Bonus Lecture #2: Digital Signatures in Blockchain Protocols (Part 1 of 2)

COMS 4995-001:
The Science of Blockchains

URL: https://timroughgarden.org/s25/

Tim Roughgarden

Trying to Make Sense of Cryptography

Trying to Make Sense of Cryptography

Question: in Ethereum, what signature scheme do users use to sign their transactions?

Trying to Make Sense of Cryptography

Question: in Ethereum, what signature scheme do users use to sign their transactions?

Answer: “ECDSA with the secp256k1 curve, except with Keccak-256 instead of SHA-256.”

Trying to Make Sense of Cryptography

Question: in Ethereum, what signature scheme do users use to sign their transactions?

Answer: “ECDSA with the secp256k1 curve, except with Keccak-256 instead of SHA-256.” [huh?]

Trying to Make Sense of Cryptography

Question: in Ethereum, what signature scheme do users use to sign their transactions?

Answer: “ECDSA with the secp256k1 curve, except with Keccak-256 instead of SHA-256.” [huh?]

Question: why not RSA signatures?

Trying to Make Sense of Cryptography

Question: in Ethereum, what signature scheme do users use to sign their transactions?

Answer: "ECDSA with the secp256k1 curve, except with Keccak-256 instead of SHA-256." [huh?]

Question: why not RSA signatures?

Answer: "for 128 bits of security, would need keys of length 3072 bits rather than 256 bits."

Trying to Make Sense of Cryptography

Question: in Ethereum, what signature scheme do users use to sign their transactions?

Answer: “ECDSA with the secp256k1 curve, except with Keccak-256 instead of SHA-256.” [huh?]

Question: why not RSA signatures?

Answer: “for 128 bits of security, would need keys of length 3072 bits rather than 256 bits.” [why 128? why 256? why 3072?]

Goals for Bonus Lecture #2

1. Bits of security.
   – what does it mean and how much is enough?
2. Groups and the discrete logarithm approach to signatures.
   – common to ECDSA, Schnorr, BLS, etc.
3. Algorithms for the discrete logarithm problem.
   – the discrete log problem is not as hard as you might have thought!

Digital Signature Schemes in Blockchains

- one of the two most ubiquitous cryptographic primitives used in blockchain protocols (along with cryptographic hash functions)

Application #1: allows a user of a blockchain to authorize a transaction (e.g., making a payment).

- fundamental to the vision of shared computer in the sky

Application #2: under the hood, allows validators of a blockchain protocol to sign their messages.

- used in most blockchain protocols for this purpose

Defining Digital Signature Schemes

Digital signature scheme: defined by 3 (efficient) algorithms:

1. *Key generation algorithm:* maps seed r → (pk,sk) pair.
   – in some cases, may generate r itself (e.g., ssh-keygen)
2. *Signing algorithm:* maps message + sk → signature.
   – signature depends on both sk and the message being signed
3. *Verification algorithm:* maps msg + sig + pk → "yes"/"no".
   – anyone who knows pk can verify correctness of an alleged signature

Defining Digital Signature Schemes

Digital signature scheme: defined by 3 (efficient) algorithms:

1. *Key generation algorithm:* maps seed r → (pk,sk) pair.
   – in some cases, may generate r itself (e.g., ssh-keygen)
2. *Signing algorithm:* maps message + sk → signature.
   – signature depends on both sk and the message being signed
3. *Verification algorithm:* maps msg + sig + pk → "yes"/"no".
   – anyone who knows pk can verify correctness of an alleged signature

Prime considerations: (i) security (how infeasible is it to forge signatures?);

Defining Digital Signature Schemes

Digital signature scheme: defined by 3 (efficient) algorithms:

1. *Key generation algorithm:* maps seed r → (pk,sk) pair.
   - in some cases, may generate r itself (e.g., ssh-keygen)
2. *Signing algorithm:* maps message + sk → signature.
   - signature depends on both sk and the message being signed
3. *Verification algorithm:* maps msg + sig + pk → "yes"/"no".
   - anyone who knows pk can verify correctness of an alleged signature

Prime considerations: (i) security (how infeasible is it to forge signatures?); (ii) performance (time and space).

Bits of Security

Question: how to quantify “degree of infeasibility” of an attack?

Bits of Security

Question: how to quantify "degree of infeasibility" of an attack?

Informal definition: *t bits of security* ➔ successful attack would require $\geq 2^t$ computer operations.

- example: if need to brute force a t-bit secret key ➔ t bits of security

Bits of Security

Question: how to quantify "degree of infeasibility" of an attack?

Informal definition: t bits of security ➔ *as far as we know,* successful attack would require ≥ 2^t computer operations.

- example: if need to brute force a t-bit secret key ➔ t bits of security
  - if there's an attack faster than brute force ➔ < t bits of security

Bits of Security

Question: how to quantify "degree of infeasibility" of an attack?

Informal definition: t bits of security ➔ *as far as we know,* successful attack would require $\geq 2^t$ computer operations.

- example: if need to brute force a t-bit secret key ➔ t bits of security

Question: how many bits of security are enough?

Bits of Security

Question: how to quantify "degree of infeasibility" of an attack?

Informal definition: t bits of security ➔ *as far as we know,* successful attack would require ≥ 2^t computer operations.

- example: if need to brute force a t-bit secret key ➔ t bits of security

Question: how many bits of security are enough?

Cartoon:

Bits of Security

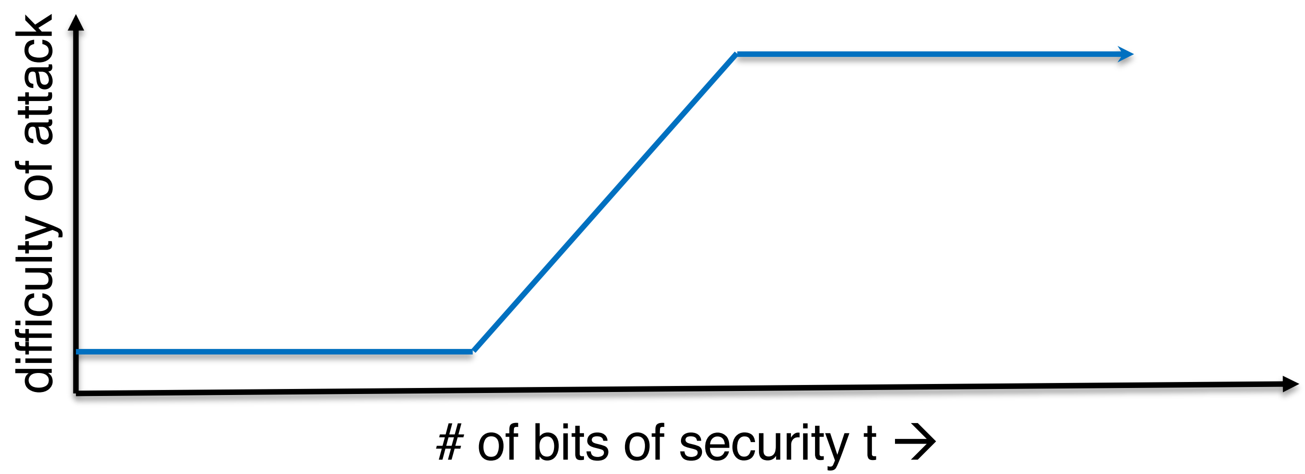
Question: how to quantify "degree of infeasibility" of an attack?

Informal definition: t bits of security ➔ *as far as we know,* successful attack would require ≥ 2^t computer operations.

- example: if need to brute force a t-bit secret key ➔ t bits of security

Question: how many bits of security are enough?

