder, etc. Exploited in, e.g., CacheBleed, despite Intel and OpenSSL claiming their code was safe.

Confidence-inspiring solution: Avoid all data flow from secrets to memory addresses.

**Table lookups via arithmetic**

Always read all table entries. Use bit operations to select the desired table entry:

```python
def scalarmult(n, b, P):
    if b == 0:
        return 0
    R = scalarmult(n // 2, b - 1, P)
    R2 = R + R
    S = [R2, R2 + P]
    mask = -(n % 2)
    return S[0] ^ (mask & (S[1] ^ S[0]))
```

**Width-2 unsigned fixed windows**

```python
def fixwin2(n, b, table):
    if b <= 0:
        return 0
    T = table[0]
    mask = -(1 ^ (n % 4)) >> 2
    T ^= ~mask & (T ^ table[1])
    mask = -(2 ^ (n % 4)) >> 2
    T ^= ~mask & (T ^ table[2])
    mask = -(3 ^ (n % 4)) >> 2
    T ^= ~mask & (T ^ table[3])
    R = fixwin2(n // 4, b - 2, table)
    R = R + R
    R = R + R
    return R + T
```
Another big problem: CPUs do not try to protect metadata regarding array indices.

Actual time for $x[i]$ affects, and is affected by, detailed state of data cache, store-to-load forwarder, etc. Exploited in, e.g., CacheBleed, despite Intel and OpenSSL claiming their code was safe.

Confidence-inspiring solution:
Avoid all data flow from secrets to memory addresses.

Table lookups via arithmetic
Always read all table entries. Use bit operations to select the desired table entry:

```python
def scalarmult(n, b, P):
    if b == 0:
        return 0
    R = scalarmult(n//2, b-1, P)
    R2 = R + R
    S = [R2, R2 + P]
    mask = -(n % 2)
    return S[0]^(mask&(S[1]^S[0]))
```

Width-2 unsigned fixed windows
```python
def fixwin2(n, b, table):
    if b <= 0:
        return 0
    T = table[0]
    mask = -(1 ^ (n % 4)) >> 2
    T ^= ~mask & (T^table[1])
    mask = -(2 ^ (n % 4)) >> 2
    T ^= ~mask & (T^table[2])
    mask = -(3 ^ (n % 4)) >> 2
    T ^= ~mask & (T^table[3])
    R = fixwin2(n//4, b-2, table)
    R = R + R
    R = R + R
    return R + T
```
Table lookups via arithmetic

Always read all table entries. Use bit operations to select the desired table entry:

```python
def scalarmult(n, b, P):
    if b == 0:
        return 0
    R = scalarmult(n//2, b-1, P)
    R2 = R + R
    S = [R2, R2 + P]
    mask = -(n % 2)
    return S[0]^(mask&(S[1]^S[0]))
```

Width-2 unsigned fixed windows

```python
def fixwin2(n, b, table):
    if b <= 0:
        return 0
    T = table[0]
    mask = (-(1 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[1])
    mask = (-(2 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[2])
    mask = (-(3 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[3])
    R = fixwin2(n//4, b-2, table)
    R = R + R
    R = R + R
    return R + T
```
Table lookups via arithmetic

Always read all table entries. Use bit operations to select the desired table entry:

```python
def scalarmult(n,b,P):
    if b == 0: return 0
    R = scalarmult(n//2,b-1,P)
    R2 = R + R
    S = [R2,R2 + P]
    mask = -(n % 2)
    return S[0]^(mask&(S[1]^S[0]))
```

Width-2 unsigned fixed windows

```python
def fixwin2(n,b,table):
    if b <= 0: return 0
    T = table[0]
    mask = (-(1 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[1])
    mask = (-(2 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[2])
    mask = (-(3 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[3])
    R = fixwin2(n//4,b-2,table)
    R = R + R
    R = R + R
    return R + T
```
Table lookups via arithmetic
Always read all table entries.
Use bit operations to select the desired table entry:

```python
def scalarmult(n, b, P):
    if b == 0:
        return 0
    R = scalarmult(n // 2, b - 1, P)
    R2 = R + R
    S = [R2, R2 + P]
    mask = -(n % 2)
    return S[0] ^ (mask & (S[1] ^ S[0]))
```

Width-2 unsigned fixed windows

```python
def fixwin2(n, b, table):
    if b <= 0:
        return 0
    T = table[0]
    mask = (-(1 ^ (n % 4))) >> 2
    T ^= ~mask & (T ^ table[1])
    mask = (-(2 ^ (n % 4))) >> 2
    T ^= ~mask & (T ^ table[2])
    mask = (-(3 ^ (n % 4))) >> 2
    T ^= ~mask & (T ^ table[3])
    R = fixwin2(n // 4, b - 2, table)
    R = R + R
    R = R + R
    return R + T
```

Public branches, public indices.
For \( b \in \mathbb{Z} \):
Always \( b \) doublings.
Always \( b = 2 \) additions of \( T \).
Always 2 additions for table.
Can similarly protect larger-width fixed windows.
Unsigned is slightly easier.
Signed is slightly faster.
Table lookups via arithmetic
Always read all table entries.
Use bit operations to select
the desired table entry:

```python
def scalarmult(n, b, P):
    if b == 0:
        return 0
    R = scalarmult(n // 2, b - 1, P)
    R2 = R + R
    S = [R2, R2 + P]
    mask = -(n % 2)
    return S[0]^(mask&(S[1]^S[0]))
```

Width-2 unsigned fixed windows

```python
def fixwin2(n, b, table):
    if b <= 0:
        return 0
    T = table[0]
    mask = (-(1 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[1])
    mask = (-(2 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[2])
    mask = (-(3 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[3])
    R = fixwin2(n // 4, b - 2, table)
    R = R + R
    R = R + R
    return R + T
```

Public branches, public indices.
For $b \in 2\mathbb{Z}$:
Always $b$ doublings.
Always $b/2$ additions.
Always 2 additions for table.
Can similarly protect larger-width fixed windows.
Unsigned is slightly easier.
Signed is slightly faster.
Width-2 unsigned fixed windows

```python
def fixwin2(n,b,table):
    if b <= 0: return 0
    T = table[0]
    mask = (-((1 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[1])
    mask = (-((2 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[2])
    mask = (-((3 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[3])
    R = fixwin2(n//4,b-2,table)
    R = R + R
    R = R + R
    return R + T
```

Public branches, public indices.

For $b \in 2\mathbb{Z}$:
Always $b$ doublings.
Always $b/2$ additions of $T$.
Always 2 additions for table.

Can similarly protect larger-width fixed windows. Unsigned is slightly easier. Signed is slightly faster.

```python
def scalarmult(n,b,P):
    P2 = P+P
    table = [0,P,P2,P2+P]
    return fixwin2(n,b,table)
```
def fixwin2(n,b,table):
    if b <= 0: return 0
    T = table[0]
    mask = (-(1 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[1])
    mask = (-(2 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[2])
    mask = (-(3 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[3])
    R = fixwin2(n//4,b-2,table)
    R = R + R
    R = R + R
    return R + T

def scalarmult(n,b,P):
    P2 = P+P
    table = [0,P,P2,P2+P]
    return fixwin2(n,b,table)

Public branches, public indices.

For $b \in 2\mathbb{Z}$:
Always $b$ doublings.
Always $b/2$ additions of $T$.
Always 2 additions for table.

Can similarly protect larger-width fixed windows.
Unsigned is slightly easier.
Signed is slightly faster.
def fixwin2(n, b, table):
    if b <= 0:
        return 0
    T = table[0]
    mask = (-(1 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[1])
    mask = (-(2 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[2])
    mask = (-(3 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[3])
    R = fixwin2(n//4, b-2, table)
    R = R + R
    R = R + R
    return R + T

def scalarmult(n, b, P):
    P2 = P+P
    table = [0, P, P2, P2+P]
    return fixwin2(n, b, table)

Public branches, public indices.

For $b \in 2\mathbb{Z}$:
- Always $b$ doublings.
- Always $b/2$ additions of $T$.
- Always 2 additions for table.

Can similarly protect larger-width fixed windows.

Unsigned is slightly easier.
Signed is slightly faster.
def scalarmult(n,b,P):
    P2 = P+P
    table = [0,P,P2,P2+P]
    return fixwin2(n,b,table)

Public branches, public indices.

For $b \in 2\mathbb{Z}$:
Always $b$ doublings.
Always $b/2$ additions of $T$.
Always 2 additions for table.

Can similarly protect larger-width fixed windows.
Unsigned is slightly easier.
Signed is slightly faster.

Fixed-base scalar multiplication
Obvious way to handle keygen $a \mapsto aB$ and signing $r \mapsto rB$:
reuse $n, P \mapsto nP$ from ECDH.
def fixwin2(n,b,table):
    if b <= 0: return 0
    T = table[0]
    mask = (-(1 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[1])
    mask = (-(2 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[2])
    mask = (-(3 ^ (n % 4))) >> 2
    T ^= ~mask & (T^table[3])
    R = fixwin2(n//4,b-2,table)
    R = R + R
    R = R + R
    return R + T

def scalarmult(n,b,P):
    P2 = P+P
    table = [0,P,P2,P2+P]
    return fixwin2(n,b,table)

Public branches, public indices.
For \( b \in 2\mathbb{Z} \):
Always \( b \) doublings.
Always \( b/2 \) additions of \( T \).
Always 2 additions for table.
Can similarly protect larger-width fixed windows.
Unsigned is slightly easier.
Signed is slightly faster.

Fixed-base scalar multiplication

Obvious way to handle keygen \( a \mapsto aB \) and signing \( r \mapsto rB \):
reuse \( n, P \mapsto nP \) from ECDH.
def scalarmult(n,b,P):
    P2 = P+P
    table = [0,P,P2,P2+P]
    return fixwin2(n,b,table)

Public branches, public indices.

