Short-lived ZK Proofs and Signatures

Arasu Arun

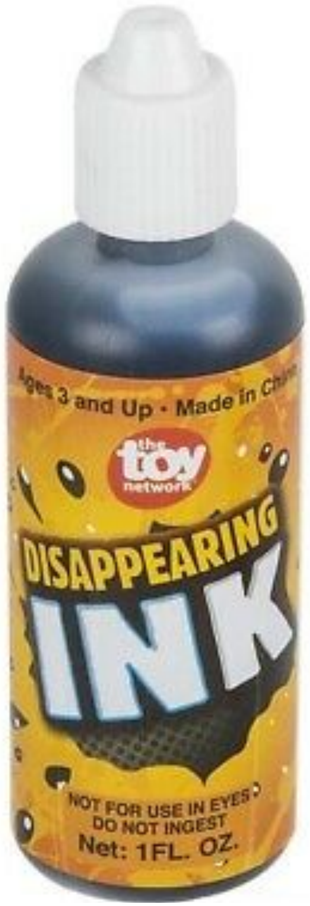
Joseph Bonneau

Jeremy Clark

NYU COURANT INSTITUTE OF MATHEMATICAL SCIENCES

UNIVERSITÉ Concordia UNIVERSITY

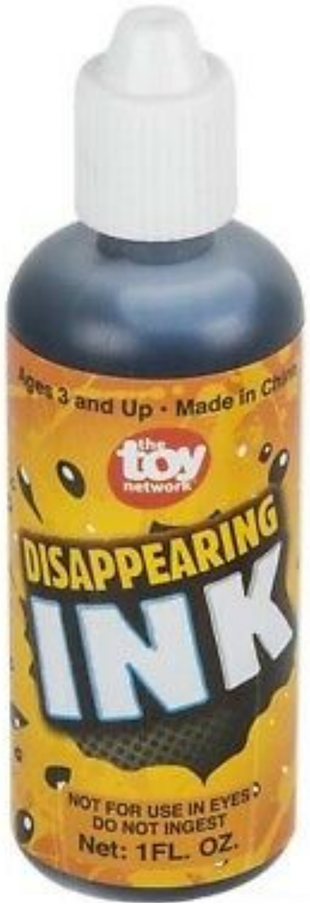
Goal: Proofs and signatures that naturally *disappear*



A signature on paper with **disappearing ink** will vanish after some time.

What's the cryptographic equivalent of disappearing?

<u>Goal</u>: Proofs and signatures that naturally *disappear*

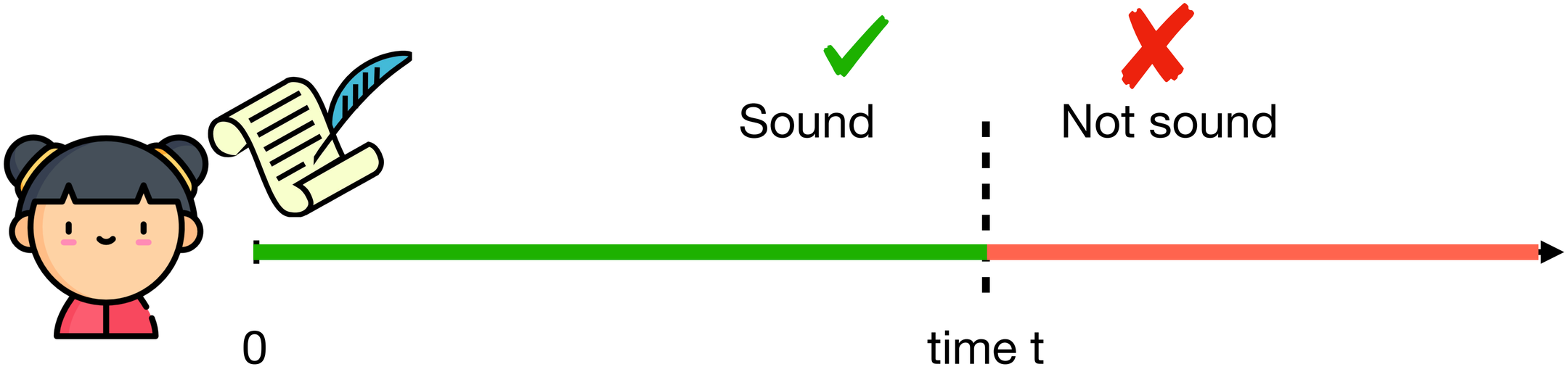


A signature on paper with **disappearing ink** will vanish after some time.

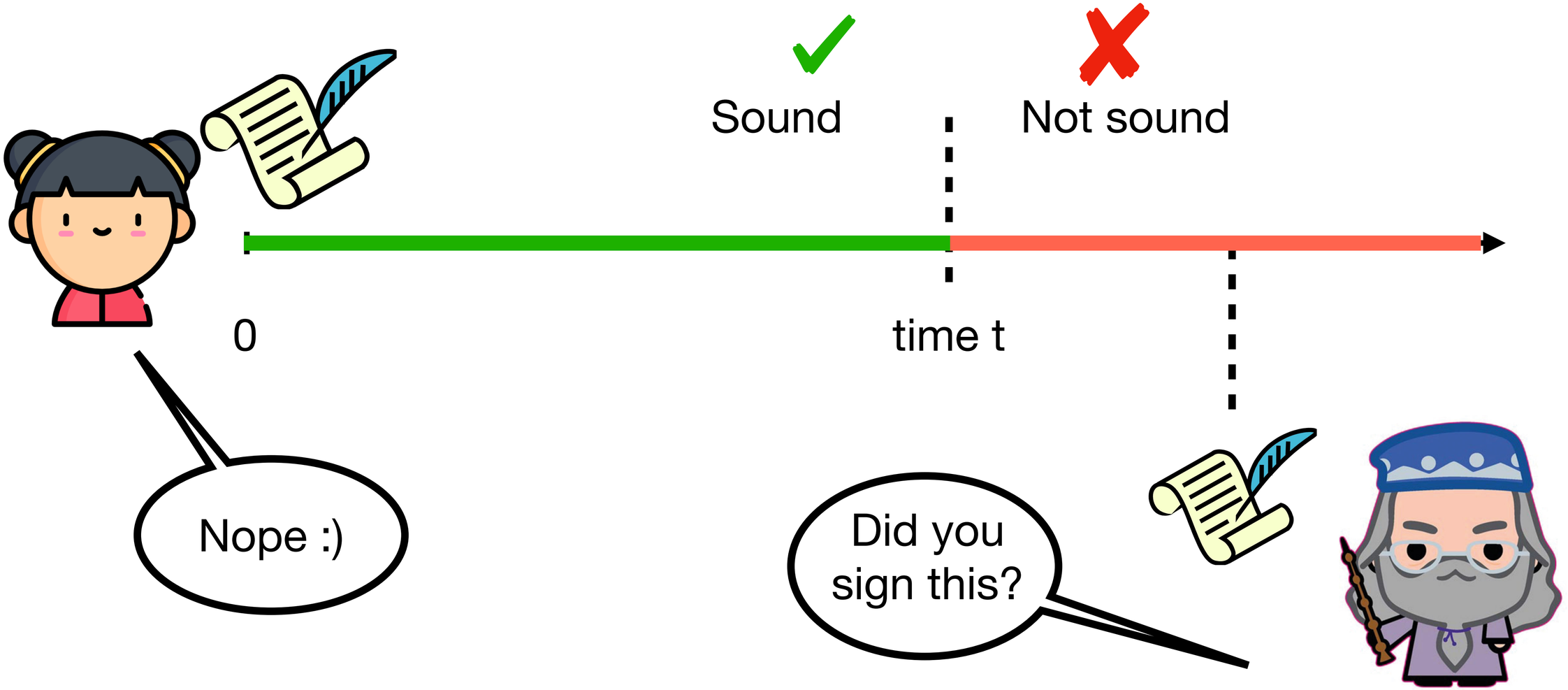
What's the cryptographic equivalent of disappearing?