Cartoon:



Bits of Security

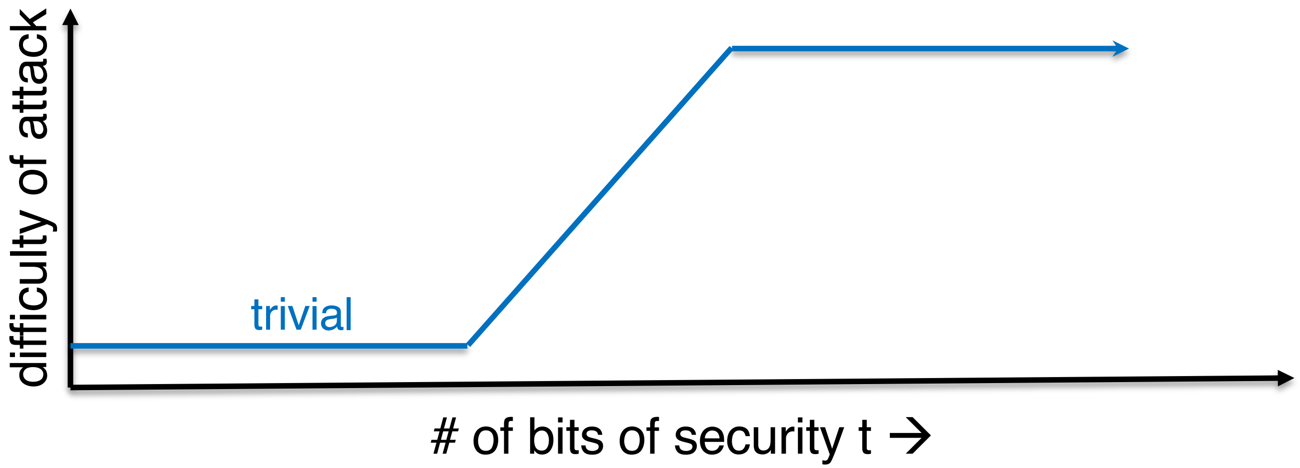
Question: how to quantify "degree of infeasibility" of an attack?

Informal definition: t bits of security ➔ *as far as we know,* successful attack would require $\geq 2^t$ computer operations.

– example: if need to brute force a t-bit secret key ➔ t bits of security

Question: how many bits of security are enough?

Cartoon:



Bits of Security

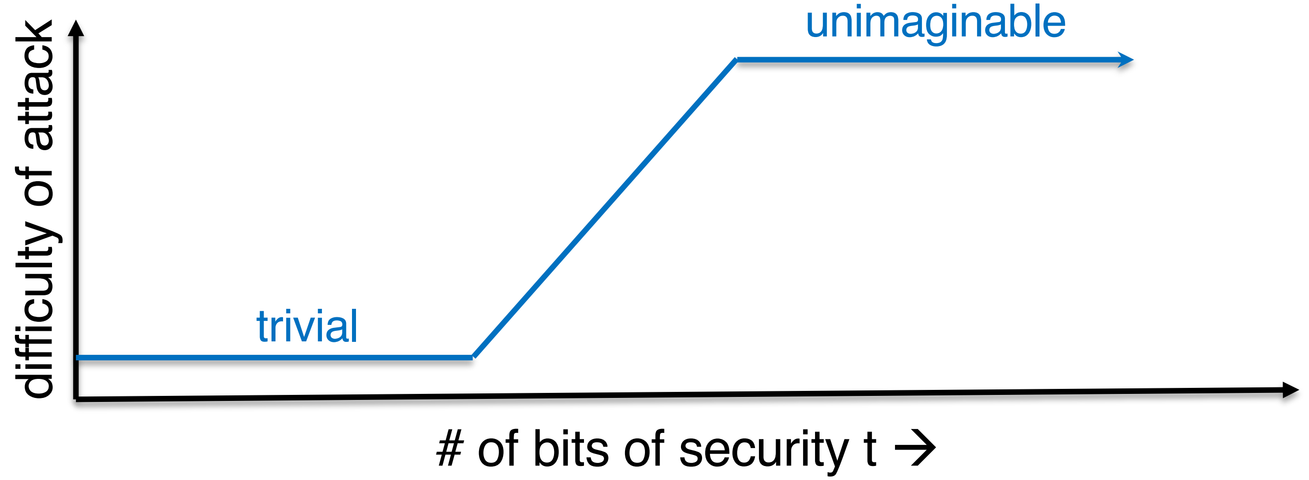
Question: how to quantify "degree of infeasibility" of an attack?

Informal definition: t bits of security ➔ *as far as we know,* successful attack would require ≥ 2^t computer operations.

- example: if need to brute force a t-bit secret key ➔ t bits of security

Question: how many bits of security are enough?

Cartoon:



Bits of Security

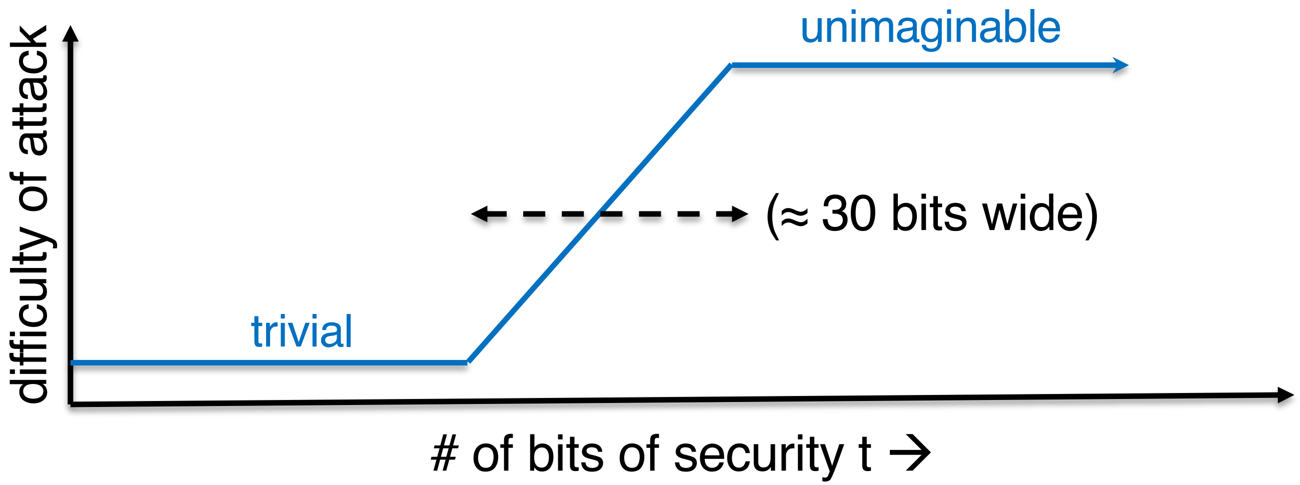
Question: how to quantify "degree of infeasibility" of an attack?

Informal definition: t bits of security ➔ *as far as we know,* successful attack would require ≥ 2^t computer operations.

- example: if need to brute force a t-bit secret key ➔ t bits of security

Question: how many bits of security are enough?

Cartoon:



How Many Bits of Security Are Enough?

Recall:

How Many Bits of Security Are Enough?

Recall: $2^{10} \approx 1000$, $2^{20} \approx$ one million, $2^{30} \approx$ one billion, etc.

How Many Bits of Security Are Enough?

Recall: $2^{10} \approx 1000$, $2^{20} \approx$ one million, $2^{30} \approx$ one billion, etc.

Time (loose approximations):

How Many Bits of Security Are Enough?

Recall: $2^{10} \approx 1000$, $2^{20} \approx$ one million, $2^{30} \approx$ one billion, etc.

Time (loose approximations):

- 1000 seconds $\approx$ 15 minutes

How Many Bits of Security Are Enough?

Recall: $2^{10} \approx 1000$, $2^{20} \approx$ one million, $2^{30} \approx$ one billion, etc.

Time (loose approximations):

- 1000 seconds ≈ 15 minutes
- one million seconds ≈ 10 days

How Many Bits of Security Are Enough?

