Engineering Cryptographic Software

Scalar Multiplication

Peter Schwabe

January 2026

New directions in cryptography

644 IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. IT-22, NO. 6, NOVEMBER 1976

New Directions in Cryptography

Invited Paper

WHITFIELD DIFFIE AND MARTIN E. HELLMAN, MEMBER, IEEE

***Abstract*—Two kinds of contemporary developments in cryptography are examined. Widening applications of teleprocessing have given rise to a need for new types of cryptographic systems, which minimize the need for secure key distribution channels and supply the equivalent of a written signature. This paper suggests ways to solve these currently open problems. It also discusses how the theories of communication and computation are beginning to provide the tools to solve cryptographic problems of long standing.**

I. Introduction

WE STAND TODAY on the brink of a revolution in cryptography. The development of cheap digital

The best known cryptographic problem is that of privacy: preventing the unauthorized extraction of information from communications over an insecure channel. In order to use cryptography to insure privacy, however, it is currently necessary for the communicating parties to share a key which is known to no one else. This is done by sending the key in advance over some secure channel such as private courier or registered mail. A private conversation between two people with no prior acquaintance is a common occurrence in business, however, and it is unrealistic to expect initial business contacts to be postponed long enough for keys to be transmitted by some physical means. The cost and delay imposed by this key distribution problem is a major barrier to the transfer of business

Groups

Definition

A set S together with an operation $\circ$ is called a *group* $G = (S, \circ)$ if

- For all $a, b \in S$: $a \circ b \in S$ (closed under $\circ$)
- For all $a, b, c \in S$: $(a \circ b) \circ c = a \circ (b \circ c)$ (associativity)
- There exists $u \in S$ such that for any $a \in S$: $a \circ u = u \circ a = a$ (identity element)
- For each $a \in S$ there exists $b \in S$ such that $a \circ b = b \circ a = u$ (inverse)

Groups

Definition

A set S together with an operation $\circ$ is called a *group* $G = (S, \circ)$ if

- For all $a, b \in S$: $a \circ b \in S$ (closed under $\circ$)
- For all $a, b, c \in S$: $(a \circ b) \circ c = a \circ (b \circ c)$ (associativity)
- There exists $u \in S$ such that for any $a \in S$: $a \circ u = u \circ a = a$ (identity element)
- For each $a \in S$ there exists $b \in S$ such that $a \circ b = b \circ a = u$ (inverse)

- The operation $\circ$ is typically written as $+$ (additive group) or as $\cdot$ (multiplicative group)

Groups

Definition

A set S together with an operation $\circ$ is called a *group* $G = (S, \circ)$ if

- For all $a, b \in S$: $a \circ b \in S$ (closed under $\circ$)
- For all $a, b, c \in S$: $(a \circ b) \circ c = a \circ (b \circ c)$ (associativity)
- There exists $u \in S$ such that for any $a \in S$: $a \circ u = u \circ a = a$ (identity element)
- For each $a \in S$ there exists $b \in S$ such that $a \circ b = b \circ a = u$ (inverse)

- The operation $\circ$ is typically written as $+$ (additive group) or as $\cdot$ (multiplicative group)
- Groups where $a \circ b = b \circ a$ for all $a, b \in S$ are called *commutative* or *Abelian*

Groups

Definition

A set S together with an operation $\circ$ is called a *group* $G = (S, \circ)$ if

- For all $a, b \in S$: $a \circ b \in S$ (closed under $\circ$)
- For all $a, b, c \in S$: $(a \circ b) \circ c = a \circ (b \circ c)$ (associativity)
- There exists $u \in S$ such that for any $a \in S$: $a \circ u = u \circ a = a$ (identity element)
- For each $a \in S$ there exists $b \in S$ such that $a \circ b = b \circ a = u$ (inverse)

- The operation $\circ$ is typically written as $+$ (additive group) or as $\cdot$ (multiplicative group)
- Groups where $a \circ b = b \circ a$ for all $a, b \in S$ are called *commutative* or *Abelian*
- The *group order* is $|S|$, the number of elements in S

Groups

Definition

A set S together with an operation $\circ$ is called a *group* $G = (S, \circ)$ if

- For all $a, b \in S$: $a \circ b \in S$ (closed under $\circ$)
- For all $a, b, c \in S$: $(a \circ b) \circ c = a \circ (b \circ c)$ (associativity)
- There exists $u \in S$ such that for any $a \in S$: $a \circ u = u \circ a = a$ (identity element)
- For each $a \in S$ there exists $b \in S$ such that $a \circ b = b \circ a = u$ (inverse)

- The operation $\circ$ is typically written as $+$ (additive group) or as $\cdot$ (multiplicative group)
- Groups where $a \circ b = b \circ a$ for all $a, b \in S$ are called *commutative* or *Abelian*
- The *group order* is $|S|$, the number of elements in S
- We call $g \in S$ a *generator* of the group if all elements of S can be written as multiples (additive group) or powers (multiplicative group) of g.

Groups

Definition

A set S together with an operation $\circ$ is called a *group* $G = (S, \circ)$ if

- For all $a, b \in S$: $a \circ b \in S$ (closed under $\circ$)
- For all $a, b, c \in S$: $(a \circ b) \circ c = a \circ (b \circ c)$ (associativity)
- There exists $u \in S$ such that for any $a \in S$: $a \circ u = u \circ a = a$ (identity element)
- For each $a \in S$ there exists $b \in S$ such that $a \circ b = b \circ a = u$ (inverse)

- The operation $\circ$ is typically written as $+$ (additive group) or as $\cdot$ (multiplicative group)
- Groups where $a \circ b = b \circ a$ for all $a, b \in S$ are called *commutative* or *Abelian*
- The *group order* is $|S|$, the number of elements in S
- We call $g \in S$ a *generator* of the group if all elements of S can be written as multiples (additive group) or powers (multiplicative group) of g.
- A group generated by a single element is called *cyclic*

Examples

- The integers with addition $(\mathbb{Z}, +)$ are a (commutative) group
  - Closed, associative ✓
  - Identity element 0
  - Inverse of a is $-a$
  - The group is cyclic with generator 1

Examples

- The integers with addition $(\mathbb{Z}, +)$ are a (commutative) group
  - Closed, associative ✓
  - Identity element 0
  - Inverse of a is $-a$
  - The group is cyclic with generator 1
- The integers without zero with multiplication $(\mathbb{Z}, \cdot)$ are *not* a group
  - Closed, associative ✓
  - Identity element is 1
  - We cannot invert 0
  - More generally we lack inverses, e.g., $\frac{1}{2} \notin \mathbb{Z}$