For $b \in \mathbb{Z}$:
Always $b$ doublings.
Always $b/2$ additions of $T$.
Always 2 additions for table.

Can similarly protect
larger-width fixed windows.
Unsigned is slightly easier.
Signed is slightly faster.

Fixed-base scalar multiplication

Obvious way to handle keygen
$a \mapsto aB$ and signing $r \mapsto rB$:
reuse $n, P \mapsto nP$ from ECDH.
def scalarmult(n,b,P):
    P2 = P+P
    table = [0,P,P2,P2+P]
    return fixwin2(n,b,table)

Public branches, public indices.

For \( b \in 2\mathbb{Z} \):
Always \( b \) doublings.
Always \( b/2 \) additions of \( T \).
Always 2 additions for table.

Can similarly protect larger-width fixed windows.
Unsigned is slightly easier.
Signed is slightly faster.

Fixed-base scalar multiplication

Obvious way to handle keygen \( a \mapsto aB \) and signing \( r \mapsto rB \):
reuse \( n, P \mapsto nP \) from ECDH.

Can do much better since \( B \) is a constant: standard base point.
e.g. For \( b = 256 \): Compute \((2^{128}n_1 + n_0)B\) as \( n_1B_1 + n_0B \)
using double-scalar fixed windows, after precomputing \( B_1 = 2^{128}B \).

Fun exercise: For each \( k \), try to minimize number of additions using \( k \) precomputed points.
def scalarmult(n, b, P):
    P2 = P + P
    table = [0, P, P2, P2 + P]
    return fixwin2(n, b, table)

Fixed-base scalar multiplication

Obvious way to handle keygen $a \mapsto aB$ and signing $r \mapsto rB$:
reuse $n, P \mapsto nP$ from ECDH.

Can do much better since $B$ is a constant: standard base point.

e.g. For $b = 256$: Compute $(2^{128}n_1 + n_0)B$ as $n_1B_1 + n_0B$
using double-scalar fixed windows, after precomputing $B_1 = 2^{128}B$.

Fun exercise: For each $k$, try to minimize number of additions
using $k$ precomputed points.

Recall Chou timings:
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.

ECDH is single-scalar mult.
Verification is double-scalar mult, somewhat slower than ECDH.
(But batch verification is faster.)

Keygen is fixed-base scalar mult, much faster than ECDH.

Signing is keygen plus overhead
depending on message length.
def scalarmult(n, b, P):
P2 = P + P
table = [0, P, P2, P2 + P]
return fixwin2(n, b, table)

Public branches, public indices.

For \( b \in \mathbb{Z}_2 \):
Always \( b \) doublings.
Always \( b = 2 \) additions of \( T \).
Always 2 additions for table.
Can similarly protect larger-width fixed windows.
Unsigned is slightly easier.
Signed is slightly faster.

Fixed-base scalar multiplication

Obvious way to handle keygen \( a \mapsto aB \) and signing \( r \mapsto rB \):
reuse \( n, P \mapsto nP \) from ECDH.
Can do much better since \( B \) is a constant: standard base point.

e.g. For \( b = 256 \):
Compute \( (2^{128} n_1 + n_0)B \) as \( n_1 B_1 + n_0 B \)
using double-scalar fixed windows,
after precomputing \( B_1 = 2^{128} B \).

Fun exercise: For each \( k \), try to minimize number of additions
using \( k \) precomputed points.

Recall Chou timings:
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.

ECDH is single-scalar mult.
Verification is double-scalar mult,
somewhat slower than ECDH.
(But batch verification is faster.)

Keygen is fixed-base scalar mult,
much faster than ECDH.

Signing is keygen plus overhead
depending on message length.
Fixed-base scalar multiplication

Obvious way to handle keygen $a \mapsto aB$ and signing $r \mapsto rB$: reuse $n, P \mapsto nP$ from ECDH.

Can do much better since $B$ is a constant: standard base point.

e.g. For $b = 256$: Compute $(2^{128}n_1 + n_0)B$ as $n_1B_1 + n_0B$ using double-scalar fixed windows, after precomputing $B_1 = 2^{128}B$.

Fun exercise: For each $k$, try to minimize number of additions using $k$ precomputed points.

Recall Chou timings:
- 57164 cycles for keygen,
- 63526 cycles for signature,
- 205741 cycles for verification,
- 159128 cycles for ECDH.

ECDH is single-scalar mult.

Verification is double-scalar somewhat slower than ECDH.
(But batch verification is faster.)

Keygen is fixed-base scalar much faster than ECDH.

Signing is keygen plus overhead depending on message length.
Fixed-base scalar multiplication

Obvious way to handle keygen $a \mapsto aB$ and signing $r \mapsto rB$: reuse $n, P \mapsto nP$ from ECDH.

Can do much better since $B$ is a constant: standard base point.

e.g. For $b = 256$: Compute $(2^{128}n_1 + n_0)B$ as $n_1B_1 + n_0B$
using double-scalar fixed windows, after precomputing $B_1 = 2^{128}B$.

Fun exercise: For each $k$, try to minimize number of additions using $k$ precomputed points.

Recall Chou timings:
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.

ECDH is single-scalar mult.

Verification is double-scalar mult, somewhat slower than ECDH.
(But batch verification is faster.)

Keygen is fixed-base scalar mult, much faster than ECDH.

Signing is keygen plus overhead depending on message length.
Fixed-base scalar multiplication

An obvious way to handle keygen $a \mapsto aB$ and signing $r \mapsto rB$: $P \mapsto nP$ from ECDH.

It can be much better since $B$ is constant: standard base point.

For example, $b = 256$: Compute $(n_1 B_1 + n_0 B)$ as $n_1 B_1 + n_0 B$ using double-scalar fixed windows, after precomputing $B_1 = 2^{128} B$. 

Exercise: For each $k$, try to minimize number of additions using $k$ precomputed points.

Recall Chou timings:
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.

ECDH is single-scalar mult.

Verification is double-scalar mult, somewhat slower than ECDH.
(But batch verification is faster.)

Keygen is fixed-base scalar mult, much faster than ECDH.

Signing is keygen plus overhead depending on message length.
Fixed-base scalar multiplication

Obvious way to handle keygen and signing $r \mapsto rB$: reuse $P \mapsto nP$ from ECDH.

Faster since $B$ is a constant: standard base point.

Compute $n_1B_1 + n_0B$ for fixed windows, e.g. $B_1 = 2^{128}B$.

For each $k$, try to minimize number of additions using $k$ precomputed points.

Recall Chou timings:
- 57164 cycles for keygen,
- 63526 cycles for signature,
- 205741 cycles for verification,
- 159128 cycles for ECDH.

ECDH is single-scalar mult.

Verification is double-scalar mult, somewhat slower than ECDH. (But batch verification is faster.)

Keygen is fixed-base scalar mult, much faster than ECDH.

Signing is keygen plus overhead depending on message length.

Let's move down a level:
ECC ops: e.g., verify $SB = R + hA$
Point ops: e.g., $P, Q \mapsto P + Q$
Field ops: e.g., $x_1, x_2 \mapsto x_1x_2$ in $F_p$
Machine insns: e.g., 32-bit multiplication
Gates: e.g., AND, OR, XOR
Recall Chou timings:
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.

ECDH is single-scalar mult.
Verification is double-scalar mult, somewhat slower than ECDH.
(But batch verification is faster.)
Keygen is fixed-base scalar mult, much faster than ECDH.
Signing is keygen plus overhead depending on message length.

Let’s move down a level:

ECC ops: e.g.,
verify $SB = R + hA$
windowing etc.

Point ops: e.g.,
$P, Q \mapsto P + Q$
faster doubling etc.

Field ops: e.g.,
$x_1, x_2 \mapsto x_1x_2$ in $\mathbb{F}_p$
delayed carries etc.

Machine insns: e.g.,
32-bit multiplication
pipelining etc.

Gates: e.g.,
AND, OR, XOR
Recall Chou timings:
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.

ECDH is single-scalar mult.

Verification is double-scalar mult, somewhat slower than ECDH.
(But batch verification is faster.)

Keygen is fixed-base scalar mult, much faster than ECDH.

Signing is keygen plus overhead depending on message length.

Let’s move down a level:

<table>
<thead>
<tr>
<th>ECC ops: e.g., verify $SB = R + hA$</th>
</tr>
</thead>
<tbody>
<tr>
<td>windowing etc.</td>
</tr>
<tr>
<td>Point ops: e.g., $P, Q \mapsto P + Q$</td>
</tr>
<tr>
<td>faster doubling etc.</td>
</tr>
<tr>
<td>Field ops: e.g., $x_1, x_2 \mapsto x_1 x_2$ in $\mathbb{F}_p$</td>
</tr>
<tr>
<td>delayed carries etc.</td>
</tr>
<tr>
<td>Machine insns: e.g., 32-bit multiplication</td>
</tr>
<tr>
<td>pipelining etc.</td>
</tr>
<tr>
<td>Gates: e.g., AND, OR, XOR</td>
</tr>
</tbody>
</table>
Recall Chou timings:
- 57,164 cycles for keygen,
- 63,526 cycles for signature,
- 205,741 cycles for verification,
- 159,128 cycles for ECDH.

- ECDH is single-scalar mult.
- Verification is double-scalar mult, somewhat slower than ECDH.
  (But batch verification is faster.)
- Keygen is fixed-base scalar mult, much faster than ECDH.
- Signing is keygen plus overhead depending on message length.

Let's move down a level:

**ECC ops:** e.g., verify $SB = R + hA$

- windowing etc.

**Point ops:** e.g., $P, Q \mapsto P + Q$

- faster doubling etc.

**Field ops:** e.g., $x_1, x_2 \mapsto x_1 x_2$ in $\mathbb{F}_p$

- delayed carries etc.