Recall: $2^{10} \approx 1000$, $2^{20} \approx$ one million, $2^{30} \approx$ one billion, etc.

Time (loose approximations):

- 1000 seconds ≈ 15 minutes
- one million seconds ≈ 10 days
- one billion seconds ≈ 30 years

How Many Bits of Security Are Enough?

Recall: $2^{10} \approx 1000$, $2^{20} \approx$ one million, $2^{30} \approx$ one billion, etc.

Time (loose approximations): 1000 seconds ≈ 15 minutes, one million seconds ≈ 10 days, one billion seconds ≈ 30 years.

How Many Bits of Security Are Enough?

Recall: $2^{10} \approx 1000$, $2^{20} \approx$ one million, $2^{30} \approx$ one billion, etc.

Time (loose approximations): 1000 seconds ≈ 15 minutes,
one million seconds ≈ 10 days, one billion seconds ≈ 30 years.

Computational power: [Moore's Law ➔ doubles every 18 months]

How Many Bits of Security Are Enough?

Recall: $2^{10} \approx 1000$, $2^{20} \approx$ one million, $2^{30} \approx$ one billion, etc.

Time (loose approximations): 1000 seconds ≈ 15 minutes, one million seconds ≈ 10 days, one billion seconds ≈ 30 years.

Computational power: [Moore's Law ➔ doubles every 18 months]

- El Capitan (world's fastest supercomputer): $\approx 2^{60}$ ops/sec

How Many Bits of Security Are Enough?

Recall: $2^{10} \approx 1000$, $2^{20} \approx$ one million, $2^{30} \approx$ one billion, etc.

Time (loose approximations): 1000 seconds ≈ 15 minutes,
one million seconds ≈ 10 days, one billion seconds ≈ 30 years.

Computational power: [Moore's Law ➔ doubles every 18 months]

- El Capitan (world's fastest supercomputer): $\approx 2^{60}$ ops/sec
- Macbook Pro (M4 Chip): $\approx 2^{45}$ ops/sec

How Many Bits of Security Are Enough?

Recall: $2^{10} \approx 1000$, $2^{20} \approx$ one million, $2^{30} \approx$ one billion, etc.

Time (loose approximations): 1000 seconds ≈ 15 minutes, one million seconds ≈ 10 days, one billion seconds ≈ 30 years.

Computational power: [Moore's Law ➔ doubles every 18 months]

- El Capitan (world's fastest supercomputer): $\approx 2^{60}$ ops/sec
- Macbook Pro (M4 Chip): $\approx 2^{45}$ ops/sec
- 1000 Macbook Pros (M4 Chip): $\approx 2^{55}$ ops/sec

How Many Bits of Security Are Enough?

Recall: $2^{10} \approx 1000$, $2^{20} \approx$ one million, $2^{30} \approx$ one billion, etc.

Time (loose approximations): 1000 seconds ≈ 15 minutes, one million seconds ≈ 10 days, one billion seconds ≈ 30 years.

Computational power: [Moore's Law ➔ doubles every 18 months]

- El Capitan (world's fastest supercomputer): $\approx 2^{60}$ ops/sec
- Macbook Pro (M4 Chip): $\approx 2^{45}$ ops/sec
- 1000 Macbook Pros (M4 Chip): $\approx 2^{55}$ ops/sec
- Antminer S21 XP: $\approx 2^{45}$ SHA-256 hashes/sec

How Many Bits of Security Are Enough?

Recall: $2^{10} \approx 1000$, $2^{20} \approx$ one million, $2^{30} \approx$ one billion, etc.

Time (loose approximations): 1000 seconds ≈ 15 minutes, one million seconds ≈ 10 days, one billion seconds ≈ 30 years.

Computational power: [Moore's Law ➔ doubles every 18 months]

- El Capitan (world's fastest supercomputer): $\approx 2^{60}$ ops/sec
- Macbook Pro (M4 Chip): $\approx 2^{45}$ ops/sec
- 1000 Macbook Pros (M4 Chip): $\approx 2^{55}$ ops/sec
- Antminer S21 XP: $\approx 2^{45}$ SHA-256 hashes/sec
  - all of Bitcoin mining: $\approx 2^{70}$ SHA-256 hashes/sec

How Many Bits of Security Are Enough?

Recall: $2^{10} \approx 1000$, $2^{20} \approx$ one million, $2^{30} \approx$ one billion, etc.

Time (loose approximations): 1000 seconds ≈ 15 minutes, one million seconds ≈ 10 days, one billion seconds ≈ 30 years.

Computational power: [Moore's Law ➔ doubles every 18 months]

- 2^{45} ops/sec easy, 2^{60} ops/sec expensive but possible

How Many Bits of Security Are Enough?

Recall: $2^{10} \approx 1000$, $2^{20} \approx$ one million, $2^{30} \approx$ one billion, etc.

Time (loose approximations): 1000 seconds ≈ 15 minutes, one million seconds ≈ 10 days, one billion seconds ≈ 30 years.

Computational power: [Moore's Law ➔ doubles every 18 months]

- 2^{45} ops/sec easy, 2^{60} ops/sec expensive but possible

Upshot:

- 80 bits of security fine 20 years ago, not good enough now
- 128 bits regarded as plenty in the short- and medium-term

The Key Generation Algorithm

Question: what do digital signature schemes typically look like?

The Key Generation Algorithm

Question: what do digital signature schemes typically look like?

Idea #1: sk = a uniformly random t-bit string. [e.g., t = 256]

The Key Generation Algorithm

Question: what do digital signature schemes typically look like?

Idea #1: sk = a uniformly random t-bit string. [e.g., t = 256]

- note: an unimaginably large number [#atoms in universe ≈ 2^{265}]
- variation: sk uniformly random from {1,2,…,p-1} for t-bit prime p

The Key Generation Algorithm

Question: what do digital signature schemes typically look like?

Idea #1: sk = a uniformly random t-bit string. [e.g., t = 256]

- note: an unimaginably large number [#atoms in universe ≈ 2^{265}]
- variation: sk uniformly random from {1,2,…,p-1} for t-bit prime p

Idea #2: pk = some deterministic function of sk: pk := f(sk).

The Key Generation Algorithm

Question: what do digital signature schemes typically look like?

Idea #1: sk = a uniformly random t-bit string. [e.g., t = 256]

- note: an unimaginably large number [#atoms in universe ≈ 2^{265}]
- variation: sk uniformly random from {1,2,…,p-1} for t-bit prime p

Idea #2: pk = some deterministic function of sk: pk := f(sk).

- f must be efficiently computable (i.e., in time polynomial in t)

The Key Generation Algorithm

Question: what do digital signature schemes typically look like?

Idea #1: sk = a uniformly random t-bit string. [e.g., t = 256]

- note: an unimaginably large number [#atoms in universe ≈ 2^{265}]
- variation: sk uniformly random from {1,2,…,p-1} for t-bit prime p

Idea #2: pk = some deterministic function of sk: pk := f(sk).

- f must be efficiently computable (i.e., in time polynomial in t)
- f must be invertible (i.e., sk ≠ sk' ➔ f(sk) ≠ f(sk'))

The Key Generation Algorithm

Question: what do digital signature schemes typically look like?

Idea #1: sk = a uniformly random t-bit string. [e.g., t = 256]

- note: an unimaginably large number [#atoms in universe ≈ 2^{265}]
- variation: sk uniformly random from {1,2,…,p-1} for t-bit prime p

Idea #2: pk = some deterministic function of sk: pk := f(sk).

- f must be efficiently computable (i.e., in time polynomial in t)
- f must be invertible (i.e., sk ≠ sk' ➔ f(sk) ≠ f(sk'))
- f^{-1} must be hard to compute (ideally, exponential in t)

The Key Generation Algorithm

Question: what do digital signature schemes typically look like?

Idea #1: sk = a uniformly random t-bit string. [e.g., t = 256]

- note: an unimaginably large number [#atoms in universe ≈ 2^{265}]
- variation: sk uniformly random from {1,2,…,p-1} for t-bit prime p

Idea #2: pk = some deterministic function of sk: pk := f(sk).