Examples

- The integers with addition $(\mathbb{Z}, +)$ are a (commutative) group
  - Closed, associative $\checkmark$
  - Identity element 0
  - Inverse of a is $-a$
  - The group is cyclic with generator 1
- The integers without zero with multiplication $(\mathbb{Z}, \cdot)$ are *not* a group
  - Closed, associative $\checkmark$
  - Identity element is 1
  - We cannot invert 0
  - More generally we lack inverses, e.g., $\frac{1}{2} \notin \mathbb{Z}$
- The rationals without zero with multiplication $(\mathbb{Q} \setminus \{0\}, \cdot)$ are a (commutative) group
  - Closed, associative $\checkmark$
  - Identity element is 1
  - Inverse of a is $\frac{1}{a}$

Examples

- For an integer $q > 1$, the set $\{0, \ldots, q-1\}$ together with addition modulo q is a (commutative) group
  - Closed, associative ✓
  - Identity element 0
  - Inverse of a is $q - a$

Examples

- For an integer $q > 1$, the set $\{0, \ldots, q-1\}$ together with addition modulo q is a (commutative) group
  - Closed, associative ✓
  - Identity element 0
  - Inverse of a is $q - a$
- For a prime q, the set $\{1, \ldots, q-1\}$ together with multiplication modulo q is a (commutative) group
  - Closed, associative ✓
  - Identity element is 1
  - More about inverses later

The discrete logarithm problem

Definition

Let G be a finite, Abelian, cyclic group of order ℓ with generator g. Let a be an element of G. The *(computational) discrete-logarithm problem (DLP)* is

- to find an integer k such that $g^k = a$ (for a multiplicatively written group)
- to find an integer k such that $kg = a$ (for an additively written group)

- g^k means $\underbrace{g \cdot g \cdot g \cdot \cdots \cdot g}_{k \text{ times}}$
- kg means $\underbrace{g + g + g + \cdots + g}_{k \text{ times}}$

The discrete logarithm problem

Definition

Let G be a finite, Abelian, cyclic group of order ℓ with generator g. Let a be an element of G. The *(computational) discrete-logarithm problem (DLP)* is

- to find an integer k such that $g^k = a$ (for a multiplicatively written group)
- to find an integer k such that $kg = a$ (for an additively written group)

- g^k means $\underbrace{g \cdot g \cdot g \cdot \cdots \cdot g}_{k \text{ times}}$
- kg means $\underbrace{g + g + g + \cdots + g}_{k \text{ times}}$
- In many groups the DLP is easy to solve (e.g., $\{0, \ldots, q-1\}$ with addition modulo q)
- In some groups the DLP is believed to be hard (e.g., $\{1, \ldots, q-1\}$ with multiplication modulo q), for certain primes q

Some notation

For the remainder of today's lecture

- consider an finite, cyclic group G, written additively,
- the generator of G is called P,
- the group order of G is ℓ,
- other elements are denoted by capital letters (e.g., P, R), and
- we assume that the DLP is hard in G.

Diffie-Hellman (DH) key exchange

Alice		Bob
Choose $a \leftarrow \{0, \ldots, \ell - 1\}$		Choose $b \leftarrow \{0, \ldots, \ell - 1\}$
$A \leftarrow aP$		$B \leftarrow bP$
	A →	
	← B	
$K \leftarrow aB = a(bP) = (ab)P$		$K \leftarrow bA = b(aP) = (ba)P$

Somes notes about DH

- Clearly insecure of DLP is not hard

Somes notes about DH

- Clearly insecure of DLP is not hard
- Also secure only against passive adversaries
- Active “man-in-the-middle” (MitM) attack:
  - Eve chooses her own keypair (e, E)
  - replaces A on the channel by E
  - replaces B on the channel by E

Somes notes about DH

- Clearly insecure of DLP is not hard
- Also secure only against passive adversaries
- Active "man-in-the-middle" (MitM) attack:
  - Eve chooses her own keypair (e, E)
  - replaces A on the channel by E
  - replaces B on the channel by E
  - Gets shared secret $(ae)P$ with Alice
  - Gets shared secret $(be)P$ with Bob

Somes notes about DH

- Clearly insecure of DLP is not hard
- Also secure only against passive adversaries
- Active “man-in-the-middle” (MitM) attack:
  - Eve chooses her own keypair (e, E)
  - replaces A on the channel by E
  - replaces B on the channel by E
  - Gets shared secret $(ae)P$ with Alice
  - Gets shared secret $(be)P$ with Bob
  - Afterwards decrypts and re-encrypts communication

Somes notes about DH

- Clearly insecure of DLP is not hard
- Also secure only against passive adversaries
- Active "man-in-the-middle" (MitM) attack:
  - Eve chooses her own keypair (e, E)
  - replaces A on the channel by E
  - replaces B on the channel by E
  - Gets shared secret $(ae)P$ with Alice
  - Gets shared secret $(be)P$ with Bob
  - Afterwards decrypts and re-encrypts communication
- DH is an **unauthenticated** key exchange

Somes notes about DH

- Clearly insecure of DLP is not hard
- Also secure only against passive adversaries
- Active “man-in-the-middle” (MitM) attack:
  - Eve chooses her own keypair (e, E)
  - replaces A on the channel by E
  - replaces B on the channel by E
  - Gets shared secret $(ae)P$ with Alice
  - Gets shared secret $(be)P$ with Bob
  - Afterwards decrypts and re-encrypts communication
- DH is an **unauthenticated** key exchange
- Consider DH a **building block** for protocols

How about authentication?

- Can build **authenticated** key exchange just from DH (plus symmetric primitives)
- Examples:
  - X3DH used by Signal (https://signal.org/docs/specifications/x3dh/)
  - Noise protocol framework (https://noiseprotocol.org/)

How about authentication?

- Can build **authenticated** key exchange just from DH (plus symmetric primitives)
- Examples:
  - X3DH used by Signal (`https://signal.org/docs/specifications/x3dh/`)
  - Noise protocol framework (`https://noiseprotocol.org/`)
- Typical alternative: **cryptographic signatures**:

How about authentication?

- Can build **authenticated** key exchange just from DH (plus symmetric primitives)
- Examples:
  - X3DH used by Signal (`https://signal.org/docs/specifications/x3dh/`)
  - Noise protocol framework (`https://noiseprotocol.org/`)
- Typical alternative: **cryptographic signatures**:
  - $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}()$ (probabilistic)

How about authentication?