**Machine insns:** e.g., 32-bit multiplication

- pipelining etc.

**Gates:** e.g., AND, OR, XOR

Eliminating divisions

Have to do many additions of curve points: $P; Q \mapsto P + Q$.

How to efficiently decompose additions into field ops?

Addition $((x_1 y_2 + y_1 x_2) + (x_1 y_2 - x_1 x_2))$ uses expensive divisions.

$$ (x_1 y_2 + y_1 x_2) = (1 + dx_1 x_2 y_1 y_2), \\ (y_1 y_2 - x_1 x_2) = (1 - dx_1 x_2 y_1 y_2) $$
Recall Chou timings:
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.
ECDH is single-scalar mult.
Verification is double-scalar mult,
somewhat slower than ECDH.
(But batch verification is faster.)
Keygen is fixed-base scalar mult,
much faster than ECDH.
Signing is keygen plus overhead
depending on message length.

Let’s move down a level:

ECC ops: e.g.,
verify $SB = R + hA$
windowing etc.

Point ops: e.g.,
$P, Q \mapsto P + Q$
faster doubling etc.

Field ops: e.g.,
$x_1, x_2 \mapsto x_1 x_2$ in $\mathbb{F}_p$
delayed carries etc.

Machine insns: e.g.,
32-bit multiplication
pipelining etc.

Gates: e.g.,
AND, OR, XOR

Eliminating divisions
Have to do many additions of curve points: $P, Q \mapsto P + Q$.
How to efficiently decompose additions into field ops?

Addition $(x_1, y_1) + (x_2, y_2) = ((x_1 y_2 + y_1 x_2)/(1 + dx_1 x_2 y_1 y_2),
( y_1 y_2 − x_1 x_2)/(1 − dx_1 x_2 y_1 y_2))$ uses expensive division.
Recall Chou timings:
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.

ECDH is single-scalar mult.
Verification is double-scalar mult,
somewhat slower than ECDH.
(But batch verification is faster.)
Keygen is fixed-base scalar mult,
much faster than ECDH.
Signing is keygen plus overhead
depending on message length.

Let's move down a level:

ECC ops: e.g.,
verify $SB = R + hA$
windowing etc.

Point ops: e.g.,
$P, Q \rightarrow P + Q$
faster doubling etc.

Field ops: e.g.,
$x_1, x_2 \rightarrow x_1 x_2$ in $\mathbb{F}_p$
delayed carries etc.

Machine insns: e.g.,
32-bit multiplication
pipelining etc.

Gates: e.g.,
AND, OR, XOR

Eliminating divisions

Have to do many additions
of curve points: $P, Q \rightarrow P + Q$.

How to efficiently decompose
additions into field ops?

Addition $(x_1, y_1) + (x_2, y_2) =
((x_1 y_2 + y_1 x_2)/(1 + dx_1 x_2 y_1 y_2),
(y_1 y_2 - x_1 x_2)/(1 - dx_1 x_2 y_1 y_2))$
uses expensive divisions.
Let’s move down a level:

**ECC ops**: e.g., verify $SB = R + hA$

- windowing etc.

**Point ops**: e.g.,

- $P, Q \mapsto P + Q$

- faster doubling etc.

**Field ops**: e.g.,

- $x_1, x_2 \mapsto x_1x_2$ in $\mathbb{F}_p$

- delayed carries etc.

**Machine insns**: e.g.,

- 32-bit multiplication

- pipelining etc.

**Gates**: e.g.,

- AND, OR, XOR

---

**Eliminating divisions**

Have to do many additions of curve points: $P, Q \mapsto P + Q$.

How to efficiently decompose additions into field ops?

Addition $(x_1, y_1) + (x_2, y_2) = ((x_1y_2 + y_1x_2)/(1 + d x_1 x_2 y_1 y_2), (y_1 y_2 - x_1 x_2)/(1 - d x_1 x_2 y_1 y_2))$ uses expensive divisions.
Let’s move down a level:

**ECC ops:** e.g., verify $SB = R + hA$
- windowing etc.

**Point ops:** e.g.,
$$P, Q \mapsto P + Q$$
- faster doubling etc.

**Field ops:** e.g.,
$$x_1, x_2 \mapsto x_1 x_2 \text{ in } F_p$$
- delayed carries etc.

**Machine insns:** e.g.,
- 32-bit multiplication
- pipelining etc.

**Gates:** e.g.,
- AND, OR, XOR

---

**Eliminating divisions**

Have to do many additions of curve points: $P, Q \mapsto P + Q$. How to efficiently decompose additions into field ops?

Addition $(x_1, y_1) + (x_2, y_2) = ((x_1 y_2 + y_1 x_2)/(1 + dx_1 x_2 y_1 y_2), (y_1 y_2 - x_1 x_2)/(1 - dx_1 x_2 y_1 y_2))$ uses expensive divisions.

Better: postpone divisions and work with fractions.

Represent $(x, y)$ as $(X : Y : Z)$ with $x = X/Z, y = Y/Z, Z \neq 0$. 
Let's move down a level:

**ECC ops**: e.g.,

\[ IB = R + hA \]

(windowing etc.)

↓ ↓

**Point ops**: e.g.,

\[ P; Q \mapsto P + Q \]

(faster doubling etc.)

↓ ↓

**Field ops**: e.g.,

\[ x_1; x_2 \mapsto x_1 x_2 \text{ in } \mathbb{F}_p \]

(delayed carries etc.)

↓ ↓

**Machine insns**: e.g.,

32-bit multiplication

(pipelining etc.)

↓ ↓

**Gates**: e.g.,

AND, OR, XOR

---

Eliminating divisions

Have to do many additions of curve points: \( P, Q \mapsto P + Q \). How to efficiently decompose additions into field ops?

Addition \((x_1, y_1) + (x_2, y_2) = \)

\[ ((x_1 y_2 + y_1 x_2)/(1 + d x_1 x_2 y_1 y_2),
(\gamma y_2 - x_1 x_2)/(1 - d x_1 x_2 y_1 y_2)) \]

uses expensive divisions.

Better: postpone divisions and work with fractions.

Represent \((x, y)\) as \((X:Y:Z)\) with \(x = X/Z, y = Y/Z, Z \neq 0\).
Let's move down a level:

\[ \text{ECC ops: e.g., verify } SB = R + hA \]

Windowing etc.

↓ ↓

Point ops: e.g., \( P, Q \mapsto P + Q \)

Faster doubling etc.

↓ ↓

Field ops: e.g., \( x_1, x_2 \mapsto x_1 x_2 \) in \( F_p \)

Delayed carries etc.

↓ ↓

Machine insns: e.g., 32-bit multiplication

Pipelining etc.

↓ ↓

Gates: e.g., AND, OR, XOR

Eliminating divisions

Have to do many additions of curve points: \( P, Q \mapsto P + Q \).

How to efficiently decompose additions into field ops?

Addition \((x_1, y_1) + (x_2, y_2) = ((x_1 y_2 + y_1 x_2)/(1 + d x_1 x_2 y_1 y_2), (y_1 y_2 - x_1 x_2)/(1 - d x_1 x_2 y_1 y_2))\)

uses expensive divisions.

Better: postpone divisions and work with fractions.

Represent \((x, y)\) as \((X : Y : Z)\) with \(x = X/Z, y = Y/Z, Z \neq 0\).

Addition now has to handle fractions as:

\[
\left( \frac{X_1}{Z_1}, \frac{Y_1}{Z_1} \right) + \left( \frac{X_2}{Z_2}, \frac{Y_2}{Z_2} \right)
\]

\[
= \left( \frac{X_1 Y_2 + Y_1 X_2}{Z_1 Z_2}, \frac{Y_1 Y_2 - X_1 X_2}{Z_1 Z_2} \right)
\]

\[
1 + d \frac{X_1}{Z_1} \frac{X_2}{Z_2} \frac{Y_1}{Z_1} \frac{Y_2}{Z_2}
\]

\[
1 - d \frac{X_1}{Z_1} \frac{X_2}{Z_2} \frac{Y_1}{Z_1} \frac{Y_2}{Z_2}
\]
### Eliminating divisions

Have to do many additions of curve points: $P, Q \mapsto P + Q$. How to efficiently decompose additions into field ops?

Addition $(x_1, y_1) + (x_2, y_2) = ((x_1y_2 + y_1x_2)/(1 + dx_1x_2y_1y_2), (y_1y_2 - x_1x_2)/(1 - dx_1x_2y_1y_2))$ uses expensive divisions.

Better: postpone divisions and work with fractions.

Represent $(x, y)$ as $(X : Y : Z)$ with $x = X/Z, y = Y/Z, Z \neq 0$. 

---

### Addition now has to handle fractions as input:

$$
\begin{align*}
\left( \frac{X_1}{Z_1}, \frac{Y_1}{Z_1} \right) + \left( \frac{X_2}{Z_2}, \frac{Y_2}{Z_2} \right) &= \\
\left( \frac{\frac{X_1}{Z_1} \frac{Y_2}{Z_2} + \frac{Y_1}{Z_1} \frac{X_2}{Z_2}}{1 + d \frac{X_1}{Z_1} \frac{X_2}{Z_2} \frac{Y_1}{Z_1} \frac{Y_2}{Z_2}}, \frac{\frac{Y_1}{Z_1} \frac{Y_2}{Z_2} - \frac{X_1}{Z_1} \frac{X_2}{Z_2}}{1 - d \frac{X_1}{Z_1} \frac{X_2}{Z_2} \frac{Y_1}{Z_1} \frac{Y_2}{Z_2}} \right) &= 
\end{align*}
$$
Eliminating divisions

Have to do many additions of curve points: $P, Q \mapsto P + Q$.

How to efficiently decompose additions into field ops?