- The **loss of soundness**.

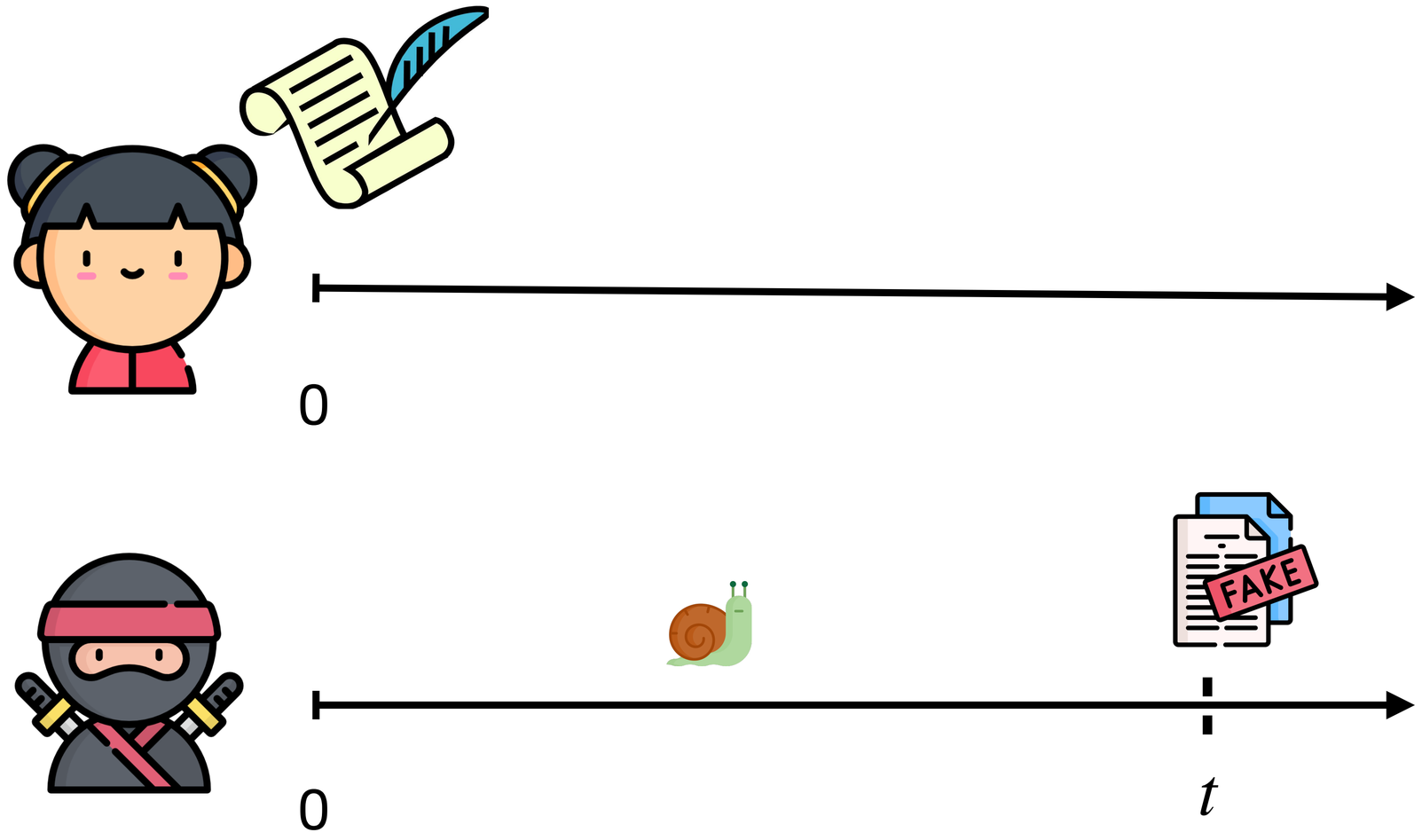
Loss of soundness over time



Loss of soundness over time + deniability



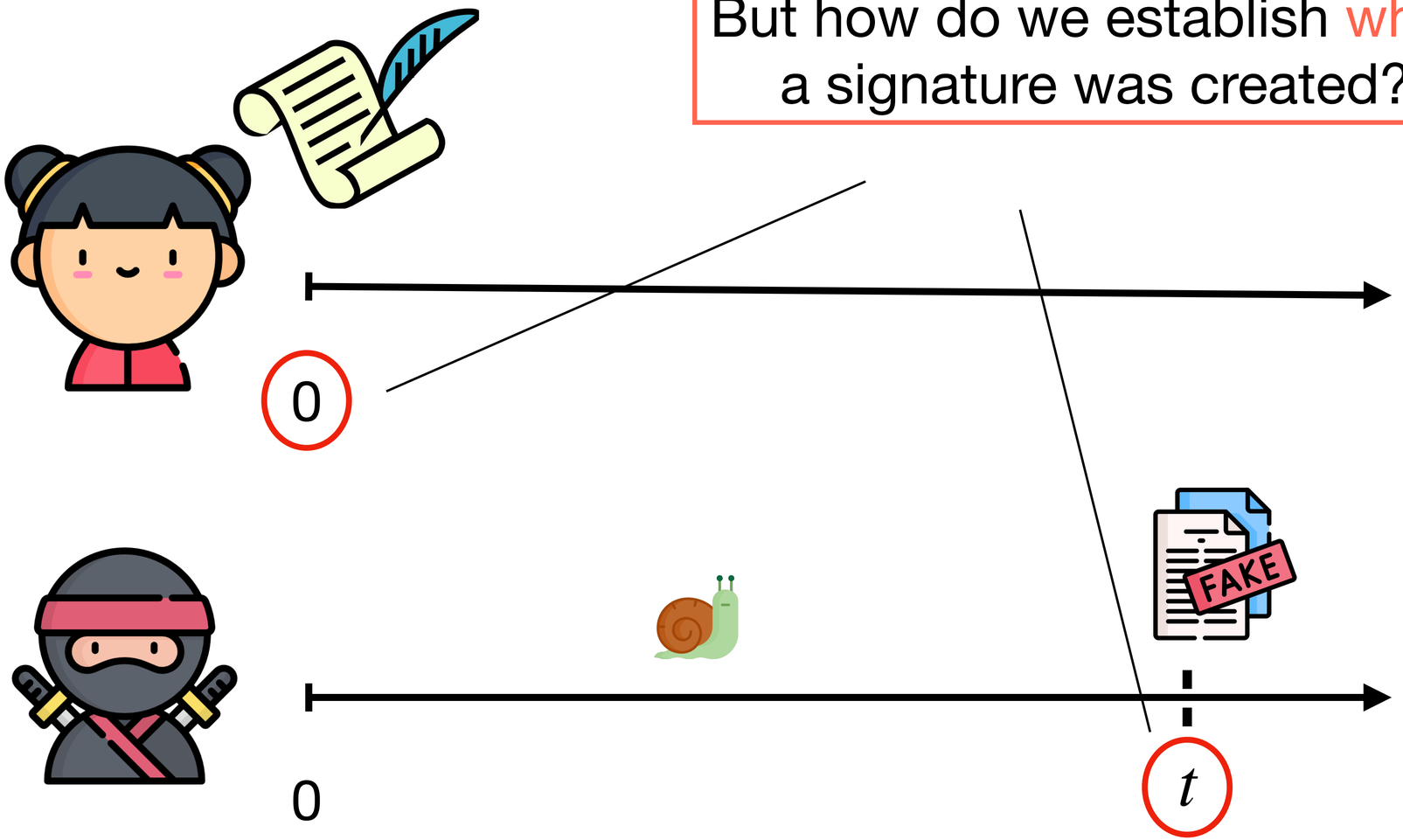
How? Allow *slow* forgeries



are indistinguishable

How? Allow *slow* forgeries

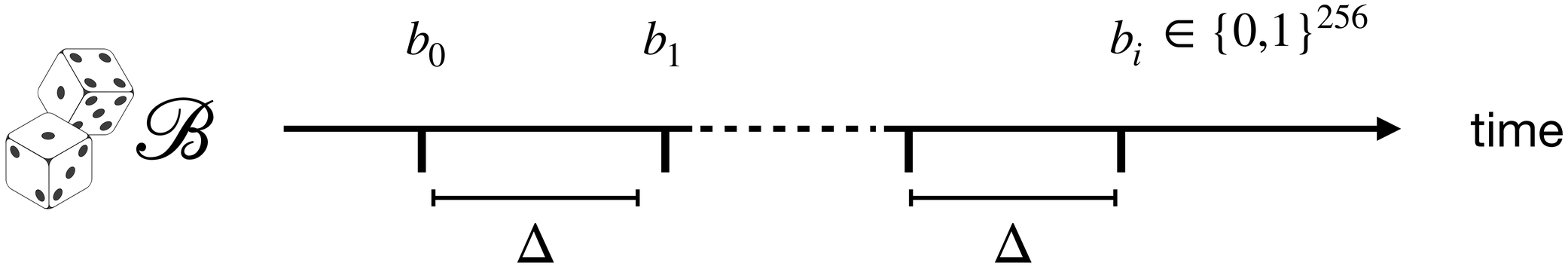
But how do we establish when a signature was created?



and

are indistinguishable

Beacons establish non-interactive time



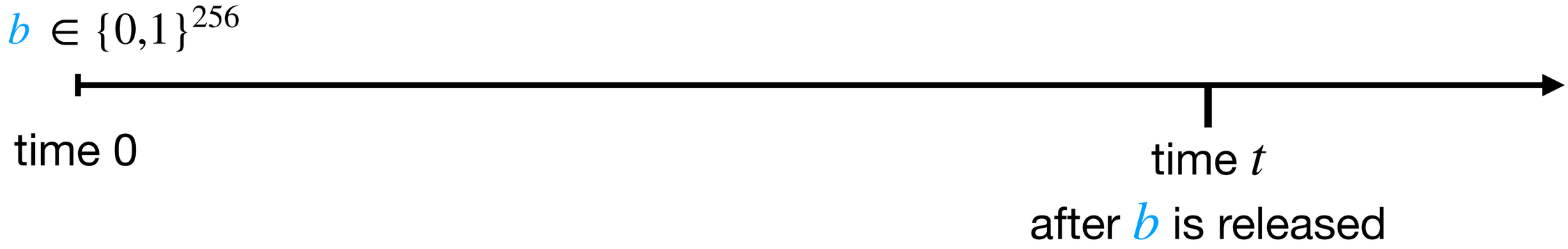
We assume a global beacon that periodically emits unpredictable randomness.

Possible beacons: Stock prices, blockchain blocks, distributed protocols…

drand

Use beacon values to specify expiration

The beacon $\mathscr{B}$
emits b at time 0.