- Can build **authenticated** key exchange just from DH (plus symmetric primitives)
- Examples:
  - X3DH used by Signal (https://signal.org/docs/specifications/x3dh/)
  - Noise protocol framework (https://noiseprotocol.org/)
- Typical alternative: **cryptographic signatures**:
  - $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}()$ (probabilistic)
  - $\mathsf{sig} \leftarrow \mathsf{Sign}(\mathsf{msg}, \mathsf{sk})$ (probabilistic)

How about authentication?

- Can build **authenticated** key exchange just from DH (plus symmetric primitives)
- Examples:
  - X3DH used by Signal (https://signal.org/docs/specifications/x3dh/)
  - Noise protocol framework (https://noiseprotocol.org/)
- Typical alternative: **cryptographic signatures**:
  - (sk, vk) ← KeyGen() (probabilistic)
  - sig ← Sign(msg, sk) (probabilistic)
  - accept/reject ← Verify(msg, sig, vk) (deterministic)

Properties of a signature scheme

Correctness

For (sk, vk) generated by KeyGen and sig generated by Sign(msg, sk), we want Verify(msg, sig, vk) to return accept (with overwhelming probability).

Properties of a signature scheme

Correctness

For (sk, vk) generated by KeyGen and sig generated by Sign(msg, sk), we want Verify(msg, sig, vk) to return accept (with overwhelming probability).

Security (intuition)

- Challenger generates (sk, vk) $\leftarrow$ KeyGen()

Properties of a signature scheme

Correctness

For (sk, vk) generated by KeyGen and sig generated by Sign(msg, sk), we want Verify(msg, sig, vk) to return accept (with overwhelming probability).

Security (intuition)

- Challenger generates (sk, vk) $\leftarrow$ KeyGen()
- Attacker gets vk and access to *signing oracle* for sk
- Can adaptively query signatures on arbitrary messages $\mathsf{msg}_1, \ldots, \mathsf{msg}_n$

Properties of a signature scheme

Correctness

For (sk, vk) generated by KeyGen and sig generated by Sign(msg, sk), we want Verify(msg, sig, vk) to return accept (with overwhelming probability).

Security (intuition)

- Challenger generates (sk, vk) $\leftarrow$ KeyGen()
- Attacker gets vk and access to *signing oracle* for sk
- Can adaptively query signatures on arbitrary messages $\text{msg}_1, \ldots, \text{msg}_n$
- Now oracle access is taken away

Properties of a signature scheme

Correctness

For (sk, vk) generated by KeyGen and sig generated by Sign(msg, sk), we want Verify(msg, sig, vk) to return accept (with overwhelming probability).

Security (intuition)

- Challenger generates (sk, vk) $\leftarrow$ KeyGen()
- Attacker gets vk and access to *signing oracle* for sk
- Can adaptively query signatures on arbitrary messages $\mathsf{msg}_1, \ldots, \mathsf{msg}_n$
- Now oracle access is taken away
- Attacker needs to produce sig_a on arbitrary msg_a, such that
  - Verify(msg_a, sig_a, vk) returns accept
  - $\mathsf{msg}_a \neq \mathsf{msg}_i$ for all $i \in \{1, \ldots, n\}$

Properties of a signature scheme

Correctness

For (sk, vk) generated by KeyGen and sig generated by Sign(msg, sk), we want Verify(msg, sig, vk) to return accept (with overwhelming probability).

Security (intuition)

- Challenger generates (sk, vk) $\leftarrow$ KeyGen()
- Attacker gets vk and access to *signing oracle* for sk
- Can adaptively query signatures on arbitrary messages $\mathsf{msg}_1, \ldots, \mathsf{msg}_n$
- Now oracle access is taken away
- Attacker needs to produce sig_a on arbitrary msg_a, such that
  - Verify(msg_a, sig_a, vk) returns accept
  - $\mathsf{msg}_a \neq \mathsf{msg}_i$ for all $i \in \{1, \ldots, n\}$
- Scheme is secure if the attacker succeeds only with negligible probability

Properties of a signature scheme

Correctness

For (sk, vk) generated by KeyGen and sig generated by Sign(msg, sk), we want Verify(msg, sig, vk) to return accept (with overwhelming probability).

Security (intuition)

- Challenger generates (sk, vk) $\leftarrow$ KeyGen()
- Attacker gets vk and access to *signing oracle* for sk
- Can adaptively query signatures on arbitrary messages $\text{msg}_1, \ldots, \text{msg}_n$
- Now oracle access is taken away
- Attacker needs to produce sig_a on arbitrary msg_a, such that
  - Verify(msg_a, sig_a, vk) returns accept
  - $\text{msg}_a \neq \text{msg}_i$ for all $i \in \{1, \ldots, n\}$
- Scheme is secure if the attacker succeeds only with negligible probability
- This is called **existential unforgeability under chosen-message attacks (EUF-CMA)**

Schnorr signatures

$(a, A) \leftarrow$ KeyGen()

- Choose a uniformly random in $\{0, \ldots, \ell\}$
- Compute $A \leftarrow aP$

Schnorr signatures

$(a, A) \leftarrow \text{KeyGen}()$

- Choose a uniformly random in $\{0, \ldots, \ell\}$
- Compute $A \leftarrow aP$

$(s, e) \leftarrow \text{Sign}(\text{msg}, \text{sk} = a)$

- Choose r uniformly random in $\{0, \ldots, \ell\}$
- Compute $R \leftarrow rP$
- Compute $e = H(R, \text{msg})$
- Compute $s = (r - a \cdot e) \mod \ell$

Schnorr signatures

$(a, A) \leftarrow \text{KeyGen}()$

- Choose a uniformly random in $\{0, \ldots, \ell\}$
- Compute $A \leftarrow aP$

$(s, e) \leftarrow \text{Sign}(\text{msg}, \text{sk} = a)$

- Choose r uniformly random in $\{0, \ldots, \ell\}$
- Compute $R \leftarrow rP$
- Compute $e = H(R, \text{msg})$
- Compute $s = (r - a \cdot e) \mod \ell$

$\text{Verify}(\text{msg}, \text{sig} = (R, S), \text{vk} = A)$

- Compute $\overline{R} \leftarrow sP + eA$
- Return accept if and only if $H(\overline{R}, \text{msg}) = e$

Scalar multiplication

- Looks like all these schemes need computation of kP.

