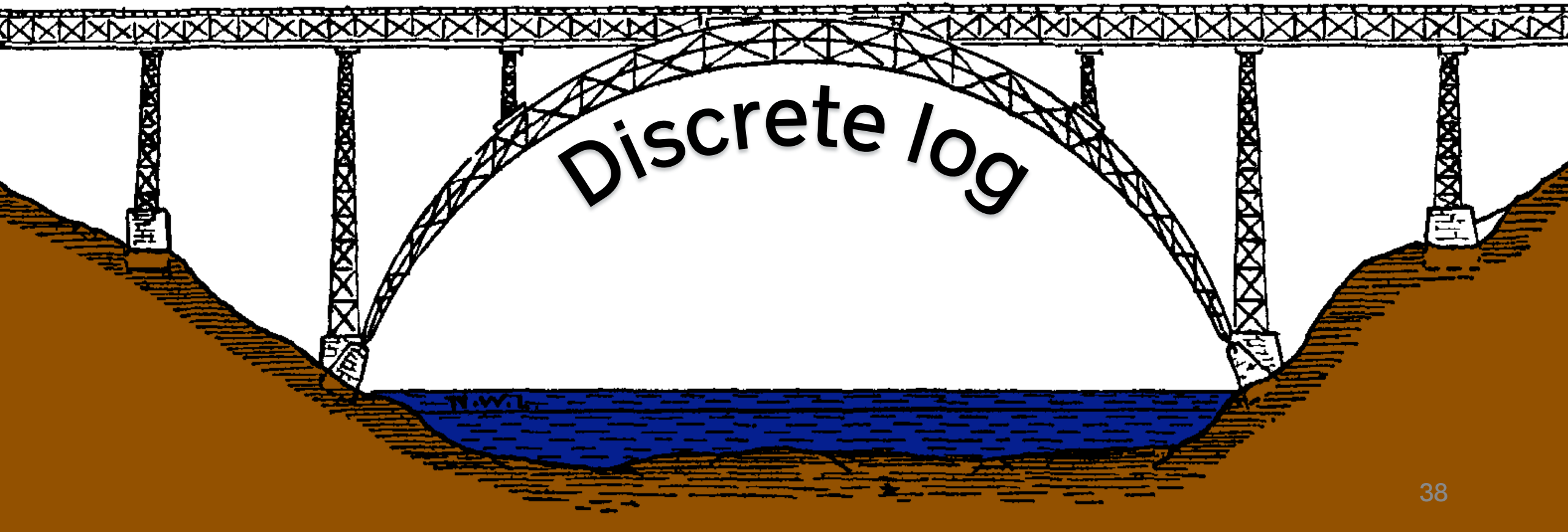
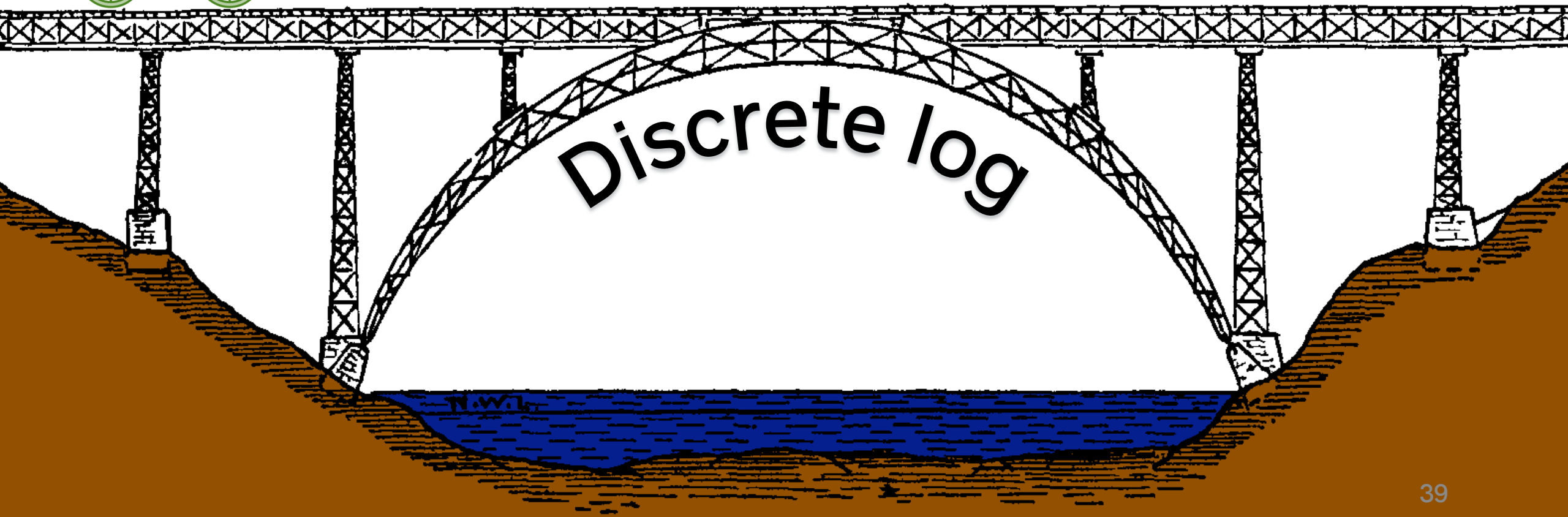
The Discrete Logarithm Problem with Preprocessing

Henry Corrigan-Gibbs and Dmitry Kogan
Stanford University

Eurocrypt – 1 May 2018
Tel Aviv, Israel

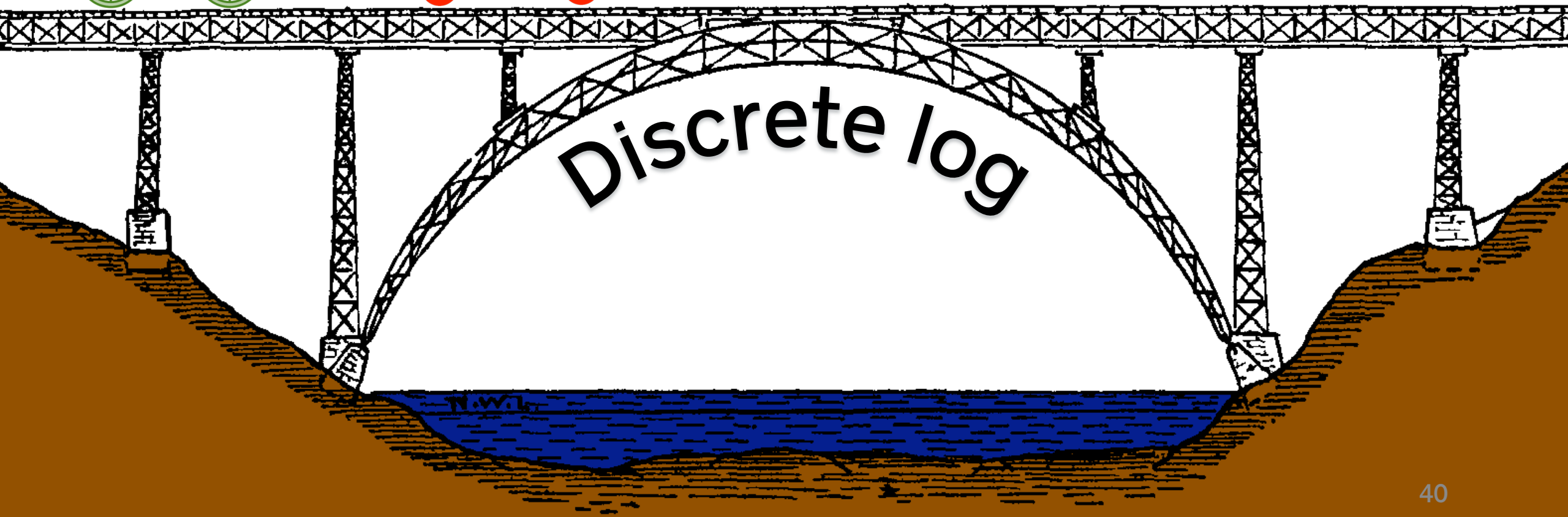

Discrete log

Signatures
(DSA and Schnorr)



Signatures
(DSA and Schnorr)

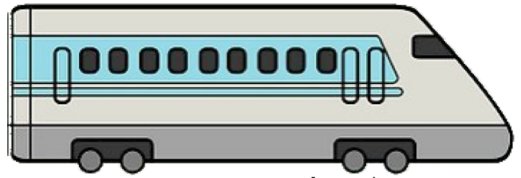
DH key
exchange

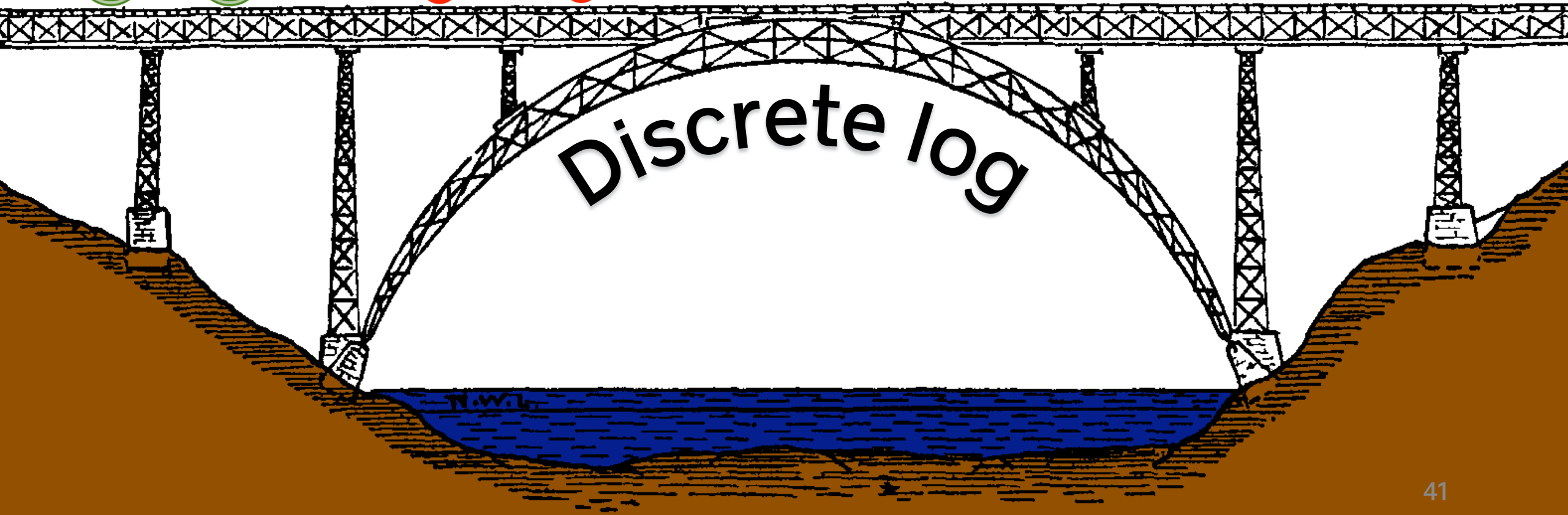


Signatures
(DSA and Schnorr)

