Verifiable Delay Functions

– Archisman Dutta

Verifiable Delay Functions

(or how to harness the power of time)

– Archisman Dutta

Plan for the afternoon

→ Timed-release crypto

→ Time-lock puzzles

→ Verifiable Delay Functions

→ Pietrzak's construction

Plan for the afternoon

→ Timed-release crypto

→ Time-lock puzzles

→ Verifiable Delay Functions

→ Pietrzak's construction

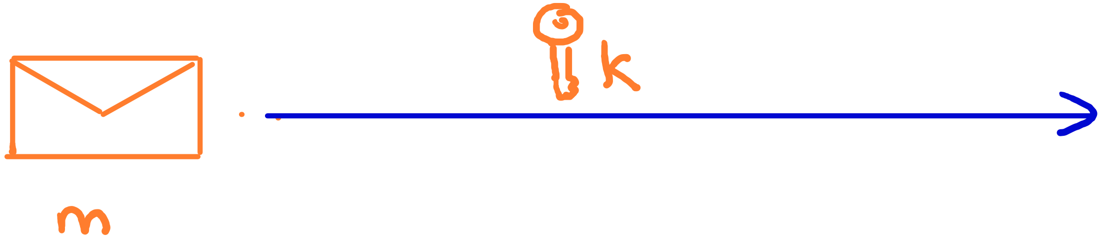
Timed-release crypto

"Send information into the future"

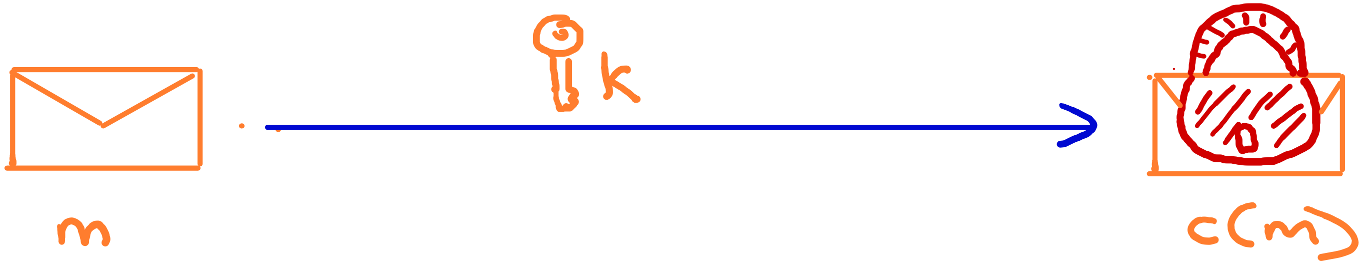
(sort of an inverse of Steins;Gate)

Timed-release crypto

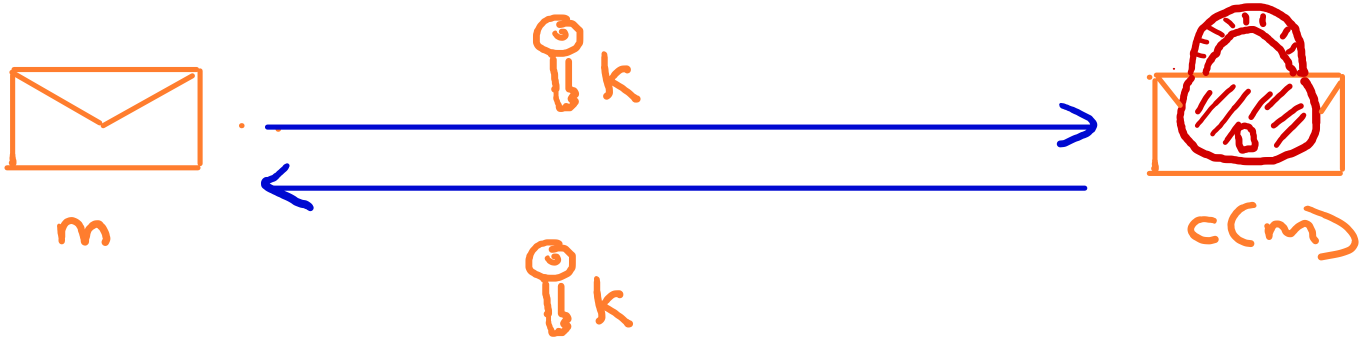
Timed-release crypto



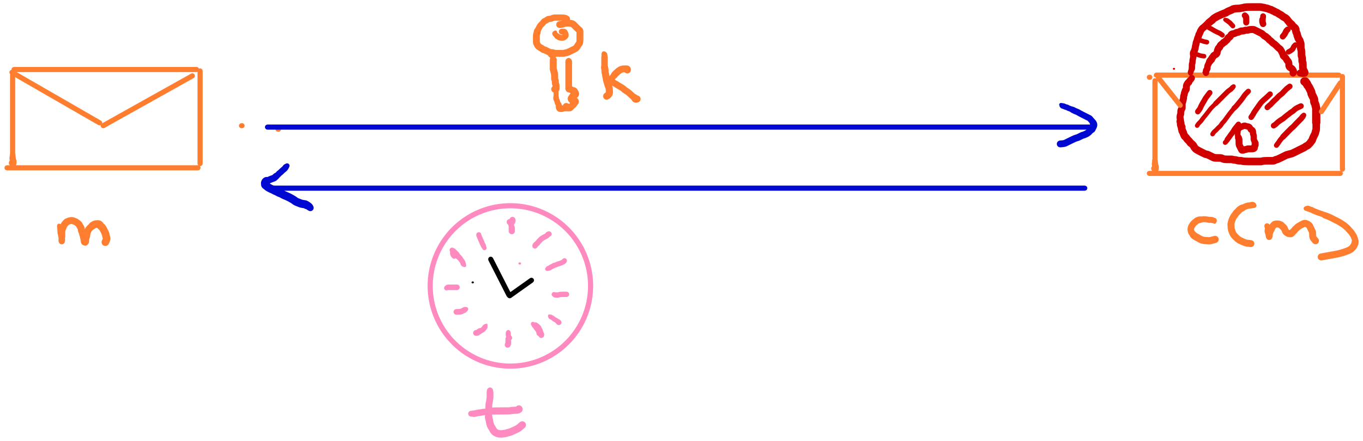
Timed-release crypto



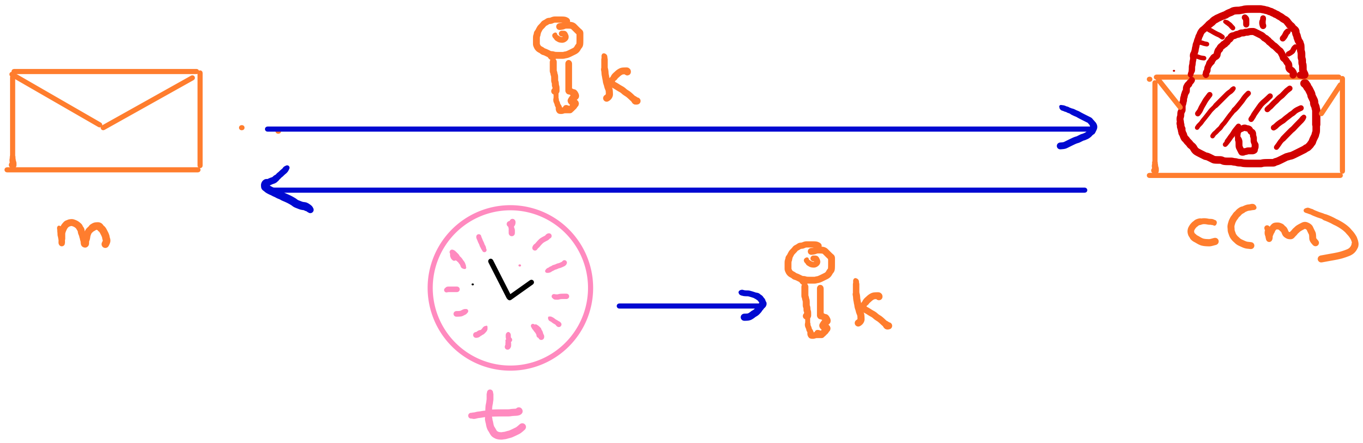
Timed-release crypto



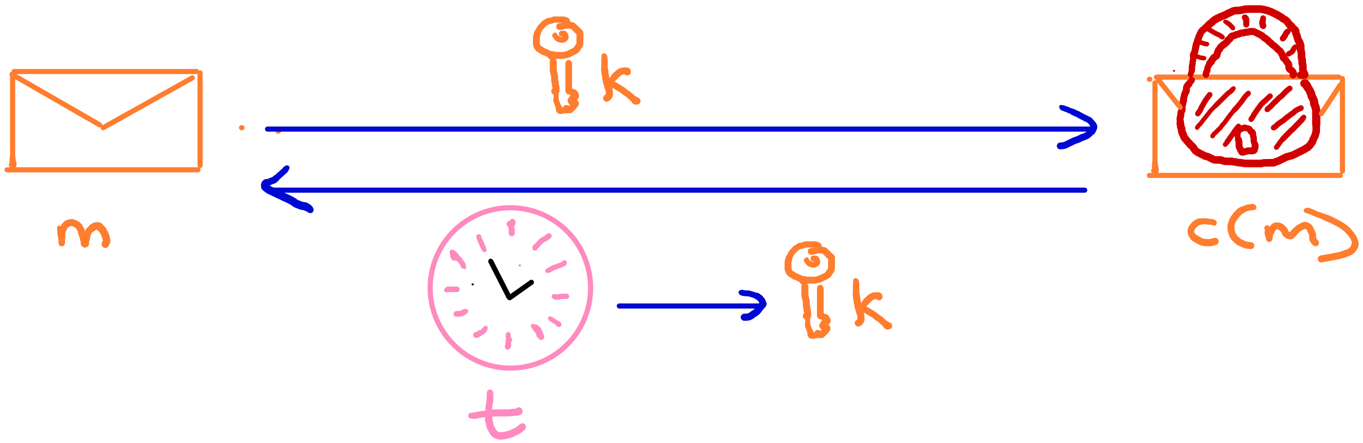
Timed-release crypto



Timed-release crypto

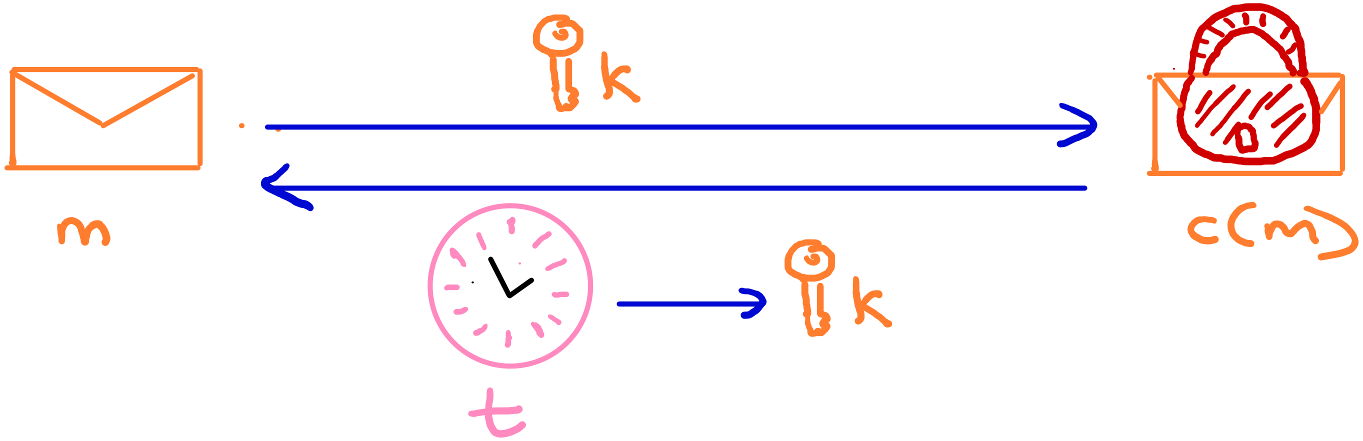


Timed-release crypto



c(m) cannot be decrypted without k

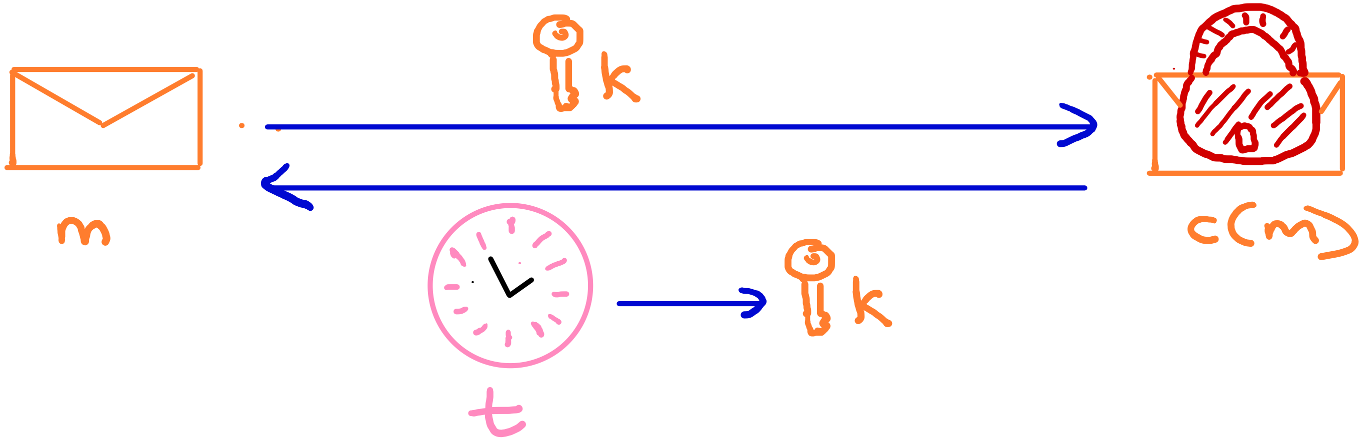
Timed-release crypto



c(m) cannot be decrypted without k

but k is only obtained after time t

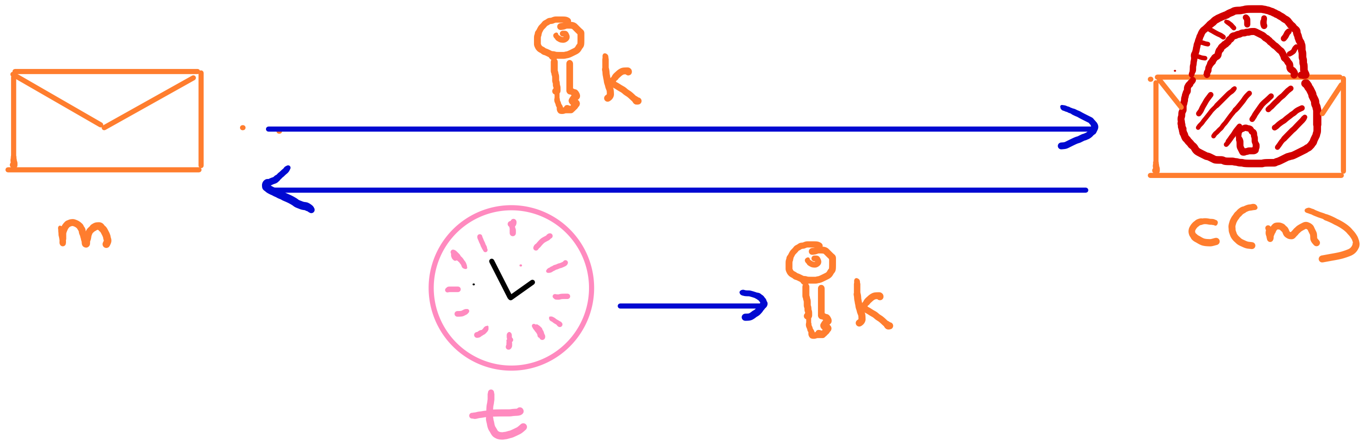
Timed-release crypto



c(m) cannot be decrypted without k

but k is only obtained after time t*

Timed-release crypto

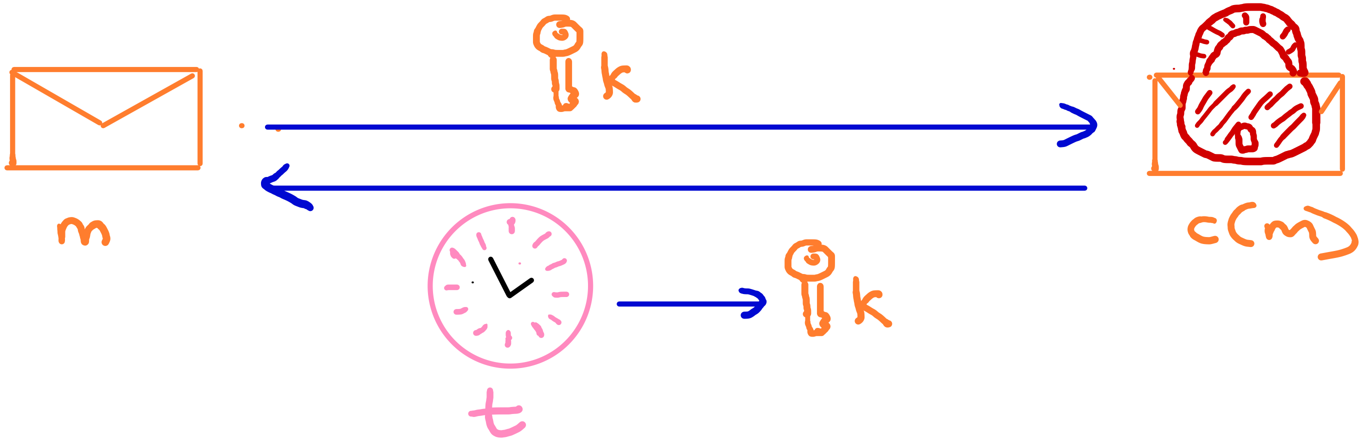


c(m) cannot be decrypted without k

but k is only obtained after time t*

* t → wall-clock time (not CPU time)

Timed-release crypto



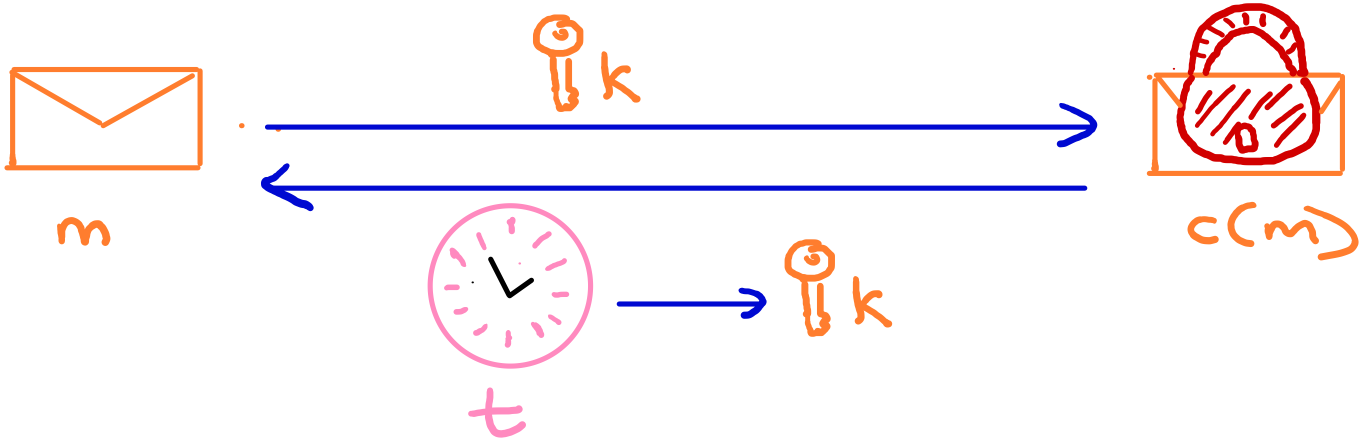
c(m) cannot be decrypted without k

but k is only obtained after time t*

* t → wall-clock time (not CPU time)

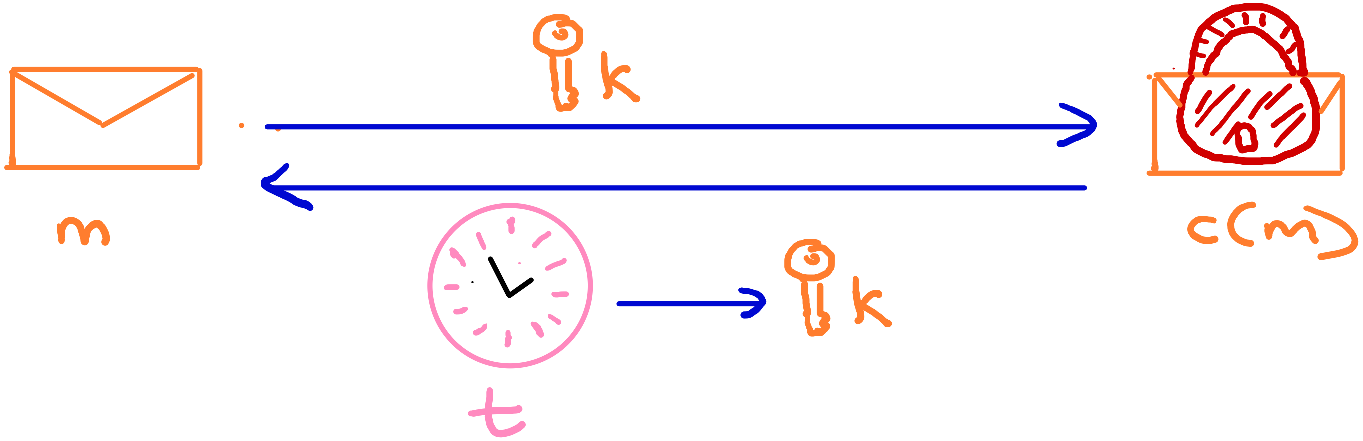
wall-clock time < CPU time for parallel processes

Timed-release crypto

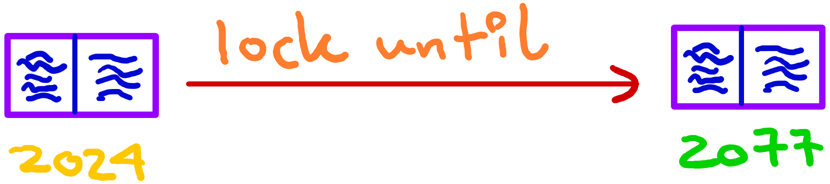


Applications?