Addition $(x_1, y_1) + (x_2, y_2) = ((x_1 y_2 + y_1 x_2)/(1 + d x_1 x_2 y_1 y_2),
(y_1 y_2 - x_1 x_2)/(1 - d x_1 x_2 y_1 y_2))$ uses expensive divisions.

Better: postpone divisions and work with fractions.

Represent $(x, y)$ as $(X : Y : Z)$ with $x = X/Z, y = Y/Z, Z \neq 0$.

Addition now has to handle fractions as input:

$\left( \frac{X_1}{Z_1}, \frac{Y_1}{Z_1} \right) + \left( \frac{X_2}{Z_2}, \frac{Y_2}{Z_2} \right) =$

$\left( \frac{X_1 Y_2 Z_1 + Y_1 X_2 Z_1}{Z_1 Z_2}, \frac{X_1 Y_2 Z_1 - Y_1 X_2 Z_1}{Z_1 Z_2} \right) =$

$\left( \frac{X_1 Y_2 Z_1 + Y_1 X_2 Z_1}{Z_1 Z_2} + \frac{Y_1 X_2 Z_1 - X_1 Y_2 Z_1}{Z_1 Z_2}, \frac{Y_1 Y_2 Z_1 - X_1 X_2 Z_1}{Z_1 Z_2} \right)$
Eliminating divisions

Have to do many additions of curve points: $P, Q \mapsto P + Q$.
How to efficiently decompose additions into field ops?

Addition $(x_1, y_1) + (x_2, y_2) =$
$$((x_1y_2 + y_1x_2)/(1 + dx_1x_2y_1y_2),
(y_1y_2 - x_1x_2)/(1 - dx_1x_2y_1y_2))$$
uses expensive divisions.

Better: postpone divisions and work with fractions.

Represent $(x, y)$ as $(X : Y : Z)$
with $x = X/Z, y = Y/Z, Z \neq 0$.

Addition now has to handle fractions as input:

$$(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}) + (\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}) =$$

$$\left(\frac{X_1Y_2 + Y_1X_2}{Z_1Z_2}, \frac{Y_1Y_2 - X_1X_2}{Z_1Z_2}\right) =$$

$$\frac{Z_1Z_2(X_1Y_2 + Y_1X_2)}{Z_1^2Z_2^2 + dX_1X_2Y_1Y_2},$$

$$\frac{Z_1Z_2(Y_1Y_2 - X_1X_2)}{Z_1^2Z_2^2 - dX_1X_2Y_1Y_2}.$$
Eliminating divisions
Have to do many additions
of curve points: \( P; Q \mapsto P + Q \).

How to efficiently decompose
additions into field ops?

Addition \((x_1; y_1) + (x_2; y_2) = \)
\( \left(\frac{x_1 y_2 + y_1 x_2}{1 + dx_1 x_2 y_1 y_2}, \frac{y_1 y_2 - x_1 x_2}{1 - dx_1 x_2 y_1 y_2}\right) \)
uses expensive divisions.

Better: postpone divisions
and work with fractions.

Represent \((x; y)\) as \((X : Y : Z)\)
with \(x = \frac{X}{Z}, y = \frac{Y}{Z}, Z \neq 0\).

Addition now has to handle fractions as input:
\( \left(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}\right) + \left(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\right) = \)
\( \left(\frac{X_1}{Z_1} \frac{Y_2}{Z_2} + \frac{Y_1}{Z_1} \frac{X_2}{Z_2}}{1 + d\frac{X_1}{Z_1} \frac{X_2}{Z_2} \frac{Y_1}{Z_1} \frac{Y_2}{Z_2}}, \right) \)
\( \left(\frac{Y_1}{Z_1} \frac{Y_2}{Z_2} - \frac{X_1}{Z_1} \frac{X_2}{Z_2}}{1 - d\frac{X_1}{Z_1} \frac{X_2}{Z_2} \frac{Y_1}{Z_1} \frac{Y_2}{Z_2}} \right) = \)
\( \left(\frac{Z_1 Z_2 (X_1 Y_2 + Y_1 X_2)}{Z_1^2 Z_2^2 + dX_1 X_2 Y_1 Y_2}, \frac{Z_1 Z_2 (Y_1 Y_2 - X_1 X_2)}{Z_1^2 Z_2^2 - dX_1 X_2 Y_1 Y_2} \right) \)

i.e. \( \left(\frac{X_3}{Z_3}, \frac{Y_3}{Z_3}\right) \)

where
\( F = Z_1^2 + dX_1 X_2 Y_1 Y_2 \)
\( G = Z_1^2 - dX_1 X_2 Y_1 Y_2 \)
\( X_3 = Z_1 Z_2 (X_1 Y_2 + Y_1 X_2) / F \)
\( Y_3 = Z_1 Z_2 (Y_1 Y_2 - X_1 X_2) / G \)
\( Z_3 = F G \)

Input to addition algorithm:
\( X_1, Y_1, Z_1; X_2, Y_2, Z_2 \).

Output from addition algorithm:
\( X_3, Y_3, Z_3 \). No divisions needed!
Eliminating divisions
Have to do many additions of curve points: \( P, Q \mapsto P + Q \).

How to efficiently decompose additions into field ops?

Addition \((x_1; y_1) + (x_2; y_2) = (x_3; y_3)\) uses expensive divisions.

Better: postpone divisions and work with fractions.

Represent \((x; y)\) as \((X : Y : Z)\) with \(x = \frac{X}{Z}, y = \frac{Y}{Z}, Z \neq 0\).

Addition now has to handle fractions as input:

\[
\left( \frac{X_1}{Z_1}, \frac{Y_1}{Z_1} \right) + \left( \frac{X_2}{Z_2}, \frac{Y_2}{Z_2} \right) = \left( \frac{X_3}{Z_3}, \frac{Y_3}{Z_3} \right)
\]

where

\[
\begin{align*}
F &= Z_1^2 Z_2^2 - dX_1 Z_1 Z_2 Y_1 Y_2, \\
G &= Z_1^2 Z_2^2 + dX_1 Z_1 Z_2 Y_1 Y_2, \\
X_3 &= Z_1 Z_2 (X_1 Y_2 + Y_1 X_2), \\
Y_3 &= Z_1 Z_2 (Y_1 Y_2 - X_1 X_2), \\
Z_3 &= F G.
\end{align*}
\]

Input to addition algorithm: \(X_1, Y_1, Z_1, X_2, Y_2, Z_2\).

Output from addition algorithm: \(X_3, Y_3, Z_3\). No divisions needed!
Eliminating divisions
Have to do many additions of curve points: \( P; Q \mapsto P + Q \).

How to efficiently decompose additions into field ops?

Addition \((x_1; y_1) + (x_2; y_2) = (x_1 y_2 + y_1 x_2; 1 + dx_1 x_2 y_1 y_2)

uses expensive divisions.

Better: postpone divisions and work with fractions.

Represent \((x; y)\) as \((X:Y:Z)\) with \(x = \frac{X}{Z}, y = \frac{Y}{Z}, Z \neq 0\).

Addition now has to handle fractions as input:

\[
\left(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}\right) + \left(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\right) = \left(\frac{X_1 Y_2 + Y_1 X_2}{Z_1 Z_2}, \frac{1 + d x_1 x_2 y_1 y_2}{Z_1 Z_2}\right)
\]

where

\[
F = Z_1^2 Z_2^2 - d X_1 X_2 Y_1 Y_2,
G = Z_1^2 Z_2^2 + d X_1 X_2 Y_1 Y_2,
X_3 = Z_1 Z_2 (X_1 Y_2 + Y_1 X_2) F,
Y_3 = Z_1 Z_2 (Y_1 Y_2 - X_1 X_2) G,
Z_3 = FG.
\]

Input to addition algorithm: \(X_1, Y_1, Z_1, X_2, Y_2, Z_2\).
Output from addition algorithm: \(X_3, Y_3, Z_3\). No divisions needed!
Addition now has to handle fractions as input:

\[
\left( \frac{X_1}{Z_1}, \frac{Y_1}{Z_1} \right) + \left( \frac{X_2}{Z_2}, \frac{Y_2}{Z_2} \right) =
\]

\[
\left( \frac{X_1 Y_2}{Z_1 Z_2} + \frac{Y_1 X_2}{Z_1 Z_2}, \frac{Y_1 Y_2 - X_1 X_2}{Z_1 Z_2} \right) =
\]

\[
\left( \frac{Z_1 Z_2 (X_1 Y_2 + Y_1 X_2)}{Z_1 Z_2 + dX_1 X_2 Y_1 Y_2}, \frac{Z_1 Z_2 (Y_1 Y_2 - X_1 X_2)}{Z_1 Z_2 - dX_1 X_2 Y_1 Y_2} \right)
\]

i.e. \( \left( \frac{X_1}{Z_1}, \frac{Y_1}{Z_1} \right) + \left( \frac{X_2}{Z_2}, \frac{Y_2}{Z_2} \right) = \left( \frac{X_3}{Z_3}, \frac{Y_3}{Z_3} \right) \)

where

\( F = Z_1 Z_2 - dX_1 X_2 Y_1 Y_2, \)
\( G = Z_1 Z_2 + dX_1 X_2 Y_1 Y_2, \)
\( X_3 = Z_1 Z_2 (X_1 Y_2 + Y_1 X_2) F, \)
\( Y_3 = Z_1 Z_2 (Y_1 Y_2 - X_1 X_2) G, \)
\( Z_3 = FG. \)

Input to addition algorithm:
\( X_1, Y_1, Z_1, X_2, Y_2, Z_2. \)

Output from addition algorithm:
\( X_3, Y_3, Z_3. \) No divisions needed!
Addition now has to handle fractions as input:
\[
\left( \frac{X_1}{Z_1}, \frac{Y_1}{Z_1} \right) + \left( \frac{X_2}{Z_2}, \frac{Y_2}{Z_2} \right) = \left( \frac{X_3}{Z_3}, \frac{Y_3}{Z_3} \right)
\]

i.e. \( \left( \frac{X_1}{Z_1}, \frac{Y_1}{Z_1} \right) + \left( \frac{X_2}{Z_2}, \frac{Y_2}{Z_2} \right) = \left( \frac{X_3}{Z_3}, \frac{Y_3}{Z_3} \right) \)

where
\[
\begin{align*}
F &= Z_1^2 Z_2^2 - dX_1 X_2 Y_1 Y_2, \\
G &= Z_1^2 Z_2^2 + dX_1 X_2 Y_1 Y_2, \\
X_3 &= Z_1 Z_2 (X_1 Y_2 + Y_1 X_2) F, \\
Y_3 &= Z_1 Z_2 (Y_1 Y_2 - X_1 X_2) G, \\
Z_3 &= FG.
\end{align*}
\]

Input to addition algorithm:
\( X_1, Y_1, Z_1, X_2, Y_2, Z_2 \).