- f must be efficiently computable (i.e., in time polynomial in t)
- f must be invertible (i.e., sk ≠ sk' ➔ f(sk) ≠ f(sk'))
- f^{-1} must be hard to compute (ideally, exponential in t)
  - otherwise, attacker can reverse engineer sk from pk!

The Key Generation Algorithm

Question: what do digital signature schemes typically look like?

Idea #1: sk = a uniformly random t-bit string. [e.g., t = 256]

- note: an unimaginably large number [#atoms in universe ≈ 2^{265}]
- variation: sk uniformly random from {1,2,…,p-1} for t-bit prime p

Idea #2: pk = some deterministic function of sk: pk := f(sk).

- f must be efficiently computable (i.e., in time polynomial in t)
- f must be invertible (i.e., sk ≠ sk' ➔ f(sk) ≠ f(sk'))
- f^{-1} must be hard to compute (ideally, exponential in t)
  - issue: no hope of proving this (would imply P ≠ NP!)

The Key Generation Algorithm

Question: what do digital signature schemes typically look like?

Idea #1: sk = a uniformly random t-bit string. [e.g., t = 256]

- note: an unimaginably large number [#atoms in universe ≈ 2^{265}]
- variation: sk uniformly random from {1,2,…,p-1} for t-bit prime p

Idea #2: pk = some deterministic function of sk: pk := f(sk).

- f must be efficiently computable (i.e., in time polynomial in t)
- f must be invertible (i.e., sk ≠ sk' ➔ f(sk) ≠ f(sk'))
- *as far as we know*, f^{-1} hard to compute (ideally, exponential in t)
  - otherwise, attacker can reverse engineer sk from pk!

Easy to Evaluate, Hard to Invert?

Attempt #1: for some fixed positive integer a, $f(x) = a \circ x$

- conceptually (not computationally!), add a to itself x times

Easy to Evaluate, Hard to Invert?

Attempt #1: for some fixed positive integer a, $f(x) = a \circ x$

- conceptually (not computationally!), add a to itself x times
- easy to evaluate [grade-school multiplication = $O(t^2)$ time]

Easy to Evaluate, Hard to Invert?

Attempt #1: for some fixed positive integer a, $f(x) = a \circ x$

- conceptually (not computationally!), add a to itself x times
- easy to evaluate [grade-school multiplication = $O(t^2)$ time]
- issue: easy to invert [long division = $O(t^2)$ time]

Easy to Evaluate, Hard to Invert?

Attempt #1: for some fixed positive integer a, $f(x) = a \circ x$

- conceptually (not computationally!), add a to itself x times
- easy to evaluate [grade-school multiplication = $O(t^2)$ time]
- issue: easy to invert [long division = $O(t^2)$ time]

Attempt #2: for some fixed ≈t-bit prime p and a in {2,3,…,p-1},
$f(x) = a \circ x \bmod p$ (remainder of $a \circ x$ after discarding multiples of p).

Easy to Evaluate, Hard to Invert?

Attempt #1: for some fixed positive integer a, $f(x) = a \circ x$

- conceptually (not computationally!), add a to itself x times
- easy to evaluate [grade-school multiplication = $O(t^2)$ time]
- issue: easy to invert [long division = $O(t^2)$ time]

Attempt #2: for some fixed ≈t-bit prime p and a in {2,3,…,p-1}, $f(x) = a \circ x \bmod p$ (remainder of $a \circ x$ after discarding multiples of p).

- note: modular arithmetic keeps sizes of all number bounded
  - why use a prime p? ensures that f is invertible (you check)

Easy to Evaluate, Hard to Invert?

Attempt #1: for some fixed positive integer a, $f(x) = a \circ x$

- conceptually (not computationally!), add a to itself x times
- easy to evaluate [grade-school multiplication = $O(t^2)$ time]
- issue: easy to invert [long division = $O(t^2)$ time]

Attempt #2: for some fixed ≈t-bit prime p and a in {2,3,…,p-1},
$f(x) = a \circ x \bmod p$ (remainder of $a \circ x$ after discarding multiples of p).

- note: modular arithmetic keeps sizes of all number bounded
  - why use a prime p? ensures that f is invertible (you check)
- issue: easy to invert [extended Euclid's algorithm = $O(t^2)$ time]

Easy to Evaluate, Hard to Invert? (con'd)

Attempt #3: for some fixed integer $a \geq 2$, $f(x) = a^x$.

- conceptually, multiply a by itself x times

Easy to Evaluate, Hard to Invert? (con'd)

Attempt #3: for some fixed integer $a \geq 2$, $f(x) = a^x$.

- conceptually, multiply a by itself x times
- issue: hard to evaluate [need $x \approx 2^t$ multiplications]

Easy to Evaluate, Hard to Invert? (con'd)

Attempt #3: for some fixed integer $a \geq 2$, $f(x) = a^x$.

- conceptually, multiply a by itself x times
- ~~issue: hard to evaluate [need $x \approx 2^t$ multiplications]~~
- repeated squaring ➔ can evaluate using $\leq 2t$ multiplications

Easy to Evaluate, Hard to Invert? (con'd)

Attempt #3: for some fixed integer $a \geq 2$, $f(x) = a^x$.

- conceptually, multiply a by itself x times
- ~~issue: hard to evaluate [need $x \approx 2^t$ multiplications]~~
- repeated squaring ➔ can evaluate using $\leq 2t$ multiplications
  - e.g., to compute a^{23}

Easy to Evaluate, Hard to Invert? (con'd)

Attempt #3: for some fixed integer $a \geq 2$, $f(x) = a^x$.

- conceptually, multiply a by itself x times
- ~~issue: hard to evaluate [need $x \approx 2^t$ multiplications]~~
- repeated squaring ➔ can evaluate using $\leq 2t$ multiplications
  - e.g., to compute a^{23}, compute $\underbrace{a, a^2, a^4, a^8, a^{16}}_{\leq \text{ t multiplications}}$,

Easy to Evaluate, Hard to Invert? (con'd)

Attempt #3: for some fixed integer $a \geq 2$, $f(x) = a^x$.

- conceptually, multiply a by itself x times
- ~~issue: hard to evaluate [need $x \approx 2^t$ multiplications]~~
- repeated squaring ➔ can evaluate using $\leq 2t$ multiplications
  - e.g., to compute a^{23}, compute $\underbrace{a, a^2, a^4, a^8, a^{16}}_{\leq t \text{ multiplications}}$, and $\underbrace{a \circ a^2 \circ a^4 \circ a^{16}}_{\leq t \text{ multiplications}}$

Easy to Evaluate, Hard to Invert? (con'd)

Attempt #3: for some fixed integer $a \geq 2$, $f(x) = a^x$.

- conceptually, multiply a by itself x times
- ~~issue: hard to evaluate [need $x \approx 2^t$ multiplications]~~
- repeated squaring ➔ can evaluate using $\leq 2t$ multiplications
  - e.g., to compute a^{23}, compute $\underbrace{a, a^2, a^4, a^8, a^{16}}_{\leq \text{ t multiplications}}$, and $\underbrace{a \circ a^2 \circ a^4 \circ a^{16}}_{\leq \text{ t multiplications}}$

- note: can similarly invert f with O(t) multiplications
  - repeatedly square, overshoot the target, divide out, repeat

Easy to Evaluate, Hard to Invert? (con'd)

Attempt #3: for some fixed integer $a \geq 2$, $f(x) = a^x$.

- conceptually, multiply a by itself x times
- issue: f not one-way ($O(t)$ multiplications to evaluate or invert)

Easy to Evaluate, Hard to Invert? (con'd)

Attempt #3: for some fixed integer $a \geq 2$, $f(x) = a^x$.