Timed-release crypto

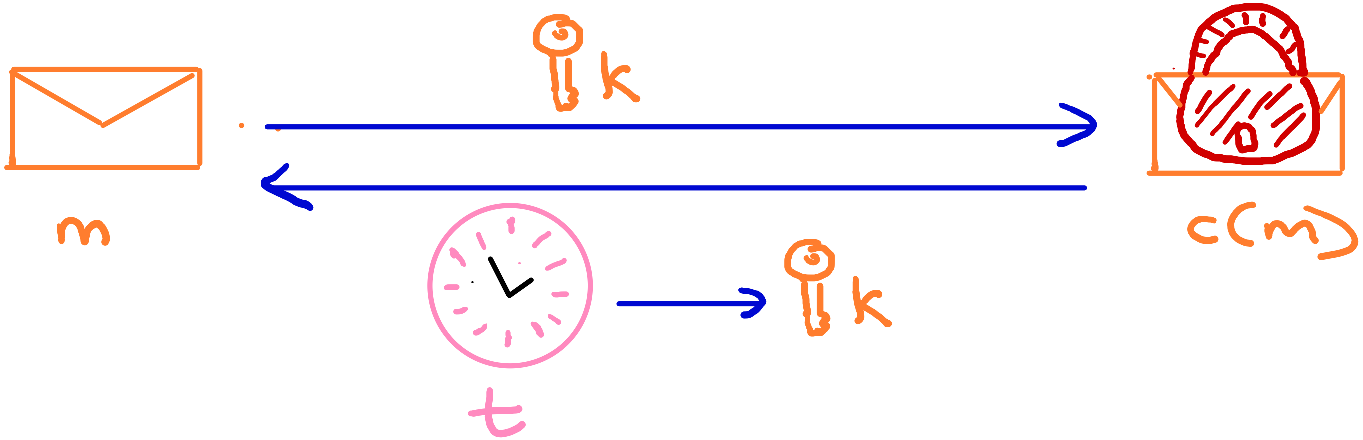


Applications?

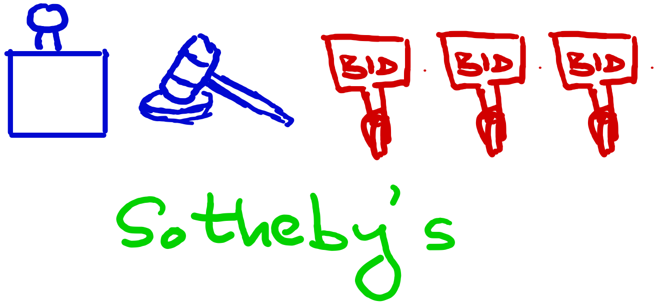


"Dear Diary..."

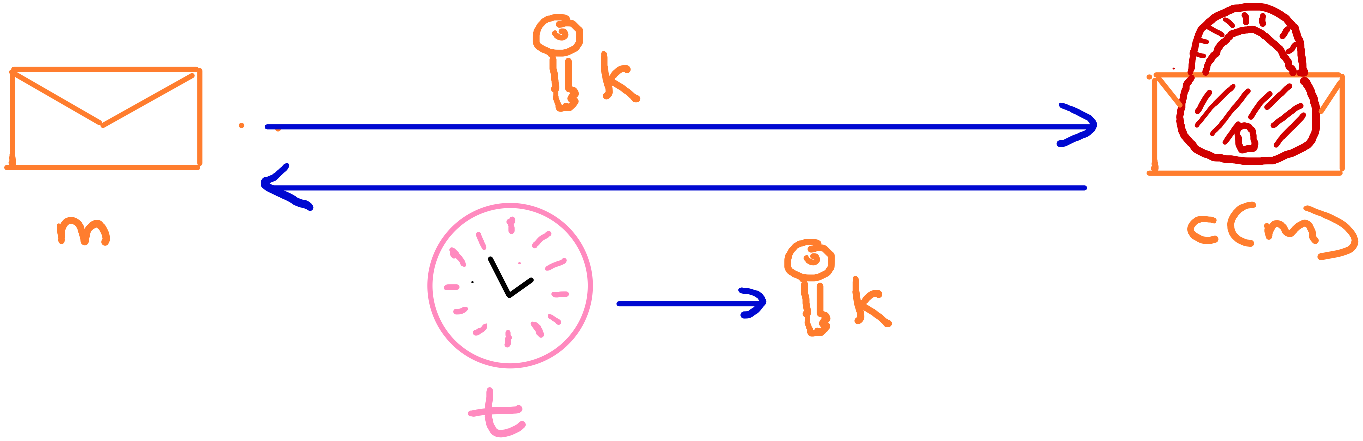
Timed-release crypto



Applications?



Timed-release crypto

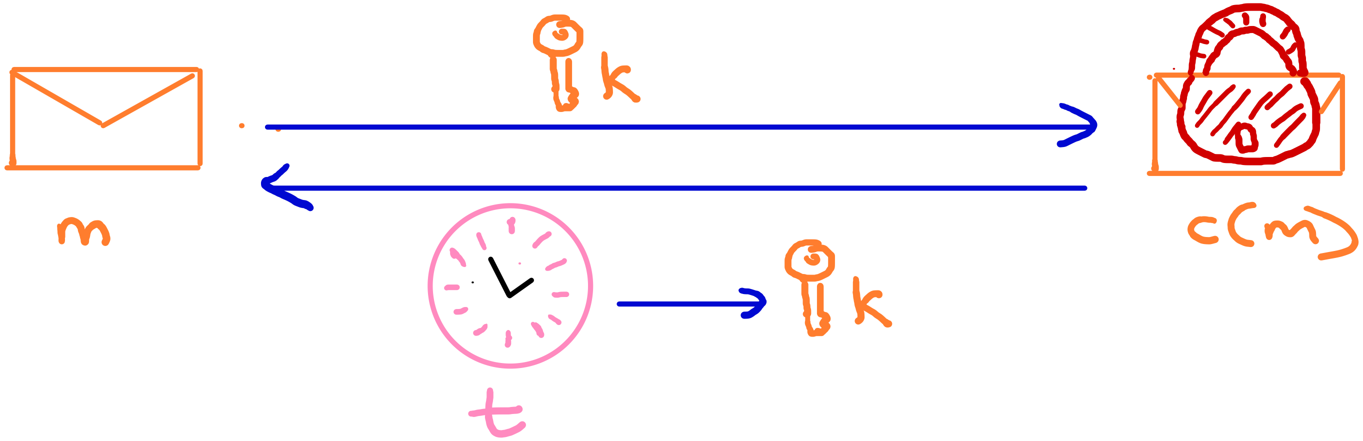


Applications?

* But only after a year....

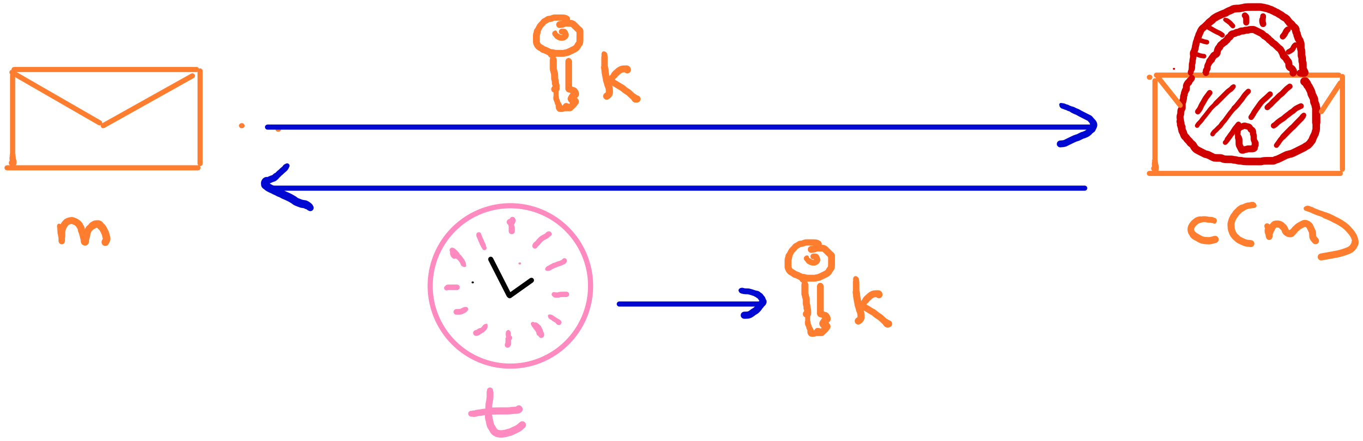
No-Spying Agency

Timed-release crypto

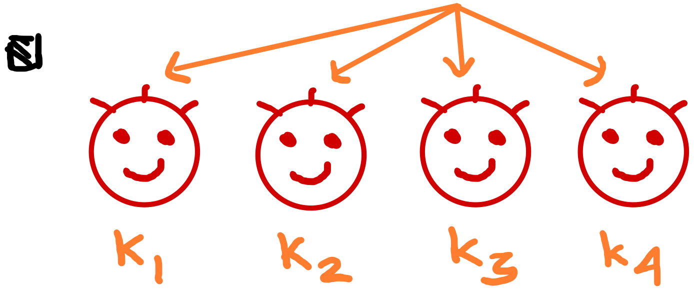


How to implement?

Timed-release crypto

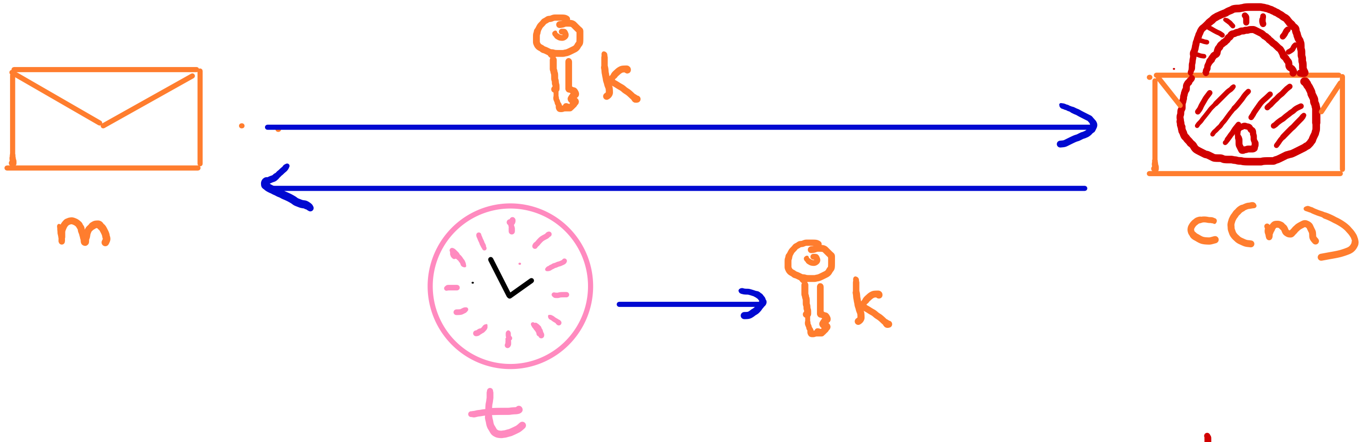


How to implement?

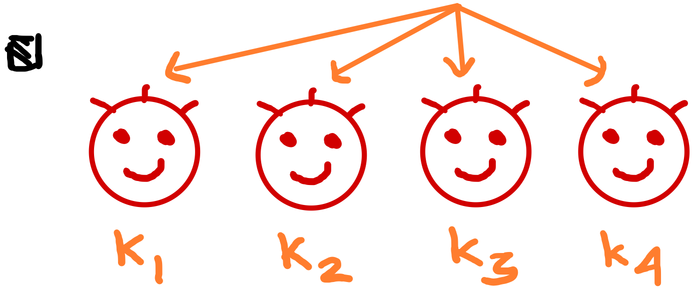


$k = (k_1, k_2, k_3, k_4)$

Timed-release crypto



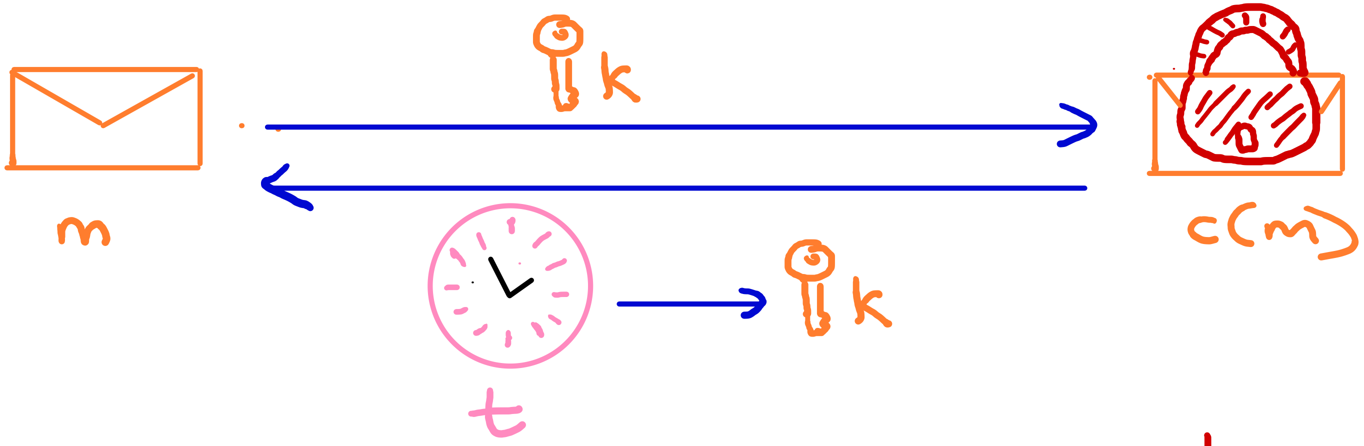
How to implement?



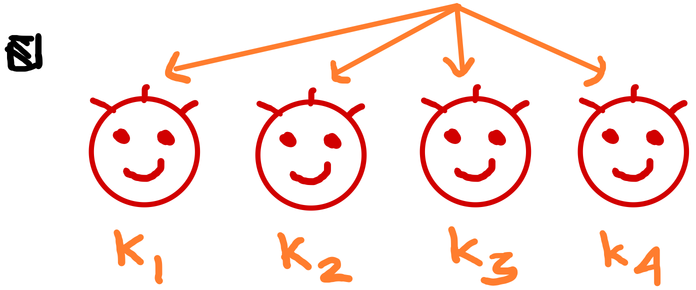
cannot reconstruct k without knowing $k_i, i=1\dots4$

 $k=(k_1,k_2,k_3,k_4)$

Timed-release crypto



How to implement?



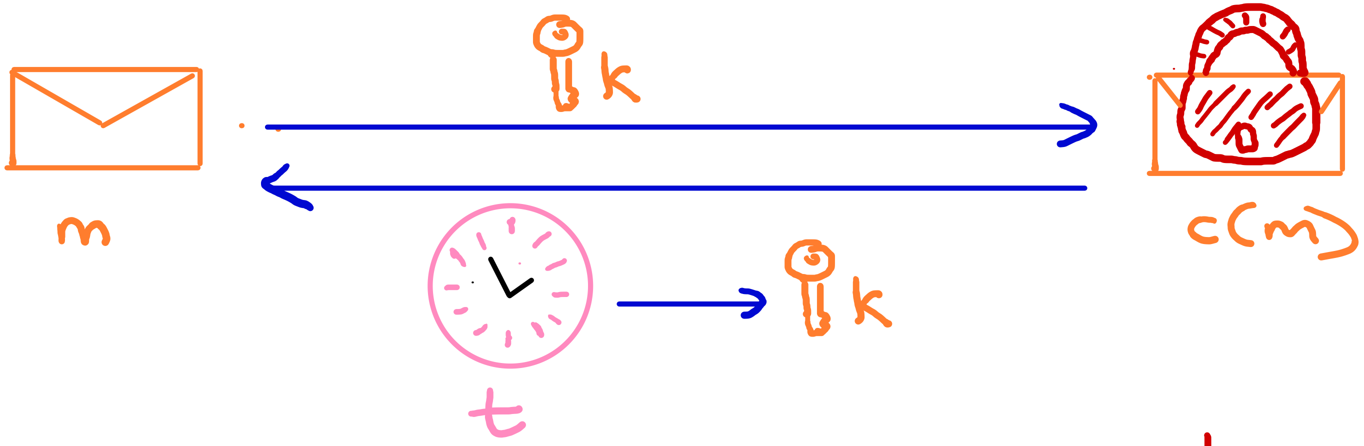
cannot reconstruct k without knowing k_i, $i=1\ldots4$

$k = (k_1, k_2, k_3, k_4)$

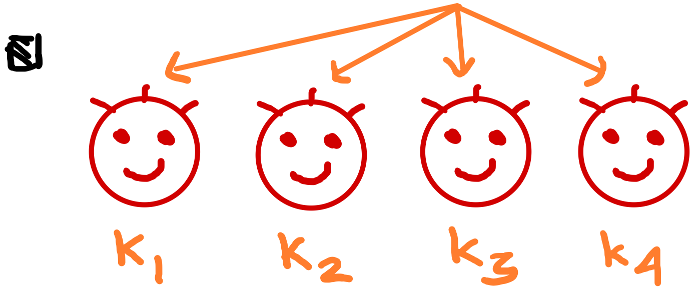
but...

collusion / death / disappearance!

Timed-release crypto



How to implement?



cannot reconstruct k without knowing $k_i, i = 1 \dots 4$

$k = (k_1, k_2, k_3, k_4)$

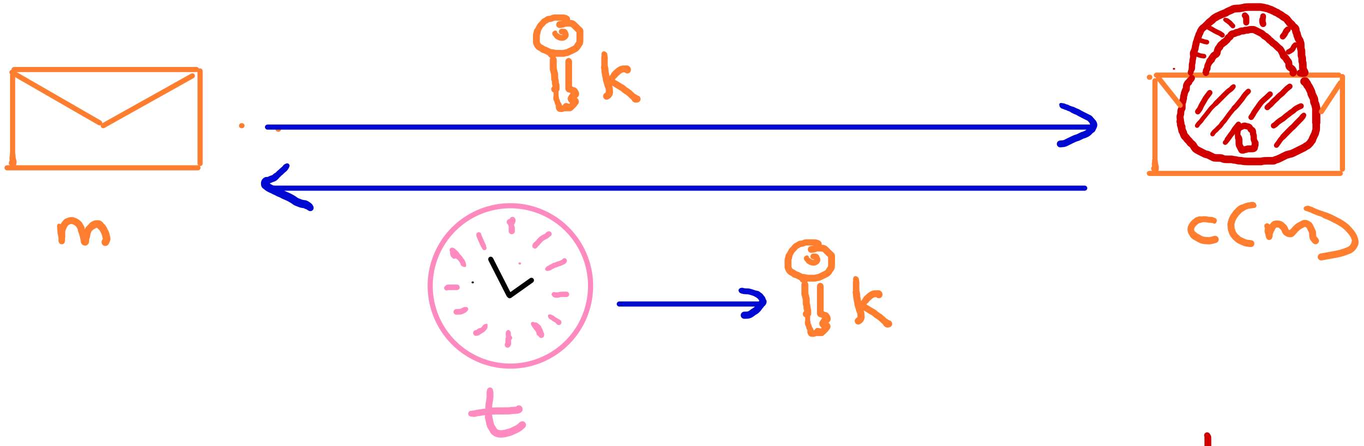
but...

collusion/death/disappearance!

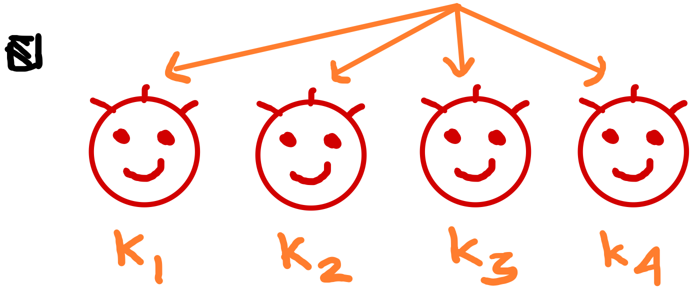
use Shamir's secret sharing!