Use beacon values to specify expiration

The beacon $\mathscr{B}$ emits b at time 0.

The latest beacon value b is specified in the signature.

Soundness is lost at time t regardless of when Alice signed.

$b \in \{0,1\}^{256}$

time 0

time t after b is released

In this talk

Applications: deniable email leaks and receipt-free voting

Tool: Verifiable Delay Functions (VDFs)

Constructions:

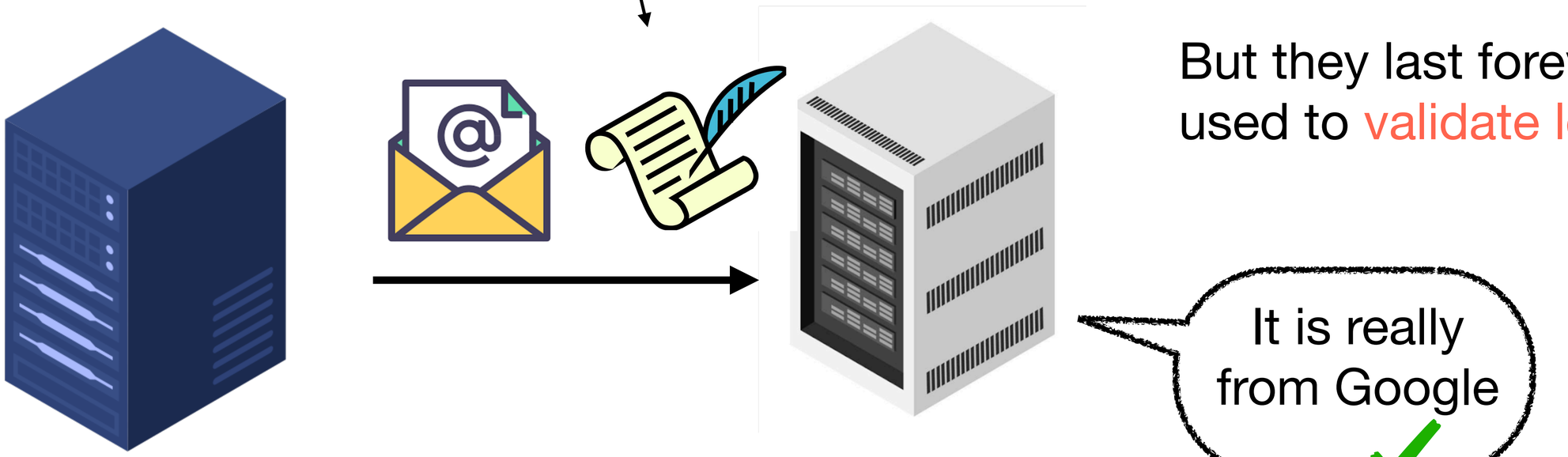
- Transforming NIZKs, ∑-protocols
- zkVDFs

Implementation

Mitigating Email Leaks

Email servers attach their signatures to out-going emails for authentication.

These signatures are only required during transfers.



But they last forever and are used to validate leaks.