Scalar multiplication

- Looks like all these schemes need computation of kP.
- Let's take a closer look:
  - For key generation, the point P is *fixed* at compile time
  - For Diffie-Hellman joint-key computation the point is received at runtime

Scalar multiplication

- Looks like all these schemes need computation of kP.
- Let's take a closer look:
  - For key generation, the point P is *fixed* at compile time
  - For Diffie-Hellman joint-key computation the point is received at runtime
  - Key generation and Diffie-Hellman need *one* scalar multiplication kP
  - Schnorr signature verification needs double-scalar multiplication $k_1P_1 + k_2P_2$

Scalar multiplication

- Looks like all these schemes need computation of kP.
- Let's take a closer look:
  - For key generation, the point P is *fixed* at compile time
  - For Diffie-Hellman joint-key computation the point is received at runtime
  - Key generation and Diffie-Hellman need *one* scalar multiplication kP
  - Schnorr signature verification needs double-scalar multiplication $k_1P_1 + k_2P_2$
  - In key generation and Diffie-Hellman joint-key computation, k is secret
  - The scalars in Schnorr signature verification are public

Scalar multiplication

- Looks like all these schemes need computation of kP.
- Let's take a closer look:
  - For key generation, the point P is *fixed* at compile time
  - For Diffie-Hellman joint-key computation the point is received at runtime
  - Key generation and Diffie-Hellman need *one* scalar multiplication kP
  - Schnorr signature verification needs double-scalar multiplication $k_1P_1 + k_2P_2$
  - In key generation and Diffie-Hellman joint-key computation, k is secret
  - The scalars in Schnorr signature verification are public
- In the following: Distinguish these cases

A first approach

- Let's compute $105 \cdot P$.

A first approach

- Let's compute $105 \cdot P$.
- Obvious: Can do that with 104 additions $P + P + P + \cdots + P$

A first approach

- Let's compute $105 \cdot P$.
- Obvious: Can do that with 104 additions $P + P + P + \cdots + P$
- Problem: 105 has 7 bits, we need roughly 2^7 additions, *cryptographic* scalars have ≈ 256 bits, we would need roughly 2^{256} additions (more expensive than solving the ECDLP!)

A first approach

- Let's compute $105 \cdot P$.
- Obvious: Can do that with 104 additions $P + P + P + \cdots + P$
- Problem: 105 has 7 bits, we need roughly 2^7 additions, *cryptographic* scalars have ≈ 256 bits, we would need roughly 2^{256} additions (more expensive than solving the ECDLP!)
- Conclusion: we need algorithms that run in polynomial time (in the size of the scalar)

Rewriting the scalar

- $105 = 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0$

Rewriting the scalar

- $105 = 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0$
- $105 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

Rewriting the scalar

- $105 = 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0$
- $105 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
- $105 = ((((((((((1 \cdot 2 + 1) \cdot 2) + 0) \cdot 2) + 1) \cdot 2) + 0) \cdot 2) + 0) \cdot 2) + 1$ (Horner's rule)

Rewriting the scalar

- $105 = 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0$
- $105 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
- $105 = ((((((((((1 \cdot 2 + 1) \cdot 2) + 0) \cdot 2) + 1) \cdot 2) + 0) \cdot 2) + 0) \cdot 2) + 1$ (Horner's rule)
- $105 \cdot P = ((((((((((P \cdot 2 + P) \cdot 2) + 0) \cdot 2) + P) \cdot 2) + 0) \cdot 2) + 0) \cdot 2) + P$

Rewriting the scalar

- $105 = 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0$
- $105 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
- $105 = ((((((((((1 \cdot 2 + 1) \cdot 2) + 0) \cdot 2) + 1) \cdot 2) + 0) \cdot 2) + 0) \cdot 2) + 1$ (Horner's rule)
- $105 \cdot P = ((((((((((P \cdot 2 + P) \cdot 2) + 0) \cdot 2) + P) \cdot 2) + 0) \cdot 2) + 0) \cdot 2) + P$
- Cost: 6 doublings, 3 additions

Rewriting the scalar

- $105 = 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0$
- $105 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
- $105 = ((((((((((1 \cdot 2 + 1) \cdot 2) + 0) \cdot 2) + 1) \cdot 2) + 0) \cdot 2) + 0) \cdot 2) + 1$ (Horner's rule)
- $105 \cdot P = ((((((((((P \cdot 2 + P) \cdot 2) + 0) \cdot 2) + P) \cdot 2) + 0) \cdot 2) + 0) \cdot 2) + P$
- Cost: 6 doublings, 3 additions
- General algorithm: "Double and add"

  $R \leftarrow P$
  for $i \leftarrow n - 2$ downto 0 **do**
      $R \leftarrow 2R$
      if $(k)_2[i] = 1$ **then**
          $R \leftarrow R + P$
      end if
  end for
  return R

Analysis of double-and-add

- Let n be the number of bits in the exponent
- Double-and-add takes $n - 1$ doublings

Analysis of double-and-add

- Let n be the number of bits in the exponent
- Double-and-add takes $n - 1$ doublings
- Let m be the number of 1 bits in the exponent
- Double-and-add takes $m - 1$ additions
- On average: $\approx n/2$ additions

Analysis of double-and-add

- Let n be the number of bits in the exponent
- Double-and-add takes $n - 1$ doublings
- Let m be the number of 1 bits in the exponent
- Double-and-add takes $m - 1$ additions
- On average: $\approx n/2$ additions
- P does not need to be known in advance, no precomputation depending on P

Analysis of double-and-add

- Let n be the number of bits in the exponent
- Double-and-add takes $n-1$ doublings
- Let m be the number of 1 bits in the exponent
- Double-and-add takes $m-1$ additions
- On average: $\approx n/2$ additions
- P does not need to be known in advance, no precomputation depending on P
- Handles single-scalar multiplication

Analysis of double-and-add

- Let n be the number of bits in the exponent
- Double-and-add takes $n-1$ doublings
- Let m be the number of 1 bits in the exponent
- Double-and-add takes $m-1$ additions
- On average: $\approx n/2$ additions
- P does not need to be known in advance, no precomputation depending on P
- Handles single-scalar multiplication
- Running time clearly depends on the scalar: insecure for secret scalars!