[S'79]

Timed-release crypto



How to implement?



cannot reconstruct k without knowing $k_i, i=1\dots4$

$k=(k_1,k_2,k_3,k_4)$

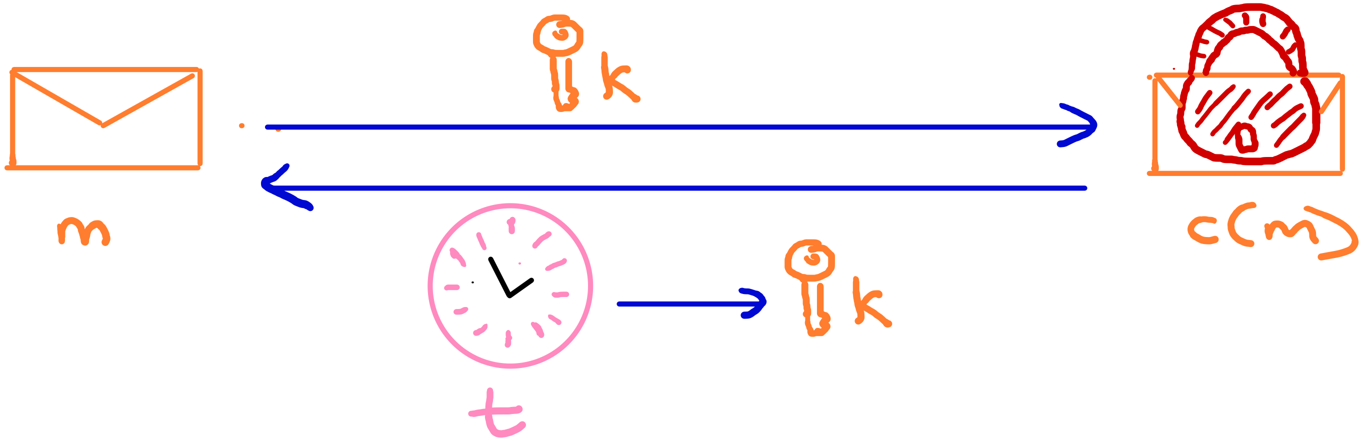
but... still need to rely/trust!

collusion/death/disappearance!

use Shamir's secret sharing!

[S'79]

Timed-release crypto



How to implement?

- Use time-lock puzzles! [RSW'96]

Plan for the afternoon

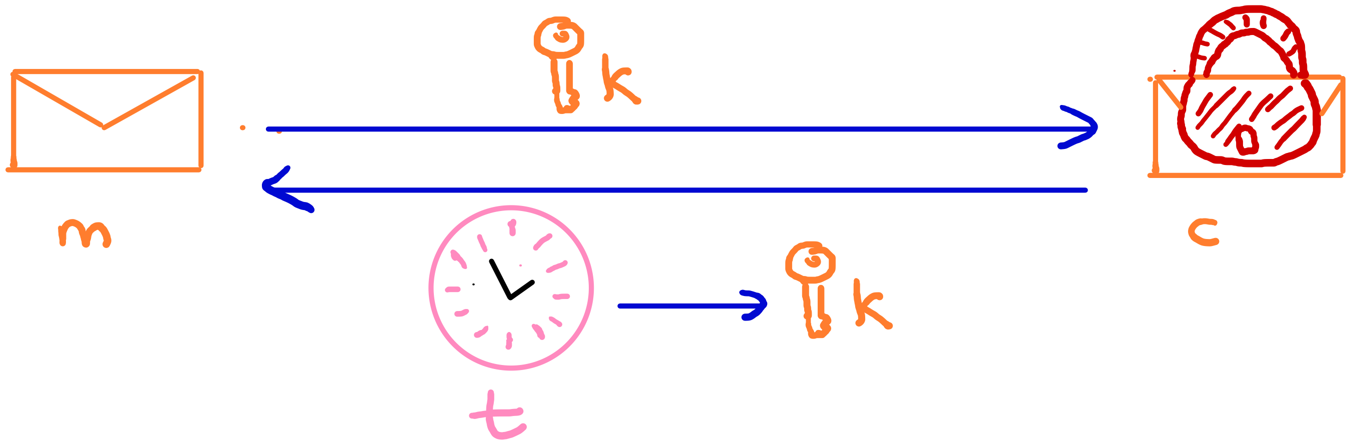
→ Timed-release crypto

→ Time-lock puzzles

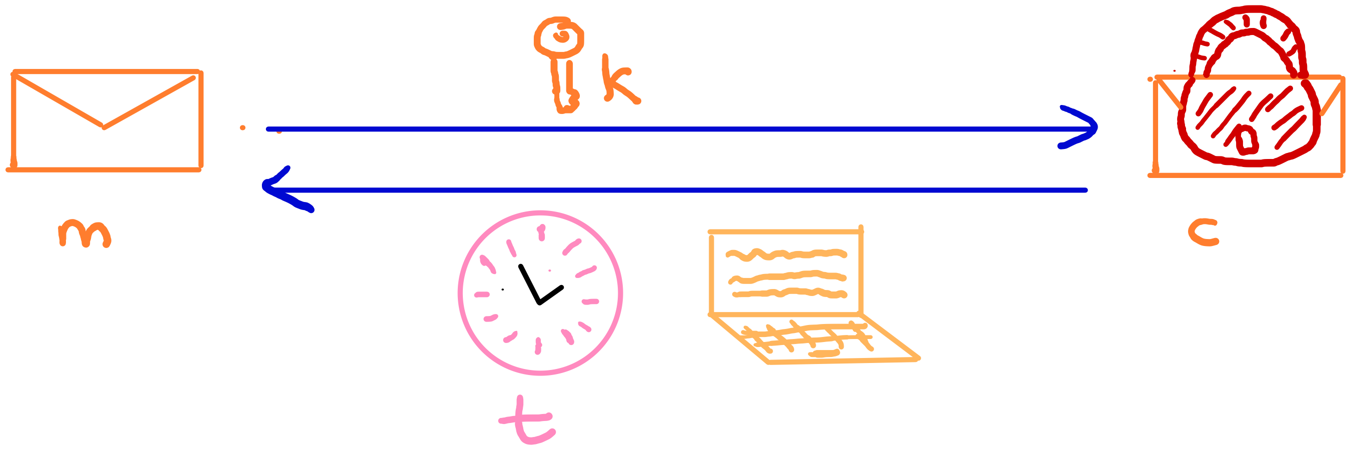
→ Verifiable Delay Functions

→ Pietrzak's construction

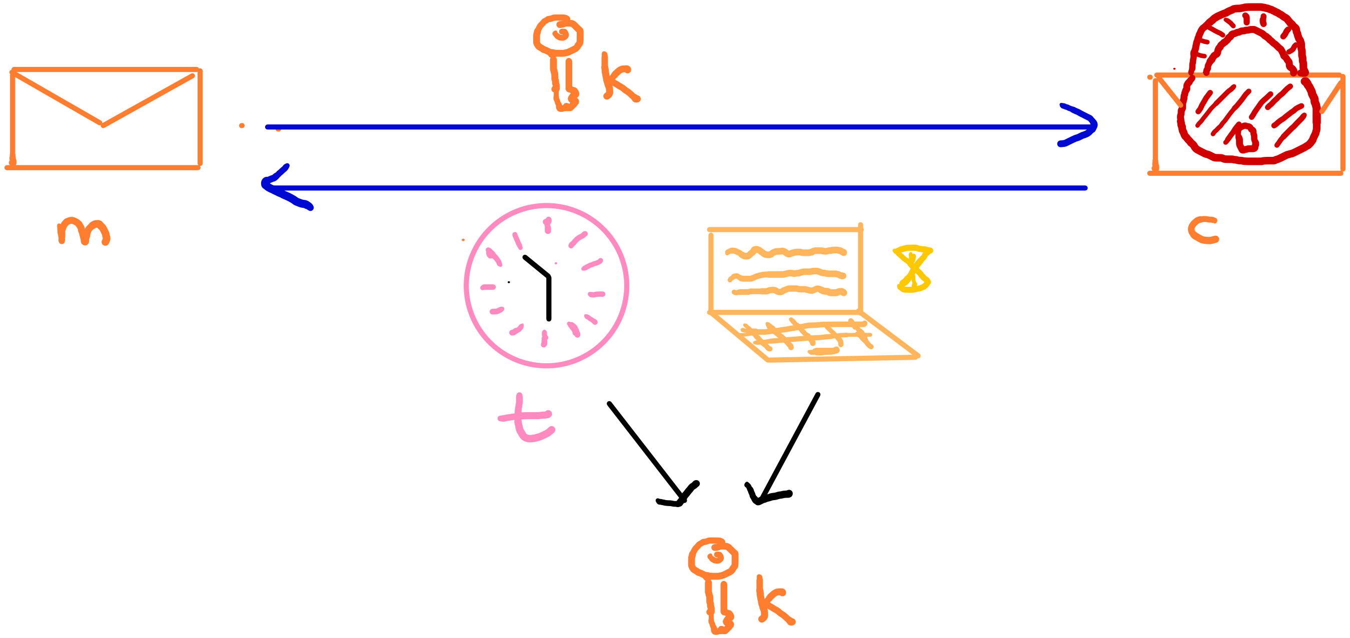
Time-lock Puzzle



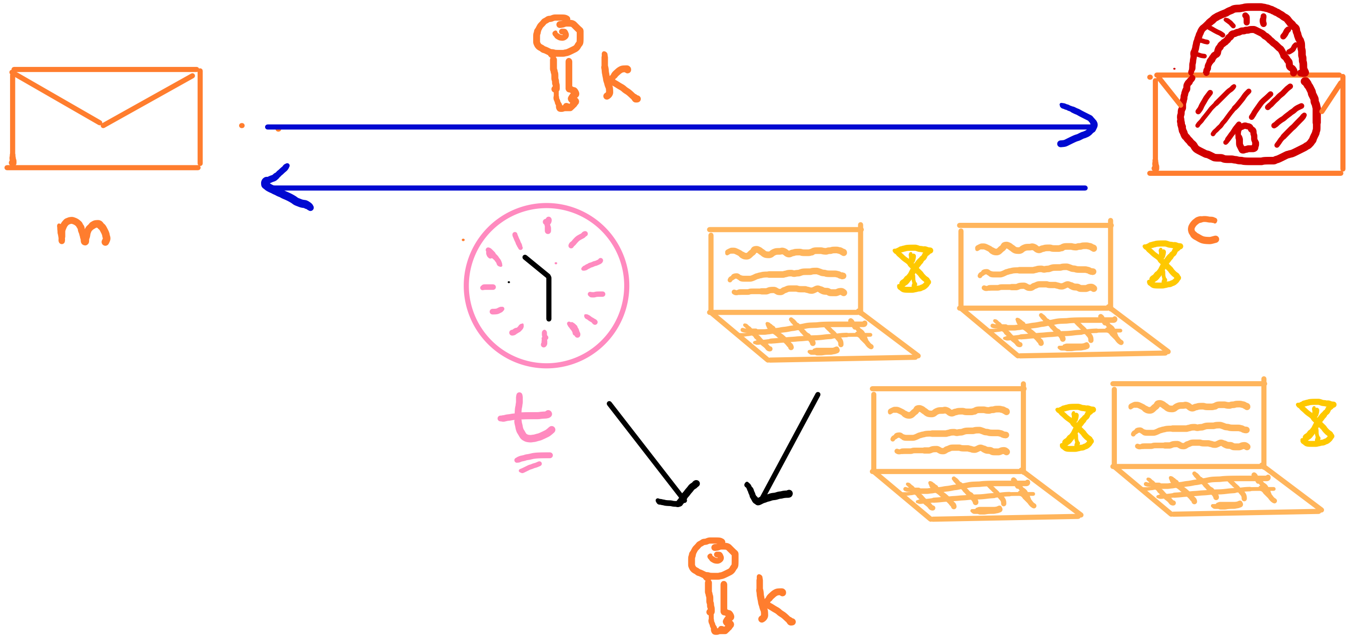
Time-lock Puzzle



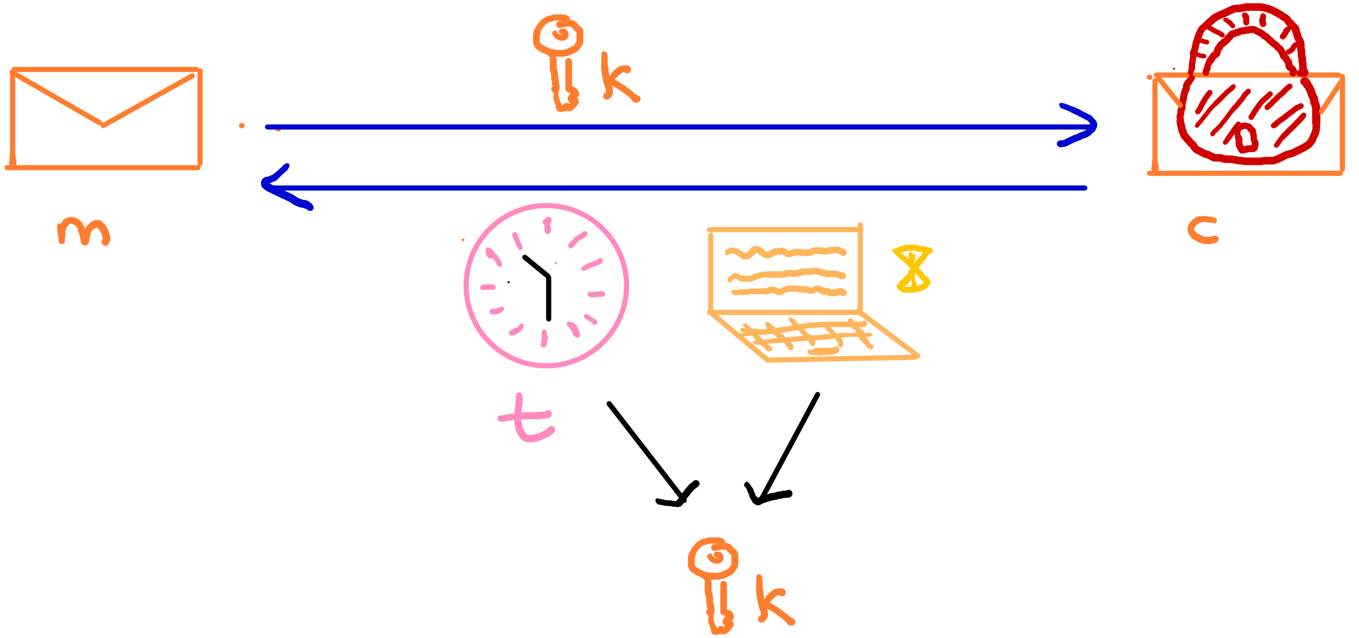
Time-lock Puzzle



Time-lock Puzzle



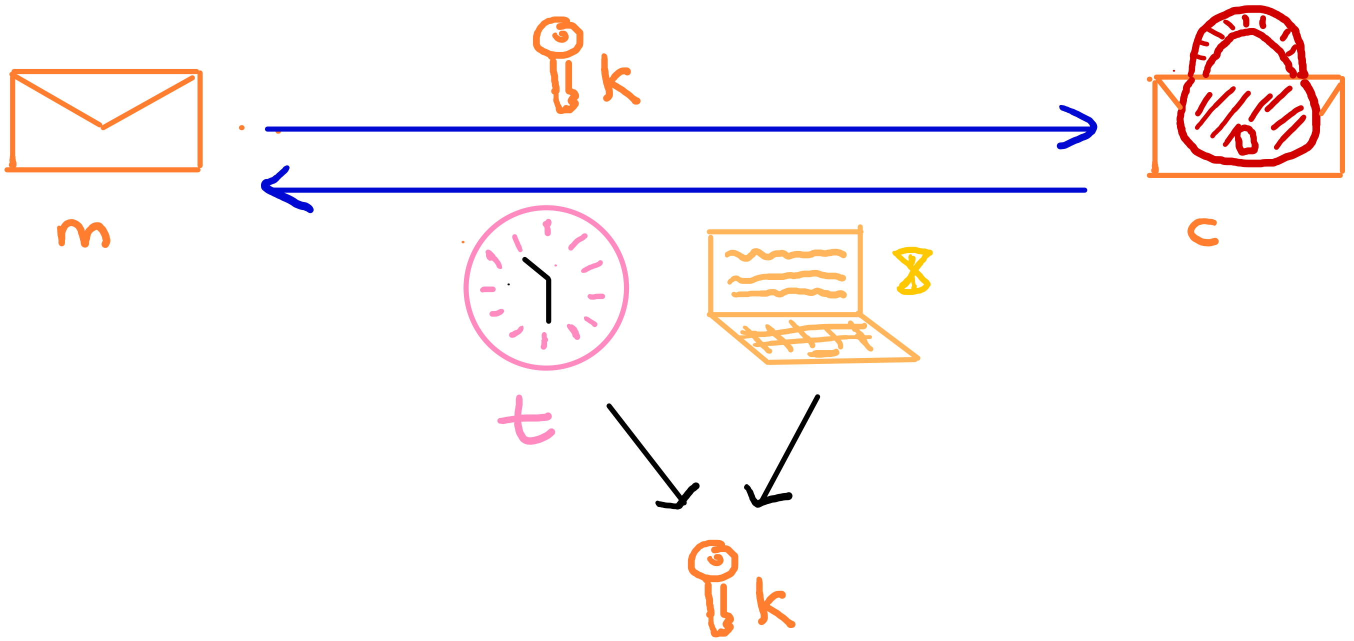
Time-lock Puzzle



Solutions?

- just AES encrypt with $|k| = \log(2st)$ bits,
  $s \rightarrow$ no. of decryptions/s

Time-lock Puzzle

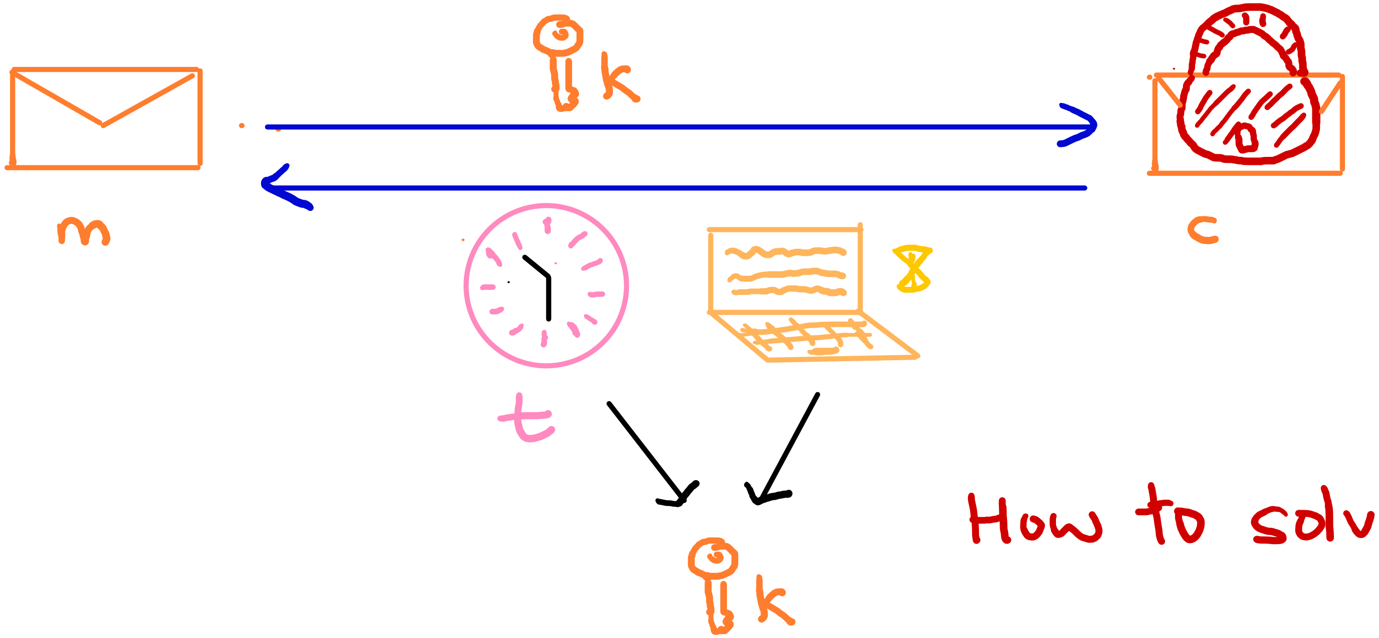


Solutions?

☒ just AES encrypt with $|k| = \log(2st)$ bits,
s → no. of decryptions/s

and 

Time-lock Puzzle



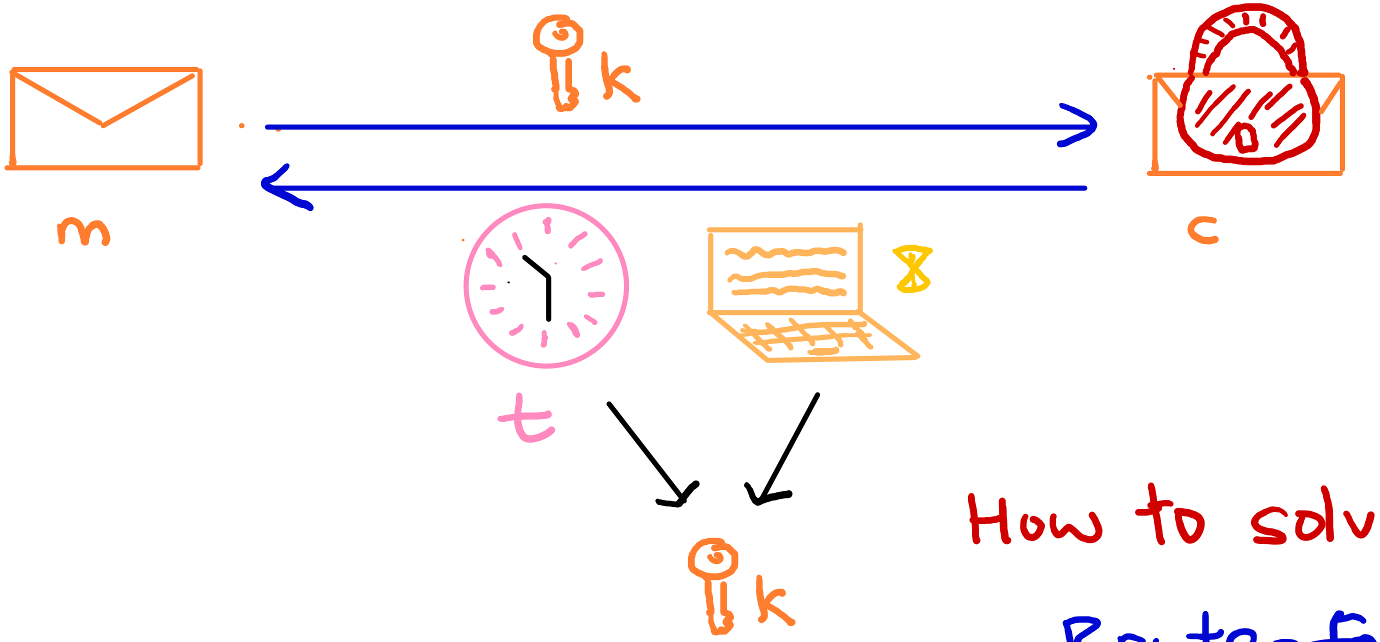
How to solve for 🔑 k?