Gmail

Yahoo

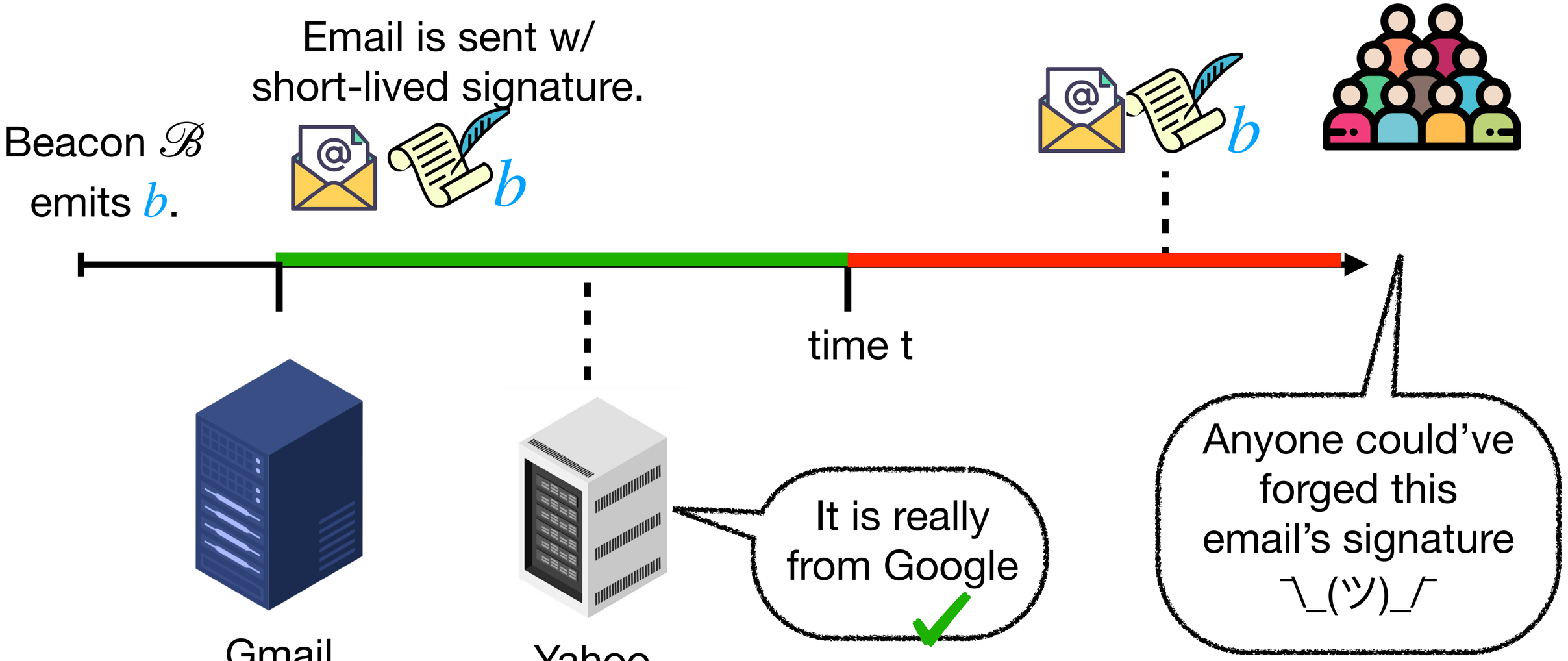
DKIM signatures validate email leaks

A email leaked by Wikileaks in 2016 that still has its signature from Google, thereby validating it.

One solution would be to have Google periodically refresh and leak keys.

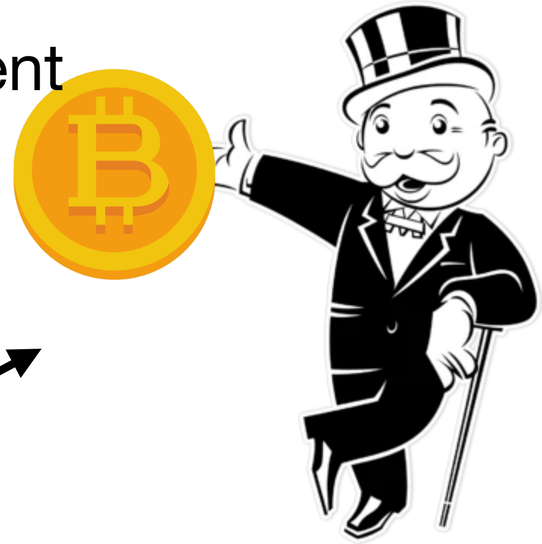
https://blog.cryptographyengineering.com/2020/11/16/ok-google-please-publish-your-dkim-secret-keys/

Short-lived sigs are a natural solution



Receipt-free Voting

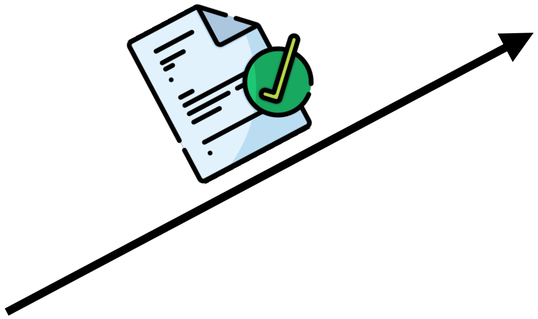
Short-lived proofs can prevent cash-for-votes.



Voting Booth

Vote

proof that your vote was included correctly



Receipt-free Voting

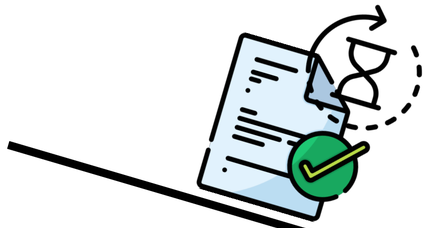
Short-lived proofs can prevent cash-for-votes.

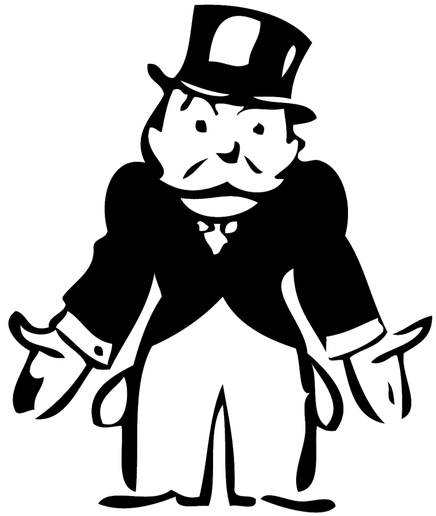
Voting Booth

Vote

proof that your vote was included correctly

The proof is no longer convincing!





Related Work

- “**Designated Verifier** Proofs” [JSI96]; “Chameleon Signatures” [KR00]
  - Designated verifier vs time-based deniability
- “**Timed** signatures” [BN00]; "Time-capsule signatures” [DY05]
  - Signatures are “locked” until time t; We solve the *inverse* problem.
- KeyForge and TimeForge [SPG21]
  - Require further actions from signer while ours is natural based on VDFs.
- Proof of Knowledge or Work [BKZZ16]
  - Based on proof-of-work, which can be parallelized (and thus, not time-based)

Formalizing short-lived proofs

- **Setup**$(\lambda, t) \rightarrow pp$
- **Prove**$(x, w, b) \rightarrow \pi$
- **Verify**$(x, b) \rightarrow \{0,1\}$
- **Forge**$(x, b) \rightarrow \pi$

Completeness: Prove runs in time $o(t)$

t-Forgeability: Forge runs in time $[t, (1+\epsilon)t]$

Indistinguishability:

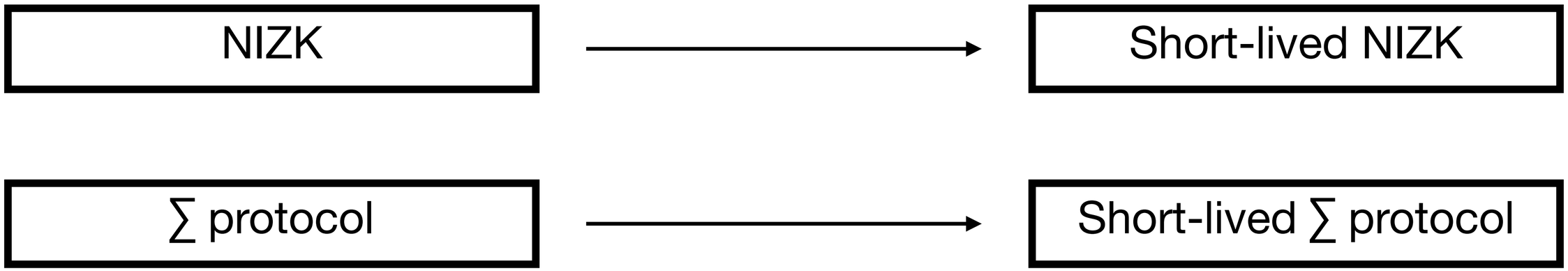
$\forall\, x, w$: $\{\textbf{Prove}(x, w, b)\} \approx \{\textbf{Forge}(x, b)\}$

ZK is implied as **Forge** is also a simulator!

t-Soundness: You can extract a witness from a prover that produces proofs in time less than t after the beacon emits b.