Double-scalar double-and-add

- Let's modify the algorithm to compute $k_1P_1 + k_2P_2$

Double-scalar double-and-add

- Let's modify the algorithm to compute $k_1 P_1 + k_2 P_2$
- Obvious solution:
  - Compute $k_1 P_1$ ($n_1 - 1$ doublings, $m_1 - 1$ additions)
  - Compute $k_2 P_2$ ($n_2 - 1$ doublings, $m_2 - 1$ additions)
  - Add the results (1 addition)

Double-scalar double-and-add

- Let's modify the algorithm to compute $k_1 P_1 + k_2 P_2$
- Obvious solution:
  - Compute $k_1 P_1$ ($n_1 - 1$ doublings, $m_1 - 1$ additions)
  - Compute $k_2 P_2$ ($n_2 - 1$ doublings, $m_2 - 1$ additions)
  - Add the results (1 addition)
- We can do better ($\mathcal{O}$ denotes the neutral element):

```
R ← O
for i ← max(n_1, n_2) − 1 downto 0 do
    R ← 2R
    if (k_1)_2[i] = 1 then
        R ← R + P_1
    end if
    if (k_2)_2[i] = 1 then
        R ← R + P_2
    end if
end for
return R
```

Double-scalar double-and-add

- Let's modify the algorithm to compute $k_1 P_1 + k_2 P_2$
- Obvious solution:
  - Compute $k_1 P_1$ ($n_1 - 1$ doublings, $m_1 - 1$ additions)
  - Compute $k_2 P_2$ ($n_2 - 1$ doublings, $m_2 - 1$ additions)
  - Add the results (1 addition)
- We can do better ($\mathcal{O}$ denotes the neutral element):

  $R \leftarrow \mathcal{O}$
  for $i \leftarrow \max(n_1, n_2) - 1$ downto 0 **do**
      $R \leftarrow 2R$
      if $(k_1)_2[i] = 1$ **then**
          $R \leftarrow R + P_1$
      end if
      if $(k_2)_2[i] = 1$ **then**
          $R \leftarrow R + P_2$
      end if
  end for
  return R
- $\max(n_1, n_2)$ doublings, $m_1 + m_2$ additions

Some precomputation helps

- Whenever k_1 and k_2 have a 1 bit at the same position, we first add P_1 and then P_2 (on average for $1/4$ of the bits)

Some precomputation helps

- Whenever k_1 and k_2 have a 1 bit at the same position, we first add P_1 and then P_2 (on average for $1/4$ of the bits)
- Let's just precompute $T = P_1 + P_2$

Some precomputation helps

- Whenever k_1 and k_2 have a 1 bit at the same position, we first add P_1 and then P_2 (on average for $1/4$ of the bits)
- Let's just precompute $T = P_1 + P_2$
- Modified algorithm (Shamir's trick, special case of Strauss' algorithm):

```
R ← O
for i ← max(n_1, n_2) − 1 downto 0 do
    R ← 2R
    if (k_1)_2[i] = 1 AND (k_2)_2[i] = 1 then
        R ← R + T
    else if (k_1)_2[i] = 1 then
        R ← R + P_1
    else if (k_2)_2[i] = 1 then
        R ← R + P_2
    end if
end for
return R
```

Even more (offline) precomputation

- What if precomputation is free (fixed basepoint, offline precomputation)?

Even more (offline) precomputation

- What if precomputation is free (fixed basepoint, offline precomputation)?
- First idea: Let's precompute a table containing $0P, P, 2P, 3P, \ldots$, when we receive k, simply look up kP.

Even more (offline) precomputation

- What if precomputation is free (fixed basepoint, offline precomputation)?
- First idea: Let's precompute a table containing $0P, P, 2P, 3P, \ldots$, when we receive k, simply look up kP.
- Problem: k is large. For a 256-bit k we would need a table of size

$$3369993333393829974333376885877453834204643052817571560137951281152\text{TB}$$

Even more (offline) precomputation

- What if precomputation is free (fixed basepoint, offline precomputation)?
- First idea: Let's precompute a table containing $0P, P, 2P, 3P, \dots,$ when we receive k, simply look up kP.
- Problem: k is large. For a 256-bit k we would need a table of size

$$3369993333393829974333376885877453834204643052817571560137951281152\text{TB}$$

- How about, for example, precompute $P, 2P, 4P, 8P, \dots, 2^{n-1}P$
- This needs only about 16KB of storage for $n = 256$ and 64-byte group elements

Even more (offline) precomputation

- What if precomputation is free (fixed basepoint, offline precomputation)?
- First idea: Let's precompute a table containing $0P, P, 2P, 3P, \ldots,$ when we receive k, simply look up kP.
- Problem: k is large. For a 256-bit k we would need a table of size

$$3369993333393829974333376885877453834204643052817571560137951281152\text{TB}$$

- How about, for example, precompute $P, 2P, 4P, 8P, \ldots, 2^{n-1}P$
- This needs only about 16KB of storage for $n = 256$ and 64-byte group elements
- Modified scalar-multiplication algorithm:

$R \leftarrow \mathcal{O}$
for $i \leftarrow 0$ to $n-1$ **do**
  if $(k)_2[i] = 1$ **then**
    $R \leftarrow R + 2^iP$
  end if
end for
return R

Even more (offline) precomputation

- What if precomputation is free (fixed basepoint, offline precomputation)?
- First idea: Let's precompute a table containing $0P, P, 2P, 3P, \dots,$ when we receive k, simply look up kP.
- Problem: k is large. For a 256-bit k we would need a table of size

$$3369993333393829974333376885877453834204643052817571560137951281152\text{TB}$$

- How about, for example, precompute $P, 2P, 4P, 8P, \dots, 2^{n-1}P$
- This needs only about 16KB of storage for $n = 256$ and 64-byte group elements
- Modified scalar-multiplication algorithm:

  $R \leftarrow \mathcal{O}$
  for $i \leftarrow 0$ to $n-1$ **do**
      if $(k)_2[i] = 1$ **then**
          $R \leftarrow R + 2^i P$
      end if
  end for
  return R

- Eliminated all doublings in fixed-basepoint scalar multiplication!

Double-and-add always

- All algorithms so far perform *conditional addition* where the condition is secret
- For secret scalars (most common case!) we need something else

Double-and-add always

- All algorithms so far perform *conditional addition* where the condition is secret
- For secret scalars (most common case!) we need something else
- Idea: Always perform addition, discard result:

$R \leftarrow P$
for $i \leftarrow n-2$ downto 0 **do**
  $R \leftarrow 2R$
  $R_t \leftarrow R + P$
  if $(k)_2[i] = 1$ **then**
    $R \leftarrow R_t$
  end if
end for

Double-and-add always

- All algorithms so far perform *conditional addition* where the condition is secret
- For secret scalars (most common case!) we need something else
- Idea: Always perform addition, discard result:
- Or simply add the neutral element $\mathcal{O}$

```
R ← P
for i ← n − 2 downto 0 do
    R ← 2R
    if (k)_2[i] = 1 then
        R ← R + P
    else
        R ← R + O
    end if
end for
return R
```

Double-and-add always

- All algorithms so far perform *conditional addition* where the condition is secret
- For secret scalars (most common case!) we need something else
- Idea: Always perform addition, discard result:
- Or simply add the neutral element $\mathcal{O}$

  $R \leftarrow P$
  for $i \leftarrow n-2$ downto 0 **do**
      $R \leftarrow 2R$
      if $(k)_2[i] = 1$ **then**
          $R \leftarrow R + P$
      else
          $R \leftarrow R + \mathcal{O}$
      end if
  end for
  return R

- Still not constant time, more later...