Solutions?

- just AES encrypt with $|k| = \log(2st)$ bits,
  $s \rightarrow$ no. of decryptions/s

  and 🔑 $k \xrightarrow{\text{discard}}$ 🗑

Time-lock Puzzle



How to solve for 🔑 k?

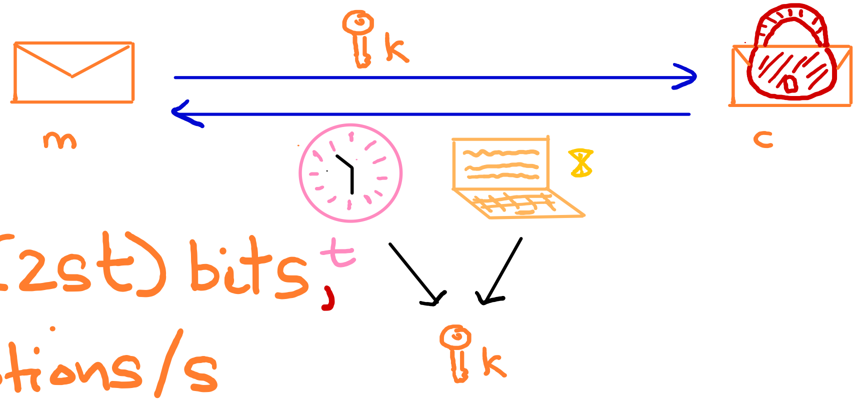
Brute-force!

Solutions?

- just AES encrypt with $|k| = \log(2st)$ bits,
  s → no. of decryptions/s

  and 🔑 k $\xrightarrow{\text{discard}}$ 🗑

Time-lock Puzzle



Solutions?

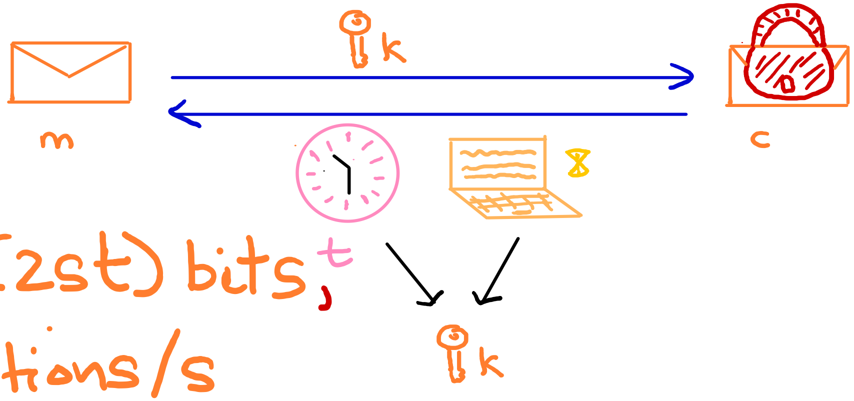
- ☒ just AES encrypt with $|k| = \log(2st)$ bits,

  $s \rightarrow$ no. of decryptions/s

  and k $\xrightarrow{\text{discard}}$  How to solve for k?

  Brute-force!

Time-lock Puzzle



Solutions?

- ☒ just AES encrypt with $|k| = \log(2st)$ bits,

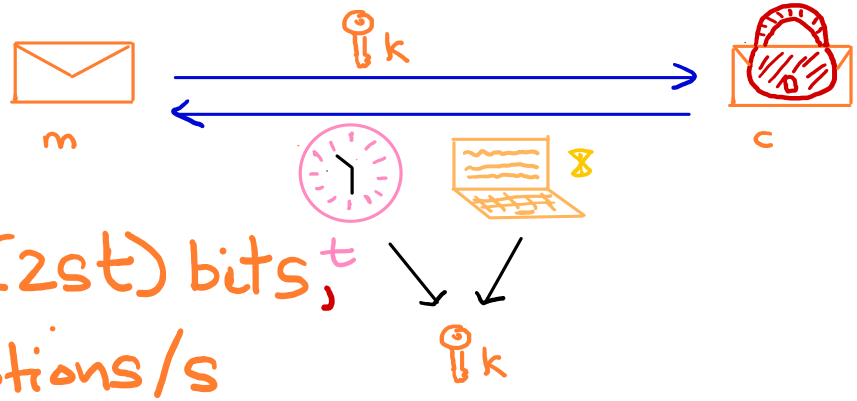
  $s \rightarrow$ no. of decryptions/s

  and k $\xrightarrow{\text{discard}}$  How to solve for k?

  Brute-force!

Problems:

Time-lock Puzzle



Solutions?

- ☒ just AES encrypt with $|k| = \log(2st)$ bits, t

  $s \rightarrow$ no. of decryptions/s

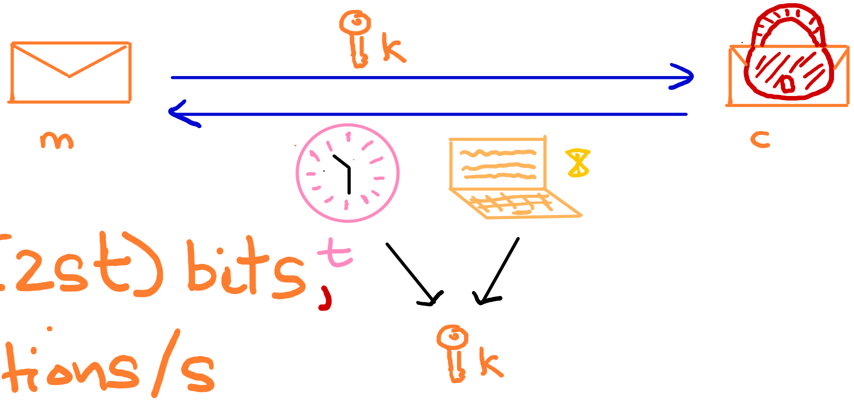
  and 🔑 k $\xrightarrow{\text{discard}}$ 🗑 How to solve for 🔑 k?

  Brute-force!

problems:

- ☒ 💻⏳ 💻⏳ 💻⏳ 💻⏳ can find 🔑 k in less than t

Time-lock Puzzle



Solutions?

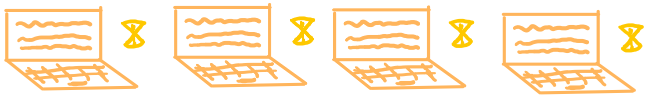
- just AES encrypt with $|k| = \log(2st)$ bits, t

  s → no. of decryptions/s

  and 🔑 k $\xrightarrow{\text{discard}}$ 🗑 How to solve for 🔑 k?

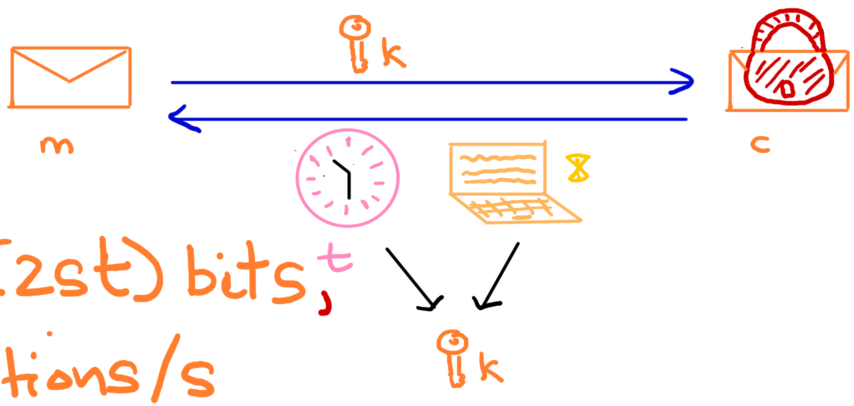
  Brute-force!

problems:

-  can find 🔑 k in less than t

needs to be sequential!

Time-lock Puzzle



Solutions?

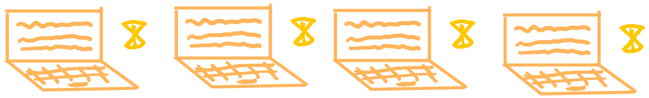
☒ just AES encrypt with $|k| = \log(2st)$ bits,

$s \rightarrow$ no. of decryptions/s

and 🔑k $\xrightarrow{\text{discard}}$ 🗑 How to solve for 🔑k?

Brute-force!

problems:

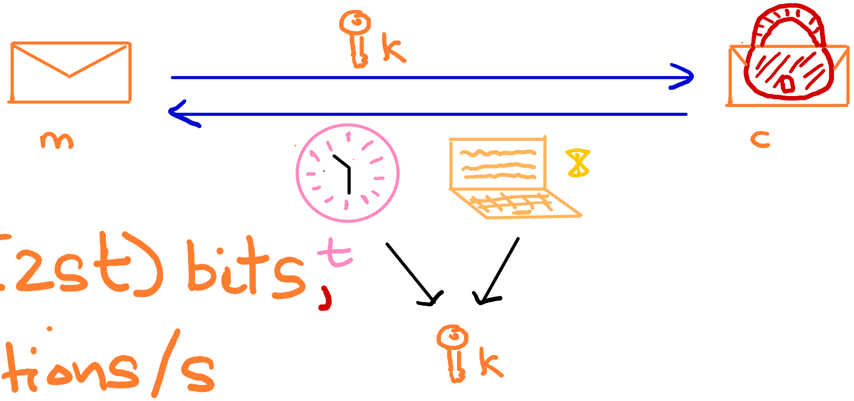
☒  can find 🔑k in less than t

needs to be sequential!

i.e. 🔑k cannot be found in, say,

$t' = t^c, c < 1$

Time-lock Puzzle



Solutions?

- ☒ just AES encrypt with $|k| = \log(2st)$ bits, t

  $s \rightarrow$ no. of decryptions/s

  and 🔑k $\xrightarrow{\text{discard}}$ 🗑 How to solve for 🔑k?

  Brute-force!

problems:

- ☒ 💻⏳ 💻⏳ 💻⏳ 💻⏳ can find 🔑k in less than t

  needs to be sequential!

  i.e. 🔑k cannot be found in, say,

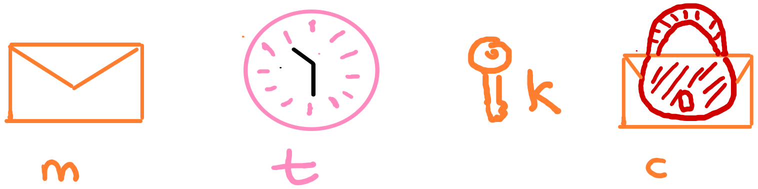
  $t' = t^c, c < 1$ (allow minor variations)

Time-lock Puzzle

→ RSW construction:

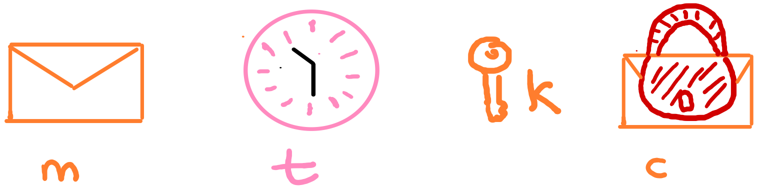
Time-lock Puzzle

→ **RSW construction:**



Time-lock Puzzle

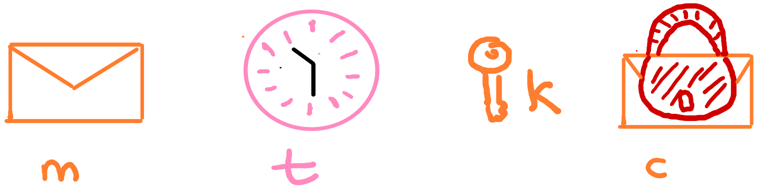
→ **RSW construction:**



- Generate $n = pq$

Time-lock Puzzle

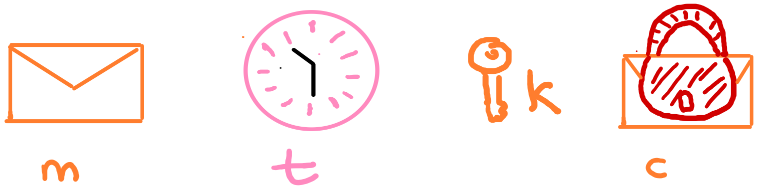
→ RSW construction:



- Generate $n = pq$ (for two large random secret primes p, q)

Time-lock Puzzle

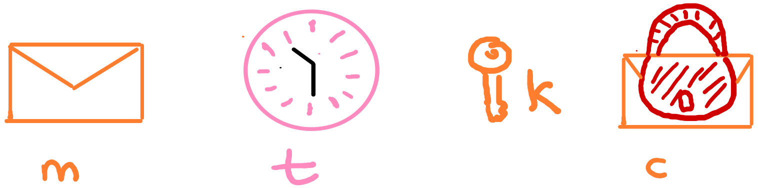
→ RSW construction:



- Generate $n = pq$ (for two large random secret primes p, q)
- Compute $\phi(n) = (p-1)(q-1)$

Time-lock Puzzle

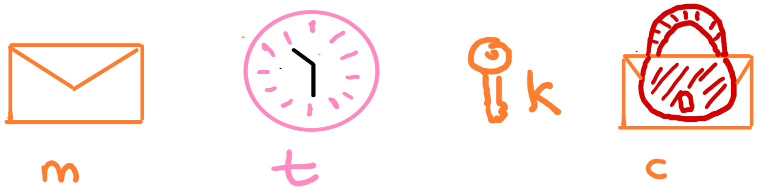
→ RSW construction:



- Generate $n=pq$ (for two large random secret primes p,q)
- Compute $\phi(n)=(p-1)(q-1)$
- Pick random a $(1<a<n)$

Time-lock Puzzle

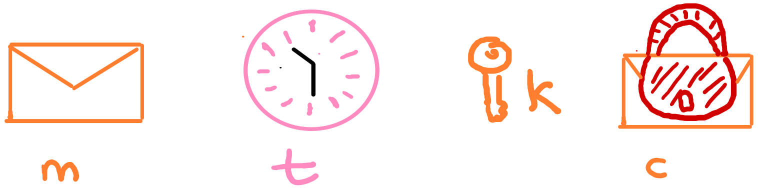
→ RSW construction:



- Generate $n = pq$ (for two large random secret primes p, q)
- Compute $\phi(n) = (p-1)(q-1)$
- Pick random a $(1 < a < n)$
- Compute $c_k = k + a^{2^t} \pmod{n}$

Time-lock Puzzle

→ **RSW construction:**

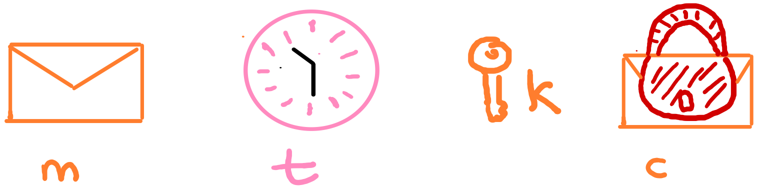


- Generate $n = pq$ (for two large random secret primes p, q)
- Compute $\phi(n) = (p-1)(q-1)$
- Pick random a $(1 < a < n)$
- Compute $c_k = k + a^{2^t} \pmod n$

How?

Time-lock Puzzle

→ RSW construction:

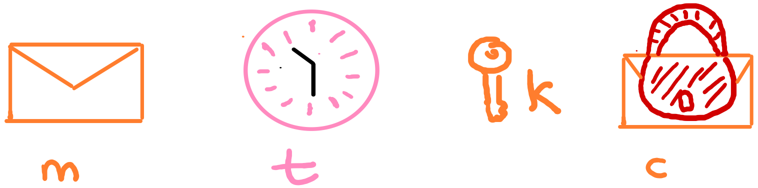


- Generate $n=pq$ (for two large random secret primes p, q)
- Compute $\phi(n) = (p-1)(q-1)$
- Pick random a $(1 < a < n)$
- Compute $c_k = k + a^{2^t} \pmod{n}$

How? compute $e = 2^T \pmod{\phi(n)}$

Time-lock Puzzle

→ **RSW construction:**



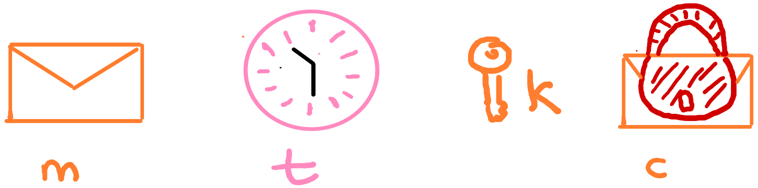
- Generate $n=pq$ (for two large random secret primes p, q)
- Compute $\phi(n)=(p-1)(q-1)$
- Pick random a $(1<a<n)$
- Compute $c_k = k + a^{2^t} \pmod n$

How? compute $e = 2^T \pmod{\phi(n)}$

$b = a^e \pmod n$

Time-lock Puzzle

→ RSW construction:



- Generate $n = pq$ (for two large random secret primes p, q)
- Compute $\phi(n) = (p-1)(q-1)$
- Pick random a $(1 < a < n)$
- Compute $c_k = k + a^{2^t} \pmod{n}$
- Output (n, a, t, c, c_k) and 🗑 anything else

Time-lock Puzzle

→ RSW construction:

Why does this work?

Time-lock Puzzle

→ RSW construction:

Why does this work?

- brute-forcing the AES key → infeasible

Time-lock Puzzle

→ RSW construction:

Why does this work?

- ☒ brute-forcing the AES key → infeasible

  fastest way to solve puzzle :

Time-lock Puzzle

→ RSW construction:

Why does this work?

- brute-forcing the AES key → infeasible

  fastest way to solve puzzle:

$$a \longrightarrow a^2 \longrightarrow a^{2^2} \longrightarrow a^{2^3} \longrightarrow \dots \longrightarrow a^{2^t} \pmod{n}$$

Time-lock Puzzle

→ RSW construction:

Why does this work?

☒ brute-forcing the AES key → infeasible

fastest way to solve puzzle:

$$\underbrace{a \longrightarrow a^2 \longrightarrow a^{2^2} \longrightarrow a^{2^3} \longrightarrow \dots \longrightarrow a^{2^t}}_{t \text{ sequential steps}} \pmod{n}$$

Time-lock Puzzle

→ RSW construction:

Why does this work?

- ☒ brute-forcing the AES key → infeasible

  fastest way to solve puzzle:

$$\underbrace{a \longrightarrow a^2 \longrightarrow a^{2^2} \longrightarrow a^{2^3} \longrightarrow \dots \longrightarrow a^{2^t}}_{t \text{ sequential steps}} \pmod{n}$$

- ☒ can easily compute if $\phi(n)$ known

Time-lock Puzzle

→ RSW construction:

Why does this work?

- brute-forcing the AES key → infeasible
  fastest way to solve puzzle:

$$\underbrace{a \longrightarrow a^2 \longrightarrow a^{2^2} \longrightarrow a^{2^3} \longrightarrow \dots \longrightarrow a^{2^t}}_{t \text{ sequential steps}} \pmod n$$

- can easily compute if $\phi(n)$ known
  but $n \longrightarrow \phi(n) \approx$ factoring n !

Time-lock Puzzle

→ RSW construction:

Why does this work?

- brute-forcing the AES key → infeasible

  fastest way to solve puzzle:

$$\underbrace{a \longrightarrow a^2 \longrightarrow a^{2^2} \longrightarrow a^{2^3} \longrightarrow \dots \longrightarrow a^{2^t}}_{t \text{ sequential steps}} \pmod{n}$$

- can easily compute if $\phi(n)$ known

  but $n \longrightarrow \phi(n) \approx$ factoring n!

  infeasible for large p, q

Time-lock Puzzle

→ **RSW construction:**

Why does this work?

- brute-forcing the AES key → infeasible

  fastest way to solve puzzle:

$$\underbrace{a \rightarrow a^2 \rightarrow a^{2^2} \rightarrow a^{2^3} \rightarrow \dots \rightarrow a^{2^t}}_{t \text{ sequential steps}} \pmod n$$

(no known algo. to parallelize)

- can easily compute if $\phi(n)$ known

  but $n \rightarrow \phi(n) \approx$ factoring n!

  infeasible for large p, q

Time-lock Puzzle

→ RSW construction:

What about verifiability of $b = a^{2^t}$?

Time-lock Puzzle

→ RSW construction:

What about verifiability of $b = a^{2^t}$?

simply check if $\mathrm{AES.Dec}(c, c_K - b \pmod{n}) = m$?

Time-lock Puzzle

→ RSW construction:

What about verifiability of $b = a^{2^t}$?

simply check if $AES.Dec(c, c_K - b \pmod n) = m$?

What about public and efficient verifiability?

Time-lock Puzzle

→ RSW construction:

What about verifiability of $b = a^{2^t}$?

simply check if $AES.Dec(c, c_K - b \pmod{n}) = m$?

What about public and efficient verifiability?

reveal $\phi(n)$ (or p, q)

Time-lock Puzzle

→ RSW construction:

What about verifiability of $b = a^{2^t}$?

simply check if $\text{AES.Dec}(c, c_k - b \pmod{n}) = m$?