Output from addition algorithm:
\( X_3, Y_3, Z_3 \). No divisions needed!

Eliminate common subexpressions to save multiplications:
\[
\begin{align*}
A &= Z_1 Z_2, \\
B &= A^2, \\
C &= X_1 X_2, \\
D &= Y_1 Y_2, \\
E &= d \cdot C \cdot D, \\
F &= B - E, \\
G &= B + E, \\
X_3 &= A \cdot F \cdot (X_1 Y_2 + Y_1 X_2), \\
Y_3 &= A \cdot G \cdot (Y_1 Y_2 - X_1 X_2), \\
Z_3 &= F G.
\end{align*}
\]

Cost: 11 \( M \) + 1 \( S \) + 1 \( M \) \( d \)

Choose small \( d \) for cheap \( M \) \( d \).
Addition now has to handle fractions as input:
\[
\frac{X_1}{Z_1} + \frac{Y_1}{Z_1} = \frac{X_2}{Z_2} + \frac{Y_2}{Z_2}
\]

\[
= \left( \frac{X_3}{Z_3}, \frac{Y_3}{Z_3} \right)
\]

where

\[
F = Z_1^2 Z_2^2 - dX_1 X_2 Y_1 Y_2,
\]

\[
G = Z_1^2 Z_2^2 + dX_1 X_2 Y_1 Y_2,
\]

\[
X_3 = Z_1 Z_2 (X_1 Y_2 + Y_1 X_2) F,
\]

\[
Y_3 = Z_1 Z_2 (Y_1 Y_2 - X_1 X_2) G,
\]

\[
Z_3 = F G.
\]

Input to addition algorithm:
\(X_1, Y_1, Z_1, X_2, Y_2, Z_2\).

Output from addition algorithm:
\(X_3, Y_3, Z_3\). No divisions needed!

Eliminate common subexpressions to save multiplications:
\(A = Z_1 \cdot Z_2; B = A^2; C = X_1 \cdot X_2; D = Y_1 \cdot Y_2; E = d \cdot C \cdot D; F = B - E; G = B + E; X_3 = A \cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot X_2) F; Y_3 = A \cdot G \cdot (D - C) G; Z_3 = F \cdot G.\)

Cost: 11\(M\) + 1\(S\) + 1\(M\)\(d\)

\(M, S\) are costs of mult, square.

Choose small \(d\) for cheap \(M\)\(d\).
Addition now has to handle fractions as input:
\[
\left( \frac{X_1}{Z_1}, \frac{Y_1}{Z_1} \right) + \left( \frac{X_2}{Z_2}, \frac{Y_2}{Z_2} \right) = \left( \frac{X_3}{Z_3}, \frac{Y_3}{Z_3} \right)
\]

where
\[
F = Z_1^2Z_2^2 - dX_1X_2Y_1Y_2, \\
G = Z_1^2Z_2^2 + dX_1X_2Y_1Y_2, \\
X_3 = Z_1Z_2(X_1Y_2 + Y_1X_2)F, \\
Y_3 = Z_1Z_2(Y_1Y_2 - X_1X_2)G, \\
Z_3 = FG.
\]

Input to addition algorithm:
\(X_1, Y_1, Z_1, X_2, Y_2, Z_2\).
Output from addition algorithm:
\(X_3, Y_3, Z_3\). No divisions needed!

Eliminate common subexpressions to save multiplications:
\[
A = Z_1 \cdot Z_2; \quad B = A^2; \\
C = X_1 \cdot X_2; \\
D = Y_1 \cdot Y_2; \\
E = d \cdot C \cdot D; \\
F = B - E; \quad G = B + E; \\
X_3 = A \cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot X_2); \\
Y_3 = A \cdot G \cdot (D - C); \\
Z_3 = F \cdot G.
\]

Cost: \(11M + 1S + 1M_d\) where \(M, S\) are costs of mult, square.
Choose small \(d\) for cheap \(M_d\).
\[(X_1, Y_1) + (X_2, Y_2) = \left(\frac{X_3}{Z_3}, \frac{Y_3}{Z_3}\right)\]

where
\[
F = Z_1^2 Z_2^2 - dX_1 X_2 Y_1 Y_2, \\
G = Z_1^2 Z_2^2 + dX_1 X_2 Y_1 Y_2, \\
X_3 = Z_1 Z_2 (X_1 Y_2 + Y_1 X_2) F, \\
Y_3 = Z_1 Z_2 (Y_1 Y_2 - X_1 X_2) G, \\
Z_3 = FG.
\]

Input to addition algorithm:
\(X_1, Y_1, Z_1, X_2, Y_2, Z_2\).

Output from addition algorithm:
\(X_3, Y_3, Z_3\). No divisions needed!

Eliminate common subexpressions to save multiplications:
\[
A = Z_1 \cdot Z_2; \quad B = A^2; \\
C = X_1 \cdot X_2; \\
D = Y_1 \cdot Y_2; \\
E = d \cdot C \cdot D; \\
F = B - E; \quad G = B + E; \\
X_3 = A \cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot X_2); \\
Y_3 = A \cdot G \cdot (D - C); \\
Z_3 = F \cdot G.
\]

Cost: \(11M + 1S + 1M_d\) where \(M, S\) are costs of mult, square. Choose small \(d\) for cheap \(M_d\).
\[
\left( \frac{X_1}{Z_1}, \frac{Y_1}{Z_1} \right) + \left( \frac{X_2}{Z_2}, \frac{Y_2}{Z_2} \right) = \left( \frac{X_3}{Z_3}, \frac{Y_3}{Z_3} \right)
\]

Eliminate common subexpressions to save multiplications:

\[
A = Z_1 \cdot Z_2; \quad B = A^2; \\
C = X_1 \cdot X_2; \\
D = Y_1 \cdot Y_2; \\
E = d \cdot C \cdot D; \\
F = B - E; \quad G = B + E; \\
X_3 = A \cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot X_2); \\
Y_3 = A \cdot G \cdot (D - C); \\
Z_3 = F \cdot G.
\]

Cost: \( 11M + 1S + 1M_d \) where \( M, S \) are costs of mult, square.

Choose small \( d \) for cheap \( M_d \).

Can do better: \( 10M + 1S + 1M_d \) }

Obvious \( 4M \) method to compute product \( C + Mt + Dt \) of polys \( X_1 + Y_1t, X_2 + Y_2t \):

\[
C = X_1 \cdot X_2; \\
D = Y_1 \cdot Y_2; \\
M = X_1 \cdot Y_2 + Y_1 \cdot X_2.
\]
\[
\begin{pmatrix}
X_2 & Y_2 \\
Z_2' & Z_2
\end{pmatrix}
\]

Eliminate common subexpressions to save multiplications:

\[ A = Z_1 \cdot Z_2; \quad B = A^2; \]
\[ C = X_1 \cdot X_2; \]
\[ D = Y_1 \cdot Y_2; \]
\[ E = d \cdot C \cdot D; \]
\[ F = B - E; \quad G = B + E; \]
\[ X_3 = A \cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot X_2); \]
\[ Y_3 = A \cdot G \cdot (D - C); \]
\[ Z_3 = F \cdot G. \]

Cost: \(11M + 1S + 1M_d\) where \(M, S\) are costs of mult, square.

Choose small \(d\) for cheap \(M_d\).

Can do better: \(10M + 1S + 1M_d\).

Obvious 4M method to compute product \(C + Mt + Dt^2\) of polys \(X_1 + Y_1t, X_2 + Y_2t\):

\[ C = X_1 \cdot X_2; \]
\[ D = Y_1 \cdot Y_2; \]
\[ M = X_1 \cdot Y_2 + Y_1 \cdot X_2. \]
Eliminate common subexpressions to save multiplications:

\[ A = Z_1 \cdot Z_2; \quad B = A^2; \]
\[ C = X_1 \cdot X_2; \]
\[ D = Y_1 \cdot Y_2; \]
\[ E = d \cdot C \cdot D; \]
\[ F = B - E; \quad G = B + E; \]
\[ X_3 = A \cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot X_2); \]
\[ Y_3 = A \cdot G \cdot (D - C); \]
\[ Z_3 = F \cdot G. \]

Cost: 11M + 1S + 1M_d where M, S are costs of mult, square. Choose small d for cheap M_d.

Can do better: 10M + 1S + 1M_d.

Obvious 4M method to compute product \( C + Mt + Dt^2 \) of polys \( X_1 + Y_1 \cdot t, X_2 + Y_2 \cdot t \):

\[ C = X_1 \cdot X_2; \]
\[ D = Y_1 \cdot Y_2; \]
\[ M = X_1 \cdot Y_2 + Y_1 \cdot X_2. \]
Eliminate common subexpressions to save multiplications:

\[ A = Z_1 \cdot Z_2; \quad B = A^2; \]
\[ C = X_1 \cdot X_2; \]
\[ D = Y_1 \cdot Y_2; \]
\[ E = d \cdot C \cdot D; \]
\[ F = B - E; \quad G = B + E; \]
\[ X_3 = A \cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot X_2); \]
\[ Y_3 = A \cdot G \cdot (D - C); \]
\[ Z_3 = F \cdot G. \]

Cost: \( 11M + 1S + 1M_d \) where \( M, S \) are costs of mult, square.