Let's rewrite that a bit . . .

- We have a table $T = (\mathcal{O}, P)$
- Notation $T[0] = \mathcal{O}$, $T[1] = P$
- Scalar multiplication is

  $R \leftarrow P$
  for $i \leftarrow n-2$ downto 0 **do**
  $\quad R \leftarrow 2R$
  $\quad R \leftarrow R + T[(k)_2[i]]$
  end for

Changing the scalar radix

- So far we considered a scalar written in radix 2
- How about radix 3?

Changing the scalar radix

- So far we considered a scalar written in radix 2
- How about radix 3?
- We precompute a Table $T = (\mathcal{O}, P, 2P)$
- Write scalar k as $(k_{n-1}, \ldots, k_0)_3$

Changing the scalar radix

- So far we considered a scalar written in radix 2
- How about radix 3?
- We precompute a Table $T = (\mathcal{O}, P, 2P)$
- Write scalar k as $(k_{n-1}, \ldots, k_0)_3$
- Compute scalar multiplication as

  $R \leftarrow T[(k)_3[n-1]]$
  for $i \leftarrow n-2$ downto 0 **do**
  $\quad R \leftarrow 3R$
  $\quad R \leftarrow R + T[(k)_3[i]]$
  end for

Changing the scalar radix

- So far we considered a scalar written in radix 2
- How about radix 3?
- We precompute a Table $T = (\mathcal{O}, P, 2P)$
- Write scalar k as $(k_{n-1}, \ldots, k_0)_3$
- Compute scalar multiplication as

  $R \leftarrow T[(k)_3[n-1]]$
  for $i \leftarrow n-2$ downto 0 **do**
  $\quad R \leftarrow 3R$
  $\quad R \leftarrow R + T[(k)_3[i]]$
  end for
- Advantage: The scalar is shorter, fewer additions
- Disadvantage: 3 is just not nice (needs triplings)

Changing the scalar radix

- So far we considered a scalar written in radix 2
- How about radix 3?
- We precompute a Table $T = (\mathcal{O}, P, 2P)$
- Write scalar k as $(k_{n-1}, \ldots, k_0)_3$
- Compute scalar multiplication as

  $R \leftarrow T[(k)_3[n-1]]$
  for $i \leftarrow n-2$ downto 0 **do**
  $\quad R \leftarrow 3R$
  $\quad R \leftarrow R + T[(k)_3[i]]$
  end for

- Advantage: The scalar is shorter, fewer additions
- Disadvantage: 3 is just not nice (needs triplings)
- How about some nice numbers, like $4, 8, 16$?

Fixed-window scalar multiplication

- Fix a window width w
- Precompute $T = (\mathcal{O}, P, 2P, \ldots, (2^w - 1)P)$

Fixed-window scalar multiplication

- Fix a window width w
- Precompute $T = (\mathcal{O}, P, 2P, \ldots, (2^w - 1)P)$
- Write scalar k as $(k_{m-1}, \ldots, k_0)_{2^w}$
- This is the same as chopping the binary scalar into “windows” of fixed length w

Fixed-window scalar multiplication

- Fix a window width w
- Precompute $T = (\mathcal{O}, P, 2P, \ldots, (2^w - 1)P)$
- Write scalar k as $(k_{m-1}, \ldots, k_0)_{2^w}$
- This is the same as chopping the binary scalar into "windows" of fixed length w
- Compute scalar multiplication as

```
R ← T[(k)_{2^w}[m − 1]]
for i ← m − 2 downto 0 do
    for j ← 1 to w do
        R ← 2R
    end for
    R ← R + T[(k)_{2^w}[i]]
end for
```

Analysis of fixed window

- For an n-bit scalar we still have $n-1$ doublings

Analysis of fixed window

- For an n-bit scalar we still have $n - 1$ doublings
- Precomputation costs us $2^w/2 - 1$ additions and $2^w/2 - 1$ doublings

Analysis of fixed window

- For an n-bit scalar we still have $n-1$ doublings
- Precomputation costs us $2^w/2-1$ additions and $2^w/2-1$ doublings
- Number of additions in the loop is $\lceil n/w \rceil - 1$

Analysis of fixed window

- For an n-bit scalar we still have $n-1$ doublings
- Precomputation costs us $2^w/2-1$ additions and $2^w/2-1$ doublings
- Number of additions in the loop is $\lceil n/w \rceil - 1$
- Larger w: More precomputation
- Smaller w: More additions inside the loop

Analysis of fixed window

- For an n-bit scalar we still have $n-1$ doublings
- Precomputation costs us $2^w/2-1$ additions and $2^w/2-1$ doublings
- Number of additions in the loop is $\lceil n/w \rceil - 1$
- Larger w: More precomputation
- Smaller w: More additions inside the loop
- For ≈ 256-bit scalars choose $w=4$ or $w=5$

Is fixed-window constant time?

- For each window of the scalar perform w doublings and one addition, sounds good.

Is fixed-window constant time?

- For each window of the scalar perform w doublings and one addition, sounds good.
- The devil is in the detail:
  - Is addition running in constant time? Also for $\mathcal{O}$? (more tomorrow)

Is fixed-window constant time?

- For each window of the scalar perform w doublings and one addition, sounds good.
- The devil is in the detail:
  - Is addition running in constant time? Also for $\mathcal{O}$? (more tomorrow)
  - Remember that table lookups are generally not constant time!
  - Need to scan through the whole table

Is fixed-window constant time?

- For each window of the scalar perform w doublings and one addition, sounds good.
- The devil is in the detail:
  - Is addition running in constant time? Also for $\mathcal{O}$? (more tomorrow)
  - Remember that table lookups are generally not constant time!
  - Need to scan through the whole table
  - Need to “select” in constant time (remove `if` statement)

Is fixed-window constant time?