DH key exchange

DDH

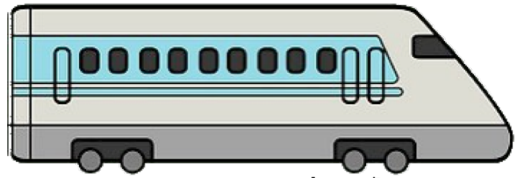




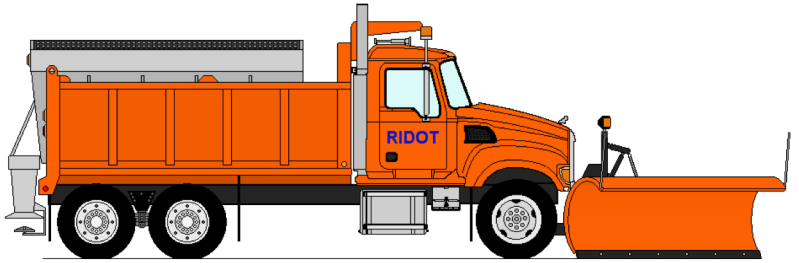
Signatures
(DSA and Schnorr)

DH key exchange

DDH



Pairings



Discrete log

The discrete-log problem

Group: $\mathbb{G} = \langle g \rangle$
of prime order N

Instance: $g^x \in \mathbb{G}$

Adversary $\mathcal{A}$

Solution:
$x \in \mathbb{Z}_N$

The discrete-log problem

Group: $\mathbb{G} = \langle g \rangle$ of prime order N

Instance: $g^x \in \mathbb{G}$

Adversary $\mathcal{A}$

Solution: $x \in \mathbb{Z}_N$

Why do we believe this problem is hard?

Generic lower bounds give us confidence

Theorem. [Shoup'97] Every **generic** discrete-log algorithm that

- operates in a group of prime order N and
- succeeds with probability at least ½

must run in time $\Omega(N^{1/2})$.

Generic lower bounds give us confidence

Theorem. [Shoup'97] Every **generic** discrete-log algorithm that

- operates in a group of prime order N and
- succeeds with probability at least ½

must run in time $\Omega(N^{1/2})$.

Generic attack in 256-bit group takes $\approx 2^{128}$ time.

Generic lower bounds give us confidence

Theorem. [Shoup'97] Every **generic** discrete-log algorithm that

- operates in a group of prime order N and
- succeeds with probability at least ½

must run in time $\Omega(N^{1/2})$.

Generic attack in 256-bit group takes $\approx 2^{128}$ time.

Best attacks on standard EC groups are generic

Generic algorithms can only make "black-box" use of the group operation

Generic-group model:

- Group is defined by an injective **"labeling" function**

$$\sigma\colon\ \mathbb{Z}_N \to \{0,1\}^*$$

- Algorithm has access to a **group-operation oracle:**

$$\mathcal{O}_\sigma\big(\sigma(i),\sigma(j)\big) \ \mapsto\ \ \sigma(i+j)$$

[Nechaev'94], [Shoup'97], [Maurer'05]

Generic dlog algorithm takes as input $\big(\sigma(1),\sigma(x)\big)$, representing (g, g^x), make queries to $\mathcal{O}_\sigma$, outputs x.

[Measure running time by query complexity]

Generic algorithms can only make "black-box" use of the group operation

Generic-group model:

- Group is defined by an injective **"labeling" function**

$$\sigma\colon\ \mathbb{Z}_N \to \{0,1\}^*$$

- Algorithm has access to a **group-operation oracle:**

$$\mathcal{O}_\sigma\big(\sigma(i),\sigma(j)\big) \ \mapsto\ \ \sigma(i+j)$$

Generic dlog algorithm takes as input $\big(\sigma(1$
(g, g^x), make queries to $\mathcal{O}_\sigma$, outputs x.

[Measure running time by query comple

Very useful way to understand hardness

[BB04,B05,M05,D06, B08,Y15,...]

Existing generic lower bounds do not account for preprocessing

- Premise of generic-group model: the adversary **knows nothing** about the structure of the group $\mathbb{G}$ in advance
- In reality: the adversary knows a lot about $\mathbb{G}$!
  - $\mathbb{G}$ is one of a small number of groups: NIST P-256, Curve25519, …
- A realistic adversary can perform $\mathbb{G}$-specific preprocessing!
- Existing generic-group lower bounds say nothing about preprocessing attacks! [H80, Yao90, FN91, …]

Existing generic lower bounds do not account for preprocessing

- Premise of generic-group model: the adversary **knows nothing** about the structure of the group $\mathbb{G}$ in advance
- In reality: the adversary knows a lot about $\mathbb{G}$!
  - ➢ $\mathbb{G}$ is one of a small number of groups: NIST P-256, Curve25519, …
- A realistic adversary can perform $\mathbb{G}$-specific preprocessing!
- Existing generic-group lower bounds say nothing about preprocessing attacks! [H80, Yao90, FN91, …]

Existing generic lower bounds do not account for preprocessing

- Premise of generic-group model: the adversary **knows nothing** about the structure of the group $\mathbb{G}$ in advance
- In reality: the adversary knows a lot about $\mathbb{G}$!
  - $\mathbb{G}$ is one of a small number of groups: NIST P-256, Curve25519, …
- A realistic adversary can perform $\mathbb{G}$-specific preprocessing!
- Existing generic-group lower bounds say nothing about preprocessing attacks! [H80, Yao90, FN91, …]

Existing generic lower bounds do not account for preprocessing

- Premise of generic-group model: the adversary **knows nothing** about the structure of the group $\mathbb{G}$ in advance
- In reality: the adversary knows a lot about $\mathbb{G}$!
  - $\mathbb{G}$ is one of a small number of groups: NIST P-256, Curve25519, …
- A realistic adversary can perform $\mathbb{G}$-specific preprocessing!
- Existing generic-group lower bounds say <u>nothing</u> about preprocessing attacks! [H80, Yao90, FN91, …]

Preprocessing phase

Group: $\mathbb{G} = \langle g \rangle$

$\mathcal{A}_0$

Advice: $st_{\mathbb{G}}$

Online phase

Instance: $g^x \in \mathbb{G}$

$\mathcal{A}_1$

Solution: $x \in \mathbb{Z}_N$

Initiated by Hellman (1980) in context of OWFs

Preprocessing phase

Group: $\mathbb{G} = \langle g \rangle$

$\mathcal{A}_0$

Advice: $st_{\mathbb{G}}$

Both algorithms are generic!

Online phase

Instance: $g^x \in \mathbb{G}$

$\mathcal{A}_1$

Solution: $x \in \mathbb{Z}_N$

Initiated by Hellman (1980) in context of OWFs

Preprocessing phase

Group: $\mathbb{G} = \langle g \rangle$

$\mathcal{A}_0$

Advice: $st_{\mathbb{G}}$

Online phase

Instance: $g^x \in \mathbb{G}$

$\mathcal{A}_1$

Solution: $x \in \mathbb{Z}_N$

Initiated by Hellman (1980) in context of OWFs

Preprocessing phase

Preprocessing time P

Group: $\mathbb{G} = \langle g \rangle$

Advice: $st_{\mathbb{G}}$

$\mathcal{A}_0$

Online phase

Instance: $g^x \in \mathbb{G}$

Solution: $x \in \mathbb{Z}_N$

$\mathcal{A}_1$

Initiated by Hellman (1980) in context of OWFs

Preprocessing phase

Preprocessing time P

Group: $\mathbb{G} = \langle g \rangle$

$\mathcal{A}_0$

Advice: $st_{\mathbb{G}}$

Advice size S

Online phase

Instance: $g^x \in \mathbb{G}$

$\mathcal{A}_1$

Solution: $x \in \mathbb{Z}_N$

Initiated by Hellman (1980) in context of OWFs

Preprocessing phase

Preprocessing time P

Group: $\mathbb{G} = \langle g \rangle$

Advice: $st_{\mathbb{G}}$

$\mathcal{A}_0$

Advice size S

Online phase

Online time T

Instance: $g^x \in \mathbb{G}$

Solution: $x \in \mathbb{Z}_N$

$\mathcal{A}_1$

Initiated by Hellman (1980) in context of OWFs

Preprocessing phase

Preprocessing time P

Group: $\mathbb{G} = \langle g \rangle$

Advice: $st_{\mathbb{G}}$

$\mathcal{A}_0$

Advice size S

Online phase

Online time T

Instance: $g^x \in \mathbb{G}$

Solution: $x \in \mathbb{Z}_N$

$\mathcal{A}_1$

Success prob. ϵ

Initiated by Hellman (1980) in context of OWFs

Rest of this talk

Background: Preprocessing attacks are relevant

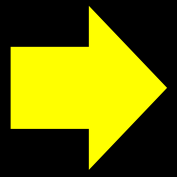
- Preexisting $S = T = \tilde{O}(N^{1/3})$ generic attack on discrete log

Our results: Preprocessing lower-bounds and attacks

- The $\tilde{O}(N^{1/3})$ generic dlog attack is optimal
- Any such attack must use lots of preprocessing: $\Omega(N^{2/3})$
- New $\tilde{O}(N^{1/5})$ preprocessing attack on DDH-like problem

Open questions

Rest of this talk



Background: Preprocessing attacks are relevant

- Preexisting $S = T = \tilde{O}(N^{1/3})$ generic attack on discrete log

Our results: Preprocessing lower-bounds and attacks

- The $\tilde{O}(N^{1/3})$ generic dlog attack is optimal
- Any such attack must use lots of preprocessing: $\Omega(N^{2/3})$
- New $\tilde{O}(N^{1/5})$ preprocessing attack on DDH-like problem

Open questions

A preexisting result...

Theorem. [Mihalcik 2010] [Lee, Cheon, Hong 2011] [Bernstein and Lange 2013]

There is a generic dlog algorithm with preprocessing that:

- uses S bits of group-specific advice,
- uses T online time, and
- succeeds with probability ϵ,

such that:

$$ST^2 = \tilde{O}(\epsilon N)$$

.... building on prior work on multiple-discrete-log algorithms [ESST99,KS01,HMCD04,BL12]

A preexisting result...

Theorem. [Mihalcik 2010] [Lee, Cheon, Hong 2011] [Bernstein and Lange 2013]

There is a generic dlog algorithm with preprocessing that:

- uses S bits of group-specific advice,
- uses T online time, and
- succeeds with probability ϵ,

such that:

$$ST^2 = \tilde{O}(\epsilon N)$$

Will sketch the algorithm for $S = T = N^{1/3}$, constant ϵ.

.... building on prior work on multiple-discrete-log algorithms [ESST99,KS01,HMCD04,BL12]

A preexisting result...