Constructions via transformations

Transformations from generic NIZK and ∑-protocols to short-lived versions.



In our paper: 4 transformations for proofs and 2 more for signatures.

Our construction use Verifiable Delay Functions (VDFs).

Tool: **Verifiable Delay Functions (VDFs)**

- **Setup**$(\lambda, t) \rightarrow pp$
- **Eval**$(x) \rightarrow y, \pi$ — time $O(t)$
- **Verify**$(x, y, \pi) \rightarrow \{0,1\}$ — fast

VDF security property: Given a random input x, it's hard any to convince the verifier in time less than t (even when given polynomial-time pre-computation).

Tool: **Verifiable Delay Functions (VDFs)**

- **Setup**$(\lambda, t) \rightarrow pp$
- **Eval**$(x) \rightarrow y, \pi$ — time $O(t)$
- **Verify**$(x, y, \pi) \rightarrow \{0,1\}$ — fast

0

Eval

$y = x^{2^t}$

y, proof π

Repeated Squaring VDFs:

$x \in \mathbb{G}$ of unknown order

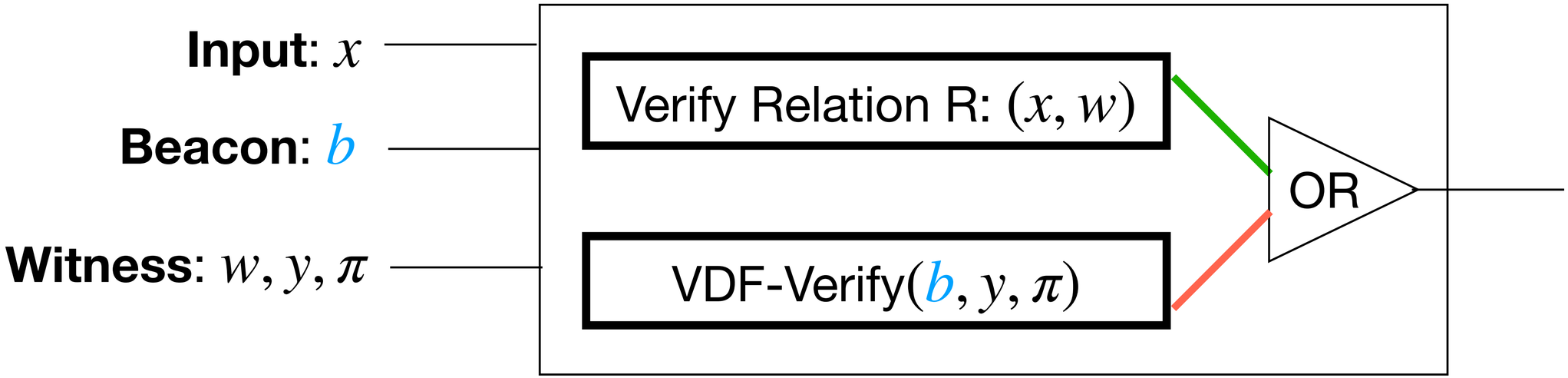
(eg: RSA2048 groups)

VDF security property: Given a random input x, it's hard any to convince the verifier in time less than t (even when given polynomial-time pre-computation).

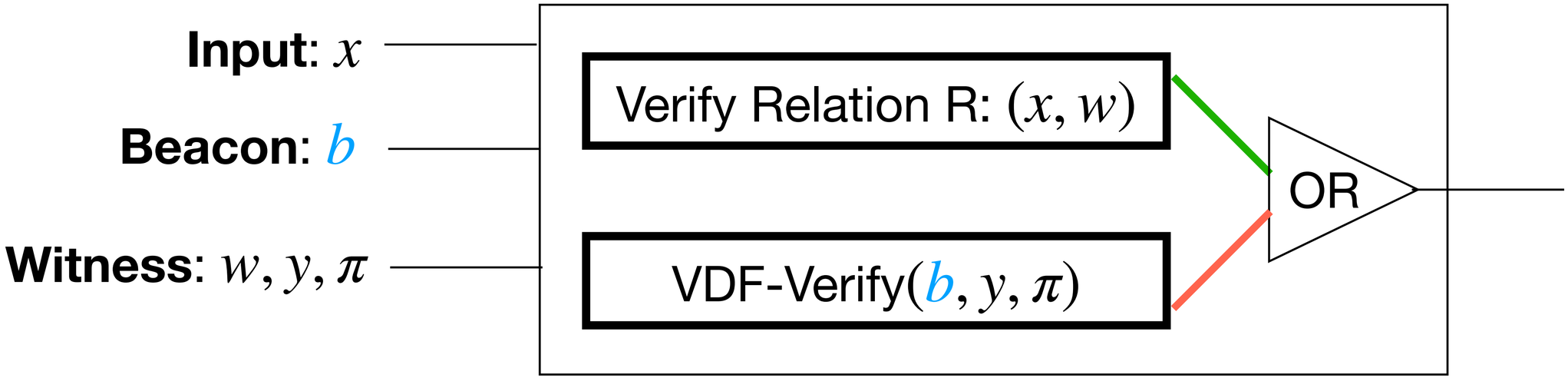
Takes t steps using the repeated squaring algorithm.