- conceptually, multiply a by itself x times
- issue: f not one-way ($O(t)$ multiplications to evaluate or invert)

Attempt #4: for some fixed $\approx$t-bit prime p and a in $\{2,3,\ldots,p-1\}$, $f(x) = a^x \bmod p$ (remainder of a^x after discarding multiples of p).

Easy to Evaluate, Hard to Invert? (con'd)

Attempt #3: for some fixed integer $a \geq 2$, $f(x) = a^x$.

- conceptually, multiply a by itself x times
- issue: f not one-way ($O(t)$ multiplications to evaluate or invert)

Attempt #4: for some fixed ≈t-bit prime p and a in {2,3,…,p-1}, $f(x) = a^x \bmod p$ (remainder of a^x after discarding multiples of p).

- note: modular arithmetic keeps sizes of all number bounded
- evaluating f by repeated squaring ➔ $O(t^3)$ time

Easy to Evaluate, Hard to Invert? (con'd)

Attempt #3: for some fixed integer $a \geq 2$, $f(x) = a^x$.

- conceptually, multiply a by itself x times
- issue: f not one-way ($O(t)$ multiplications to evaluate or invert)

Attempt #4: for some fixed ≈t-bit prime p and a in {2,3,…,p-1}, $f(x) = a^x \bmod p$ (remainder of a^x after discarding multiples of p).

- note: modular arithmetic keeps sizes of all number bounded
- evaluating f by repeated squaring ➔ $O(t^3)$ time
- fact: for carefully chosen a and p, not known how to invert this f in polynomial time! (➔ candidate for a signature scheme)

A Candidate One-Way Function

Attempt #4: for some fixed ≈t-bit prime p and a in {2,3,…,p-1}, $f(x) = a^x \bmod p$ (remainder of a^x after discarding multiples of p).

- evaluating f by repeated squaring ➔ $O(t^3)$ time
- fact: for carefully chosen a and p, not known how to invert this f in polynomial time! (➔ candidate for a signature scheme)

A Candidate One-Way Function

Attempt #4: for some fixed ≈t-bit prime p and a in {2,3,…,p-1}, $f(x) = a^x \bmod p$ (remainder of a^x after discarding multiples of p).

- evaluating f by repeated squaring ➔ $O(t^3)$ time
- fact: for carefully chosen a and p, not known how to invert this f in polynomial time! (➔ candidate for a signature scheme)

Question: is f invertible?

A Candidate One-Way Function

Attempt #4: for some fixed ≈t-bit prime p and a in {2,3,…,p-1}, $f(x) = a^x \bmod p$ (remainder of a^x after discarding multiples of p).

- evaluating f by repeated squaring ➔ $O(t^3)$ time
- fact: for carefully chosen a and p, not known how to invert this f in polynomial time! (➔ candidate for a signature scheme)

Question: is f invertible?

Answer: yes, provided:

- $p \geq 2^t + 1$ [obviously necessary]

A Candidate One-Way Function

Attempt #4: for some fixed ≈t-bit prime p and a in {2,3,…,p-1}, $f(x) = a^x \bmod p$ (remainder of a^x after discarding multiples of p).

- evaluating f by repeated squaring ➔ $O(t^3)$ time
- fact: for carefully chosen a and p, not known how to invert this f in polynomial time! (➔ candidate for a signature scheme)

Question: is f invertible?

Answer: yes, provided:

- $p \geq 2^t + 1$ [obviously necessary]
- a chosen as a "generator of Z_p^*" [explained next]

Generators of Cyclic Groups

Notation: $Z_p^* = \{1,2,\ldots,p\text{-}1\}$.

Example: 3 is a generator of $Z_7^* = \{1,2,\ldots,6\}$:

Generators of Cyclic Groups

Notation: $Z_p^* = \{1,2,\ldots,p\text{-}1\}$.

Example: 3 is a generator of $Z_7^* = \{1,2,\ldots,6\}$:

- $3^1 = 3 \pmod 7$

Generators of Cyclic Groups

Notation: $Z_p^* = \{1,2,\ldots,p\text{-}1\}$.

Example: 3 is a generator of $Z_7^* = \{1,2,\ldots,6\}$:

- $3^1 = 3 \pmod 7$
- $3^2 = 2 \pmod 7$

Generators of Cyclic Groups

Notation: $Z_p^* = \{1,2,\ldots,p\text{-}1\}$.

Example: 3 is a generator of $Z_7^* = \{1,2,\ldots,6\}$:

- $3^1 = 3 \pmod 7$
- $3^2 = 2 \pmod 7$
- $3^3 = 6 \pmod 7$

Generators of Cyclic Groups

Notation: $Z_p^* = \{1,2,\ldots,p\text{-}1\}$.

Example: 3 is a generator of $Z_7^* = \{1,2,\ldots,6\}$:

- $3^1 = 3 \pmod 7$
- $3^2 = 2 \pmod 7$
- $3^3 = 6 \pmod 7$
- $3^4 = 4 \pmod 7$

Generators of Cyclic Groups

Notation: $Z_p^* = \{1,2,\ldots,p-1\}$.

Example: 3 is a generator of $Z_7^* = \{1,2,\ldots,6\}$:

- $3^1 = 3 \pmod 7$
- $3^2 = 2 \pmod 7$
- $3^3 = 6 \pmod 7$
- $3^4 = 4 \pmod 7$
- $3^5 = 5 \pmod 7$

Generators of Cyclic Groups

Notation: $Z_p^* = \{1,2,\ldots,p-1\}$.

Example: 3 is a generator of $Z_7^* = \{1,2,\ldots,6\}$:

- $3^1 = 3 \pmod 7$
- $3^2 = 2 \pmod 7$
- $3^3 = 6 \pmod 7$
- $3^4 = 4 \pmod 7$
- $3^5 = 5 \pmod 7$
- $3^6 = 1 \pmod 7$

Generators of Cyclic Groups

Notation: $Z_p^* = \{1,2,\ldots,p\text{-}1\}$.

Example: 3 is a generator of $Z_7^* = \{1,2,\ldots,6\}$:

- $3^1 = 3 \pmod 7$
- $3^2 = 2 \pmod 7$
- $3^3 = 6 \pmod 7$
- $3^4 = 4 \pmod 7$
- $3^5 = 5 \pmod 7$
- $3^6 = 1 \pmod 7$

i.e., can relabel elements of Z_7^* as distinct powers of 3

Generators of Cyclic Groups

Notation: $Z_p^* = \{1,2,\ldots,p-1\}$.

Example: 3 is a generator of $Z_7^* = \{1,2,\ldots,6\}$. [but 2 is not]

- $3^1 = 3 \pmod 7$
- $3^2 = 2 \pmod 7$
- $3^3 = 6 \pmod 7$
- $3^4 = 4 \pmod 7$
- $3^5 = 5 \pmod 7$
- $3^6 = 1 \pmod 7$

i.e., can relabel elements of Z_7^* as distinct powers of 3

Generators of Cyclic Groups

Notation: $Z_p^* = \{1,2,\ldots,p\text{-}1\}$.

Example: 3 is a generator of $Z_7^* = \{1,2,\ldots,6\}$. [but 2 is not]

Example: 2 is a generator of $Z_{11}^* = \{1,2,\ldots,10\}$:

Generators of Cyclic Groups

Notation: $Z_p^* = \{1,2,\ldots,p\text{-}1\}$.

Example: 3 is a generator of $Z_7^* = \{1,2,\ldots,6\}$. [but 2 is not]

Example: 2 is a generator of $Z_{11}^* = \{1,2,\ldots,10\}$:

- $2^1 = 2 \pmod{11}$
- $2^2 = 4 \pmod{11}$
- $2^3 = 8 \pmod{11}$
- $2^4 = 5 \pmod{11}$
- $2^5 = 10 \pmod{11}$

Generators of Cyclic Groups

Notation: $Z_p^* = \{1,2,\ldots,p-1\}$.