Theorem. [Mihalcik 2010] [Lee, Cheon, Hong 2011] [Bernstein and Lange 2013]

There is a generic dlog algorithm with preprocessing that:

- uses S bits of group-specific advice,
- uses T online time, and
- succeeds with probability ϵ,

such that:

$$ST^2 = \tilde{O}(\epsilon N)$$

.... building on prior work on multiple-discrete-log algorithms [ESST99,KS01,HMCD04,BL12]

Preliminaries

Define a pseudo-random walk on $\mathbb{G}$:

$$g^x \mapsto g^{x+\alpha} \quad \text{where} \quad \alpha = \mathsf{Hash}(g^x)$$

is a random function

[M10, LCH11, BL13]

Preliminaries

Define a pseudo-random walk on $\mathbb{G}$:

$$g^x \mapsto g^{x+\alpha} \quad \text{where} \quad \alpha = \mathsf{Hash}(g^x)$$

is a random function

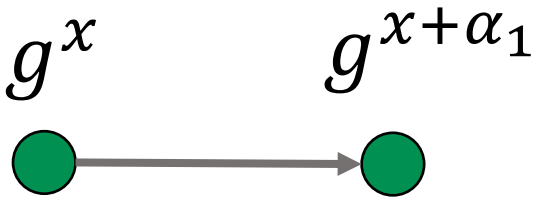
[M10, LCH11, BL13]

Preliminaries

Define a pseudo-random walk on $\mathbb{G}$:

$$g^x \mapsto g^{x+\alpha} \quad \text{where} \quad \alpha = \mathsf{Hash}(g^x)$$

is a random function



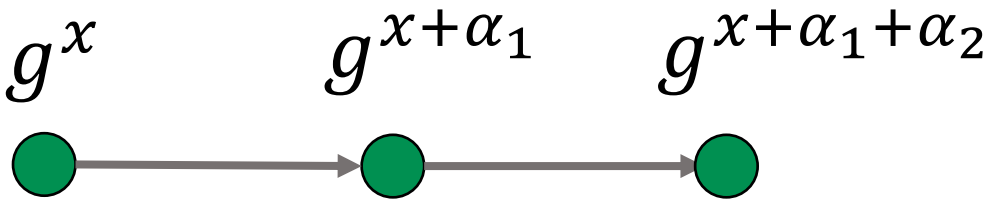
[M10, LCH11, BL13]

Preliminaries

Define a pseudo-random walk on $\mathbb{G}$:

$$g^x \mapsto g^{x+\alpha} \quad \text{where} \quad \alpha = \mathsf{Hash}(g^x)$$

is a random function



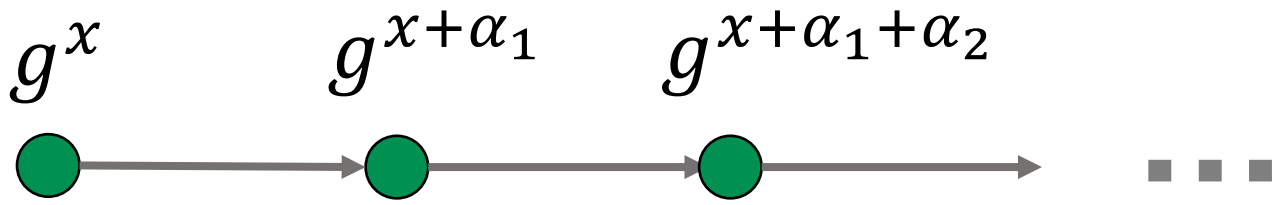
[M10, LCH11, BL13]

Preliminaries

Define a pseudo-random walk on $\mathbb{G}$:

$$g^x \mapsto g^{x+\alpha} \quad \text{where} \quad \alpha = \mathsf{Hash}(g^x)$$

is a random function



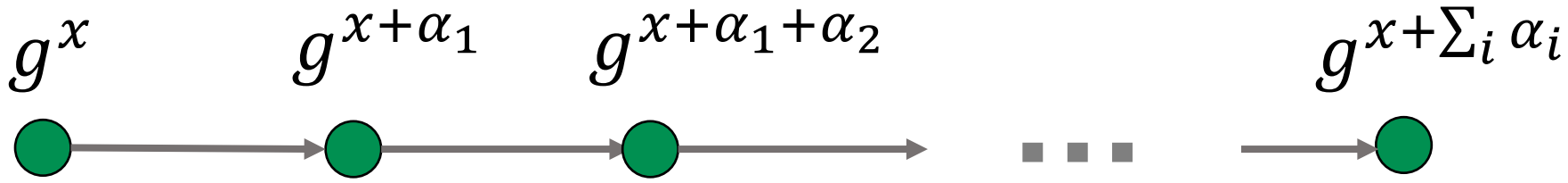
[M10, LCH11, BL13]

Preliminaries

Define a pseudo-random walk on $\mathbb{G}$:

$$g^x \mapsto g^{x+\alpha} \quad \text{where} \quad \alpha = \mathsf{Hash}(g^x)$$

is a random function



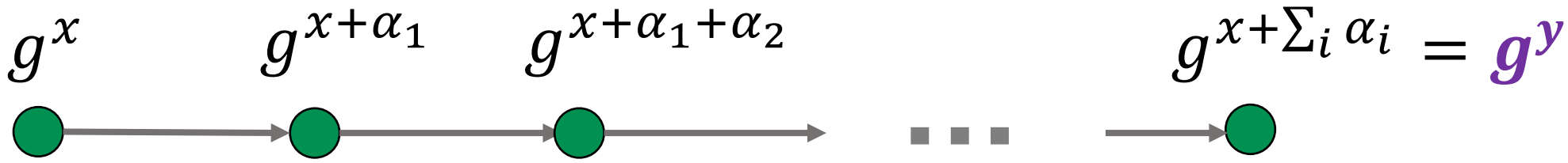
[M10, LCH11, BL13]

Preliminaries

Define a pseudo-random walk on $\mathbb{G}$:

$$g^x \mapsto g^{x+\alpha} \quad \text{where} \quad \alpha = \mathsf{Hash}(g^x)$$

is a random function



If you know the dlog of the endpoint of a walk, you know the dlog of the starting point!

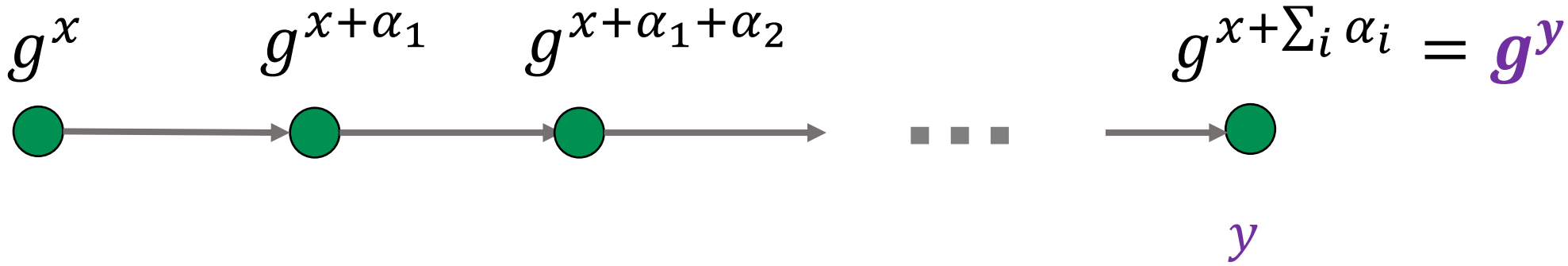
[M10, LCH11, BL13]

Preliminaries

Define a pseudo-random walk on $\mathbb{G}$:

$$g^x \mapsto g^{x+\alpha} \quad \text{where} \quad \alpha = \mathsf{Hash}(g^x)$$

is a random function



If you know the dlog of the endpoint of a walk, you know the dlog of the starting point!

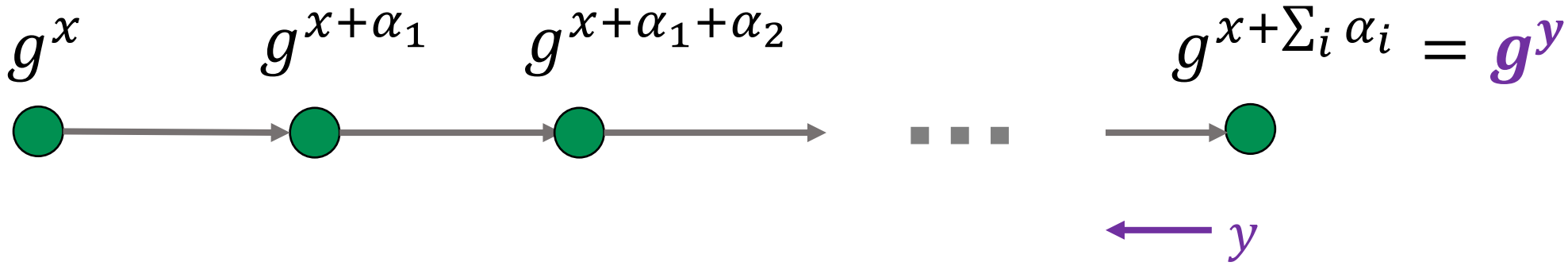
[M10, LCH11, BL13]

Preliminaries

Define a pseudo-random walk on $\mathbb{G}$:

$$g^x \mapsto g^{x+\alpha} \quad \text{where} \quad \alpha = \mathsf{Hash}(g^x)$$

is a random function



If you know the dlog of the endpoint of a walk, you know the dlog of the starting point!

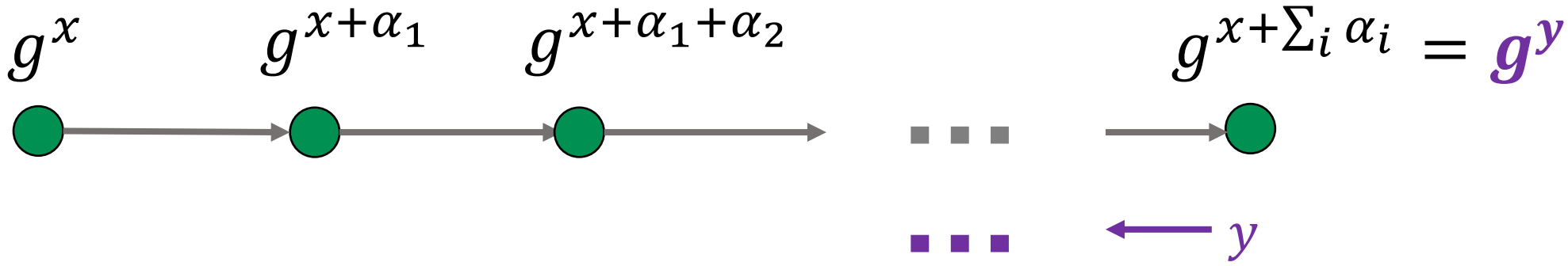
[M10, LCH11, BL13]

Preliminaries

Define a pseudo-random walk on $\mathbb{G}$:

$$g^x \mapsto g^{x+\alpha} \quad \text{where} \quad \alpha = \mathsf{Hash}(g^x)$$

is a random function



If you know the dlog of the endpoint of a walk, you know the dlog of the starting point!