Choose small \( d \) for cheap \( M_d \).

Can do better: \( 10M + 1S + 1M_d \).

Obvious 4M method to compute product \( C + Mt + Dt^2 \) of polys \( X_1 + Y_1 t, X_2 + Y_2 t \):

\[ C = X_1 \cdot X_2; \]
\[ D = Y_1 \cdot Y_2; \]
\[ M = X_1 \cdot Y_2 + Y_1 \cdot X_2. \]
Eliminate common subexpressions to save multiplications:

\[ A = Z_1 \cdot Z_2; \quad B = A^2; \]
\[ C = X_1 \cdot X_2; \]
\[ D = Y_1 \cdot Y_2; \]
\[ E = d \cdot C \cdot D; \]
\[ F = B - E; \quad G = B + E; \]
\[ X_3 = A \cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot X_2); \]
\[ Y_3 = A \cdot G \cdot (D - C); \]
\[ Z_3 = F \cdot G. \]

Cost: \(11M + 1S + 1M_d\) where \(M, S\) are costs of mult, square. Choose small \(d\) for cheap \(M_d\).

Can do better: \(10M + 1S + 1M_d\).

Obvious 4M method to compute product \(C + Mt + Dt^2\) of polys \(X_1 + Y_1 t, X_2 + Y_2 t\):

\[ C = X_1 \cdot X_2; \]
\[ D = Y_1 \cdot Y_2; \]
\[ M = X_1 \cdot Y_2 + Y_1 \cdot X_2. \]

Karatsuba’s 3M method:

\[ C = X_1 \cdot X_2; \]
\[ D = Y_1 \cdot Y_2; \]
\[ M = (X_1 + Y_1) \cdot (X_2 + Y_2) - C - D. \]
Eliminate common subexpressions to save multiplications:
\[ A = Z_1 \cdot Z_2; \quad B = A^2; \]
\[ C = X_1 \cdot X_2; \quad D = Y_1 \cdot Y_2; \]
\[ E = d \cdot C \cdot D; \]
\[ F = B - E; \quad G = B + E; \]
\[ C \cdot D; \]
\[ G \cdot (D - C); \]
\[ G. \]

1M + 1S + 1M_d where \( M \) and \( S \) are costs of mult, square.
Choose small \( d \) for cheap \( M_d \).

Can do better: \( 10M + 1S + 1M_d \).

Obvious \( 4M \) method to compute product \( C + Mt + Dt^2 \) of polys \( X_1 + Y_1t, \ X_2 + Y_2t \):
\[ C = X_1 \cdot X_2; \]
\[ D = Y_1 \cdot Y_2; \]
\[ M = X_1 \cdot Y_2 + Y_1 \cdot X_2. \]

Karatsuba’s \( 3M \) method:
\[ C = X_1 \cdot X_2; \]
\[ D = Y_1 \cdot Y_2; \]
\[ M = (X_1 + Y_1) \cdot (X_2 + Y_2) - C - D. \]

Faster doubling:
\[ (x_1, y_1) + (x_1, y_1) = ((x_1y_1 + y_1x_1) = (1 + dx_1x_1y_1y_1), \]
\[ (y_1y_1 - x_1x_1) = (1 - dx_1y_1y_1x_1); \]
\[ ((2x_1y_1) = (1 + dx_2x_2y_2), \]
\[ (y^2_1 - x^2_1) = (1 - dx_2y_2x_2y_2). \]
Eliminate common subexpressions to save multiplications:

\[ A = Z_1 \cdot Z_2; \]
\[ B = A^2; \]
\[ C = X_1 \cdot X_2; \]
\[ D = Y_1 \cdot Y_2; \]
\[ E = d \cdot C \cdot D; \]
\[ F = B - E; \]
\[ G = B + E; \]
\[ X_3 = A \cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot X_2); \]
\[ Y_3 = A \cdot G \cdot (D - C); \]
\[ Z_3 = F \cdot G. \]

Cost: \(11M + 1S + 1M_d\) where \(M\) and \(S\) are costs of mult, square. Choose small \(d\) for cheap \(M_d\).

Can do better: \(10M + 1S + 1M_d\).

Obvious 4M method to compute product \(C + Mt + Dt^2\) of polys \(X_1 + Y_1 t, X_2 + Y_2 t\):

\[ C = X_1 \cdot X_2; \]
\[ D = Y_1 \cdot Y_2; \]
\[ M = X_1 \cdot Y_2 + Y_1 \cdot X_2. \]

Karatsuba’s 3M method:

\[ C = X_1 \cdot X_2; \]
\[ D = Y_1 \cdot Y_2; \]
\[ M = (X_1 + Y_1) \cdot (X_2 + Y_2) - C - D. \]

(\(x_1, y_1\) + \((x_1, y_1) = ((x_1y_1+y_1x_1)/(1+d), (y_1y_1-x_1x_1)/(1-d), ((2x_1y_1)/(1+d), (y_1^2-x_1^2)/(1-d). \)

Faster doubling:

(\(x_1, y_1\) + \((x_1, y_1) = ((x_1y_1+y_1x_1)/(1+d), (y_1y_1-x_1x_1)/(1-d), ((2x_1y_1)/(1+d), (y_1^2-x_1^2)/(1-d). \)

(\(x_1, y_1\) + \((x_1, y_1) = ((x_1y_1+y_1x_1)/(1+d), (y_1y_1-x_1x_1)/(1-d), ((2x_1y_1)/(1+d), (y_1^2-x_1^2)/(1-d). \)
Eliminate common subexpressions to save multiplications:

\[ A = Z_1 \cdot Z_2; \]
\[ B = A^2; \]
\[ C = X_1 \cdot X_2; \]
\[ D = Y_1 \cdot Y_2; \]
\[ E = d \cdot C \cdot D; \]
\[ F = B - E; \]
\[ G = B + E; \]
\[ X_3 = A \cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot X_2); \]
\[ Y_3 = A \cdot G \cdot (D - C); \]
\[ Z_3 = F \cdot G. \]

Cost: \(11M + 1S + 1M_d\) where \(M, S\) are costs of mult, square.

Choose small \(d\) for cheap \(M_d\).

Can do better: \(10M + 1S + 1M_d\).

Obvious 4M method to compute product \(C + Mt + Dt^2\) of polys \(X_1 + Y_1t, X_2 + Y_2t:\)

\[ C = X_1 \cdot X_2; \]
\[ D = Y_1 \cdot Y_2; \]
\[ M = X_1 \cdot Y_2 + Y_1 \cdot X_2. \]

Karatsuba’s 3M method:

\[ C = X_1 \cdot X_2; \]
\[ D = Y_1 \cdot Y_2; \]
\[ M = (X_1 + Y_1) \cdot (X_2 + Y_2) - C - D. \]

Faster doubling

\((x_1, y_1) + (x_1, y_1) = ((x_1 y_1 + y_1 x_1)/(1 + dx_1 x_1 y_1 y_1), (y_1 y_1 - x_1 x_1)/(1 - dx_1 x_1 y_1 y_1)) = ((2x_1 y_1)/(1 + dx_1^2 y_1^2), (y_1^2 - x_1^2)/(1 - dx_1^2 y_1^2)).\)
Can do better: $10\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$.

Obvious $4\mathbf{M}$ method to compute product $C + Mt + Dt^2$ of polys $X_1 + Y_1 t$, $X_2 + Y_2 t$:

$$C = X_1 \cdot X_2;$$
$$D = Y_1 \cdot Y_2;$$
$$M = X_1 \cdot Y_2 + Y_1 \cdot X_2.$$

Karatsuba’s $3\mathbf{M}$ method:

$$C = X_1 \cdot X_2;$$
$$D = Y_1 \cdot Y_2;$$
$$M = (X_1 + Y_1) \cdot (X_2 + Y_2) - C - D.$$

Faster doubling

$$(x_1, y_1) + (x_1, y_1) =$$
$$((x_1 y_1 + y_1 x_1)/(1 + d x_1 x_1 y_1 y_1),$$
$$(y_1 y_1 - x_1 x_1)/(1 - d x_1 x_1 y_1 y_1)) =$$
$$((2 x_1 y_1)/(1 + d x_1^2 y_1^2),$$
$$(y_1^2 - x_1^2)/(1 - d x_1^2 y_1^2)).$$
Can do better: $10\text{M} + 1\text{S} + 1\text{M}_d$.

Obvious $4\text{M}$ method to compute product $C + Mt + Dt^2$
of polys $X_1 + Y_1 t$, $X_2 + Y_2 t$:

\[
C = X_1 \cdot X_2;
D = Y_1 \cdot Y_2;
M = X_1 \cdot Y_2 + Y_1 \cdot X_2.
\]

Karatsuba’s $3\text{M}$ method:

\[
C = X_1 \cdot X_2;
D = Y_1 \cdot Y_2;
M = (X_1 + Y_1) \cdot (X_2 + Y_2) - C - D.
\]

Faster doubling

\[
(x_1, y_1) + (x_1, y_1) =

((x_1 y_1 + y_1 x_1)/(1 + d x_1 x_1 y_1 y_1),
(y_1 y_1 - x_1 x_1)/(1 - d x_1 x_1 y_1 y_1)) =

((2 x_1 y_1)/(1 + d x_1^2 y_1^2),
(y_1^2 - x_1^2)/(1 - d x_1^2 y_1^2)).
\]

$x_1^2 + y_1^2 = 1 + d x_1^2 y_1^2$ so

\[
(x_1, y_1) + (x_1, y_1) =

((2 x_1 y_1)/(x_1^2 + y_1^2),
(y_1^2 - x_1^2)/(2 - x_1^2 - y_1^2)).
\]
Can do better: $10M + 1S + 1M_d$.