Example: 3 is a generator of $Z_7^* = \{1,2,\ldots,6\}$. [but 2 is not]

Example: 2 is a generator of $Z_{11}^* = \{1,2,\ldots,10\}$:

- $2^1 = 2 \pmod{11}$
- $2^2 = 4 \pmod{11}$
- $2^3 = 8 \pmod{11}$
- $2^4 = 5 \pmod{11}$
- $2^5 = 10 \pmod{11}$
- $2^6 = 9 \pmod{11}$
- $2^7 = 7 \pmod{11}$
- $2^8 = 3 \pmod{11}$
- $2^9 = 6 \pmod{11}$
- $2^{10} = 1 \pmod{11}$

Generators of Cyclic Groups

Notation: $Z_p^* = \{1,2,\ldots,p-1\}$.

Example: 3 is a generator of $Z_7^* = \{1,2,\ldots,6\}$. [but 2 is not]

Example: 2 is a generator of $Z_{11}^* = \{1,2,\ldots,10\}$:

- $2^1 = 2 \pmod{11}$
- $2^2 = 4 \pmod{11}$
- $2^3 = 8 \pmod{11}$
- $2^4 = 5 \pmod{11}$
- $2^5 = 10 \pmod{11}$
- $2^6 = 9 \pmod{11}$
- $2^7 = 7 \pmod{11}$
- $2^8 = 3 \pmod{11}$
- $2^9 = 6 \pmod{11}$
- $2^{10} = 1 \pmod{11}$

i.e., can relabel elements of Z_{11}^* as distinct powers of 2

Generators of Cyclic Groups

Notation: $Z_p^* = \{1,2,\ldots,p-1\}$.

Example: 3 is a generator of $Z_7^* = \{1,2,\ldots,6\}$. [but 2 is not]

Example: 2 is a generator of $Z_{11}^* = \{1,2,\ldots,10\}$.

In general: for every prime p, Z_p^* has a generator.

Generators of Cyclic Groups

Notation: $Z_p^* = \{1,2,\ldots,p-1\}$.

Example: 3 is a generator of $Z_7^* = \{1,2,\ldots,6\}$. [but 2 is not]

Example: 2 is a generator of $Z_{11}^* = \{1,2,\ldots,10\}$.

In general: for every prime p, Z_p^* has a generator.

Terminology: for every prime p, Z_p^* is a *cyclic group*:

Generators of Cyclic Groups

Notation: $Z_p^* = \{1,2,\ldots,p-1\}$.

Example: 3 is a generator of $Z_7^* = \{1,2,\ldots,6\}$. [but 2 is not]

Example: 2 is a generator of $Z_{11}^* = \{1,2,\ldots,10\}$.

In general: for every prime p, Z_p^* has a generator.

Terminology: for every prime p, Z_p^* is a *cyclic group*:

- has a well-defined binary operation (here, multiplication mod p)
- each element has an inverse (can always "undo" an operation)

Generators of Cyclic Groups

Notation: $Z_p^* = \{1,2,\ldots,p-1\}$.

Example: 3 is a generator of $Z_7^* = \{1,2,\ldots,6\}$. [but 2 is not]

Example: 2 is a generator of $Z_{11}^* = \{1,2,\ldots,10\}$.

In general: for every prime p, Z_p^* has a generator.

Terminology: for every prime p, Z_p^* is a *cyclic group*:

- has a well-defined binary operation (here, multiplication mod p)
- each element has an inverse (can always “undo” an operation)
- has a generator g (like 3 or 2, above)

The Discrete Logarithm (DL) Approach

Cyclic group:

- has a well-defined binary operation (here, multiplication mod p)
- each element has an inverse (can always "undo" an operation)
- has a generator g (like 3 or 2, above)

General approach to key generation:

The Discrete Logarithm (DL) Approach

Cyclic group:

- has a well-defined binary operation (here, multiplication mod p)
- each element has an inverse (can always “undo” an operation)
- has a generator g (like 3 or 2, above)

General approach to key generation:

- choose G = cyclic group with generator g and order/size $q \geq 2^t$

The Discrete Logarithm (DL) Approach

Cyclic group:

- has a well-defined binary operation (here, multiplication mod p)
- each element has an inverse (can always "undo" an operation)
- has a generator g (like 3 or 2, above)

General approach to key generation:

- choose G = cyclic group with generator g and order/size $q \geq 2^t$
- sk := random t-bit string

The Discrete Logarithm (DL) Approach

Cyclic group:

- has a well-defined binary operation (here, multiplication mod p)
- each element has an inverse (can always "undo" an operation)
- has a generator g (like 3 or 2, above)

General approach to key generation:

- choose G = cyclic group with generator g and order/size $q \geq 2^t$
- sk := random t-bit string
- pk := g^{sk} (can be computed efficiently by repeated squaring)
  - conceptually, multiply (i.e., group operation) g by itself sk times

The Discrete Logarithm (DL) Approach

Cyclic group:

- has a well-defined binary operation (here, multiplication mod p)
- each element has an inverse (can always “undo” an operation)
- has a generator g (like 3 or 2, above)

General approach to key generation:

- choose G = cyclic group with generator g and order/size $q \geq 2^t$
- sk := random t-bit string, pk := g^{sk}

Discrete log (DL) assumption for G: there is no (randomized) polynomial-time algorithm that can recover x from g and g^x (with non-negligible probability). [necessary condition for security]

How Hard Is the DL Problem?

Discrete log (DL) assumption for G: there is no (randomized) polynomial-time algorithm that can recover x from g and g^x (with non-negligible probability). [necessary condition for security]

- note: false in some groups (e.g., addition modulo p)

How Hard Is the DL Problem?

Discrete log (DL) assumption for G: there is no (randomized) polynomial-time algorithm that can recover x from g and g^x (with non-negligible probability). [necessary condition for security]

- note: false in some groups (e.g., addition modulo p)

Fact: if the group G has order q, can solve the discrete log problem for G with $O(\sqrt{q})$ group operations.

How Hard Is the DL Problem?

Discrete log (DL) assumption for G: there is no (randomized) polynomial-time algorithm that can recover x from g and g^x (with non-negligible probability). [necessary condition for security]

- note: false in some groups (e.g., addition modulo p)

Fact: if the group G has order q, can solve the discrete log problem for G with $O(\sqrt{q})$ group operations.

Consequence: with DL approach, need key size ≥ 256 bits to get 128 bits of security [no matter what G is].

A Black-Box Discrete Log Algorithm

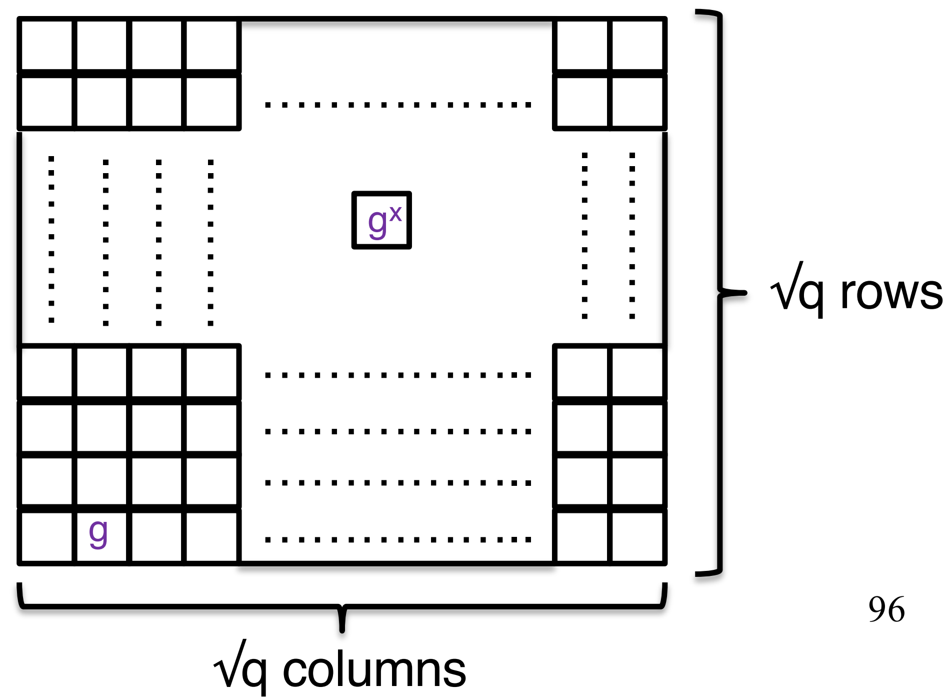
Fact: if the group G has order q, can solve the discrete log problem for G with $O(\sqrt{q})$ group operations.