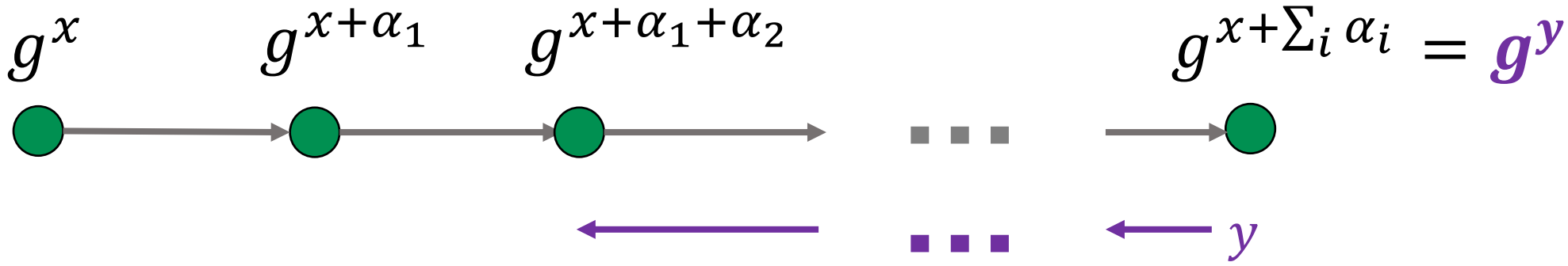
[M10, LCH11, BL13]

Preliminaries

Define a pseudo-random walk on $\mathbb{G}$:

$$g^x \mapsto g^{x+\alpha} \quad \text{where} \quad \alpha = \mathsf{Hash}(g^x)$$

is a random function



If you know the dlog of the endpoint of a walk, you know the dlog of the starting point!

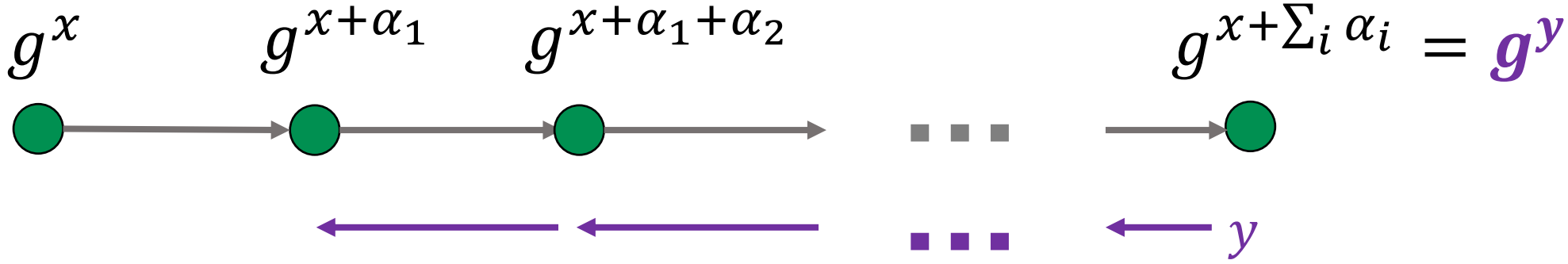
[M10, LCH11, BL13]

Preliminaries

Define a pseudo-random walk on $\mathbb{G}$:

$$g^x \mapsto g^{x+\alpha} \quad \text{where} \quad \alpha = \mathsf{Hash}(g^x)$$

is a random function



If you know the dlog of the endpoint of a walk, you know the dlog of the starting point!

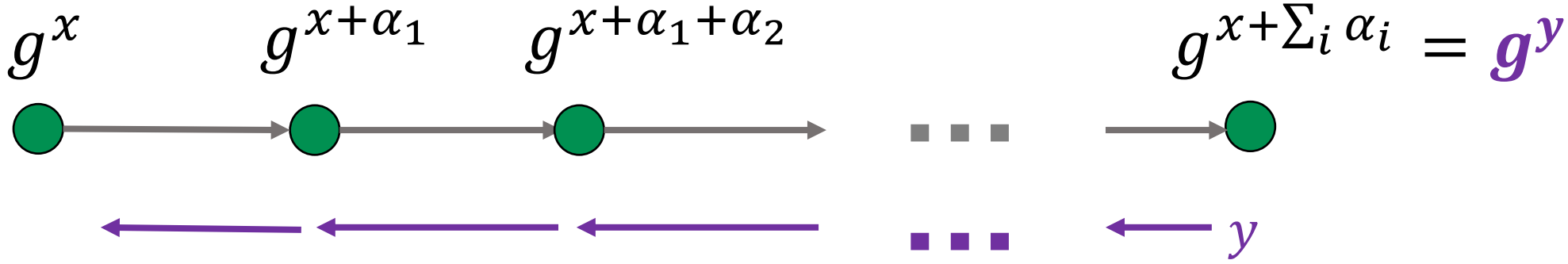
[M10, LCH11, BL13]

Preliminaries

Define a pseudo-random walk on $\mathbb{G}$:

$$g^x \mapsto g^{x+\alpha} \quad \text{where} \quad \alpha = \mathsf{Hash}(g^x)$$

is a random function



If you know the dlog of the endpoint of a walk, you know the dlog of the starting point!

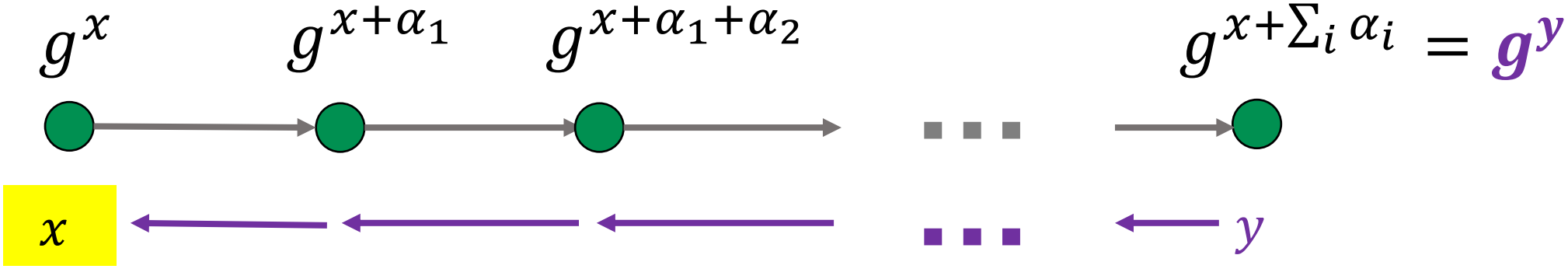
[M10, LCH11, BL13]

Preliminaries

Define a pseudo-random walk on $\mathbb{G}$:

$$g^x \mapsto g^{x+\alpha} \quad \text{where} \quad \alpha = \mathsf{Hash}(g^x)$$

is a random function



If you know the dlog of the endpoint of a walk, you know the dlog of the starting point!

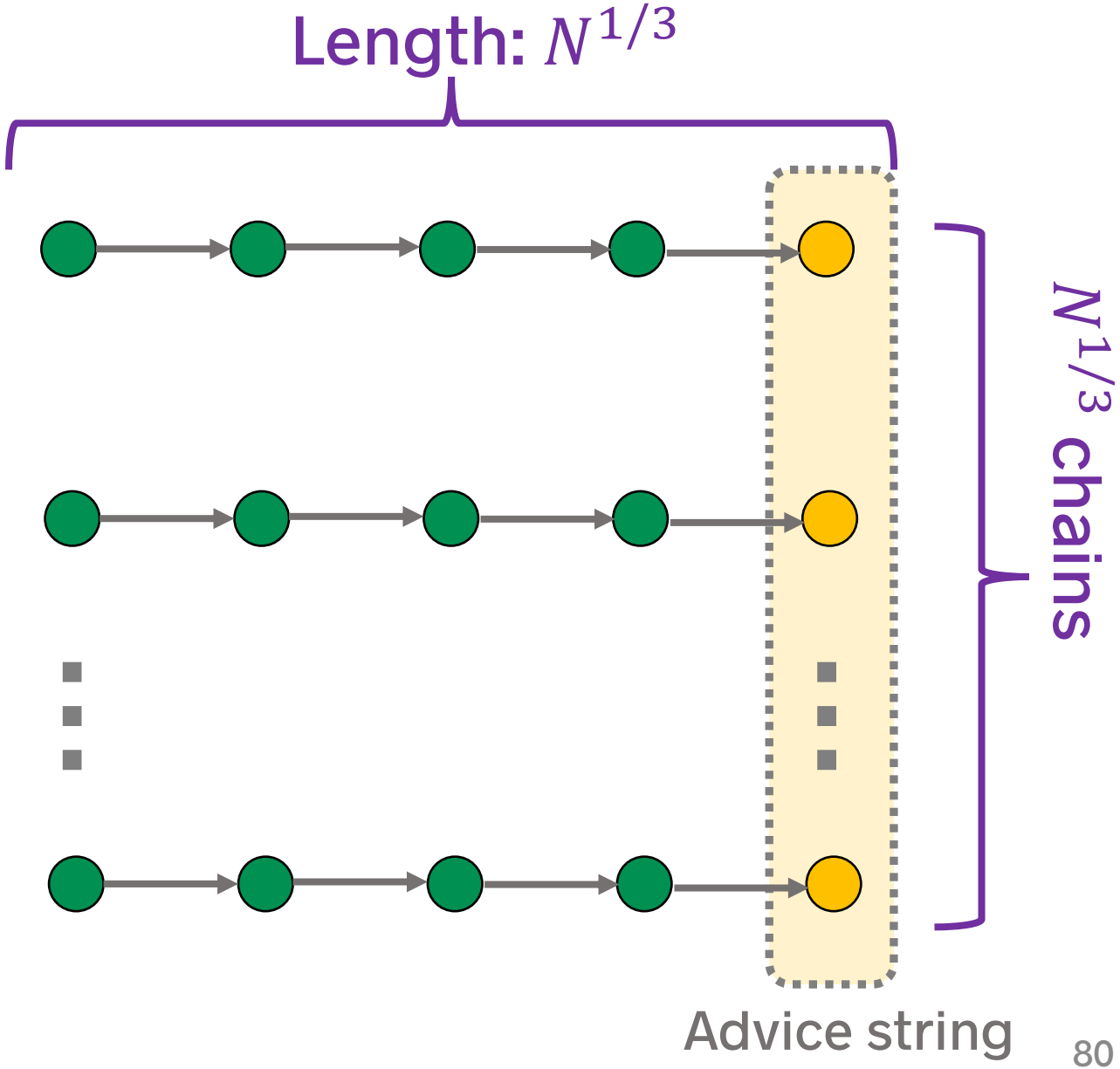
[M10, LCH11, BL13]