Obvious 4M method to compute product $C + Mt + Dt^2$ of polys $X_1 + Y_1 t$, $X_2 + Y_2 t$:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = X_1 \cdot Y_2 + Y_1 \cdot X_2$.

Karatsuba’s 3M method:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = (X_1 + Y_1) \cdot (X_2 + Y_2) − C − D$.

Faster doubling

$(x_1, y_1) + (x_1, y_1) =$

\[
\begin{align*}
((x_1 y_1 + y_1 x_1)/(1 + dx_1 x_1 y_1 y_1), \\
(y_1 y_1 - x_1 x_1)/(1 - dx_1 x_1 y_1 y_1)) = \\
((2x_1 y_1)/(1 + dx_1^2 y_1^2), \\
(y_1^2 - x_1^2)/(1 - dx_1^2 y_1^2)).
\end{align*}
\]

$x_1^2 + y_1^2 = 1 + dx_1^2 y_1^2$ so

$(x_1, y_1) + (x_1, y_1) =$

\[
\begin{align*}
((2x_1 y_1)/(x_1^2 + y_1^2), \\
(y_1^2 - x_1^2)/(2 - x_1^2 - y_1^2)).
\end{align*}
\]

Again eliminate divisions using $(X : Y : Z)$: only $3M + 4S$. Much faster than addition.
Can do better: \(10M + 1S + 1M_d\).

4M method to compute product \(C + Mt + Dt^2\) of \(X_1 + Y_1 t\), \(X_2 + Y_2 t\):

\[
\begin{align*}
\cdot X_2; \\
\cdot Y_2; \\
\cdot Y_2 + Y_1 \cdot X_2.
\end{align*}
\]

Karatsuba’s 3M method:

\[
\begin{align*}
\cdot X_2; \\
\cdot Y_2; \\
\cdot (X_1 + Y_1) \cdot (X_2 + Y_2) - C - D.
\end{align*}
\]

Faster doubling

\[
\begin{align*}
(x_1, y_1) + (x_1, y_1) = \\
((x_1 y_1 + y_1 x_1)/(1 + d x_1 x_1 y_1 y_1), \\
(y_1 y_1 - x_1 x_1)/(1 - d x_1 x_1 y_1 y_1)) = \\
((2x_1 y_1)/(1 + dx_1^2 y_1^2), \\
(y_1^2 - x_1^2)/(1 - dx_1^2 y_1^2)).
\end{align*}
\]

\[
x_1^2 + y_1^2 = 1 + dx_1^2 y_1^2 \text{ so}
\]

\[
\begin{align*}
(x_1, y_1) + (x_1, y_1) = \\
((2x_1 y_1)/(x_1^2 + y_1^2), \\
(y_1^2 - x_1^2)/(2 - x_1^2 - y_1^2)).
\end{align*}
\]

Again eliminate divisions using \((X : Y : Z)\): only \(3M + 4S\).

Much faster than addition.

More addition strategies

Dual addition formula:

\[
\begin{align*}
(x_1, y_1) + (x_1, y_1) = \\
((x_1 y_1 + y_1 x_1)/(1 + dx_1 x_1 y_1 y_1), \\
(y_1 y_1 - x_1 x_1)/(1 - dx_1 x_1 y_1 y_1)) = \\
((2x_1 y_1)/(1 + dx_1^2 y_1^2), \\
(y_1^2 - x_1^2)/(1 - dx_1^2 y_1^2)).
\end{align*}
\]

Low degree, no need for \(d\).
Can do better: $10M + 1S + 1M_d$.

Obvious $4M$ method to compute product $C + Mt + Dt$

Karatsuba's $3M$ method:

$$C = x_1 \cdot x_2;$$
$$D = y_1 \cdot y_2;$$
$$M = (x_1 + y_1) \cdot (x_2 + y_2) - C - D.$$ 

Faster doubling

$$(x_1, y_1) + (x_1, y_1) = ((x_1y_1 + y_1x_1)/(1 + dx_1x_1y_1),$$
$$(y_1y_1 - x_1x_1)/(1 - dx_1x_1y_1)) = ((2x_1y_1)/(1 + dx_1^2 y_1^2),$$
$$(y_1^2 - x_1^2)/(1 - dx_1^2 y_1^2)).$$

$$x_1^2 + y_1^2 = 1 + dx_1^2 y_1^2$$ so

$$(x_1, y_1) + (x_1, y_1) = ((2x_1y_1)/(x_1^2 + y_1^2),$$
$$(y_1^2 - x_1^2)/(2 - x_1^2 - y_1^2)).$$

Again eliminate divisions using $(X : Y : Z)$: only $3M + 4S$.

Much faster than addition.

More addition strategies

Dual addition formula:

$$(x_1, y_1) + (x_2, y_2) = ((x_1y_1 + x_2y_2)/(x_1y_1 + x_2y_2),$$
$$(x_1y_1 - x_2y_2)/(x_1y_1 - x_2y_2)).$$

Low degree, no need for $d$. 

Again eliminate divisions using $(X : Y : Z)$: only $3M + 4S$.

Much faster than addition.
Faster doubling

\[(x_1, y_1) + (x_1, y_1) =
((x_1 y_1 + y_1 x_1)/(1 + d x_1 x_1 y_1 y_1),
(y_1 y_1 - x_1 x_1)/(1 - d x_1 x_1 y_1 y_1)) =
((2x_1 y_1)/(1 + d x_1^2 y_1^2),
(y_1^2 - x_1^2)/(1 - d x_1^2 y_1^2)).
\]

\[x_1^2 + y_1^2 = 1 + d x_1^2 y_1^2 \text{ so}
(x_1, y_1) + (x_1, y_1) =
((2x_1 y_1)/(x_1^2 + y_1^2),
(y_1^2 - x_1^2)/(2 - x_1^2 - y_1^2)).
\]

Again eliminate divisions using \((X : Y : Z)\): only \(3\text{M} + 4\text{S}\).

Much faster than addition.

---

More addition strategies

Dual addition formula:

\[(x_1, y_1) + (x_2, y_2) =
((x_1 y_1 + x_2 y_2)/(x_1 x_2 + y_1 y_2),
(x_1 y_1 - x_2 y_2)/(x_1 y_2 - x_2 y_1)) =
((2x_1 y_1)/(x_1^2 + y_1^2),
(y_1^2 - x_1^2)/(2 - x_1^2 - y_1^2)).
\]

Low degree, no need for \(d\).
**Faster doubling**

\[(x_1, y_1) + (x_1, y_1) = ((x_1 y_1 + y_1 x_1)/(1 + dx_1 x_1 y_1 y_1),
\quad (y_1 y_1 - x_1 x_1)/(1 - dx_1 x_1 y_1 y_1)) =
\quad ((2x_1 y_1)/(1 + dx_1^2 y_1^2),
\quad (y_1^2 - x_1^2)/(1 - dx_1^2 y_1^2)).\]

\[x_1^2 + y_1^2 = 1 + dx_1^2 y_1^2 \quad \text{so} \]
\[x_1, y_1) + (x_1, y_1) =
\quad ((2x_1 y_1)/(x_1^2 + y_1^2),
\quad (y_1^2 - x_1^2)/(2 - x_1^2 - y_1^2)).\]

Again eliminate divisions
using \((X : Y : Z)\): only \(3\text{M} + 4\text{S}\).
Much faster than addition.

**More addition strategies**

**Dual addition formula:**

\[(x_1, y_1) + (x_2, y_2) =
\quad ((x_1 y_1 + x_2 y_2)/(x_1 x_2 + y_1 y_2),
\quad (x_1 y_1 - x_2 y_2)/(x_1 y_2 - x_2 y_1)).\]
Low degree, no need for \(d\).
Faster doubling

\[(x_1, y_1) + (x_1, y_1) =
((x_1 y_1 + y_1 x_1)/(1 + dx_1 x_1 y_1 y_1),
(y_1 y_1 - x_1 x_1)/(1 - dx_1 x_1 y_1 y_1)) =
((2x_1 y_1)/(1 + dx_1^2 y_1^2),
(y_1^2 - x_1^2)/(1 - dx_1^2 y_1^2)).\]

\[x_1^2 + y_1^2 = 1 + dx_1^2 y_1^2 \text{ so}
(x_1, y_1) + (x_1, y_1) =
((2x_1 y_1)/(x_1^2 + y_1^2),
(y_1^2 - x_1^2)/(2 - x_1^2 - y_1^2)).\]

Again eliminate divisions
using \((X : Y : Z)\): only \(3M + 4S\).
Much faster than addition.

More addition strategies

Dual addition formula:

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

Low degree, no need for \(d\).

Warning: fails for doubling!

Is this really “addition”?

Most EC formulas have failures.
Faster doubling

\[(x_1, y_1) + (x_1, y_1) =
((x_1 y_1 + y_1 x_1)/(1 + dx_1 x_1 y_1 y_1),
(y_1 y_1 - x_1 x_1)/(1 - dx_1 x_1 y_1 y_1)) =
((2x_1 y_1)/(1 + dx_1^2 y_1^2),
(y_1^2 - x_1^2)/(1 - dx_1^2 y_1^2))).
\]

\[x_1^2 + y_1^2 = 1 + dx_1^2 y_1^2 \text{ so}
(x_1, y_1) + (x_1, y_1) =
((2x_1 y_1)/(x_1^2 + y_1^2),
(y_1^2 - x_1^2)/(2 - x_1^2 - y_1^2))).
\]

Again eliminate divisions
using \((X : Y : Z)\): only \(3M + 4S\).
Much faster than addition.