What about public and efficient verifiability?

reveal $\phi(n)$ (or p, q)

↑ breaks sequentiality!

Time-lock Puzzle

→ **RSW construction:**

What about verifiability of $b = a^{2^t}$?

simply check if $\text{AES.Dec}(c, c_K - b \pmod{n}) = m$?

What about public and efficient verifiability?

reveal $\phi(n)$ (or p, q)

↖ breaks sequentiality!

can trivially compute b if $\phi(n)$ known!

Time-lock Puzzle

→ RSW construction:

What about verifiability of $b = a^{2^t}$?

simply check if $AES.Dec(c, c_K - b \pmod n) = m$?

What about public and efficient verifiability?

given (a, b, t), $b = a^{2^t}$

Time-lock Puzzle

→ RSW construction:

What about verifiability of $b = a^{2^t}$?

simply check if $\text{AES.Dec}(c, c_K - b \pmod n) = m$?

What about public and efficient verifiability?

given (a, b, t), $b = a^{2^t}$, some proof π so that

Time-lock Puzzle

→ RSW construction:

What about verifiability of $b = a^{2^t}$?

simply check if $AES.Dec(c, c_K - b \pmod{n}) = m$?

What about public and efficient verifiability?

given (a, b, t), $b = a^{2^t}$, some proof π so that

(a, b, t, π)

Time-lock Puzzle

→ **RSW construction:**

What about verifiability of $b = a^{2^t}$?

simply check if $\text{AES.Dec}(c, c_K - b \pmod{n}) = m$?

What about **public** and **efficient** verifiability?

given (a, b, t), $b = a^{2^t}$, some proof π so that

(a, b, t, π)

Time-lock Puzzle

→ RSW construction:

What about verifiability of $b = a^{2^t}$?

simply check if $\text{AES.Dec}(c, c_k - b \pmod{n}) = m$?

What about public and efficient verifiability?

given (a, b, t), $b = a^{2^t}$, some proof π so that

(a, b, t, π) ← should take time $t' <<< t$

Time-lock Puzzle

→ RSW construction:

What about verifiability of $b = a^{2^t}$?

simply check if $AES.Dec(c, c_K - b \pmod{n}) = m$?

What about public and efficient verifiability?

given (a, b, t), $b = a^{2^t}$, some proof π so that

(a, b, t, π) ← should take time $t' <<< t$

Possible?

Time-lock Puzzle

→ RSW construction:

What about verifiability of $b = a^{2^t}$?

simply check if $\text{AES.Dec}(c, c_K - b \pmod{n}) = m$?

What about public and efficient verifiability?

given (a, b, t), $b = a^{2^t}$, some proof π so that

(a, b, t, π) ← should take time $t' <<< t$

Possible? Yes, using VDFs!

Plan for the afternoon

⟶ Timed-release crypto

⟶ Time-lock puzzles

⟶ Verifiable Delay Functions

⟶ Pietrzak's construction

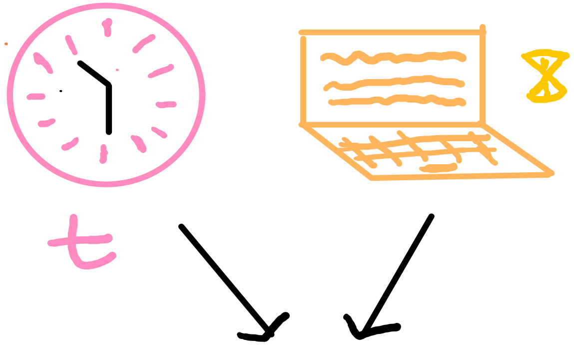
Verifiable Delay Functions [BBBF'18]

Verifiable Delay Functions

→ $f : \mathcal{X} \rightarrow \mathcal{Y}$

Verifiable Delay Functions

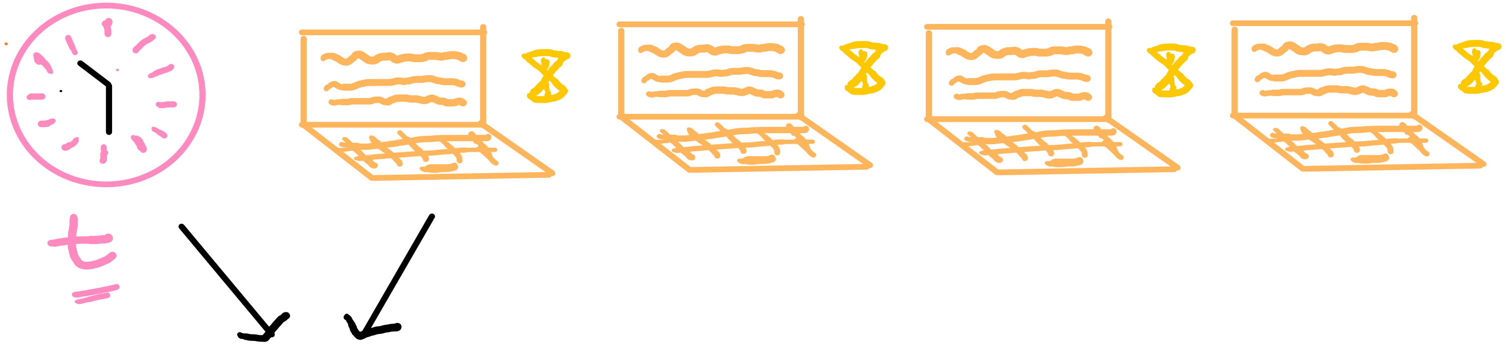
→ $f : \mathcal{X} \to \mathcal{Y}$



$f(x) \; \forall x \in \mathcal{X}$

Verifiable Delay Functions

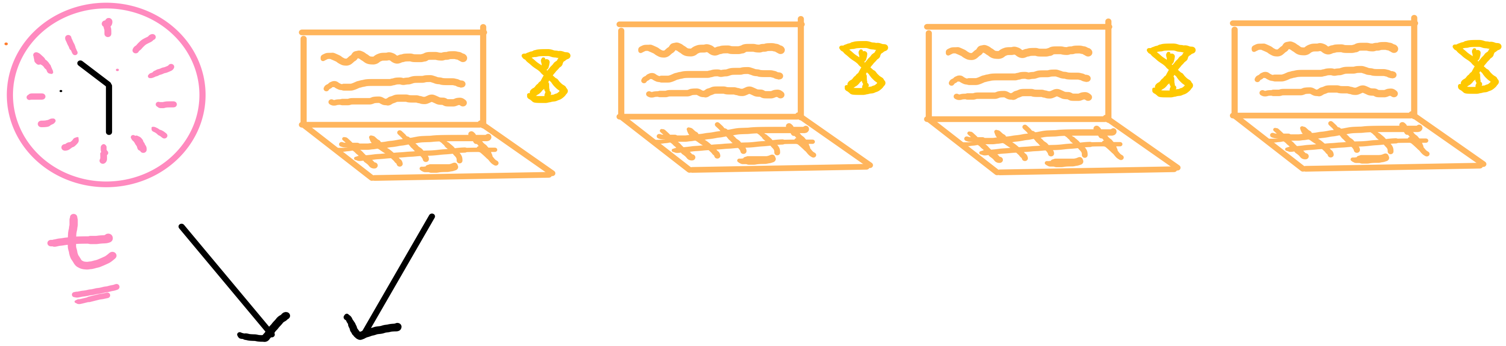
→ $f : \mathcal{X} \to \mathcal{Y}$



$f(x) \ \forall x \in \mathcal{X}$

Verifiable Delay Functions

→ $f : \mathcal{X} \to \mathcal{Y}$



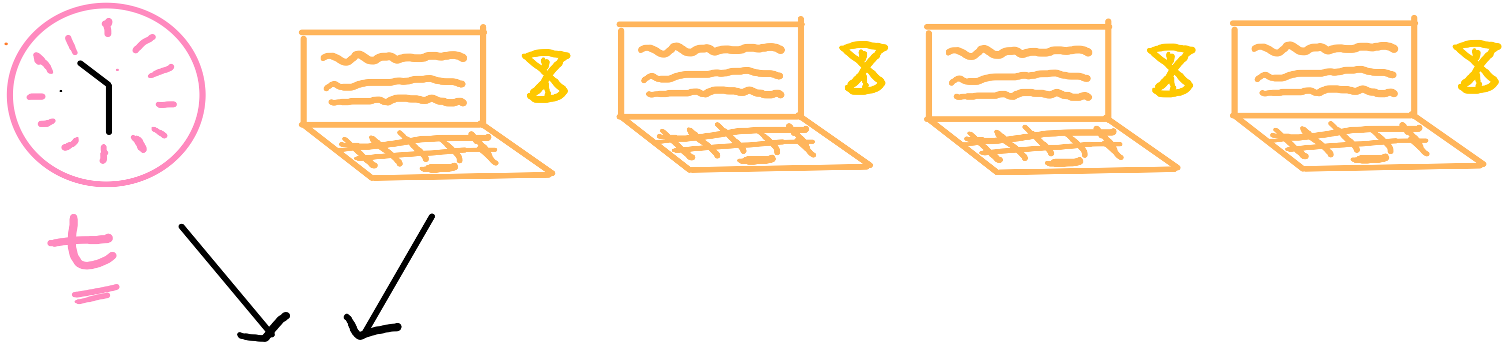
$f(x)\ \forall x \in \mathcal{X}$

$(x, y, t, \pi),\ y = f(x)$

(correctness)

Verifiable Delay Functions

→ $f: \mathcal{X} \to \mathcal{Y}$



$f(x)\ \forall x \in \mathcal{X}$

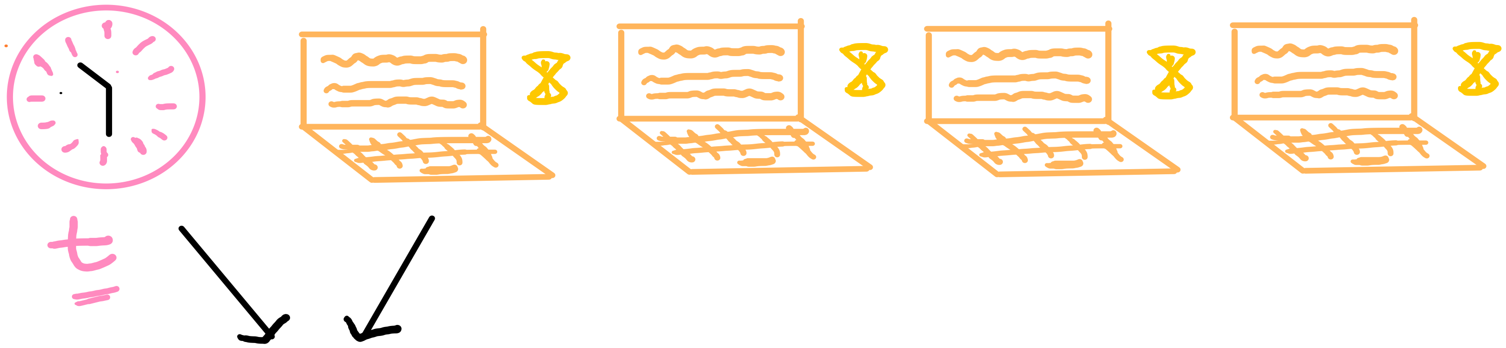
$(x, y, t, \pi),\ y = f(x)$

$t' \lll t$

↑ efficiency

Verifiable Delay Functions

→ $f : \mathcal{X} \rightarrow \mathcal{Y}$



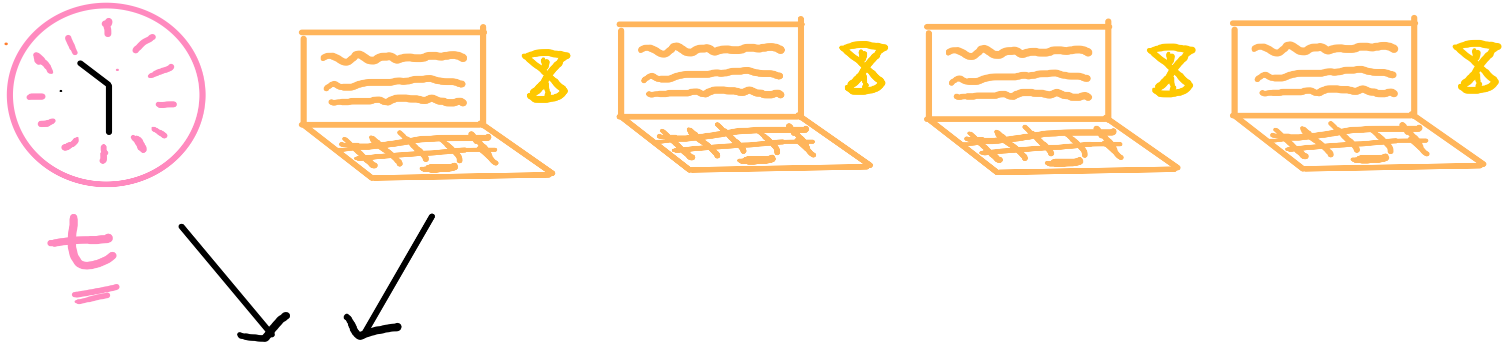
$f(x) \; \forall x \in \mathcal{X}$

$(x, y, t, \pi), \; y \neq f(x)$

(soundness)

Verifiable Delay Functions

→ $f: \mathcal{X} \to \mathcal{Y}$



$f(x)\ \forall x \in \mathcal{X}$

No Body

$(x, y, t, \pi),\ y \neq f(x)$

(soundness)

unique valid output

for all $x \in \mathcal{X}$

Verifiable Delay Functions

→ More specifically, a tuple of 3 algorithms:

Verifiable Delay Functions

→ More specifically, a tuple of 3 algorithms:

- Setup$(\lambda, T) \rightarrow pp$

Verifiable Delay Functions

→ More specifically, a tuple of 3 algorithms:

- $\text{Setup}(\underset{\substack{\uparrow \\ \text{security}}}{\lambda}, T) \to pp$

Verifiable Delay Functions

→ More specifically, a tuple of 3 algorithms:

- $\text{Setup}(\lambda, T) \rightarrow pp$ ← public param

$\uparrow$ security $\nwarrow$ delay

Verifiable Delay Functions

→ More specifically, a tuple of 3 algorithms:

- $\text{Setup}(\lambda, T) \xrightarrow{\$} pp$ ← public param (set x, y)

  security delay

Verifiable Delay Functions

→ More specifically, a tuple of 3 algorithms:

- $\text{Setup}(\underset{\substack{\uparrow \\ \text{security}}}{\lambda}, \underset{\substack{\nwarrow \\ \text{delay}}}{T}) \xrightarrow{\$} pp$ ← public param (set x, y)

- $\text{Eval}(pp, x) \longrightarrow (y, \pi)$

Verifiable Delay Functions

→ More specifically, a tuple of 3 algorithms:

- ☒ $\mathrm{Setup}(\underset{\uparrow\ \text{security}}{\lambda}, \underset{\nwarrow\ \text{delay}}{T}) \xrightarrow{\$} pp$ ← public param (set $\mathcal{X}, \mathcal{Y}$)

- ☒ $\mathrm{Eval}(pp, \underset{\uparrow\ \text{input}}{x}) \longrightarrow (\underset{\uparrow\ \text{output}}{y}, \overset{\text{proof}\ \swarrow}{\pi})$

Verifiable Delay Functions

→ More specifically, a tuple of 3 algorithms:

- $\text{Setup}(\lambda, T) \xrightarrow{\$} pp$ ← public param (set x, y)

  security delay

- $\text{Eval}(pp, x) \longrightarrow (y, \pi)$

  proof

  input output

- $\text{Verify}(pp, x, y, \pi) \rightarrow \{0, 1\}$

Verifiable Delay Functions

→ More specifically, a tuple of 3 algorithms:

- ☒ $\mathrm{Setup}(\underset{\text{security}}{\lambda}, \underset{\text{delay}}{T}) \xrightarrow{\$} pp$ ← public param (set $\mathcal{X}, \mathcal{Y}$)

- ☒ $\mathrm{Eval}(pp, \underset{\text{input}}{x}) \longrightarrow (\underset{\text{output}}{y}, \overset{\text{proof}}{\pi})$

- ☒ $\mathrm{Verify}(pp, x, y, \pi) \rightarrow \{0, \underset{\text{output if } (y,\pi) \leftarrow \mathrm{Eval}(pp,x)}{1}\}$

Verifiable Delay Functions

→ More specifically, a tuple of 3 algorithms:

- ☒ $\mathrm{Setup}(\underset{\text{security}}{\lambda}, \underset{\text{delay}}{T}) \xrightarrow{\$} pp$ ← public param (set x, y)

- ☒ $\mathrm{Eval}(pp, \underset{\text{input}}{x}) \longrightarrow (\underset{\text{output}}{y}, \overset{\text{proof}}{\pi})$

- ☒ $\mathrm{Verify}(pp, x, y, \pi) \to \{\underset{\text{otherwise}}{0}, \underset{\text{output if } (y,\pi) \leftarrow \mathrm{Eval}(pp,x)}{1}\}$

Verifiable Delay Functions

→ More specifically, a tuple of 3 algorithms:

- $\text{Setup}(\underset{\text{security}}{\lambda}, \underset{\text{delay}}{T}) \xrightarrow{\$} pp \leftarrow$ public param (set $\mathcal{X}, \mathcal{Y}$)
- $\text{Eval}(pp, \underset{\text{input}}{x}) \longrightarrow (\underset{\text{output}}{y}, \overset{\text{proof}}{\pi})$
- $\text{Verify}(pp, x, y, \pi) \to \{\underset{\text{otherwise}}{0}, 1\}$

  output if $(y, \pi) \leftarrow \text{Eval}(pp, x)$
  $pp \leftarrow \text{Setup}(\lambda, t)$
  $x \in \mathcal{X}$

Verifiable Delay Functions

→ More specifically, a tuple of 3 algorithms:

☒ $\mathrm{Setup}(\lambda, T) \xrightarrow{\$} pp$ ← public param (set $\mathcal{X}, \mathcal{Y}$)

security delay

proof

☒ $\mathrm{Eval}(pp, x) \longrightarrow (y, \pi)$

input output

☒ $\mathrm{Verify}(pp, x, y, \pi) \rightarrow \{0, 1\}$

otherwise output if $(y, \pi) \leftarrow \mathrm{Eval}(pp, x)$

$pp \leftarrow \mathrm{Setup}(\lambda, t)$

$x \in \mathcal{X}$

Sometimes, also

☒ $\mathrm{TEval}(td, pp, x) \rightarrow (y, \pi)$

Verifiable Delay Functions

→ More specifically, a tuple of 3 algorithms:

- ☒ $\text{Setup}(\underset{\text{security}}{\lambda}, \underset{\text{delay}}{T}) \xrightarrow{\$} pp$ ← public param (set $\mathcal{X}, \mathcal{Y}$)
- ☒ $\text{Eval}(pp, \underset{\text{input}}{x}) \longrightarrow (\underset{\text{output}}{y}, \overset{\text{proof}}{\pi})$
- ☒ $\text{Verify}(pp, x, y, \pi) \rightarrow \{\underset{\text{otherwise}}{0}, 1\}$

  1 ← output if
  $(y, \pi) \leftarrow \text{Eval}(pp, x)$
  $pp \leftarrow \text{Setup}(\lambda, t)$
  $x \in \mathcal{X}$

sometimes, also

- ☒ $\text{TEval}(\underset{\text{trapdoor}}{td}, pp, x) \rightarrow (y, \pi)$

Verifiable Delay Functions

→ More specifically, a tuple of 3 algorithms:

☒ $\text{Setup}(\lambda, T) \xrightarrow{\$} pp$ ← public param

(set $\mathcal{X}, \mathcal{Y}$)

security delay

☒ $\text{Eval}(pp, x) \longrightarrow (y, \pi)$

proof

input output

☒ $\text{Verify}(pp, x, y, \pi) \rightarrow \{0, 1\}$

otherwise

output if $(y, \pi) \leftarrow \text{Eval}(pp, x)$

$pp \leftarrow \text{Setup}(\lambda, t)$

$x \in \mathcal{X}$

sometimes, also

☒ $\text{TEval}(td, pp, x) \rightarrow (y, \pi)$

trapdoor → use to compute (y, π) in $T' <<< T$

Verifiable Delay Functions

→ More specifically, a tuple of 3 algorithms:

- $\text{Setup}(\lambda, T) \xrightarrow{\$} pp$ ← public param (set $\mathcal{X}, \mathcal{Y}$)

  security delay

- $\text{Eval}(pp, x) \longrightarrow (y, \pi)$

  proof

  input output

- $\text{Verify}(pp, x, y, \pi) \to \{0, 1\}$

  otherwise

  output if $(y, \pi) \leftarrow \text{Eval}(pp, x)$

  $pp \leftarrow \text{Setup}(\lambda, t)$

  $x \in \mathcal{X}$

Sometimes, also

- $\text{TEval}(td, pp, x) \to (y, \pi)$

Keep secret! trapdoor → use to compute (y, π) in $T' <<<< T$

Verifiable Delay Functions

→ Now, a concrete construction?

Verifiable Delay Functions

→ Now, a concrete construction?

RSW goes publicly verifiable!

Plan for the afternoon

→ Timed-release crypto

→ Time-lock puzzles

→ Verifiable Delay Functions

→ Pietrzak's construction

Pietrzak's VDF [Pie'19]

Pietrzak's VDF

→ Let's keep things simple!

Pietrzak's VDF

→ Let's keep things simple!

Without loss of generality

set $T = 2^k, k \in \mathbb{N}$

Pietrzak's VDF

→ Let's keep things simple!

Without loss of generality

set $T = 2^k, k \in \mathbb{N}$

$f : \mathbb{G} \longrightarrow \mathbb{G} \;, \; h = f(g) = g^{2^T} \; (\text{in } \mathbb{G})$

Pietrzak's VDF

→ Let's keep things simple!