Idea of algorithm: [given generator g and g^x, need to recover x]

A Black-Box Discrete Log Algorithm

Fact: if the group G has order q, can solve the discrete log problem for G with $O(\sqrt{q})$ group operations.

Idea of algorithm: [given generator g and g^x, need to recover x]

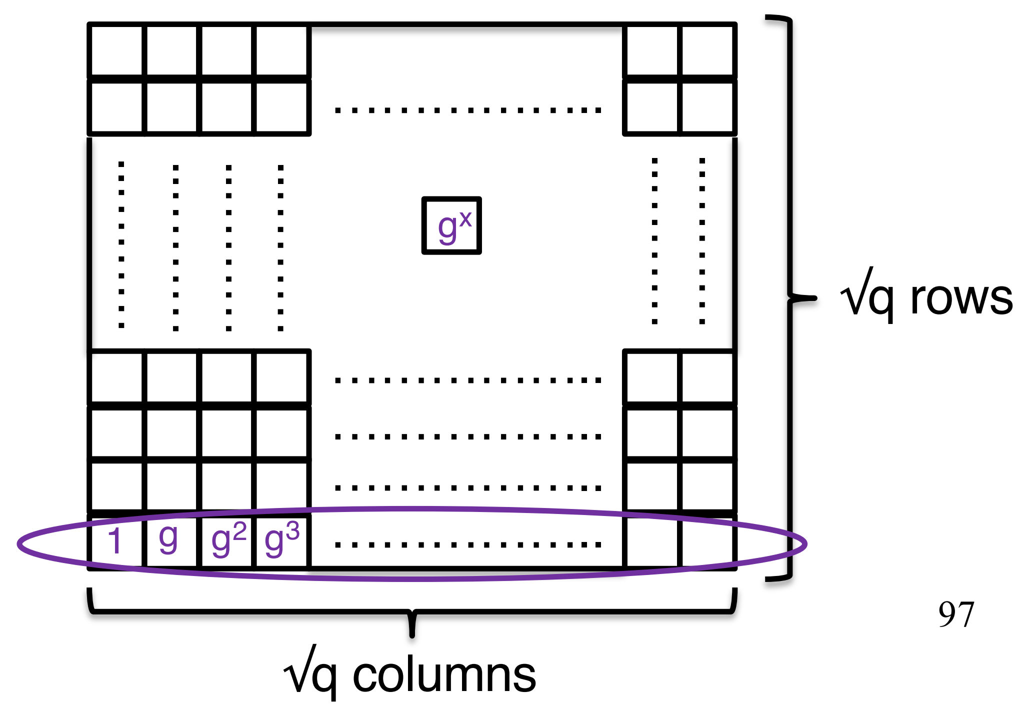


A Black-Box Discrete Log Algorithm

Fact: if the group G has order q, can solve the discrete log problem for G with $O(\sqrt{q})$ group operations.

Idea of algorithm: [given generator g and g^x, need to recover x]

- step 1: compute $g^2, g^3, g^4, \ldots, g^{\sqrt{q}}$
  [i.e., all entries in bottom row]

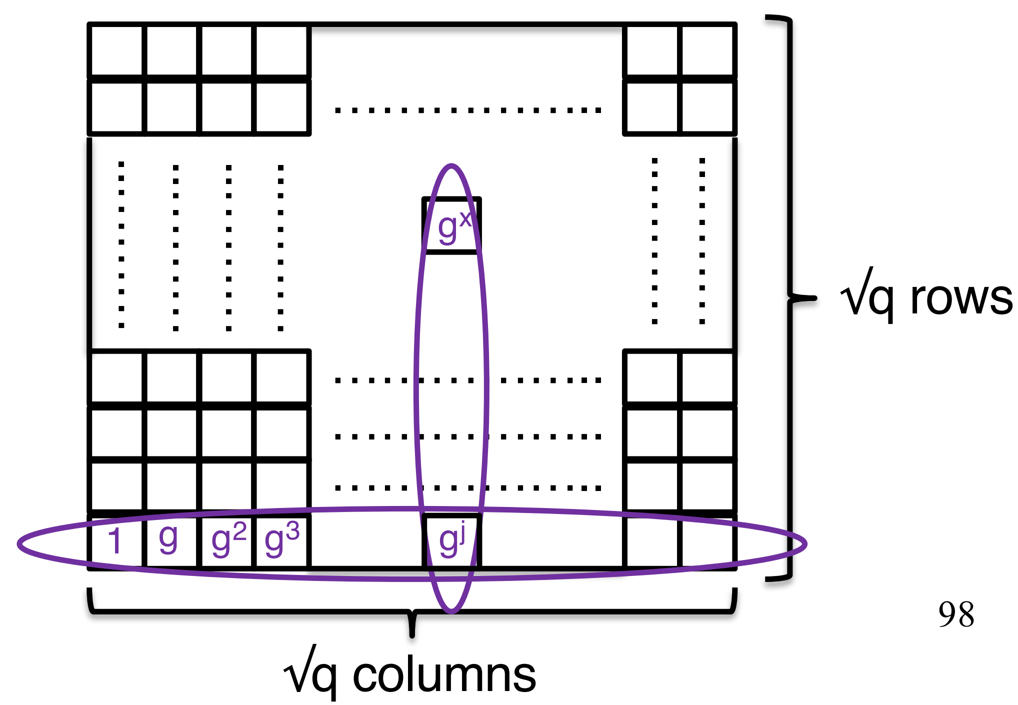


A Black-Box Discrete Log Algorithm

Fact: if the group G has order q, can solve the discrete log problem for G with $O(\sqrt{q})$ group operations.

Idea of algorithm: [given generator g and g^x, need to recover x]

- step 1: compute $g^2, g^3, g^4, \ldots, g^{\sqrt{q}}$
  [i.e., all entries in bottom row]
- step 2: compute $g^{x-\sqrt{q}}$, $g^{x-2\sqrt{q}}$, $g^{x-3\sqrt{q}}, \ldots$ i times until see repeat value g^j from step 1 [i.e., go down from g^x until you hit bottom row]