Eg: Wesolowski and Pietrzak VDFs

Generic Construction: NIZK $\vee$ VDF

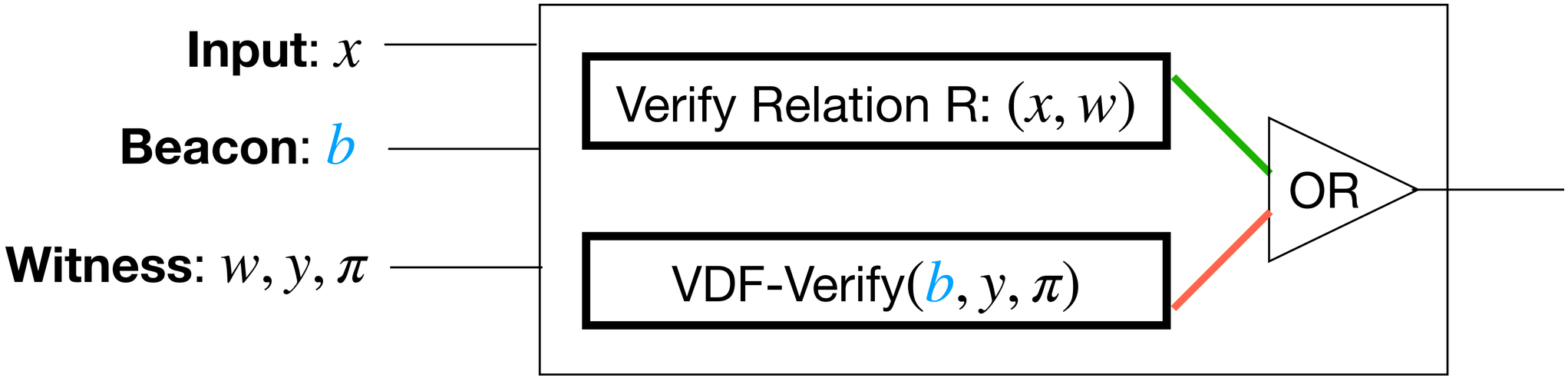


Generic Construction: NIZK $\vee$ VDF



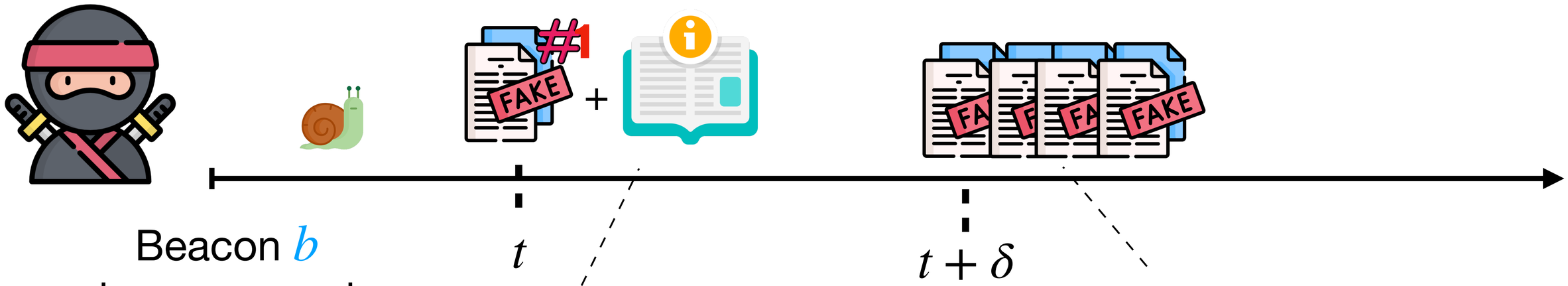
- **Completeness**: Honest prover can satisfy the original statement
- **t-Forgeability**: Forger can compute VDF-Eval$(b) = y, \pi$
  - takes t steps from when b was emitted

Generic Construction: NIZK $\vee$ VDF



- **Completeness**: Honest prover can satisfy the original statement
- **t-Forgeability**: Forger can compute VDF-Eval$(b) = y, \pi$
  - takes t steps from when b was emitted
- **Indistinguishability** & **t-Soundness**: reduces to NIZK $\wedge$ VDF security

Stronger notion: Reusable Forgeability



: Advice string for b

With reusable forgeability, only **one long computation** per beacon value b required! Advice now helps forge

- signatures of *any key*
- and proofs of *any statement*.

Background: Σ-protocols

Relation R, **input**: x

Prove:

1. Compute and send a
2. Receive challenge c
3. Compute response z

Witness: w

a

challenge c

z

Verify(x, a, c, z)

$\exists$ a simulator such that **Simulate**$(x) = a', c', z'$ such that:

(1) Verify passes (2) $(a', c', z') \approx (a, c, z)$ from an honest execution

Making ∑ protocols short-lived

$\sum_R$ for relation R	**Input**: x	**Proof**: a, z, c

Making Σ protocols short-lived

$\sum_R$ for relation R	**Input**: x	**Proof**: a, z, c
Short-lived $\sum_R$ ⧖	**Input**: x **Beacon**: b	**Proof**: a, z, c_1, c_2

Prover splits $c = c_1 \oplus c_2$

Making ∑ protocols short-lived

$\sum_R$ for relation R	**Input**: x	**Proof**: a, z, c
Short-lived $\sum_R$ ⧖	**Input**: x **Beacon**: b	**Proof**: a, z, c_1, c_2, y, π

Prover splits $c = c_1 \oplus c_2$

Verifier checks: $\sum_R.\text{Verify}(a, z, c_1) \bigwedge \text{VDF-Verify}(c_2 \oplus b, y, \pi)$

Cheat on $\sum_R$ by simulating it

Cheat on VDF by pre-computing it

Making ∑ protocols short-lived

Doesn't have reusable forgeability.

$\sum_R$ for relation R	**Input**: x	**Proof**: a, z, c
Short-lived $\sum_R$ ⏳	**Input**: x **Beacon**: b	**Proof**: a, z, c_1, c_2, y, π

Prover splits $c = c_1 \oplus c_2$

Verifier checks: $\sum_R.\text{Verify}(a, z, c_1) \bigwedge \text{VDF-Verify}(c_2 \oplus b, y, \pi)$