Preprocessing phase

- Build $N^{1/3}$ chains of length $N^{1/3}$
- Store dlogs of chain endpoints

Online phase

- Walk $O(N^{1/3})$ steps
- When you hit a stored point, output the discrete log



[M10, LCH11, BL13]

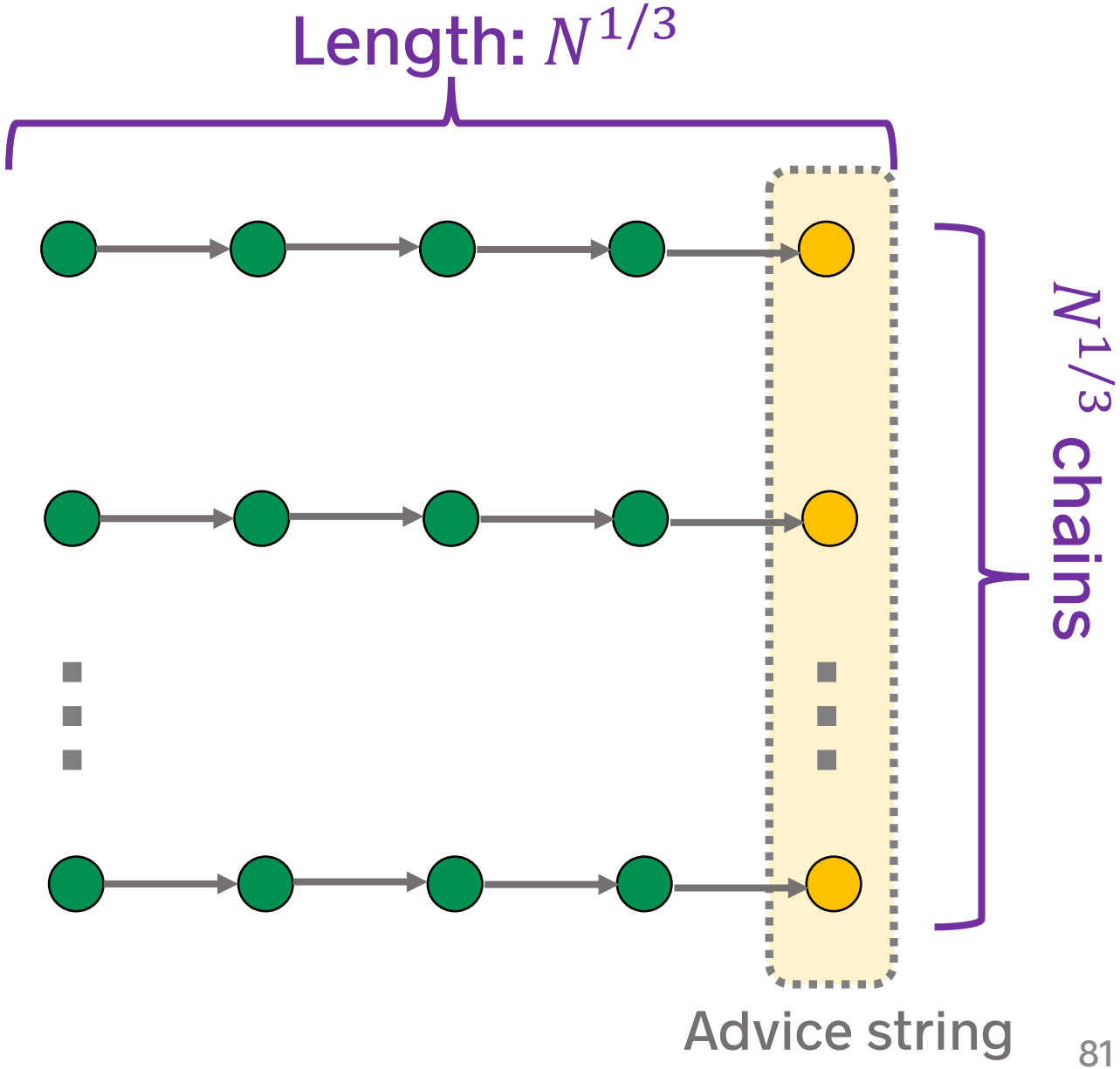
Preprocessing phase

- Build $N^{1/3}$ chains of length $N^{1/3}$
- Store dlogs of chain endpoints

Advice: $\tilde{O}(N^{1/3})$ bits

Online phase

- Walk $O(N^{1/3})$ steps
- When you hit a stored point, output the discrete log



[M10, LCH11, BL13]

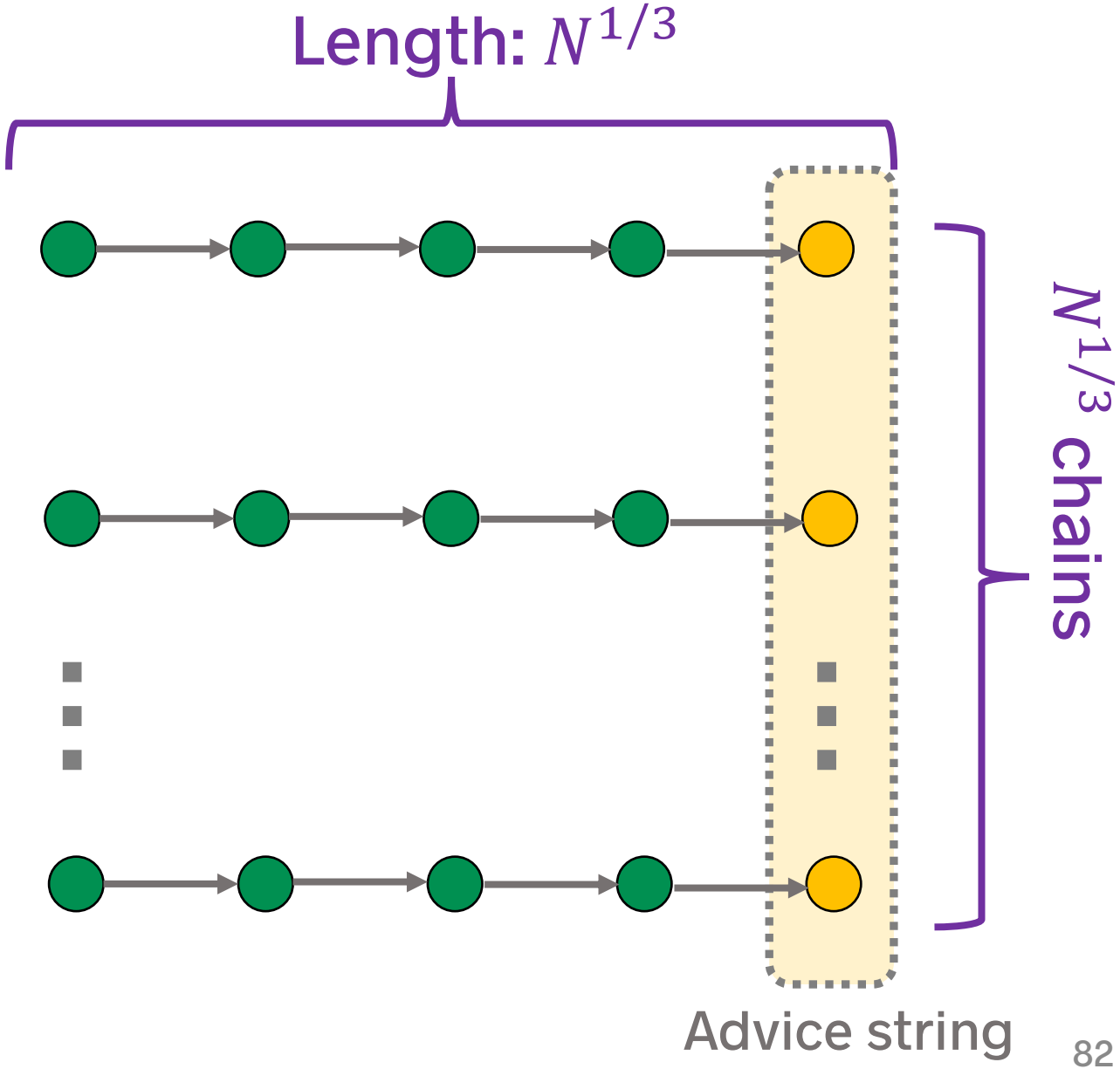
Preprocessing phase

- Build $N^{1/3}$ chains of length $N^{1/3}$
- Store dlogs of chain endpoints

Advice: $\tilde{O}(N^{1/3})$ bits

Online phase

- Walk $O(N^{1/3})$ steps
- When you hit a stored point, output the discrete log



[M10, LCH11, BL13]