Without loss of generality

set $T = 2^k, k \in \mathbb{N}$

$f: \mathbb{G} \longrightarrow \mathbb{G}$, $h = f(g) = g^{2^T}$ (in $\mathbb{G}$)

$\text{Setup}(\lambda, T) \longrightarrow pp := (\mathbb{G}, T)$

Pietrzak's VDF

→ Let's keep things simple!

Without loss of generality

set $T = 2^k, k \in \mathbb{N}$

$f : \mathbb{G} \longrightarrow \mathbb{G}, \ h = f(g) = g^{2^T}$ (in $\mathbb{G}$)

$\text{Setup}(\lambda, T) \longrightarrow pp := (\mathbb{G}, T)$

$\text{Eval}(pp, g) \longrightarrow (h, \pi)$

Pietrzak's VDF

→ Let's keep things simple!

Without loss of generality

set $T = 2^k, k \in \mathbb{N}$

$f : \mathbb{G} \longrightarrow \mathbb{G}$, $h = f(g) = g^{2^T}$ (in $\mathbb{G}$)

$\text{Setup}(\lambda, T) \longrightarrow pp := (\mathbb{G}, T)$

$\text{Eval}(pp, g) \longrightarrow (h, \pi)$

But how does $\text{Verify}(pp, g, h, \pi)$ work?

Pietrzak's VDF

→ Let's keep things simple!

Without loss of generality

set $T = 2^k, k \in \mathbb{N}$

$f: \mathbb{G} \longrightarrow \mathbb{G}, \quad h = f(g) = g^{2^T}$ (in $\mathbb{G}$)

$\text{Setup}(\lambda, T) \longrightarrow pp := (\mathbb{G}, T)$

$\text{Eval}(pp, g) \longrightarrow (h, \pi)$

But how does $\text{Verify}(pp, g, h, \pi)$ work?

let's make it interactive for now...

Pietrzak's VDF

→ Let's keep things simple!

Without loss of generality

set $T = 2^k, k \in \mathbb{N}$

$f : \mathbb{G} \longrightarrow \mathbb{G}$, $h = f(g) = g^{2^T}$ (in $\mathbb{G}$)

$\text{Setup}(\lambda, T) \longrightarrow pp := (\mathbb{G}, T)$

$\text{Eval}(pp, g) \longrightarrow (h, \pi)$

But how does $\text{Verify}(pp, g, h, \pi)$ work?

Pietrzak's VDF

→ Let's keep things simple!

Without loss of generality set $T = 2^k,\ k \in \mathbb{N}$

$f : \mathbb{G} \longrightarrow \mathbb{G},\quad h = f(g) = g^{2^T}$ (in $\mathbb{G}$)

$\text{Setup}(\lambda, T) \longrightarrow pp := (\mathbb{G}, T)$

$\text{Eval}(pp, g) \longrightarrow (h, \pi)$

But how does $\text{Verify}(pp, g, h, \pi)$ work?

Pietrzak's VDF

→ Let's keep things simple!

Without loss of generality

set $T = 2^k, k \in \mathbb{N}$

$f : \mathbb{G} \longrightarrow \mathbb{G}$, $h = f(g) = g^{2^T}$ (in $\mathbb{G}$)

$\text{Setup}(\lambda, T) \longrightarrow pp := (\mathbb{G}, T)$

$\text{Eval}(pp, g) \longrightarrow (h, \pi)$

But how does $\text{Verify}(pp, g, h, \pi)$ work?

Pietrzak's VDF

→ Let's keep things simple!

Without loss of generality

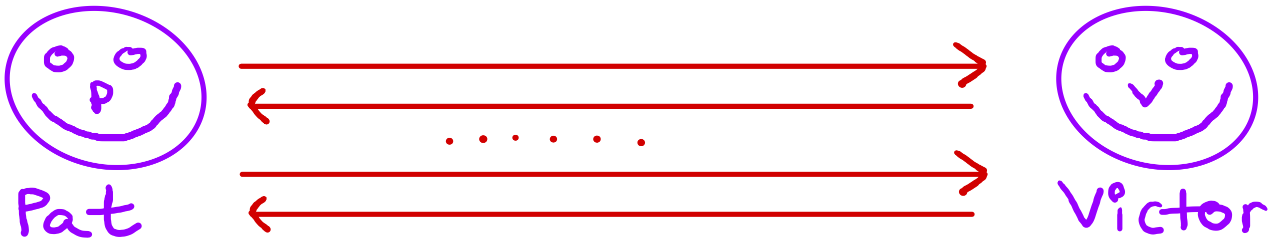
set $T = 2^k,\ k \in \mathbb{N}$

$f : \mathbb{G} \longrightarrow \mathbb{G}\ ,\ h = f(g) = g^{2^T}$ (in $\mathbb{G}$)

$\mathrm{Setup}(\lambda, T) \longrightarrow pp := (\mathbb{G}, T)$

$\mathrm{Eval}(pp, g) \longrightarrow (h, \pi)$

But how does $\mathrm{Verify}(pp, g, h, \pi)$ work?



Pietrzak's VDF

→ Let's keep things simple!

Without loss of generality

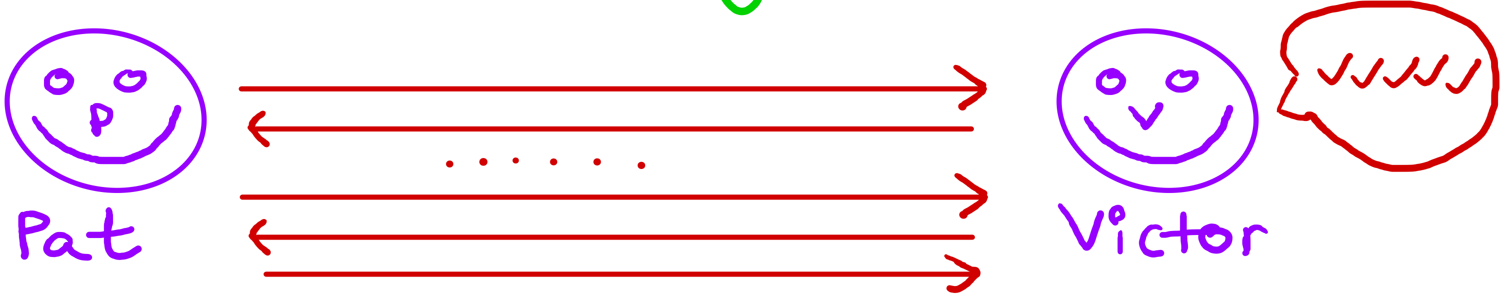
set $T = 2^k, k \in \mathbb{N}$

$f : \mathbb{G} \longrightarrow \mathbb{G}, \quad h = f(g) = g^{2^T} \text{ (in } \mathbb{G})$

$\mathrm{Setup}(\lambda, T) \longrightarrow pp := (\mathbb{G}, T)$

$\mathrm{Eval}(pp, g) \longrightarrow (h, \pi)$

But how does $\mathrm{Verify}(pp, g, h, \pi)$ work?



Pietrzak's VDF

Halving subprotocol:

(or meet-in-the-middle)

Pietrzak's VDF

Halving subprotocol:

(or meet-in-the-middle)

$$\begin{tikzcd}[column sep=small] & h = g^{2^T} \text{ (in } \mathbb{G}\text{)} \arrow[dl] \arrow[dr] & \\ v = g^{2^{T/2}} & & h = v^{2^{T/2}} \end{tikzcd}$$

Pietrzak's VDF

Halving subprotocol:

(or meet-in-the-middle)

$$\begin{tikzcd}
 & h = g^{2^T} \quad (\text{in } \mathbb{G}) \arrow[dl] \arrow[dr] & \\
v = g^{2^{T/2}} \arrow[dr] & & h = v^{2^{T/2}} \arrow[dl] \\
 & g^r v = g^{r+2^{T/2}} &
\end{tikzcd}$$

$$v^r h = g^{r \cdot 2^{T/2}+2^T}$$

(random r in $\{0, \ldots, 2^\lambda\}$)

Pietrzak's VDF

Halving subprotocol:

(or meet-in-the-middle)

$$\begin{tikzcd}
& h = g^{2^T} \ (\text{in } \mathbb{G}) \arrow[dl] \arrow[dr] & \\
v = g^{2^{T/2}} \arrow[dr] & & h = v^{2^{T/2}} \arrow[dl] \\
& g^r v = g^{r+2^{T/2}} \arrow[d, bend right, "T/2 \text{ sq.}"'] & \\
& v^r h = g^{r \cdot 2^{T/2} + 2^T} &
\end{tikzcd}$$

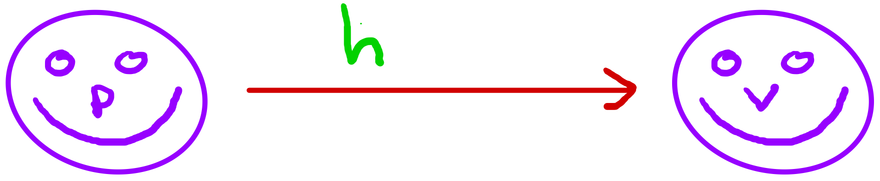
(random r in $\{0, \dots, 2^\lambda\}$)

Pietrzak's VDF

Halving subprotocol:

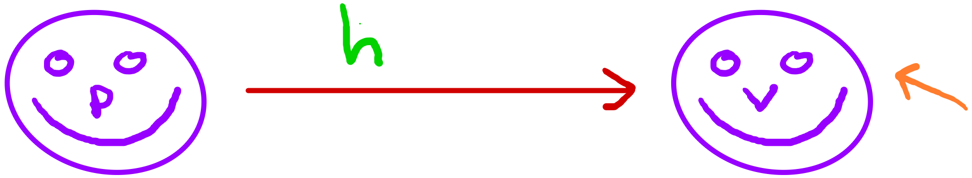
Pietrzak's VDF

Halving subprotocol:



Pietrzak's VDF

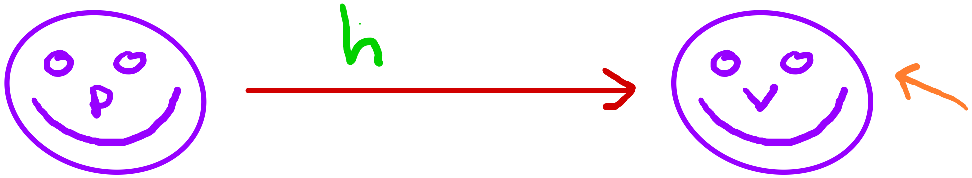
Halving subprotocol:



check $g, h \in \mathbb{G}$

Pietrzak's VDF

Halving subprotocol:

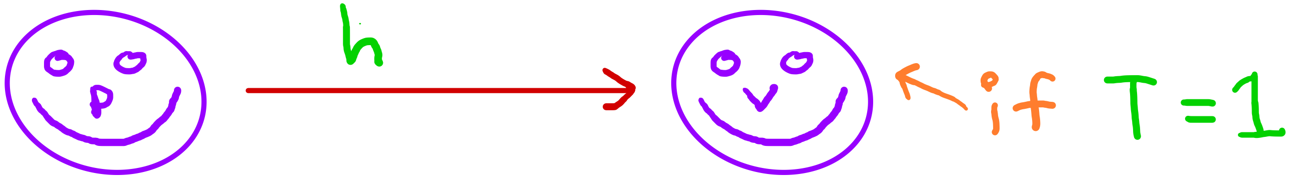


check $g, h \in \mathbb{G}$

output 0 if not

Pietrzak's VDF

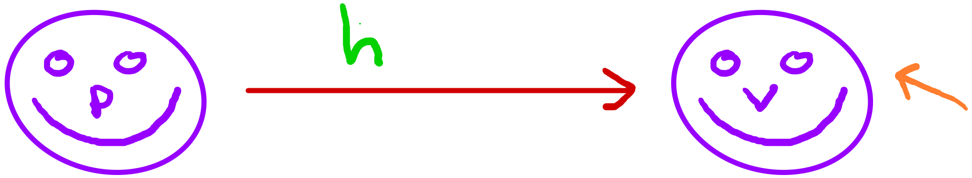
Halving subprotocol:



if $T=1$

Pietrzak's VDF

Halving subprotocol:

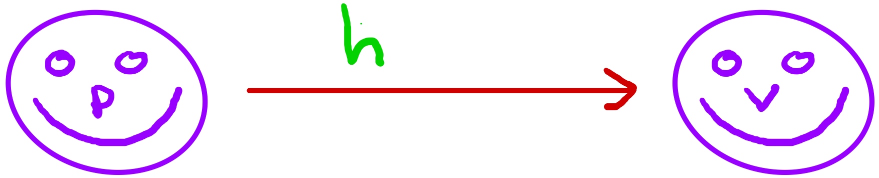


if $T=1$

check $h = g^2$ (in $\mathbb{G}$)

Pietrzak's VDF

Halving subprotocol:



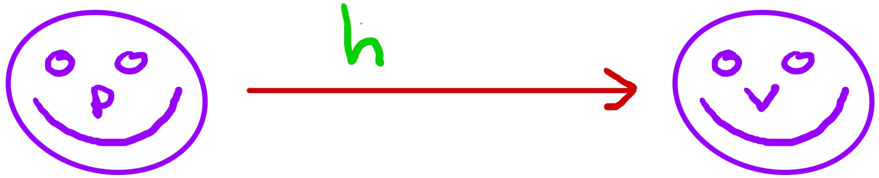
← if $T=1$

check $h = g^2$ (in $\mathbb{G}$)

output 1 if so, else 0

Pietrzak's VDF

Halving subprotocol:



if $T=1$

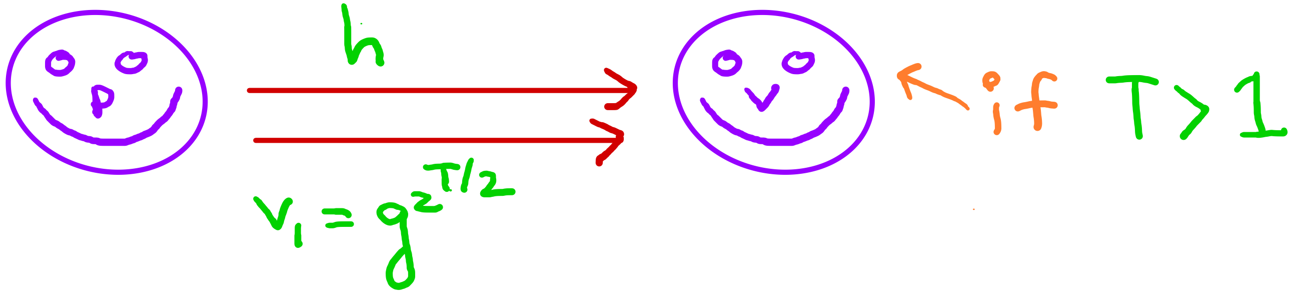
check $h = g^2$ (in $\mathbb{G}$)

output 1 if so, else 0

STOP

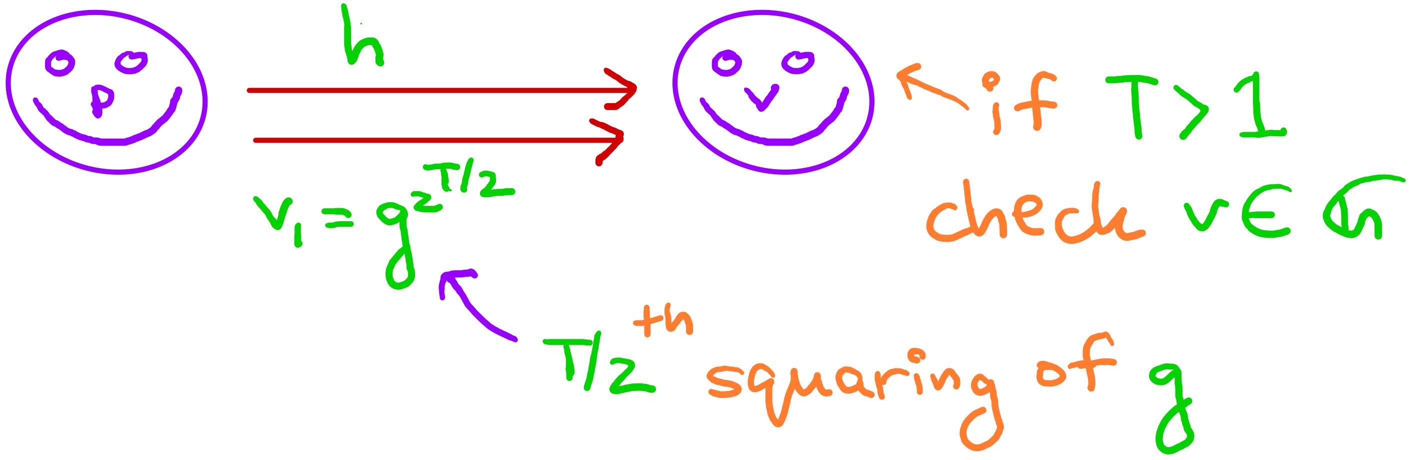
Pietrzak's VDF

Halving subprotocol:



Pietrzak's VDF

Halving subprotocol:



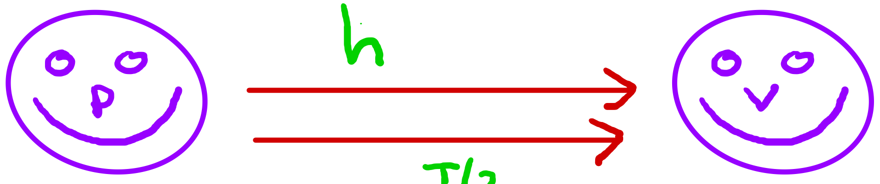
$v_1 = g^{2^{T/2}}$

if $T>1$
check $v \in \mathbb{G}$

$T/2^{th}$ squaring of g

Pietrzak's VDF

Halving subprotocol:



$v_1 = g^{2^{T/2}}$

$T/2^{th}$ squaring of g

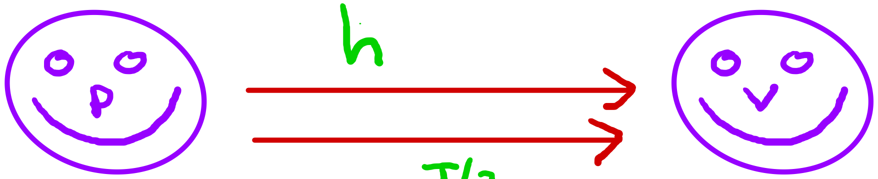
if $T > 1$

check $v \in \mathbb{G}$

output 0 if not

Pietrzak's VDF

Halving subprotocol:



$v_1 = g^{2^{T/2}}$

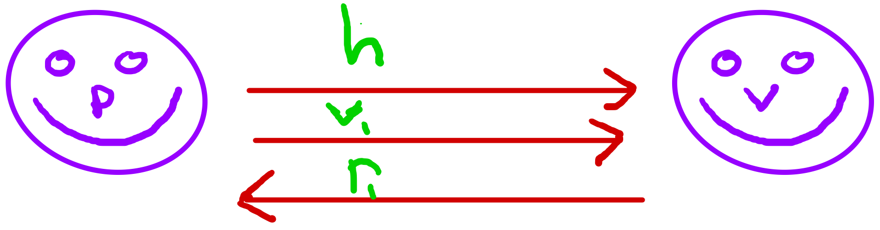
$T/2^{\text{th}}$ squaring of g

if $T > 1$

pick $r_1 \xleftarrow{\$} \{1, \ldots, 2^{\lambda}\}$

Pietrzak's VDF

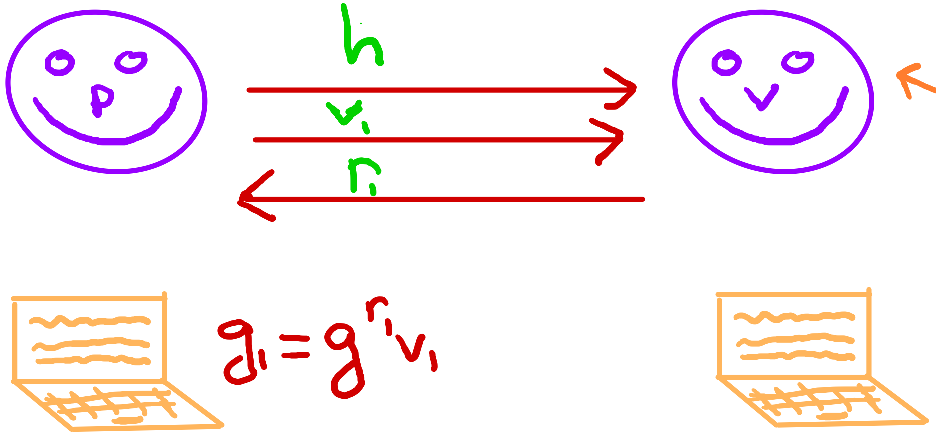
Halving subprotocol:



← if $T > 1$
pick $r_1 \xleftarrow{\$} \{1, \ldots, 2^\lambda\}$

Pietrzak's VDF

Halving subprotocol:



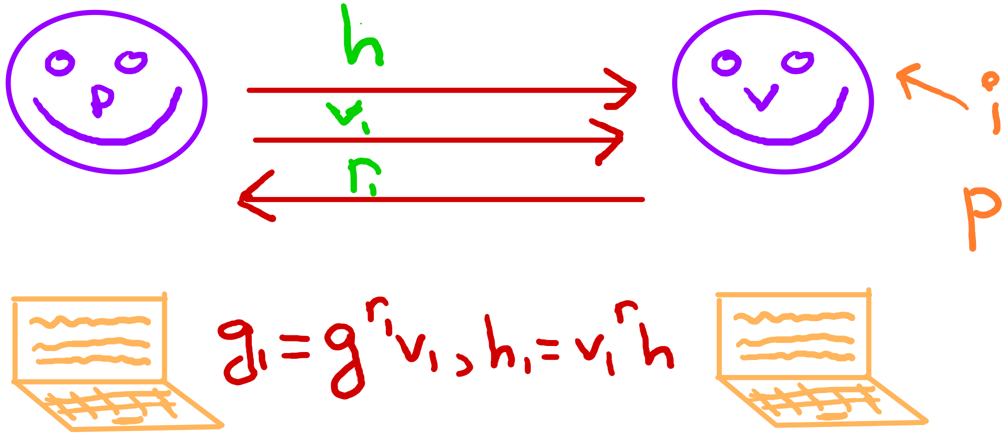
$g_1 = g^{r_1} v_1$

if $T > 1$

pick $r_1 \xleftarrow{\$} \{1, \dots, 2^\lambda\}$

Pietrzak's VDF

Halving subprotocol:



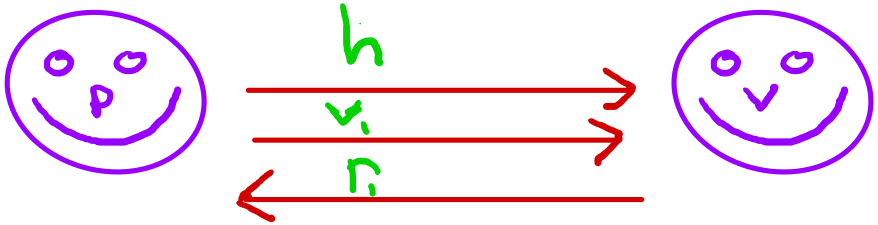
$g_1 = g^{r_1} v_1$, $h_1 = v_1^{r_1} h$

if $T > 1$

pick $r_1 \xleftarrow{\$} \{1, \ldots, 2^\lambda\}$

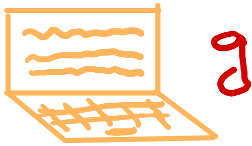
Pietrzak's VDF

Halving subprotocol:



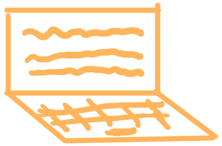
if $T>1$

pick $r_1 \xleftarrow{\$} \{1, \dots, 2^\lambda\}$



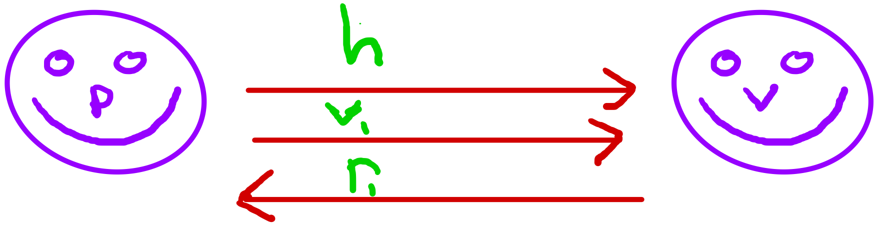
$g_1 = g^{r_1} v_1, \; h_1 = v_1^{r_1} h$

(in $\mathbb{G}$)



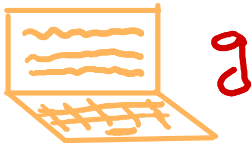
Pietrzak's VDF

Halving subprotocol:



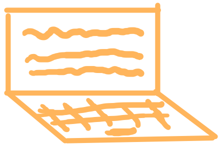
if $T>1$

pick $r_1 \xleftarrow{\$} \{1,\dots,2^\lambda\}$



$g_1 = g^{r_1} v_1, h_1 = v_1^{r_1} h$

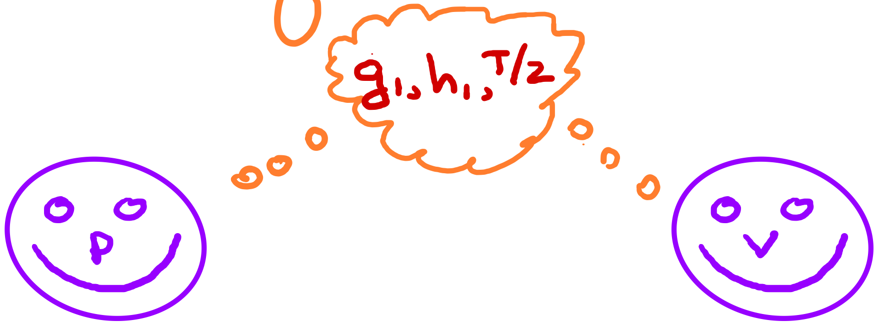
(in $\mathbb{G}$)



→ input for next recursive layer

Pietrzak's VDF

Halving subprotocol:



Pietrzak's VDF

Halving subprotocol:

if $T = 1$

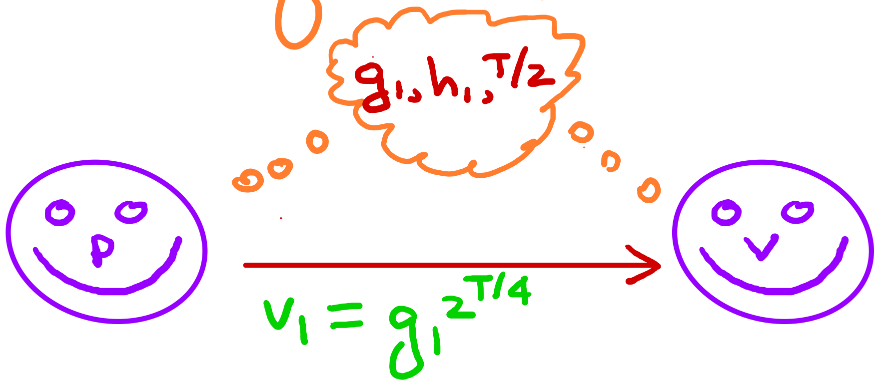
check $h_1 = g_1^2$ (in $\mathbb{G}$)

output 1 if so, else 0

STOP

Pietrzak's VDF

Halving subprotocol:

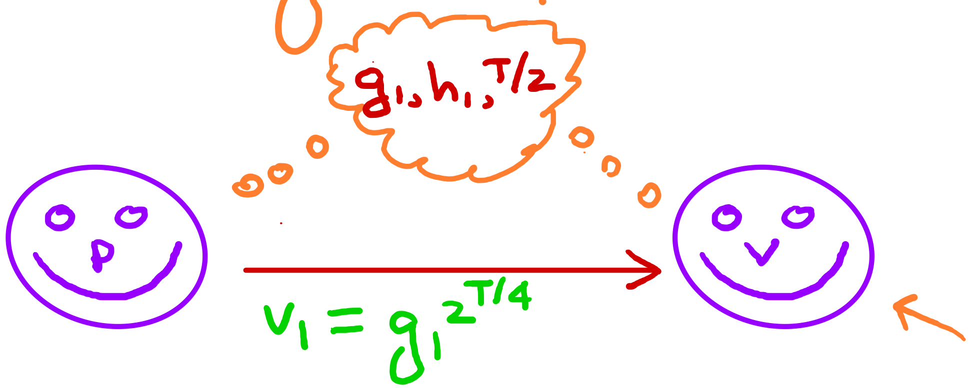


$v_1 = g_1^{2^{T/4}}$

if $T > 1$

Pietrzak's VDF

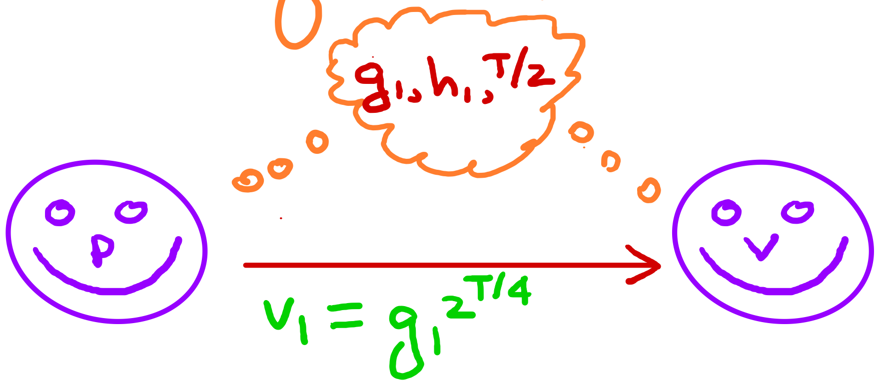
Halving subprotocol:



if $T > 1$
check $v_1 \in \mathbb{G}$
output 0 if not

Pietrzak's VDF

Halving subprotocol:

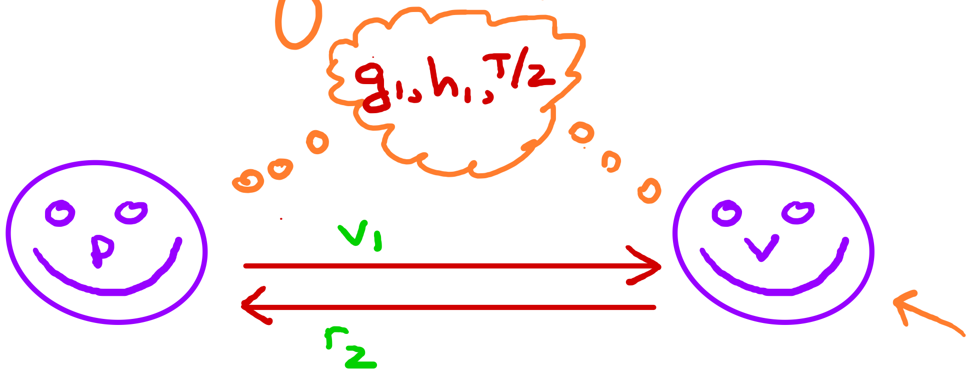


if $T > 1$

pick $r_2 \xleftarrow{\$} \{1, \ldots, 2^\lambda\}$

Pietrzak's VDF

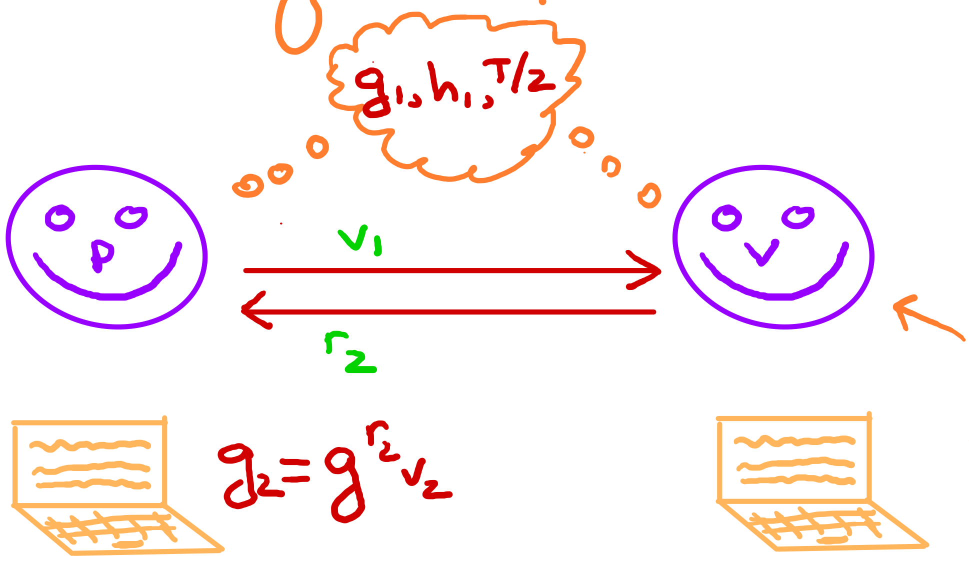
Halving subprotocol:



if $T > 1$
pick $r_2 \xleftarrow{\$} \{1, \ldots, 2^\lambda\}$

Pietrzak's VDF

Halving subprotocol:

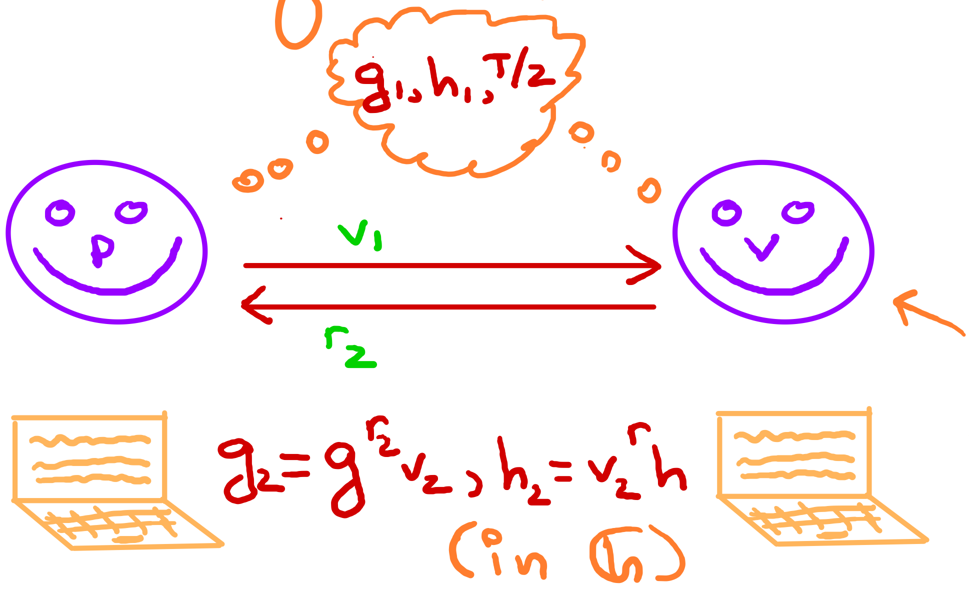


$g_2 = g^{r_2} v_2$

if $T > 1$
pick $r_2 \xleftarrow{\$} \{1, \dots, 2^\lambda\}$

Pietrzak's VDF

Halving subprotocol:



$g_2 = g^{r_2} v_2$, $h_2 = v_2^{r} h$

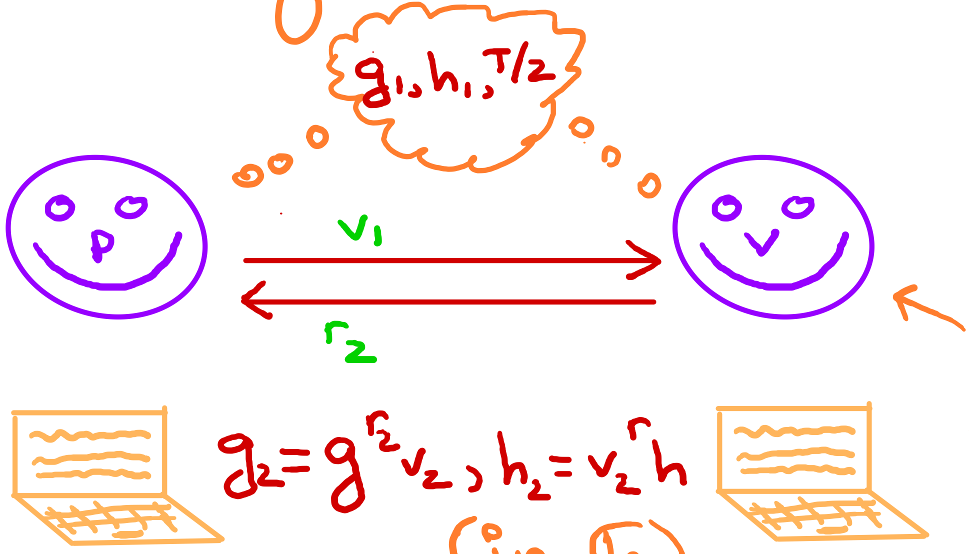
(in $\mathbb{G}$)

if $T > 1$

pick $r_2 \xleftarrow{\$} \{1, \ldots, 2^{\lambda}\}$

Pietrzak's VDF

Halving subprotocol:



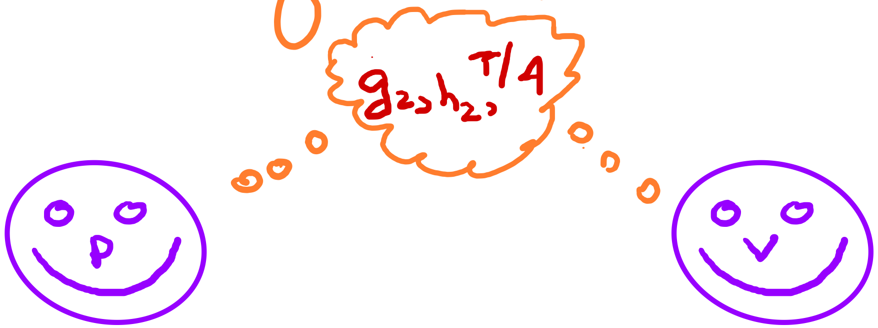
$g_2 = g^{r_2} v_2 , h_2 = v_2^{r} h$ (in $\mathbb{G}$)

← if $T > 1$ pick $r_2 \xleftarrow{\$} \{1, \ldots, 2^\lambda\}$

→ input for next recursive layer

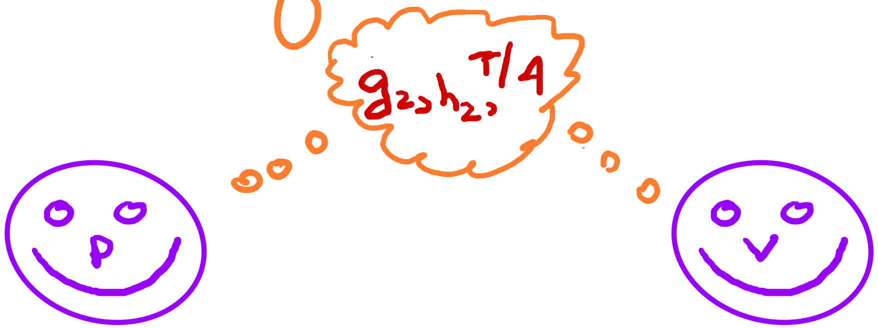
Pietrzak's VDF

Halving subprotocol:



Pietrzak's VDF

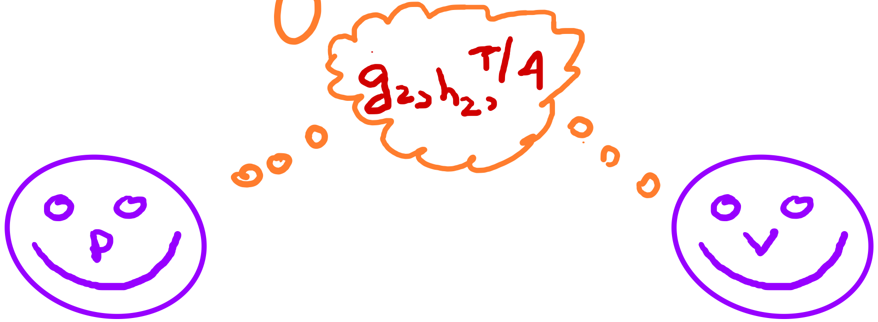
Halving subprotocol:



rinse and repeat!

Pietrzak's VDF

Halving subprotocol:

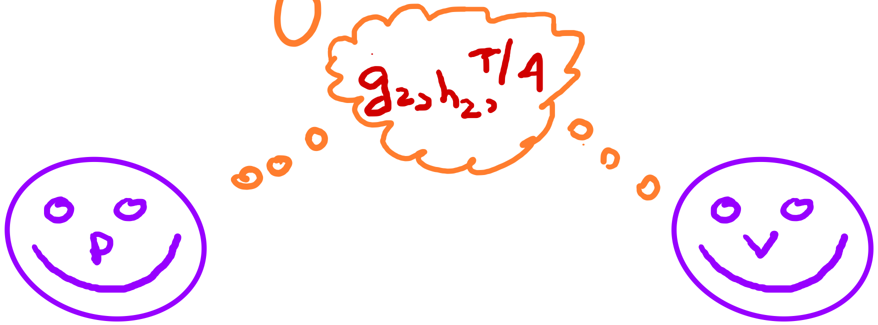


rinse and repeat!

$\log_2 T$ rounds

Pietrzak's VDF

Halving subprotocol:

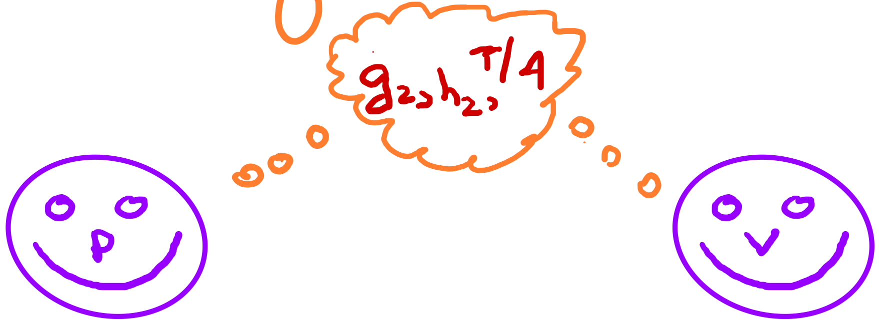


rinse and repeat!

$\log_2 T$ rounds ⟶ can be made non-interactive

Pietrzak's VDF

Halving subprotocol:



rinse and repeat!