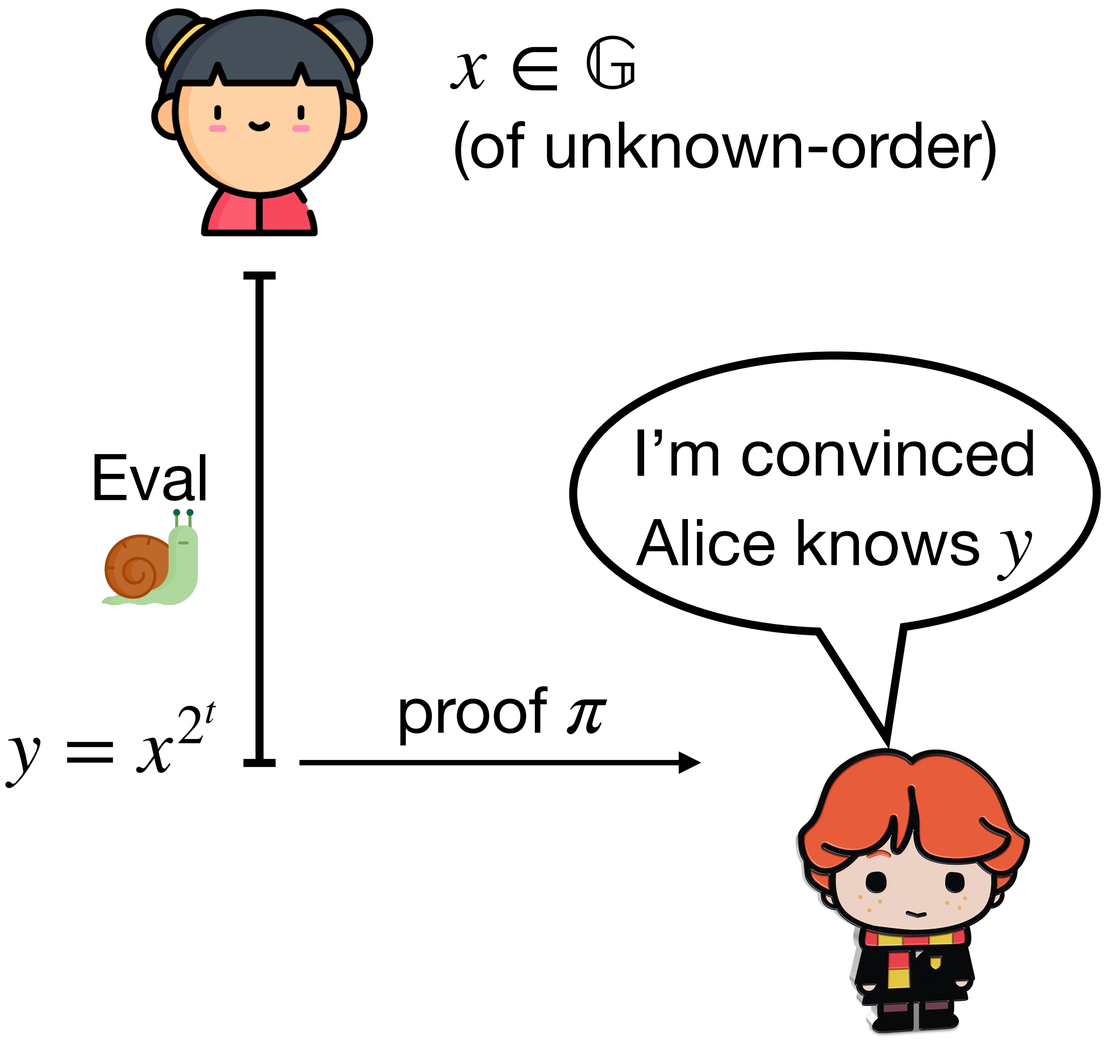
Cheat on $\sum_R$ by simulating it

Cheat on VDF by pre-computing it

Zero-Knowledge VDFs (zkVDFs)

Prove knowledge of a VDF output *without* revealing it.

- zkVDF-Eval$(x) \rightarrow y, \pi$
- zkVDF-Verify$(x, \pi) \rightarrow \{0,1\}$



Zero-Knowledge VDFs (zkVDFs)

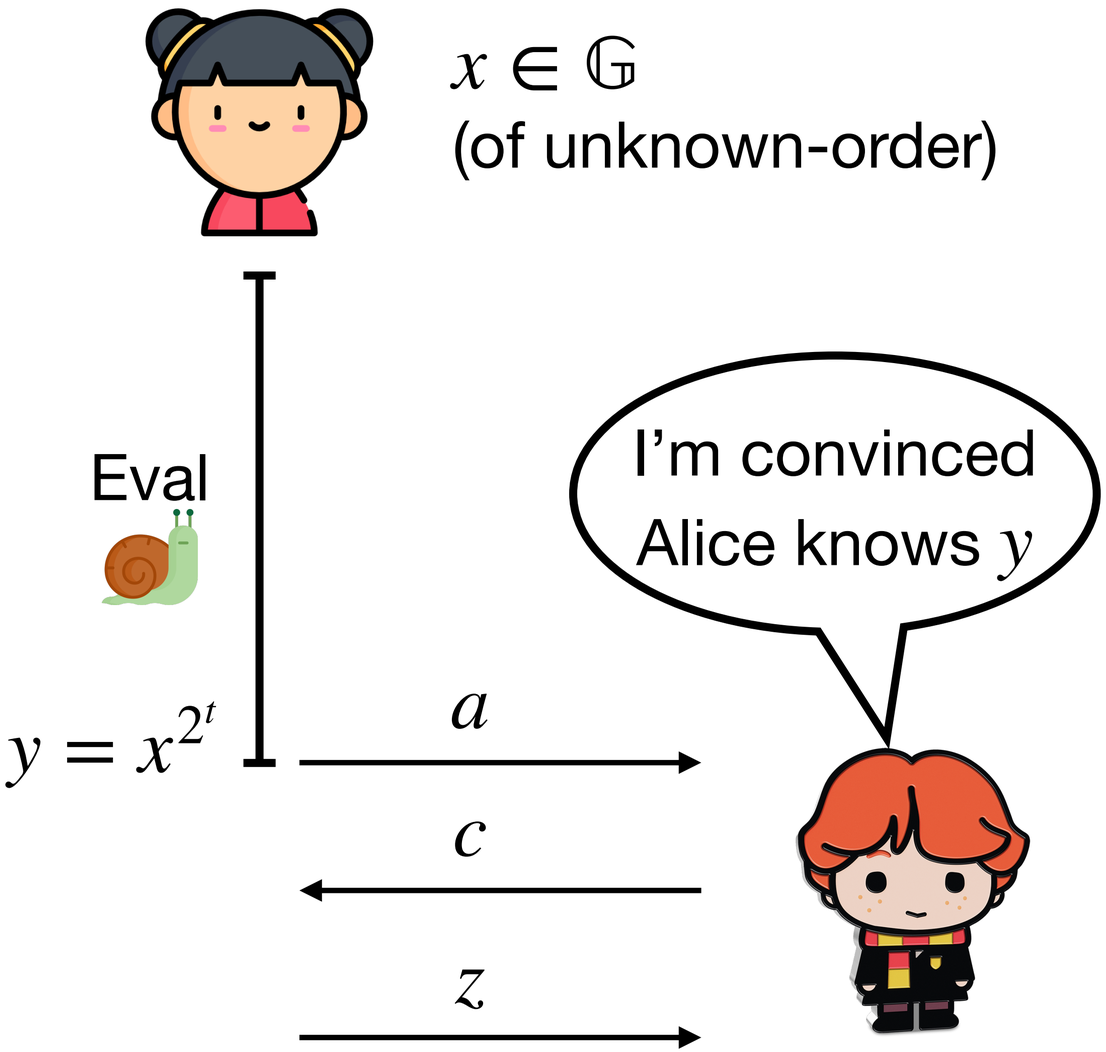
Prove knowledge of a VDF output *without* revealing it.

- zkVDF-Eval$(x) \rightarrow y, \pi$
- zkVDF-Verify$(x, \pi) \rightarrow \{0,1\}$

zkVDFs can be written as a Σ-protocol with an efficient **simulator**.

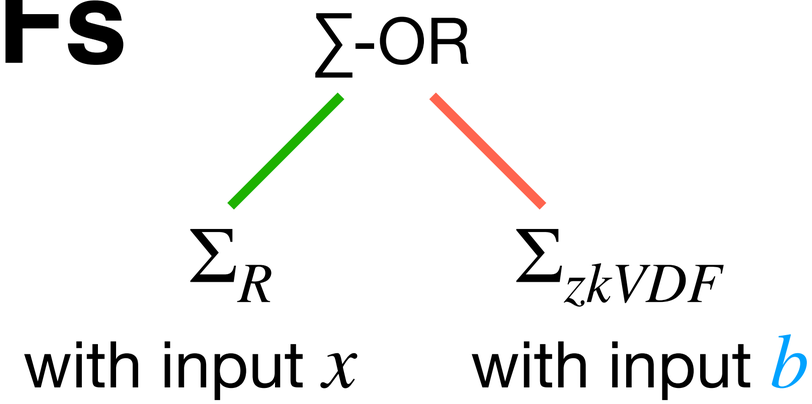
- Simulate$(x) = a', c', z'$ w/o knowledge of y!

Constructions are based on PoKE from BBF19.



Construction 2: Using zkVDFs

The classic ∑-OR composition works now.