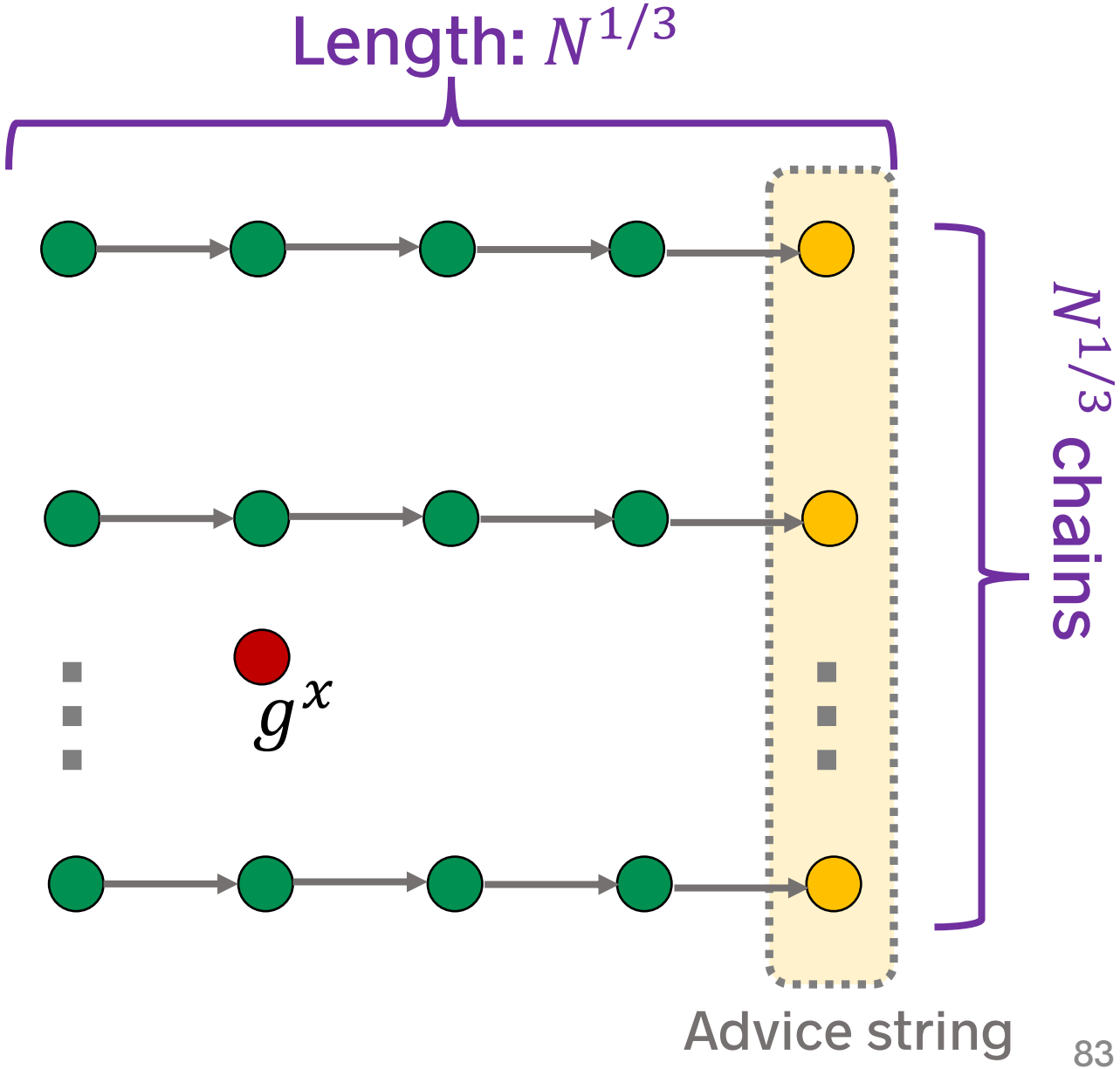
Preprocessing phase

- Build $N^{1/3}$ chains of length $N^{1/3}$
- Store dlogs of chain endpoints

Advice: $\tilde{O}(N^{1/3})$ bits

Online phase

- Walk $O(N^{1/3})$ steps
- When you hit a stored point, output the discrete log



[M10, LCH11, BL13]

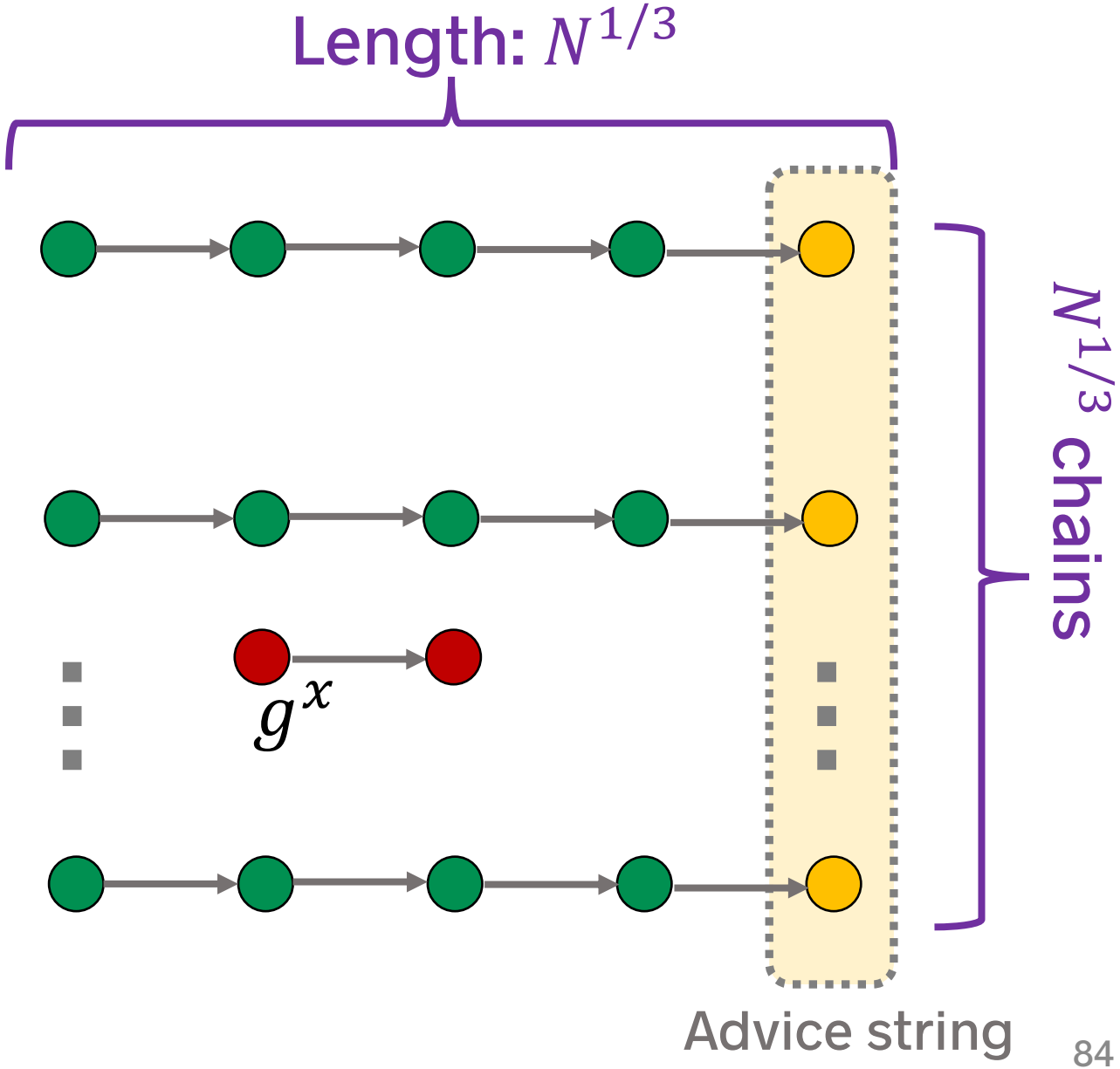
Preprocessing phase

- Build $N^{1/3}$ chains of length $N^{1/3}$
- Store dlogs of chain endpoints

Advice: $\tilde{O}(N^{1/3})$ bits

Online phase

- Walk $O(N^{1/3})$ steps
- When you hit a stored point, output the discrete log



[M10, LCH11, BL13]

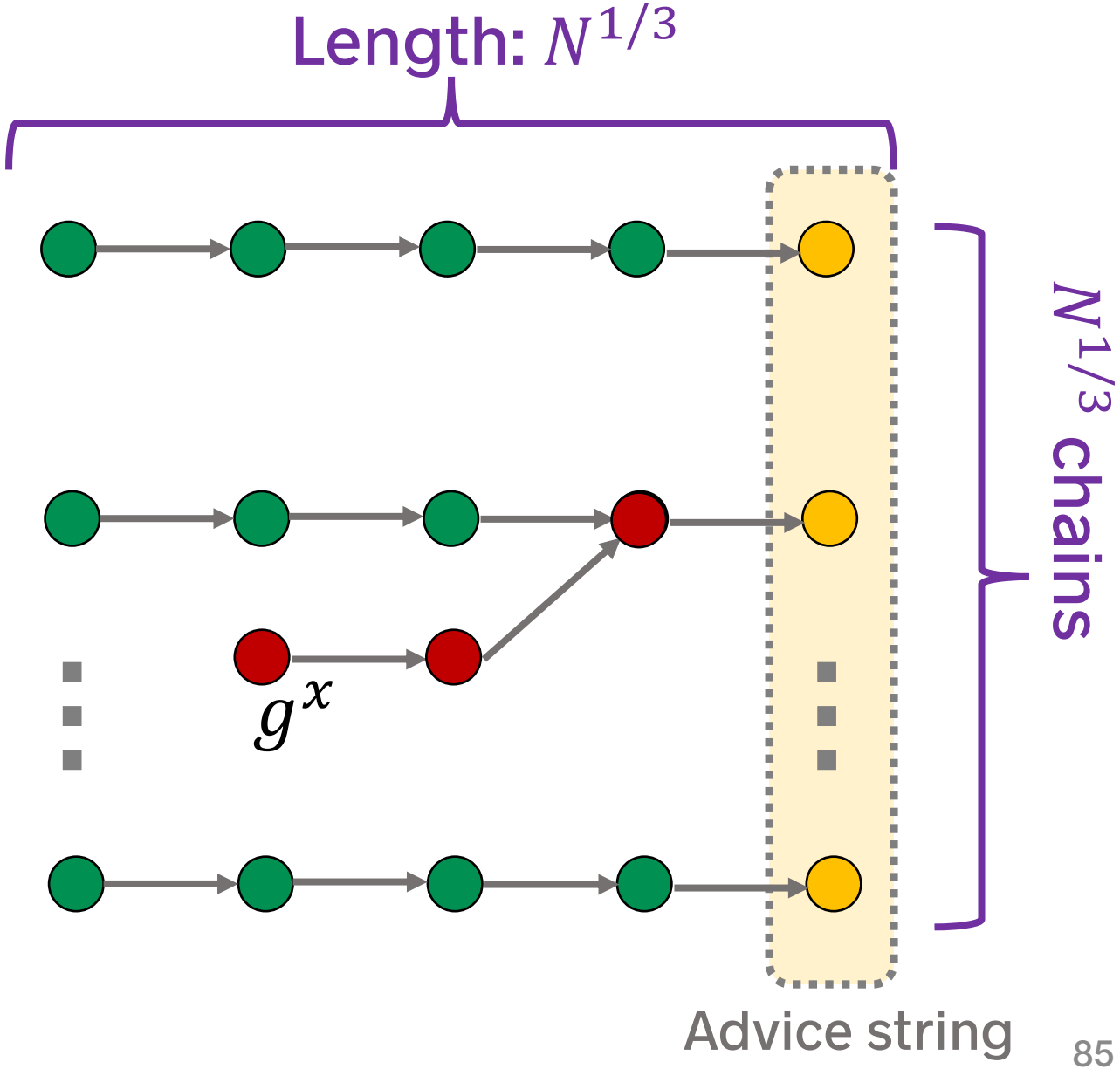
Preprocessing phase

- Build $N^{1/3}$ chains of length $N^{1/3}$
- Store dlogs of chain endpoints

Advice: $\tilde{O}(N^{1/3})$ bits

Online phase

- Walk $O(N^{1/3})$ steps
- When you hit a stored point, output the discrete log



[M10, LCH11, BL13]