- For each window of the scalar perform w doublings and one addition, sounds good.
- The devil is in the detail:
  - Is addition running in constant time? Also for $\mathcal{O}$? (more tomorrow)
  - Remember that table lookups are generally not constant time!
  - Need to scan through the whole table
  - Need to “select” in constant time (remove `if` statement)
- See `assignment2-ecdh25519`

More offline precomputation

- Let's get back to fixed-basepoint multiplication
- So far we precomputed $P, 2P, 4P, 8P, \ldots$

More offline precomputation

- Let's get back to fixed-basepoint multiplication
- So far we precomputed $P, 2P, 4P, 8P, \ldots$
- We can combine that with fixed-window scalar multiplication
- Precompute $T_i = (\mathcal{O}, P, 2P, 3P, \ldots, (2^w - 1)P) \cdot 2^i$ for $i = 0, w, 2w, 3w, \lceil n/w \rceil - 1$

More offline precomputation

- Let's get back to fixed-basepoint multiplication
- So far we precomputed $P, 2P, 4P, 8P, \dots$
- We can combine that with fixed-window scalar multiplication
- Precompute $T_i = (\mathcal{O}, P, 2P, 3P, \dots, (2^w - 1)P) \cdot 2^i$ for $i = 0, w, 2w, 3w, \lceil n/w \rceil - 1$
- Perform scalar multiplication as

  $R \leftarrow T_0[(k)_{2^w}[0]]$
  for $i \leftarrow 1$ **to** $\lceil n/w \rceil - 1$ **do**
  $\quad R \leftarrow R + T_{iw}[(k)_{2^w}[i]]$
  end for

More offline precomputation

- Let's get back to fixed-basepoint multiplication
- So far we precomputed $P, 2P, 4P, 8P, \ldots$
- We can combine that with fixed-window scalar multiplication
- Precompute $T_i = (\mathcal{O}, P, 2P, 3P, \ldots, (2^w - 1)P) \cdot 2^i$ for $i = 0, w, 2w, 3w, \lceil n/w \rceil - 1$
- Perform scalar multiplication as

  $R \leftarrow T_0[(k)_{2^w}[0]]$
  for $i \leftarrow 1$ to $\lceil n/w \rceil - 1$ **do**
  $\quad R \leftarrow R + T_{iw}[(k)_{2^w}[i]]$
  end for
- No doublings, only $\lceil n/w \rceil - 1$ additions

More offline precomputation

- Let's get back to fixed-basepoint multiplication
- So far we precomputed $P, 2P, 4P, 8P, \dots$
- We can combine that with fixed-window scalar multiplication
- Precompute $T_i = (\mathcal{O}, P, 2P, 3P, \dots, (2^w - 1)P) \cdot 2^i$ for $i = 0, w, 2w, 3w, \lceil n/w \rceil - 1$
- Perform scalar multiplication as

  $R \leftarrow T_0[(k)_{2^w}[0]]$
  for $i \leftarrow 1$ to $\lceil n/w \rceil - 1$ **do**
  $\quad R \leftarrow R + T_{iw}[(k)_{2^w}[i]]$
  end for

- No doublings, only $\lceil n/w \rceil - 1$ additions
- Can use huge w, but:
  - at some point the precomputed tables don't fit into cache anymore.
  - constant-time loads get slow for large w

Fixed-window limitations

- Consider the scalar $22 = (1\,01\,10)_2$ and window size 2
  - Initialize R with P
  - Double, double, add P
  - Double, double, add $2P$

Fixed-window limitations

- Consider the scalar $22 = (1\,01\,10)_2$ and window size 2
  - Initialize R with P
  - Double, double, add P
  - Double, double, add $2P$
- More efficient:
  - Initialize R with P
  - Double, double, double, add $3P$
  - Double

Fixed-window limitations

- Consider the scalar $22 = (1\,01\,10)_2$ and window size 2
  - Initialize R with P
  - Double, double, add P
  - Double, double, add $2P$
- More efficient:
  - Initialize R with P
  - Double, double, double, add $3P$
  - Double
- Problem with fixed window: it's fixed.

Fixed-window limitations

- Consider the scalar $22 = (1\,01\,10)_2$ and window size 2
  - Initialize R with P
  - Double, double, add P
  - Double, double, add $2P$
- More efficient:
  - Initialize R with P
  - Double, double, double, add $3P$
  - Double
- Problem with fixed window: it's fixed.
- Idea: "slide" the window over the scalar

Sliding window scalar multiplication

- Choose window size w
- Rewrite scalar k as $k = (k_0, \ldots, k_m)$ with k_i in $\{0, 1, 3, 5, \ldots, 2^w - 1\}$ with at most one non-zero entry in each window of length w

Sliding window scalar multiplication

- Choose window size w
- Rewrite scalar k as $k = (k_0, \ldots, k_m)$ with k_i in $\{0, 1, 3, 5, \ldots, 2^w - 1\}$ with at most one non-zero entry in each window of length w
- Do this by scanning k from right to left, expand window from each 1-bit

Sliding window scalar multiplication

- Choose window size w
- Rewrite scalar k as $k = (k_0, \ldots, k_m)$ with k_i in $\{0, 1, 3, 5, \ldots, 2^w - 1\}$ with at most one non-zero entry in each window of length w
- Do this by scanning k from right to left, expand window from each 1-bit
- Precompute $P, 3P, 5P, \ldots, (2^w - 1)P$

Sliding window scalar multiplication

- Choose window size w
- Rewrite scalar k as $k = (k_0, \ldots, k_m)$ with k_i in $\{0, 1, 3, 5, \ldots, 2^w - 1\}$ with at most one non-zero entry in each window of length w
- Do this by scanning k from right to left, expand window from each 1-bit
- Precompute $P, 3P, 5P, \ldots, (2^w - 1)P$
- Perform scalar multiplication

$R \leftarrow \mathcal{O}$
for $i \leftarrow m$ to 0 **do**
  $R \leftarrow 2R$
  if $k_i \neq 0$ **then**
    $R \leftarrow R + k_i P$
  end if
end for

Analysis of sliding window

- We still do $n-1$ doublings for an n-bit scalar
- Precomputation needs $2^{w-1}-1$ additions
- Expected number of additions in the main loop: $n/(w+1)$

Analysis of sliding window

- We still do $n-1$ doublings for an n-bit scalar
- Precomputation needs $2^{w-1}-1$ additions
- Expected number of additions in the main loop: $n/(w+1)$
- For the same w only half the precomputation compared to fixed-window scalar multiplication
- For the same w fewer additions in the main loop

Analysis of sliding window

- We still do $n-1$ doublings for an n-bit scalar
- Precomputation needs $2^{w-1}-1$ additions
- Expected number of additions in the main loop: $n/(w+1)$
- For the same w only half the precomputation compared to fixed-window scalar multiplication
- For the same w fewer additions in the main loop
- But: It's not running in constant time!
- Still nice (in double-scalar version) for signature verification

Multi-scalar multiplication

- Consider computation $Q = \sum_1^n k_i P_i$
- We looked at $n = 2$ before, how about $n = 128$?