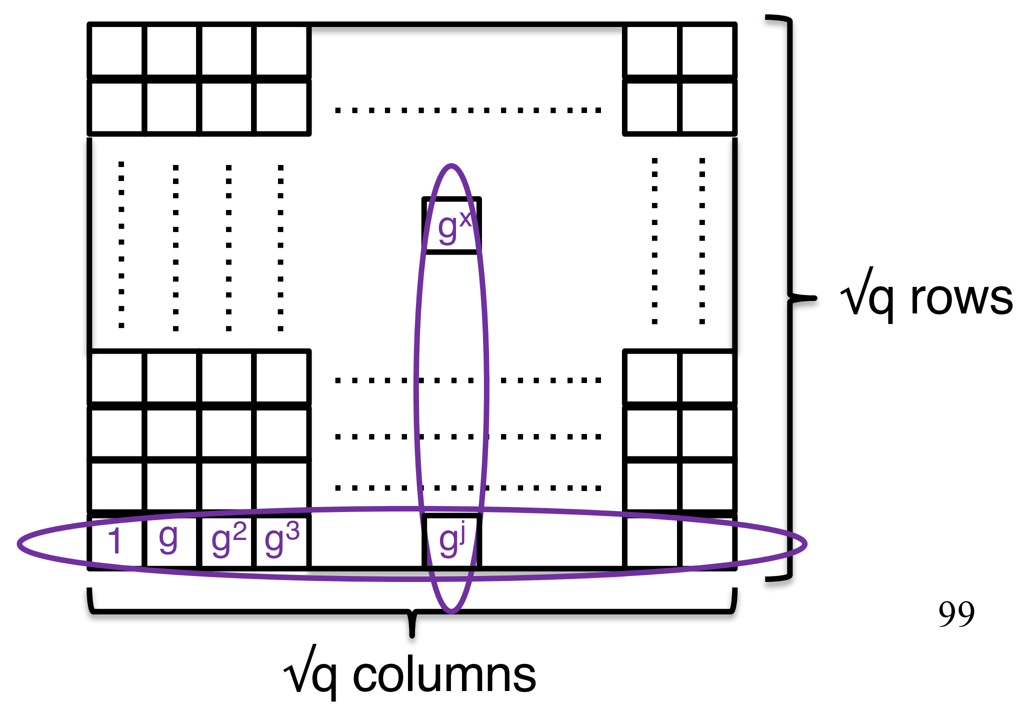


A Black-Box Discrete Log Algorithm

Fact: if the group G has order q, can solve the discrete log problem for G with $O(\sqrt{q})$ group operations.

Idea of algorithm: [given generator g and g^x, need to recover x]

- step 1: compute $g^2, g^3, g^4, \ldots, g^{\sqrt{q}}$ [i.e., all entries in bottom row]
- step 2: compute $g^{x-\sqrt{q}}$, $g^{x-2\sqrt{q}}$, $g^{x-3\sqrt{q}}, \ldots$ i times until see repeat value g^j from step 1 [i.e., go down from g^x until you hit bottom row]
  - ➔ x must be $(i\sqrt{q}) + j$



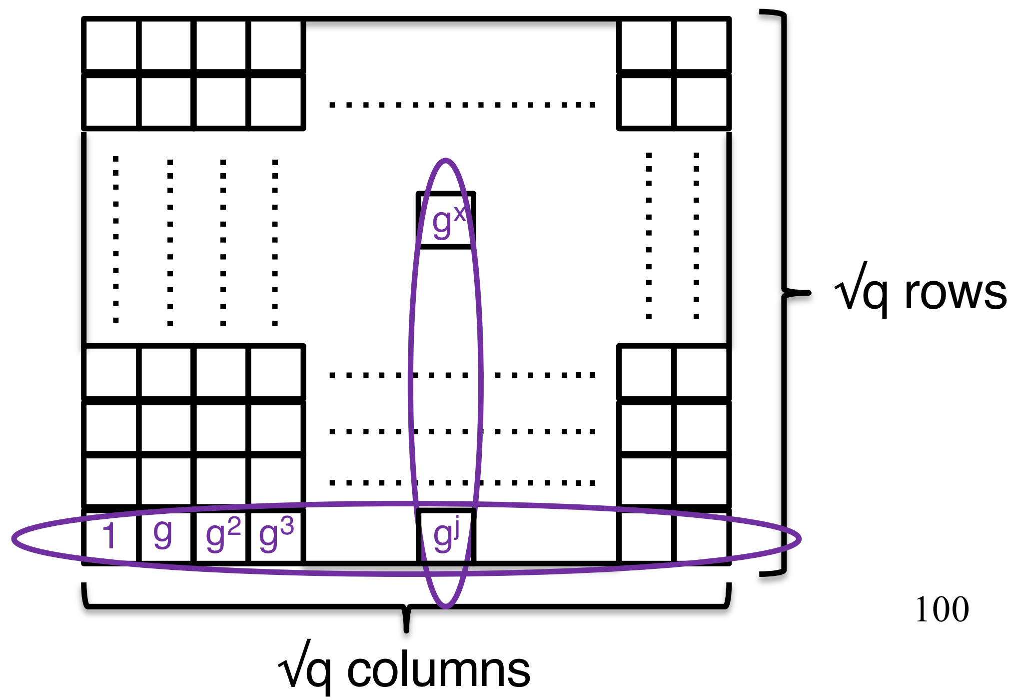
A Black-Box Discrete Log Algorithm

Fact: if the group G has order q, can solve the discrete log problem for G with $O(\sqrt{q})$ group operations.

Idea of algorithm: [given generator g and g^x, need to recover x]

- step 1: compute $g^2, g^3, g^4, \ldots, g^{\sqrt{q}}$ [i.e., all entries in bottom row]
- step 2: compute $g^{x-\sqrt{q}}$, $g^{x-2\sqrt{q}}$, $g^{x-3\sqrt{q}}, \ldots$ i times until see repeat value g^j from step 1 [i.e., go down from g^x until you hit bottom row]
  - ➔ x must be $(i\sqrt{q}) + j$

➔ uses at most $2\sqrt{q}$ group operations!



How Hard Is the DL Problem? (con’d)

Fact: “black-box/generic” algorithms cannot solve the discrete log problem with $o(\sqrt{q})$ group operations [where q = order of group].

- i.e., to do better, must exploit the structure of the group

How Hard Is the DL Problem? (con'd)

Fact: "black-box/generic" algorithms cannot solve the discrete log problem with $o(\sqrt{q})$ group operations [where q = order of group].

- i.e., to do better, must exploit the structure of the group

Fact: the discrete log problem in Z_p^* can be solved with $\approx \exp\{1.92 \times (\ln p)^{1/3} \times (\ln \ln p)^{2/3}\}$ group operations [where q is the order of group]. (via the "general number field sieve (GNFS)")

How Hard Is the DL Problem? (con’d)

Fact: “black-box/generic” algorithms cannot solve the discrete log problem with $o(\sqrt{q})$ group operations [where q = order of group].

- i.e., to do better, must exploit the structure of the group

Fact: the discrete log problem in Z_p^* can be solved with $\approx \exp\{1.92 \times (\ln p)^{1/3} \times (\ln \ln p)^{2/3}\}$ group operations [where q is the order of group]. (via the “general number field sieve (GNFS)”)

Consequence: with this group, need key size ≥ 3072 bits to get 128 bits of security.

How Hard Is the DL Problem? (con'd)

Fact: "black-box/generic" algorithms cannot solve the discrete log problem with $o(\sqrt{q})$ group operations [where q = order of group].

- i.e., to do better, must exploit the structure of the group

Fact: the discrete log problem in Z_p^* can be solved with $\approx \exp\{1.92 \times (\ln p)^{1/3} \times (\ln \ln p)^{2/3}\}$ group operations [where q is the order of group]. (via the "general number field sieve (GNFS)")

Consequence: with this group, need key size ≥ 3072 bits to get 128 bits of security. [similar conclusion for RSA signatures]

How Hard Is the DL Problem? (con’d)

Fact: if the group G has order q, can solve the discrete log problem for G on a *quantum* computer with $O((\log q)^3)$ group operations. [Shor’s algorithm, 1994]

How Hard Is the DL Problem? (con'd)

Fact: if the group G has order q, can solve the discrete log problem for G on a *quantum* computer with $O((\log q)^3)$ group operations. [Shor's algorithm, 1994]

Consequence: the DL approach to signatures is fundamentally broken if reasonably large quantum computers are available.

How Hard Is the DL Problem? (con'd)

Fact: if the group G has order q, can solve the discrete log problem for G on a *quantum* computer with $O((\log q)^3)$ group operations. [Shor's algorithm, 1994]

Consequence: the DL approach to signatures is fundamentally broken if reasonably large quantum computers are available.

- all blockchain protocols will likely need to upgrade to post-quantum-secure signature schemes in the next decade or two
  - arguably, less urgent than for e.g. encryption of sensitive data
  - likely to cause a non-trivial performance hit