More addition strategies

Dual addition formula:

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

Low degree, no need for \(d\).

Warning: fails for doubling!
Is this really “addition”?
Most EC formulas have failures.

Can test for failure cases.
Can produce constant-time code
by eliminating branches.
For some ECC ops, can prove
that failure cases never happen.
More addition strategies

Dual addition formula:
\[(x_1, y_1) + (x_2, y_2) =
\left((x_1 y_1 + x_2 y_2)/(x_1 x_2 + y_1 y_2),
(x_1 y_1 - x_2 y_2)/(x_1 y_2 - x_2 y_1)\right)\].
Low degree, no need for \(d\).

Warning: fails for doubling!

Is this really “addition”?

Most EC formulas have failures.

Can test for failure cases.

Can produce constant-time code by eliminating branches.

For some ECC ops, can prove that failure cases never happen.

More coordinate systems: e.g.,
- inverted: \(x = z/x, y = z/y\).
- extended: \(x = z/x, y = z/y\).
- completed: \(x = z/x, y = z/y, x y = t/z\).

“−1-twisted Edwards curves”− \(x^2 + y^2 = 1 + dx^2 y^2\):

further speedups related to

\[-x^2 + y^2 = (y - x)(y + x)\].

Inside modern ECC operations:

8\(M\) for addition,
3\(M\) + 4\(S\) for doubling.
More addition strategies

Dual addition formula:

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1 y_1 + x_2 y_2}{x_1 x_2 + y_1 y_2}, \frac{x_1 y_1 - x_2 y_2}{x_1 y_2 - x_2 y_1}\right).$$

Low degree, no need for $d$.

Warning: fails for doubling!

Is this really "addition"?

Most EC formulas have failures.

Can test for failure cases.

Can produce constant-time code by eliminating branches.

For some ECC ops, can prove that failure cases never happen.

More coordinate systems: e.g.,

- inverted: $x = Z = X$, $y = Z = Y$.
- extended: $x = X = Z$, $y = Y = T$.
- completed: $x = X = Z$, $y = Y = Z$, $x y = T = Z$.

"−1-twisted Edwards curves"

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

further speedups related to

$$-x^2 + y^2 = (y - x)(y + x).$$

Inside modern ECC operations:

$8M$ for addition,

$3M + 4S$ for doubling.
More addition strategies

Dual addition formula:
\((x_1, y_1) + (x_2, y_2) = ((x_1y_1 + x_2y_2)/(x_1x_2 + y_1y_2), (x_1y_1 - x_2y_2)/(x_1y_2 - x_2y_1))\).

Low degree, no need for \(d\).

Warning: fails for doubling!

Is this really “addition”?

Most EC formulas have failures.

Can test for failure cases.
Can produce constant-time code by eliminating branches.

For some ECC ops, can prove that failure cases never happen.

More coordinate systems: e.g.,
• inverted: \(x = Z/X, y = Z/Y\).
• extended: \(x = X/Z, y = Y/T\).
• completed: \(x = X/Z, y = Y/Z, xy = T/Z\).

“−1-twisted Edwards curves”

\(-x^2 + y^2 = 1 + dx^2y^2\):
further speedups related to

\(-x^2 + y^2 = (y - x)(y + x)\).

Inside modern ECC operations:

8M for addition,
3M + 4S for doubling.
More addition strategies

Dual addition formula:

\[(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1 y_1 + x_2 y_2}{x_1 x_2 + y_1 y_2}, \frac{x_1 y_1 - x_2 y_2}{x_1 y_2 - x_2 y_1}\right)\].

Low degree, no need for \(d\).

Warning: fails for doubling!

Is this really “addition”?

Most EC formulas have failures.

Can test for failure cases.

Can produce constant-time code by eliminating branches.

For some ECC ops, can prove that failure cases never happen.

More coordinate systems: e.g.,

- inverted: \(x = Z/X, y = Z/Y\).
- extended: \(x = X/Z, y = Y/T\).
- completed: \(x = X/Z, y = Y/Z, xy = T/Z\).

“−1-twisted Edwards curves”

\[-x^2 + y^2 = 1 + dx^2 y^2:\]

further speedups related to

\[-x^2 + y^2 = (y - x)(y + x)\].

Inside modern ECC operations:

8M for addition,

3M + 4S for doubling.
More addition strategies

Addition formula:
\[(x_1; y_1) + (x_2, y_2) = ((x_1 y_1 + x_2 y_2)/(x_1 x_2 + y_1 y_2), \]
\[+ (x_2 y_2)/(x_1 y_2 - x_2 y_1)).\]

Low degree, no need for \(d\).

Warning: fails for doubling!

Is this really “addition”?

Most EC formulas have failures.

Can test for failure cases.

Can produce constant-time code by eliminating branches.

For some ECC ops, can prove that failure cases never happen.

More coordinate systems: e.g.,

• inverted: \(x = Z/X, y = Z/Y\).
• extended: \(x = X/Z, y = Y/T\).
• completed: \(x = X/Z, y = Y/Z, xy = T/Z\).

“−1-twisted Edwards curves”

\(-x^2 + y^2 = 1 + dx^2 y^2:\)

further speedups related to

\(-x^2 + y^2 = (y - x)(y + x)\).

Inside modern ECC operations:

8M for addition,

3M + 4S for doubling.

NIST curves (e.g., P-256) were standardized before Edwards curves were published.

Much slower additions.
More addition strategies

Dual addition formula:

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

No need for \(d\).

Warning: fails for doubling!

Is this really "addition"?

Most EC formulas have failures.

Can test for failure cases.

Can produce constant-time code by eliminating branches.

For some ECC ops, can prove that failure cases never happen.

More coordinate systems: e.g.,

- inverted: \(x = Z/X, y = Z/Y\).
- extended: \(x = X/Z, y = Y/T\).
- completed: \(x = X/Z, y = Y/Z,\)
  \(xy = T/Z\).

"−1-twisted Edwards curves"

\[-x^2 + y^2 = 1 + dx^2 y^2:\]

further speedups related to

\[-x^2 + y^2 = (y - x)(y + x)\].

Inside modern ECC operations:

8M for addition,

3M + 4S for doubling.

NIST curves (e.g., P-256) were standardized before Edwards curves were.

Much slower additions.
More coordinate systems: e.g.,
- inverted: \( x = Z/X, \ y = Z/Y \).
- extended: \( x = X/Z, \ y = Y/T \).
- completed: \( x = X/Z, \ y = Y/Z, \ xy = T/Z \).

"−1-twisted Edwards curves"
\[-x^2 + y^2 = 1 + dx^2y^2:\]
further speedups related to
\[-x^2 + y^2 = (y - x)(y + x).\]

Inside modern ECC operations:
8M for addition,
3M + 4S for doubling.

NIST curves (e.g., P-256)
were standardized before
Edwards curves were published.
Much slower additions.
More coordinate systems: e.g.,
• inverted: \( x = Z/X, y = Z/Y \).
• extended: \( x = X/Z, y = Y/T \).
• completed: \( x = X/Z, y = Y/Z, \ xy = T/Z \).

“−1-twisted Edwards curves”
\(-x^2 + y^2 = 1 + dx^2y^2:\)
further speedups related to
\(-x^2 + y^2 = (y - x)(y + x).\)

Inside modern ECC operations:
\(8M\) for addition,
\(3M + 4S\) for doubling.

NIST curves (e.g., P-256) were standardized before Edwards curves were published.
Much slower additions.
More coordinate systems: e.g.,
• inverted: $x = \frac{Z}{X}$, $y = \frac{Z}{Y}$.
• extended: $x = \frac{X}{Z}$, $y = \frac{Y}{T}$.
• completed: $x = \frac{X}{Z}$, $y = \frac{Y}{Z}$, $xy = \frac{T}{Z}$.

“−1-twisted Edwards curves”
$-x^2 + y^2 = 1 + dx^2y^2$:
further speedups related to
$-x^2 + y^2 = (y - x)(y + x)$.

Inside modern ECC operations:
8M for addition,
3M + 4S for doubling.

NIST curves (e.g., P-256) were standardized before Edwards curves were published.
Much slower additions.
Express as Edwards curves using a field extension: slow.
More coordinate systems: e.g.,
- inverted: $x = Z/X$, $y = Z/Y$.
- extended: $x = X/Z$, $y = Y/T$.
- completed: $x = X/Z$, $y = Y/Z$,
  $xy = T/Z$.

“−1-twisted Edwards curves”
$-x^2 + y^2 = 1 + dx^2y^2$: further speedups related to
$-x^2 + y^2 = (y - x)(y + x)$.

Inside modern ECC operations:
$8M$ for addition,
$3M + 4S$ for doubling.

NIST curves (e.g., P-256) were standardized before
Edwards curves were published.

Much slower additions.

Express as Edwards curves using a field extension: slow.

How did Curve25519 obtain good speeds for ECDH?

“Montgomery curve with the Montgomery ladder.”
More coordinate systems: e.g.,
- inverted: \(x = Z/X, y = Z/Y\).
- extended: \(x = X/Z, y = Y/T\).
- completed: \(x = X/Z, y = Y/Z, xy = T/Z\).

“−1-twisted Edwards curves”
\(-x^2 + y^2 = 1 + dx^2y^2:\)
further speedups related to
\(-x^2 + y^2 = (y - x)(y + x)\).

Inside modern ECC operations:
8M for addition,
3M + 4S for doubling.

NIST curves (e.g., P-256) were standardized before Edwards curves were published.

Much slower additions.

Express as Edwards curves using a field extension: slow.

How did Curve25519 obtain good speeds for ECDH?

“Montgomery curve with the Montgomery ladder.”

Why did NIST not choose Montgomery curves? Unclear.