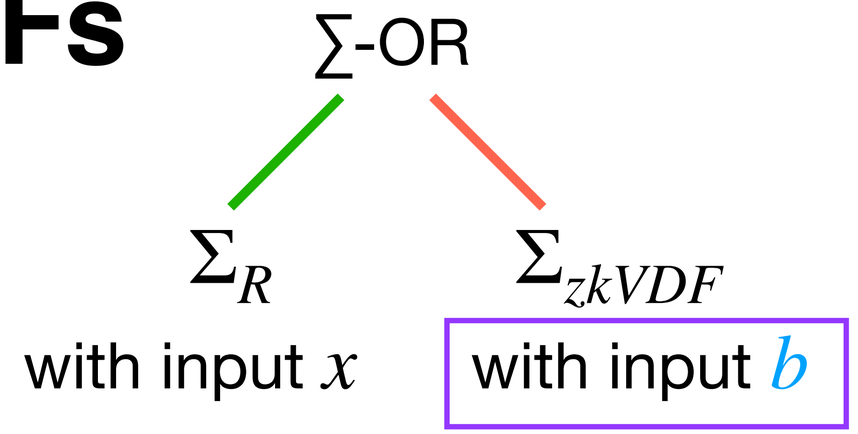


	Prove(x, w, b): 	**Forge**(x, b): 
1. Simulate	Simulate Σ_{zkVDF} w/o y	Simulate Σ_R w/o witness
2. Honestly compute	Run Σ_R using witness w	Run Σ_{zkVDF} slowly 

No more pre-computation; Supports reusable forgeability!

Construction 2: Using zkVDFs

The classic ∑-OR composition works now.



	Prove(x, w, b): 	**Forge**(x, b): 
1. Simulate	Simulate Σ_{zkVDF} w/o y	Simulate Σ_R w/o witness
2. Honestly compute	Run Σ_R using witness w	Run Σ_{zkVDF} slowly 

No more pre-computation; Supports reusable forgeability!

Other constructions

Short-lived Proofs

- From generic NIZK *
- Σ with precomputation
- Σ with rrVDF
- Σ-OR with zkVDF *

Short-lived Signatures

Any proof construction can double as a signature scheme.

More efficient constructions:

- Sign - trapdoor VDFs
- Sign - watermarkable VDFs *

* has reusable forgeability

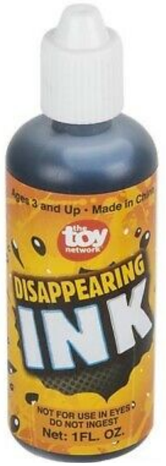
Implementation

Using Wesolowski VDFs with RSA2048 group

SCHEME	SIZE OVERHEAD	PROVER TIME OVERHEAD
SNARK-OR w/ Groth16 *	0	~ 60 sec
∑-precomp	544 Bytes	precomp = O(t) + online = O(1)
∑-rrVDF	544 Bytes	time-space tradeoff: O(1) -- O(t)
∑-zkVDF *	576 Bytes	120 ms
RSA Sig - trapdoor VDF	288 Bytes	10 ms
RSA Sig - watermark VDF *	288 Bytes	10 ms

* has reusable forgeability

Conclusion



- Short-lived zk proofs and signatures ***naturally*** **lose soundness** after some time.

- Achieved by allowing slow forgeries that require evaluating a VDF.

- Can be used to design deniable messaging and email to mitigates the effects of leaks; and to design receipt-free e-voting schemes.
- Formalize re-randomizable VDFs and introduce zkVDFs.
- Making ∑-protocols and RSA signatures short-lived is practical!

Backup Slides

Short-lived ∑-protocols w/ pre-computation

Input: x

Beacon: b

Proof: a, z, c_1, c_2, y, π

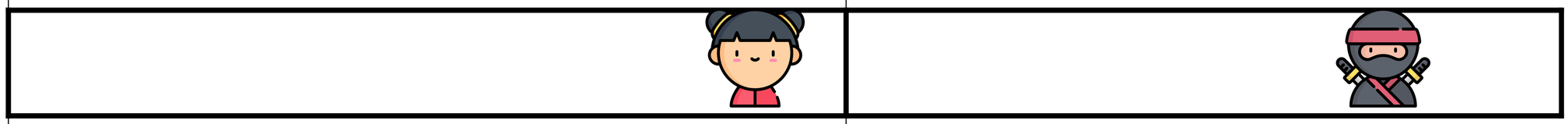
$\sum_R.\text{Verify}(a, z, c_1) \bigwedge \text{VDF-Verify}(c_2 \oplus b, y, \pi)$

0. Pre-compute

1. Simulate

2. Fix challenge

3. Honest step



Short-lived ∑-protocols w/ pre-computation

Input: x

Beacon: b

Proof: a, z, c_1, c_2, y, π

$\sum_R.\text{Verify}(a, z, c_1) \bigwedge \text{VDF-Verify}(c_2 \oplus b, y, \pi)$

0. Pre-compute	Pre-compute tuple $y_*, \pi_* = \text{VDF-Eval}(x_*)$	–
	Prove$(x, b,$ witness: $w)$ 	
1. Simulate	–	
2. Fix challenge	Fix $c_2 = x_* \oplus b \implies c_1$ is random	
3. Honest step	Respond to $\sum_R$ honestly using c_1 with w	

Output proof: $a, z, c_1, c_2, y_*, \pi_*$

Short-lived ∑-protocols w/ pre-computation

Input: x

Beacon: b

Proof: a, z, c_1, c_2, y, π

$\sum_R.\text{Verify}(a, z, c_1) \bigwedge \text{VDF-Verify}(c_2 \oplus b, y, \pi)$

0. Pre-compute	Pre-compute tuple $y_*, \pi_* = \text{VDF-Eval}(x_*)$	–
	Prove$(x, b,$ witness: $w)$ 	**Forge**(x, b) 
1. Simulate	–	Simulate $\sum_R$ to get (a', z', c_1')
2. Fix challenge	Fix $c_2 = x_* \oplus b \implies c_1$ is random	Use above $c_1 \implies c_2$ is random
3. Honest step	Respond to $\sum_R$ honestly using c_1 with w	Compute VDF-Eval$(c_2 \oplus b)$ honestly
	Output proof: $a, z, c_1, c_2, y_*, \pi_*$	**Output proof**: $a', z', c_1', c_2, y, \pi$

Reusing pre-computation w/ rrVDFs

Proof 1: $a, z, c_1, c_2, y_*, \pi_*$

Proof 2: $a', z', c_1', c_2', y_*, \pi_*$

Pre-computation cannot be re-used as it loses deniability.

Reusing pre-computation w/ rrVDFs

Proof 1: $a, z, c_1, c_2, y_*, \pi_*$

Proof 2: $a', z', c_1', c_2', y_*, \pi_*$

Pre-computation cannot be re-used as it loses deniability.

rrVDFs support a "randomize" operation:

Randomize$(x, y, \pi) \rightarrow x', y', \pi'$

- Much faster than Eval
- x, y, π is valid $\Longrightarrow$ x', y', π' is valid

Wesolowski and Pietrzak are rrVDFs involving a **time-space tradeoff.**

Reusing pre-computation w/ rrVDFs

Proof 1: $a, z, c_1, c_2, y_*, \pi_*$

Proof 2: $a', z', c_1', c_2', y_*, \pi_*$

Pre-computation cannot be re-used as it loses deniability.

rrVDFs support a "randomize" operation:

Randomize$(x, y, \pi) \rightarrow x', y', \pi'$

- Much faster than Eval
- x, y, π is valid $\Longrightarrow$ x', y', π' is valid

0. Precompute

Pre-compute tuple $y_*, \pi_* = \text{VDF-Eval}(x_*)$

Prove$(x, b,$ witness: $w, (x_*, y_*, \pi_*))$

1. Randomize

$(x, y, \pi) \leftarrow$ **Randomize**(x_*, y_*, π_*)

2. Fix challenge

Fix $c_2 = x \oplus b \Longrightarrow c_1$ is random

3. Honest step

Respond to $\sum_R$ honestly using c_1

Wesolowski and Pietrzak are rrVDFs involving a **time-space tradeoff.**