De-Rooij algorithm

- Assume $k_1 > k_2 > \cdots > k_n$.
- Use that $k_1 P_1 + k_2 P_2 = (k_1 - k_2) P_1 + k_2 (P_1 + P_2)$
- Replace:
  - $(k_1 P_1)$ and $(k_2 P_2)$, with
  - $(k_1 - k_2) P_1$ and $k_2 (P_1 + P_2)$
- Each step requires one scalar subtraction and one point addition
- Each step typically “eliminates” multiple scalar bits
- Can be very fast (but not constant-time)

Multi-scalar multiplication

- Consider computation $Q = \sum_1^n k_i P_i$
- We looked at $n = 2$ before, how about $n = 128$?

De-Rooij algorithm

- Assume $k_1 > k_2 > \cdots > k_n$.
- Use that $k_1 P_1 + k_2 P_2 = (k_1 - k_2) P_1 + k_2 (P_1 + P_2)$
- Replace:
  - $(k_1 P_1)$ and $(k_2 P_2)$, with
  - $(k_1 - k_2) P_1$ and $k_2 (P_1 + P_2)$
- Each step requires one scalar subtraction and one point addition
- Each step typically “eliminates” multiple scalar bits
- Can be very fast (but not constant-time)
- Requires fast access to the two largest scalars: put scalars into a heap
- Crucial for good performance: fast heap implementation

A fast heap

- Heap is a binary tree, each parent node is larger than the two child nodes
- Data structure is stored as a simple array, positions in the array determine positions in the tree
- Root is at position 0, left child node at position 1, right child node at position 2 etc.
- For node at position i, child nodes are at position $2 \cdot i + 1$ and $2 \cdot i + 2$, parent node is at position $\lfloor (i-1)/2 \rfloor$

A fast heap

- Heap is a binary tree, each parent node is larger than the two child nodes
- Data structure is stored as a simple array, positions in the array determine positions in the tree
- Root is at position 0, left child node at position 1, right child node at position 2 etc.
- For node at position i, child nodes are at position $2 \cdot i + 1$ and $2 \cdot i + 2$, parent node is at position $\lfloor (i-1)/2 \rfloor$
- Typical heap root replacement (pop operation): start at the root, swap down for a variable amount of times

A fast heap

- Heap is a binary tree, each parent node is larger than the two child nodes
- Data structure is stored as a simple array, positions in the array determine positions in the tree
- Root is at position 0, left child node at position 1, right child node at position 2 etc.
- For node at position i, child nodes are at position $2 \cdot i + 1$ and $2 \cdot i + 2$, parent node is at position $\lfloor (i-1)/2 \rfloor$
- Typical heap root replacement (pop operation): start at the root, swap down for a variable amount of times
- Floyd's heap: swap down to the bottom, swap up for a variable amount of times, advantages:
  - Each swap-down step needs only one comparison (instead of two)
  - Swap-down loop is more friendly to branch predictors

How about fixed scalar

- So far we have considered:
  - **variable** point, **variable** scalar
  - **fixed** point, **variable** scalar

How about fixed scalar

- So far we have considered:
  - **variable** point, **variable** scalar
  - **fixed** point, **variable** scalar
- How about **variable** point, **fixed** scalar?

How about fixed scalar

- So far we have considered:
  - **variable** point, **variable** scalar
  - **fixed** point, **variable** scalar
- How about **variable** point, **fixed** scalar?
- Optimizing for the scalar means that the scalar has to be public
- Not what we have in DH or Schnorr

How about fixed scalar

- So far we have considered:
  - **variable** point, **variable** scalar
  - **fixed** point, **variable** scalar
- How about **variable** point, **fixed** scalar?
- Optimizing for the scalar means that the scalar has to be public
- Not what we have in DH or Schnorr
- Some applications:
  - Inversion in finite fields (later in this course)
  - Elliptic-curve factorization method (not in this lecture)

Addition chains

Definition

For an integer $k > 1$ a sequence $s_1, s_2, \ldots, s_m$ is called an *addition chain of length* m *for* k if

- $s_1 = 1$
- $s_m = k$
- for each s_i with $i > 1$ it holds that $s_i = s_j + s_\ell$ for some $j, \ell < i$

Addition chains

Definition

For an integer $k > 1$ a sequence $s_1, s_2, \ldots, s_m$ is called an *addition chain of length m for k* if

- $s_1 = 1$
- $s_m = k$
- for each s_i with $i > 1$ it holds that $s_i = s_j + s_\ell$ for some $j, \ell < i$

- An addition chain immediately translates into a scalar-multiplication algorithm:
  - Start with $s_1 P = P$
  - Compute $s_i P = s_j P + s_\ell P$ for $i = 2, \ldots, m$

Addition chains

Definition

For an integer $k > 1$ a sequence $s_1, s_2, \ldots, s_m$ is called an *addition chain of length* m *for* k if

- $s_1 = 1$
- $s_m = k$
- for each s_i with $i > 1$ it holds that $s_i = s_j + s_\ell$ for some $j, \ell < i$

- An addition chain immediately translates into a scalar-multiplication algorithm:
  - Start with $s_1 P = P$
  - Compute $s_i P = s_j P + s_\ell P$ for $i = 2, \ldots, m$
- All algorithms so far just computed additions chains “on the fly”
- Signed-scalar representations are “addition-subtraction chains”

Addition chains

Definition

For an integer $k > 1$ a sequence $s_1, s_2, \dots, s_m$ is called an *addition chain of length m for k* if

- $s_1 = 1$
- $s_m = k$
- for each s_i with $i > 1$ it holds that $s_i = s_j + s_\ell$ for some $j, \ell < i$

- An addition chain immediately translates into a scalar-multiplication algorithm:
  - Start with $s_1 P = P$
  - Compute $s_i P = s_j P + s_\ell P$ for $i = 2, \dots, m$
- All algorithms so far just computed additions chains "on the fly"
- Signed-scalar representations are "addition-subtraction chains"
- For fixed scalar we can spend a lot of time to find a good addition chain at compile time
- Computing good addition chains? See https://github.com/mmcloughlin/addchain