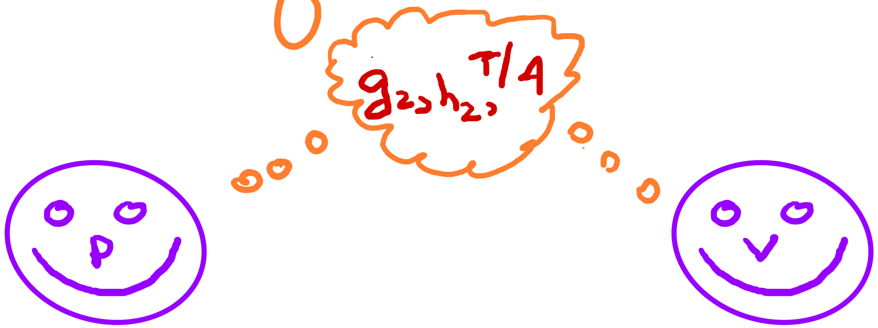
$\log_2 T$ rounds → can be made non-interactive

(Fiat-Shamir in ROM)

[FS'86]

Pietrzak's VDF

Halving subprotocol:



rinse and repeat!

$\log_2 T$ rounds → can be made **non-interactive**

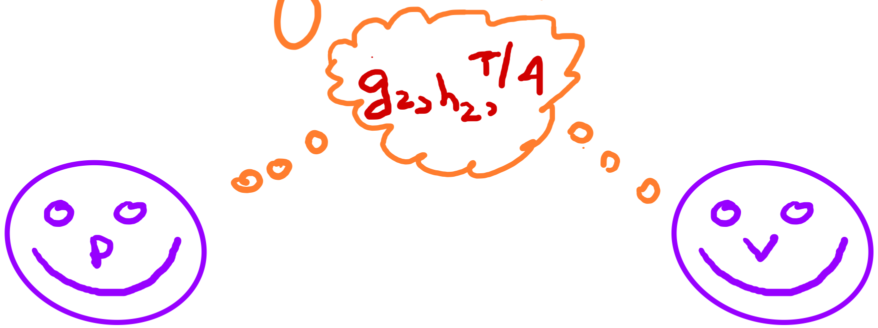
(Fiat-Shamir in ROM)

[FS'86]

then π has $\log_2 T$ elements

Pietrzak's VDF

Halving subprotocol:



rinse and repeat!

$\log_2 T$ rounds → can be made non-interactive

(Fiat-Shamir in ROM)

[FS'86]

then π has $\log_2 T$ elements

let's unwind the recursion for $T=8$!

Pietrzak's VDF

Halving subprotocol for $T = 8$:

Pietrzak's VDF

Halving subprotocol for $T = 8$:

$h = g^{2^8}$

Pietrzak's VDF

Halving subprotocol for $T=8$:

$h = g^{2^8} = g^{256}$

Pietrzak's VDF

Halving subprotocol for $T = 8$:

$h = g^{2^8} = g^{256}$

$v_1 = g^{2^{8/2}}$

Pietrzak's VDF

Halving subprotocol for $T = 8$:

$$h = g^{2^{8}} = g^{256}$$

$$v_1 = g^{2^{8/2}} = g^{16}$$

$$P \xrightarrow{v_1} V$$

Pietrzak's VDF

Halving subprotocol for T = 8 :

$$h = g^{2^8} = g^{256}$$

$$v_1 = g^{2^{8/2}} = g^{16}$$

$$\begin{tikzcd} P \arrow[r, shift left, "v_1"] & V \arrow[l, shift left, "r_1"] \end{tikzcd}$$

Pietrzak's VDF

Halving subprotocol for $T = 8$:

$$h = g^{2^8} = g^{256}$$

$$v_1 = g^{2^{8/2}} = g^{16}$$

$$g_1 = g^{r_1} v_1$$

$$\begin{tikzcd} P \arrow[r, shift left, "v_1"] & V \arrow[l, shift left, "r_1"] \end{tikzcd}$$

Pietrzak's VDF

Halving subprotocol for T = 8:

$$h = g^{2^8} = g^{256}$$

$$v_1 = g^{2^{8/2}} = g^{16}$$

$$\begin{tikzcd} P \arrow[r, shift left, "v_1"] & V \arrow[l, shift left, "r_1"] \end{tikzcd}$$

$$g_1 = g^{r_1} v_1 = g^{r_1} \cdot g^{16} = g^{r_1+16}$$

Pietrzak's VDF

Halving subprotocol for $T = 8$:

$$h = g^{2^8} = g^{256}$$

$$v_1 = g^{2^{8/2}} = g^{16}$$

$$\begin{tikzcd} P \arrow[r, shift left, "v_1"] & V \arrow[l, shift left, "r_1"] \end{tikzcd}$$

$$g_1 = g^{r_1} v_1 = g^{r_1} \cdot g^{16} = g^{r_1+16}$$

$$h_1 = v_1^{r_1} h$$

Pietrzak's VDF

Halving subprotocol for T = 8:

$$h = g^{2^8} = g^{256}$$

$$v_1 = g^{2^{8/2}} = g^{16}$$

$$\begin{tikzcd} P \arrow[r, shift left, "v_1"] & V \arrow[l, shift left, "r_1"] \end{tikzcd}$$

$$g_1 = g^{r_1} v_1 = g^{r_1} \cdot g^{16} = g^{r_1+16}$$

$$h_1 = v_1^{r_1} h = g^{16 r_1} \cdot g^{256} = g^{16r_1+256}$$

Pietrzak's VDF

Halving subprotocol for $T=8$:

$$h = g^{2^8} = g^{256}$$

$$v_1 = g^{2^{8/2}} = g^{16}$$

$$\begin{tikzcd} P \arrow[r, shift left, "v_1"] & V \arrow[l, shift left, "r_1"] \end{tikzcd}$$

$$g_1 = g^{r_1} v_1 = g^{r_1} \cdot g^{16} = g^{r_1+16}$$

$$h_1 = v_1^{r_1} h = g^{16 r_1} \cdot g^{256} = g^{16r_1+256}$$

input for next recursive layer:

$T=4,\ g_1,\ h_1$

Pietrzak's VDF

Halving subprotocol for T=4 :

$g_1 = g^{r_1 + 16}$

$h_1 = g^{16 r_1 + 256}$

$v_2 = g_1^{2^{4/2}}$

Pietrzak's VDF

Halving subprotocol for T=4 :

$g_1 = g^{r_1+16}$

$h_1 = g^{16r_1+256}$

$v_2 = g_1^{2^{4/2}} = (g^{r_1+16})^4 = g^{4r_1+64}$

Pietrzak's VDF

Halving subprotocol for T=4 :

$g_1 = g^{r_1+16}$

$h_1 = g^{16r_1+256}$

$v_2 = g_1^{2^{4/2}} = (g^{r_1+16})^4 = g^{4r_1+64}$

$$P \xrightarrow{v_2} V$$

Pietrzak's VDF

Halving subprotocol for T=4 :

$$g_1 = g^{r_1+16}$$

$$h_1 = g^{16r_1+256}$$

$$v_2 = g_1^{2^{4/2}} = (g^{r_1+16})^4 = g^{4r_1+64}$$

$$\begin{tikzcd} P \arrow[r, shift left, "v_2"] & V \arrow[l, shift left, "r_2"] \end{tikzcd}$$

Pietrzak's VDF

Halving subprotocol for $T=4$:

$g_1 = g^{r_1+16}$

$h_1 = g^{16r_1+256}$

$v_2 = g_1^{2^{4/2}} = (g^{r_1+16})^4 = g^{4r_1+64}$

$$\begin{tikzcd} P \arrow[r, shift left, "v_2"] & V \arrow[l, shift left, "r_2"] \end{tikzcd} \qquad g_2 = g_1^{r_2} v_2$$

Pietrzak's VDF

Halving subprotocol for T=4 :

$g_1 = g^{r_1+16}$

$h_1 = g^{16r_1+256}$

$v_2 = g_1^{2^{4/2}} = (g^{r_1+16})^4 = g^{4r_1+64}$

$$\begin{tikzcd} P \arrow[r, shift left, "v_2"] & V \arrow[l, shift left, "r_2"] \end{tikzcd}$$

$$g_2 = g_1^{r_2} v_2 = g^{r_1r_2+16r_2} \cdot g^{4r_1+64}$$

$$= g^{4r_1+r_1r_2+16r_2+64}$$

Pietrzak's VDF

Halving subprotocol for T=4 :

$$g_1 = g^{r_1+16}$$

$$h_1 = g^{16r_1+256}$$

$$v_2 = g_1^{2^{4/2}} = (g^{r_1+16})^4 = g^{4r_1+64}$$

$$P \underset{r_2}{\overset{v_2}{\rightleftarrows}} V \qquad g_2 = g_1^{r_2} v_2 = g^{r_1 r_2 + 16 r_2} \cdot g^{4r_1+64}$$

$$= g^{4r_1 + r_1 r_2 + 16 r_2 + 64}$$

$$h_2 = v_2^{r_2} h_1$$

Pietrzak's VDF

Halving subprotocol for T=4 :

$$g_1 = g^{r_1+16}$$

$$h_1 = g^{16r_1+256}$$

$$v_2 = g_1^{2^{4/2}} = (g^{r_1+16})^4 = g^{4r_1+64}$$

$$\begin{tikzcd} P \arrow[r, shift left, "v_2"] & V \arrow[l, shift left, "r_2"] \end{tikzcd}$$

$$g_2 = g_1^{r_2} v_2 = g^{r_1r_2+16r_2} \cdot g^{4r_1+64}$$

$$= g^{4r_1+r_1r_2+16r_2+64}$$

$$h_2 = v_2^{r_2} h_1 = g^{4r_1r_2+64r_2} \cdot g^{16r_1+256}$$

$$= g^{16r_1+4r_1r_2+64r_2+256}$$

Pietrzak's VDF

Halving subprotocol for $T=4$:

$$g_1 = g^{r_1+16}$$

$$h_1 = g^{16r_1+256}$$

$$v_2 = g_1^{2^{4/2}} = (g^{r_1+16})^4 = g^{4r_1+64}$$

$$\begin{tikzcd} P \arrow[r, shift left, "v_2"] & V \arrow[l, shift left, "r_2"] \end{tikzcd}$$

$$g_2 = g_1^{r_2} v_2 = g^{r_1r_2+16r_2} \cdot g^{4r_1+64}$$

$$= g^{4r_1+r_1r_2+16r_2+64}$$

$$h_2 = v_2^{r_2} h_1 = g^{4r_1r_2+64r_2} \cdot g^{16r_1+256}$$

$$= g^{16r_1+4r_1r_2+64r_2+256}$$

input for next recursive layer:

$T=2,\ g_2,\ h_2$

Pietrzak's VDF

Halving subprotocol for $T=2$:

$$g_2 = g^{4r_1 + r_1 r_2 + 16 r_2 + 64}$$

$$h_2 = g^{16 r_1 + 4 r_1 r_2 + 64 r_2 + 256}$$

$$v_3 = g_2^{2^{2/2}}$$

Pietrzak's VDF

Halving subprotocol for T=2:

$$g_2 = g^{4r_1 + r_1 r_2 + 16r_2 + 64}$$

$$h_2 = g^{16r_1 + 4r_1 r_2 + 64r_2 + 256}$$

$$v_3 = g_2^{2^{2/2}} = g^{8r_1 + 2r_1 r_2 + 32r_2 + 128}$$

$$P \xrightarrow{v_3} V$$

Pietrzak's VDF

Halving subprotocol for T=2:

$$g_2 = g^{4r_1 + r_1 r_2 + 16 r_2 + 64}$$

$$h_2 = g^{16 r_1 + 4 r_1 r_2 + 64 r_2 + 256}$$

$$v_3 = g_2^{2^{2/2}} = g^{8 r_1 + 2 r_1 r_2 + 32 r_2 + 128}$$

$$\begin{tikzcd} P \arrow[r, shift left, "v_3"] & V \arrow[l, shift left, "r_3"] \end{tikzcd} \qquad g_3 = g_2^{r_3} v_3$$

Pietrzak's VDF

Halving subprotocol for T=2:

$$g_2 = g^{4r_1 + r_1 r_2 + 16 r_2 + 64}$$

$$h_2 = g^{16 r_1 + 4 r_1 r_2 + 64 r_2 + 256}$$

$$v_3 = g_2^{2^{2/2}} = g^{8 r_1 + 2 r_1 r_2 + 32 r_2 + 128}$$

$$\begin{tikzcd} P \arrow[r, shift left, "v_3"] & V \arrow[l, shift left, "r_3"] \end{tikzcd}$$

$$g_3 = g_2^{r_3} v_3 = g^{4 r_1 r_3 + r_1 r_2 r_3 + 16 r_2 r_3 + 64 r_3} \cdot g^{8 r_1 + 2 r_1 r_2 + 32 r_2 + 128}$$

$$= g^{8 r_1 + 32 r_2 + 64 r_3 + 2 r_1 r_2 + 4 r_1 r_3 + 16 r_2 r_3 + r_1 r_2 r_3 + 128}$$

Pietrzak's VDF

Halving subprotocol for T=2:

$$g_2 = g^{4r_1 + r_1 r_2 + 16 r_2 + 64}$$

$$h_2 = g^{16 r_1 + 4 r_1 r_2 + 64 r_2 + 256}$$

$$v_3 = g_2^{2^{2/2}} = g^{8r_1 + 2 r_1 r_2 + 32 r_2 + 128}$$

$$\begin{tikzcd} P \arrow[r, shift left, "v_3"] & V \arrow[l, shift left, "r_3"] \end{tikzcd}$$

$$g_3 = g_2^{r_3} v_3 = g^{4 r_1 r_3 + r_1 r_2 r_3 + 16 r_2 r_3 + 64 r_3} \cdot g^{8 r_1 + 2 r_1 r_2 + 32 r_2 + 128}$$

$$= g^{8 r_1 + 32 r_2 + 64 r_3 + 2 r_1 r_2 + 4 r_1 r_3 + 16 r_2 r_3 + r_1 r_2 r_3 + 128}$$

$$h_3 = v_3^{r_3} h_2$$

Pietrzak's VDF

Halving subprotocol for T=2:

$$g_2 = g^{4r_1 + r_1 r_2 + 16r_2 + 64}$$

$$h_2 = g^{16r_1 + 4r_1 r_2 + 64r_2 + 256}$$

$$v_3 = g_2^{2^{2/2}} = g^{8r_1 + 2r_1 r_2 + 32r_2 + 128}$$

$$\begin{tikzcd} P \arrow[r, bend left, "v_3"] & V \arrow[l, bend left, "r_3"] \end{tikzcd}$$

$$g_3 = g_2^{r_3} v_3 = g^{4r_1 r_3 + r_1 r_2 r_3 + 16r_2 r_3 + 64r_3} \cdot g^{8r_1 + 2r_1 r_2 + 32r_2 + 128}$$

$$= g^{8r_1 + 32r_2 + 64r_3 + 2r_1 r_2 + 4r_1 r_3 + 16r_2 r_3 + r_1 r_2 r_3 + 128}$$

$$h_3 = v_3^{r_3} h_2 = g^{8r_1 r_3 + 2r_1 r_2 r_3 + 32r_2 r_3 + 128r_3} \cdot g^{16r_1 + 4r_1 r_2 + 64r_2 + 256}$$

$$= g^{16r_1 + 64r_2 + 128r_3 + 4r_1 r_2 + 8r_1 r_3 + 32r_2 r_3 + 2r_1 r_2 r_3 + 256}$$

Pietrzak's VDF

Halving subprotocol for T=2:

$$g_2 = g^{4r_1 + r_1 r_2 + 16 r_2 + 64}$$

$$h_2 = g^{16 r_1 + 4 r_1 r_2 + 64 r_2 + 256}$$

$$v_3 = g_2^{2^{2/2}} = g^{8r_1 + 2r_1 r_2 + 32 r_2 + 128}$$

input for final recursive layer:

$T=1$, g_3, h_3

$$\begin{tikzcd} P \arrow[r, shift left, "v_3"] & V \arrow[l, shift left, "r_3"] \end{tikzcd}$$

$$\begin{aligned} g_3 = g_2^{r_3} v_3 &= g^{4r_1 r_3 + r_1 r_2 r_3 + 16 r_2 r_3 + 64 r_3} \cdot g^{8r_1 + 2r_1 r_2 + 32 r_2 + 128} \\ &= g^{8r_1 + 32 r_2 + 64 r_3 + 2 r_1 r_2 + 4 r_1 r_3 + 16 r_2 r_3 + r_1 r_2 r_3 + 128} \end{aligned}$$

$$\begin{aligned} h_3 = v_3^{r_3} h_2 &= g^{8r_1 r_3 + 2 r_1 r_2 r_3 + 32 r_2 r_3 + 128 r_3} \cdot g^{16 r_1 + 4 r_1 r_2 + 64 r_2 + 256} \\ &= g^{16 r_1 + 64 r_2 + 128 r_3 + 4 r_1 r_2 + 8 r_1 r_3 + 32 r_2 r_3 + 2 r_1 r_2 r_3 + 256} \end{aligned}$$

Pietrzak's VDF

Halving subprotocol for $T=1$:

$$g_3 = g^{8r_1+32r_2+64r_3+2r_1r_2+4r_1r_3+16r_2r_3+r_1r_2r_3+128}$$

$$h_3 = g^{16r_1+64r_2+128r_3+4r_1r_2+8r_1r_3+32r_2r_3+2r_1r_2r_3+256}$$

base case → check $h_3=(g_3)^2$

Pietrzak's VDF

Halving subprotocol for $T=1$:

$$g_3 = g^{8r_1+32r_2+64r_3+2r_1r_2+4r_1r_3+16r_2r_3+r_1r_2r_3+128}$$

$$h_3 = g^{16r_1+64r_2+128r_3+4r_1r_2+8r_1r_3+32r_2r_3+2r_1r_2r_3+256}$$

base case → check $h_3=(g_3)^2$

Yes, indeed!

Pietrzak's VDF

Halving subprotocol for $T=1$:

$$g_3 = g^{8r_1+32r_2+64r_3+2r_1r_2+4r_1r_3+16r_2r_3+r_1r_2r_3+128}$$

$$h_3 = g^{16r_1+64r_2+128r_3+4r_1r_2+8r_1r_3+32r_2r_3+2r_1r_2r_3+256}$$

base case $\longrightarrow$ check $h_3 = (g_3)^2$

Yes, indeed! V outputs 1

Pietrzak's VDF

Halving subprotocol for $T=1$:

$$g_3 = g^{8r_1+32r_2+64r_3+2r_1r_2+4r_1r_3+16r_2r_3+r_1r_2r_3+128}$$

$$h_3 = g^{16r_1+64r_2+128r_3+4r_1r_2+8r_1r_3+32r_2r_3+2r_1r_2r_3+256}$$

base case $\longrightarrow$ check $h_3=(g_3)^2$

Yes, indeed! V outputs 1

(takes $2\log_2 T$ exponentiations)

Pietrzak's VDF

Security:

Pietrzak's VDF

Security:

Can construct VDF assuming hardness of iterated squaring in $\mathbb{G}$

Pietrzak's VDF

Security:

Can construct VDF assuming hardness of iterated squaring in $\mathbb{G}$ and ideal hash fn. (ROM)

Pietrzak's VDF

Security:

Can construct VDF assuming hardness of iterated squaring in $\mathbb{G}$ and ideal hash fn. (ROM)

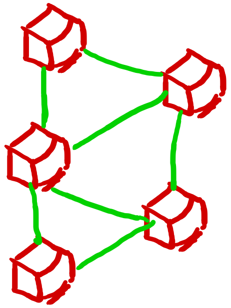
Applications:

Pietrzak's VDF

Security:

Can construct VDF assuming hardness of iterated squaring in $\mathbb{G}$ and ideal hash fn. (ROM)

Applications:

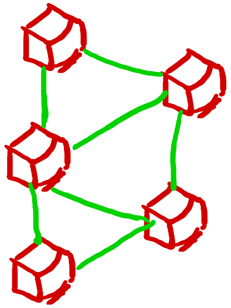


Pietrzak's VDF

Security:

Can construct VDF assuming hardness of iterated squaring in $\mathbb{G}$ and ideal hash fn. (ROM)

Applications:



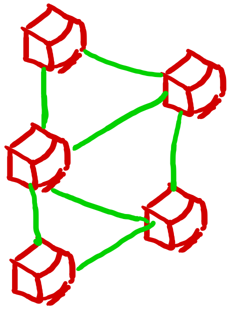
Chia network

Pietrzak's VDF

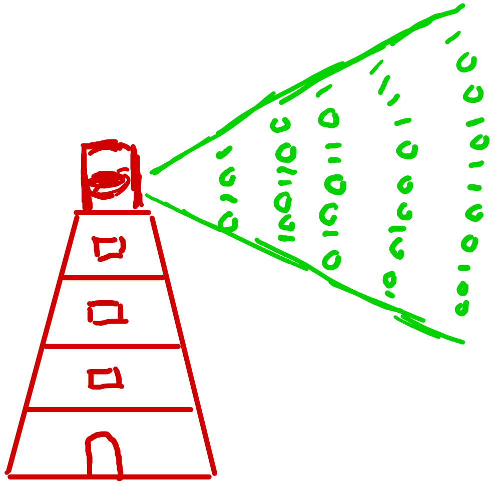
Security:

Can construct VDF assuming hardness of iterated squaring in $\mathbb{G}$ and ideal hash fn. (ROM)

Applications:



Chia network



RandRunner

Thank

You!

References

→ [S'79]: Shamir, Adi. "How to share a secret." Communications of the ACM 22.11 (1979): 612-613

→ [RSW'96]: Rivest, Ronald L., Adi Shamir, and David A. Wagner. "Time-lock puzzles and timed-release crypto." (1996)

→ [BBBF'18]: Boneh, Dan, et al. "Verifiable delay functions." Annual international cryptology conference. Cham: Springer Intl. Publishing, 2018

→ [Pie'18]: Pietrzak, Krzysztof. "Simple verifiable delay functions." 10th innovations in theoretical computer science conference (itcs 2019). Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2019

→ [FS'86]: Fiat, Amos and Adi Shamir. "How to prove yourself: Practical solutions to identification and signature schemes." Conference on the theory and application of cryptographic techniques. Springer Berlin Heidelberg, 1986.