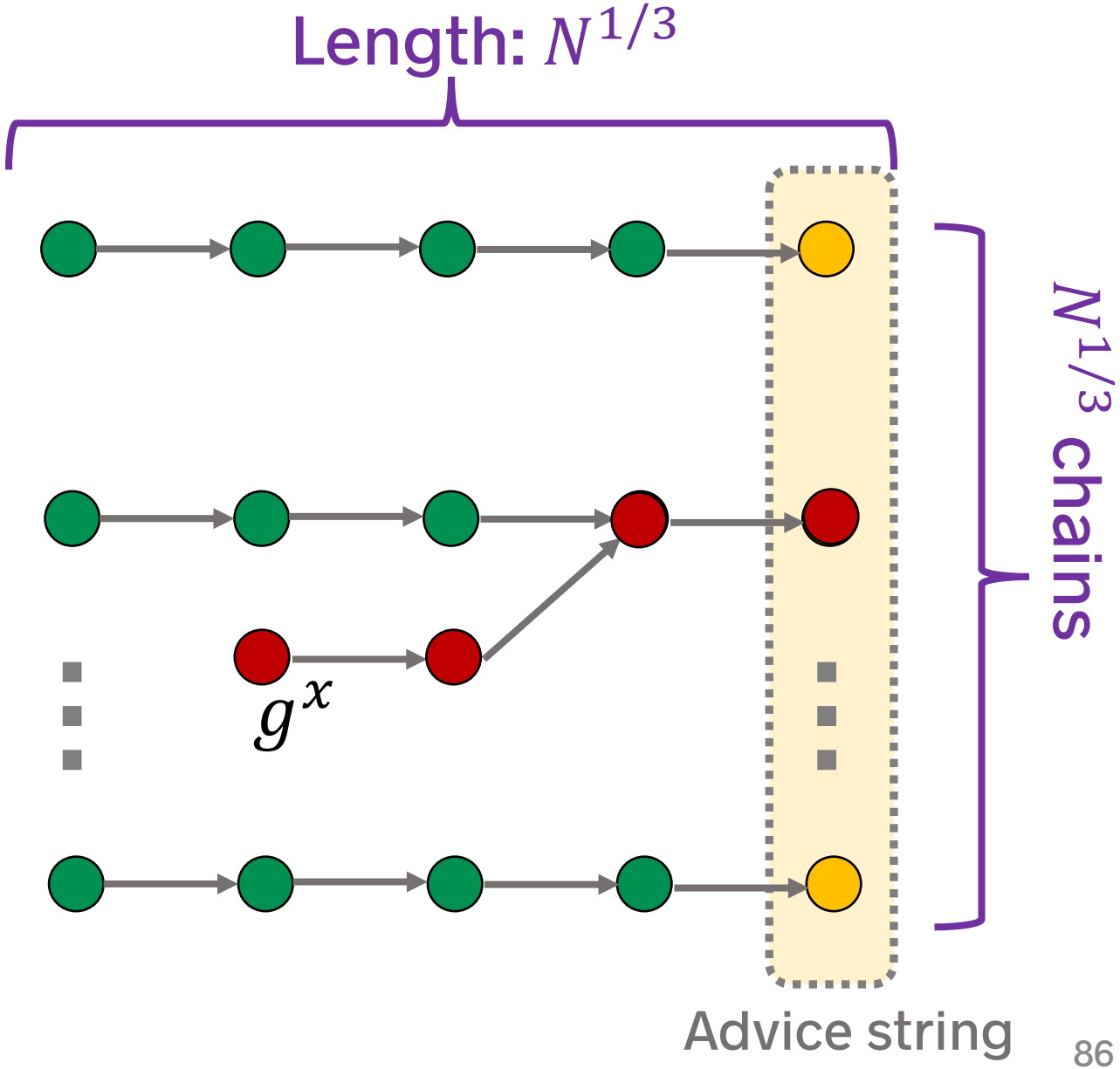
Preprocessing phase

- Build $N^{1/3}$ chains of length $N^{1/3}$
- Store dlogs of chain endpoints

Advice: $\tilde{O}(N^{1/3})$ bits

Online phase

- Walk $O(N^{1/3})$ steps
- When you hit a stored point, output the discrete log



[M10, LCH11, BL13]

Preprocessing phase

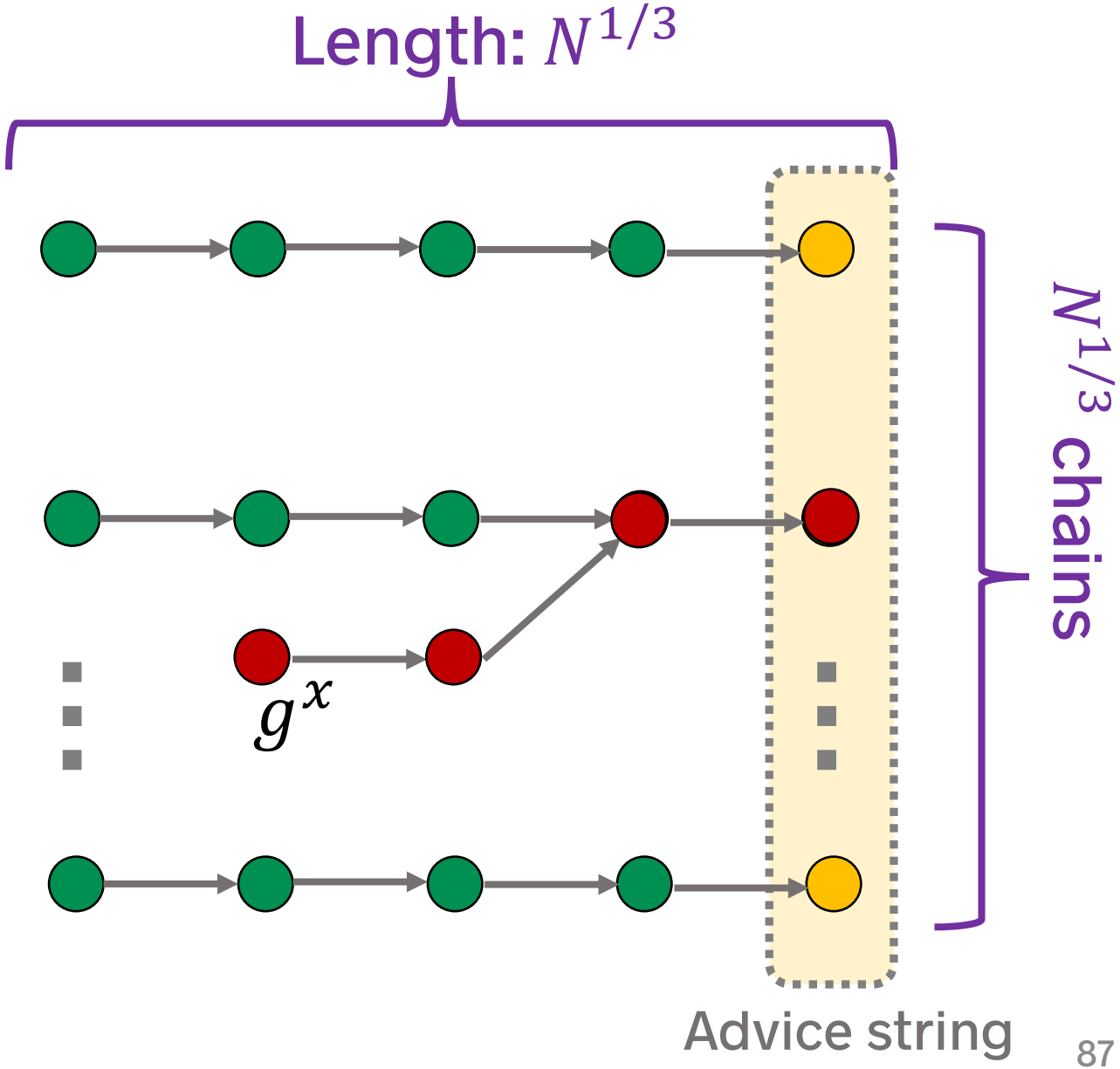
- Build $N^{1/3}$ chains of length $N^{1/3}$
- Store dlogs of chain endpoints

Advice: $\tilde{O}(N^{1/3})$ bits

Online phase

- Walk $O(N^{1/3})$ steps
- When you hit a stored point, output the discrete log

Time: $\tilde{O}(N^{1/3})$ steps



[M10, LCH11, BL13]

Preprocessing phase

- Build $N^{1/3}$ chains of length $N^{1/3}$
- Store dlogs of chain endpoints

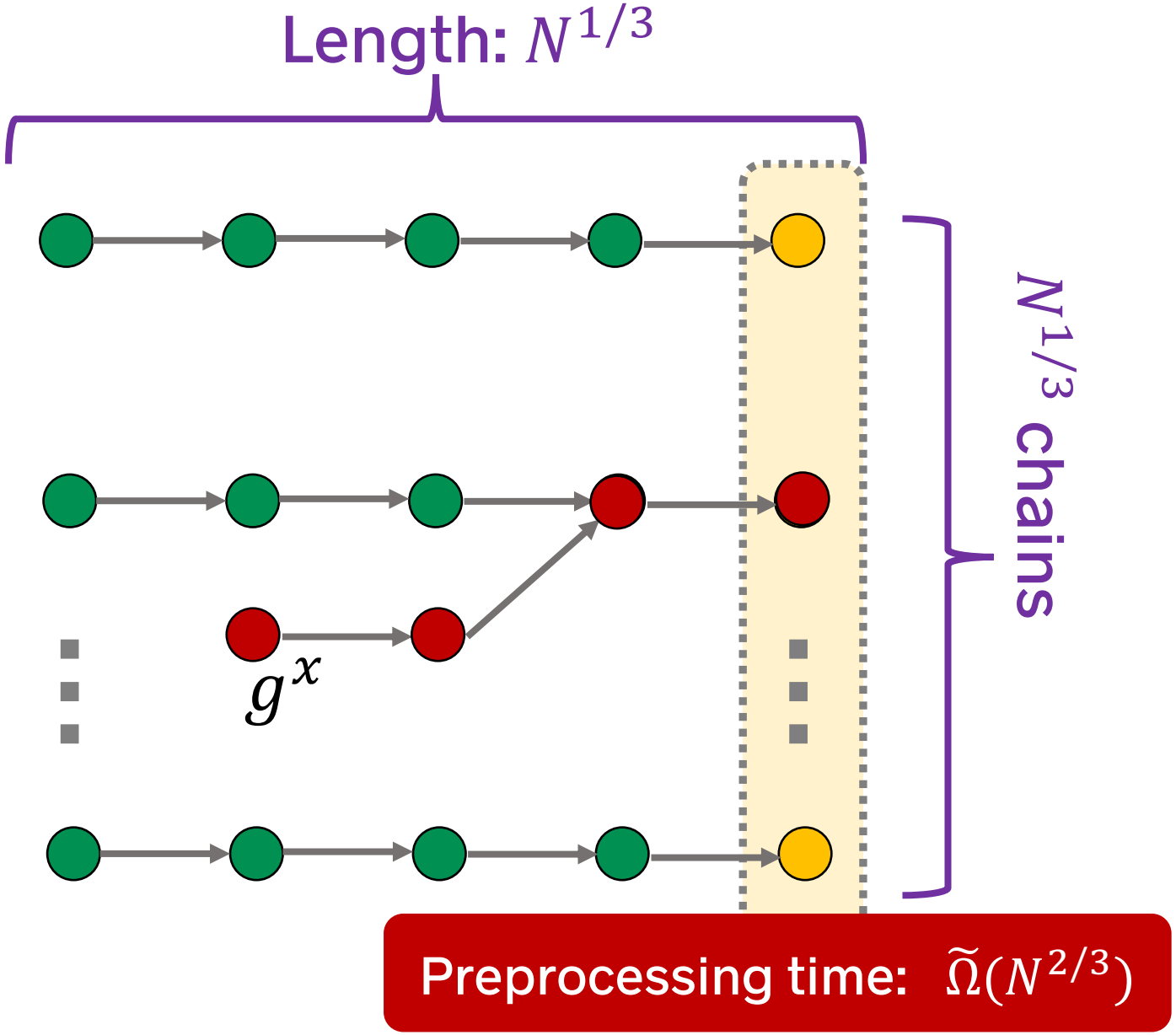
Advice: $\tilde{O}(N^{1/3})$ bits

Online phase

- Walk $O(N^{1/3})$ steps
- When you hit a stored point, output the discrete log

Time: $\tilde{O}(N^{1/3})$ steps

[M10, LCH11, BL13]



Generic discrete log

		256-bit ECDL
→ Without preprocessing:	$\Omega(N^{1/2})$	2^{128} time
→ With preprocessing:	$\tilde{O}(N^{1/3})$	2^{86} time

Related preprocessing attacks break:

- Multiple discrete log problem [This paper]
- One-round Even-Mansour cipher [FJM14]
- Merkle-Damgård hash with random IV [CDGS17]

Generic discrete log — 256-bit ECDL

→ Without preprocessing: $\Omega\left(N^{1/2}\right)$ — 2^{128} time

→ With preprocessing: $\tilde{O}(N^{1/3})$ — 2^{86} time

Related preprocessing attacks break:

- Multiple discrete log problem [This paper]
- One-round Even-Mansour cipher [FJM14]
- Merkle-Damgård hash with random IV [CDGS17]

Generic discrete log

→ Without preprocessing: $\Omega(N^{1/2})$

→ With preprocessing: $\tilde{O}(N^{1/3})$

256-bit ECDL

2^{128} time

2^{86} time

Is this dlog attack the best possible?!

Related preprocessing attacks break:

- Multiple discrete log problem [This paper]
- One-round Even-Mansour cipher [FJM14]
- Merkle-Damgård hash with random IV [CDGS17]

Generic discrete log — 256-bit ECDL

→ Without preprocessing: $\Omega\left(N^{1/2}\right)$ — 2^{128} time

→ With preprocessing: $\tilde{O}(N^{1/3})$ — 2^{86} time

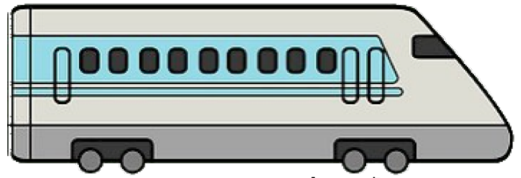
Related preprocessing attacks break:

- Multiple discrete log problem [This paper]
- One-round Even-Mansour cipher [FJM14]
- Merkle-Damgård hash with random IV [CDGS17]

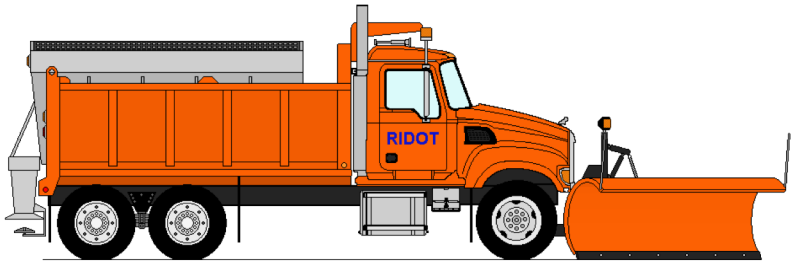
Signatures
(DSA and Schnorr)

DH key exchange

DDH



Pairings

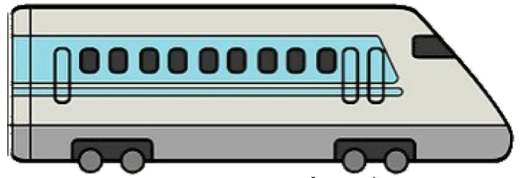


Discrete log

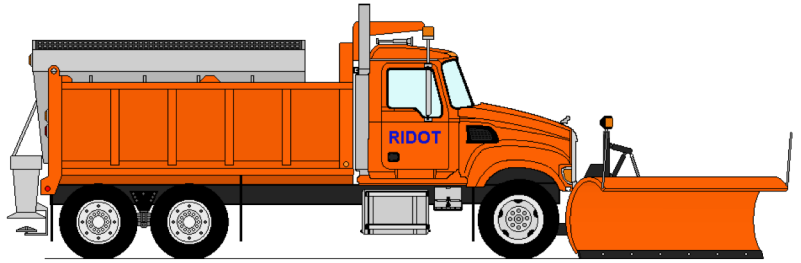
Signatures
(DSA and Schnorr)

DH key exchange

DDH



Pairings



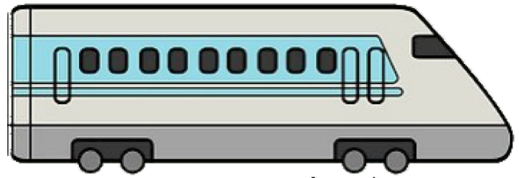
Discrete log

Could there exist a generic dlog preprocessing attack with $S = T = N^{1/10}$?

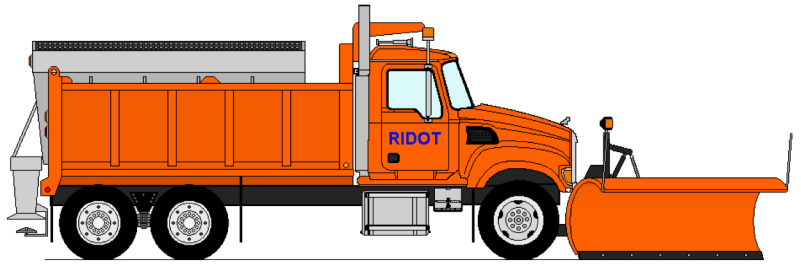
Signatures
(DSA and Schnorr)

DH key exchange

DDH



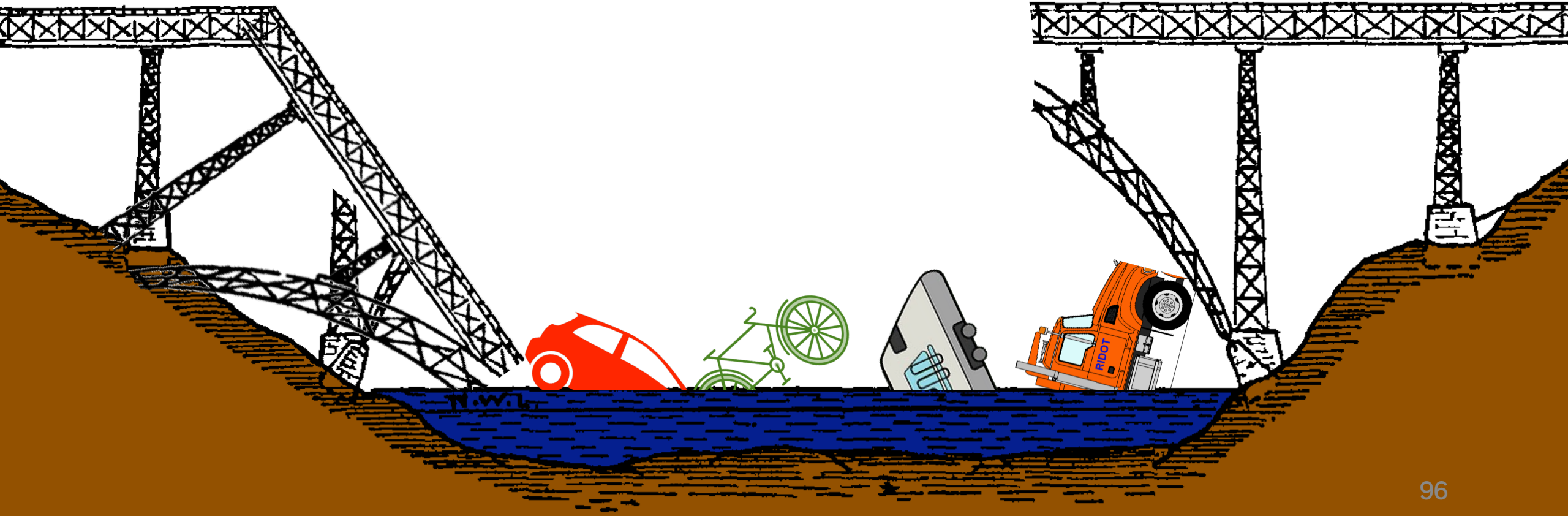
Pairings



Discrete log

Could there exist a generic dlog preprocessing attack with $S = T = N^{1/10}$?

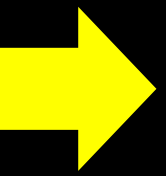
Preprocessing attacks might make us worry about 256-bit EC groups


T.W.L.
RIDOT

This talk

Background: Preprocessing attacks are relevant

- Preexisting $S = T = \tilde{O}(N^{1/3})$ generic attack on discrete log



Our results: Preprocessing lower-bounds and attacks

- The $\tilde{O}(N^{1/3})$ generic dlog attack is optimal
- Any such attack must use lots of preprocessing: $\Omega(N^{2/3})$
- New $\tilde{O}(N^{1/5})$ preprocessing attack on DDH-like problem

Open questions

Theorem. [Our paper]

Every generic dlog algorithm with preprocessing that:

- uses S bits of group-specific advice,
- uses T online time, and
- succeeds with probability ϵ,

must satisfy:

$$ST^2 = \widetilde{\Omega}(\epsilon N)$$

Theorem. [Our paper]

Every generic dlog algorithm with preprocessing that:

- uses S bits of group-specific advice,
- uses T online time, and
- succeeds with probability ϵ,

must satisfy:

$$ST^2 = \widetilde{\Omega}(\epsilon N)$$

This bound is tight for the full range of parameters (up to log factors)

Theorem. [Our paper]

Every generic dlog algorithm with preprocessing that:

- uses S bits of group-specific advice,
- uses T online time, and
- succeeds with probability ϵ,

must satisfy:

$$ST^2 = \widetilde{\Omega}(\epsilon N)$$

Theorem. [Our paper]

Every generic dlog algorithm with preprocessing that:

- uses S bits of group-specific advice,
- uses T online time, and
- succeeds with probability ϵ,

must satisfy:

$$ST^2 = \widetilde{\Omega}(\epsilon N)$$

Shoup's proof technique (1997) relies on $\mathcal{A}$ having no information about the group $\mathbb{G}$ when it starts running

→ Need different proof technique

Theorem. [Our paper]

Every generic dlog algorithm with preprocessing that:

- uses S bits of group-specific advice,
- uses T online time, and
- succeeds with probability ϵ,

must satisfy:

$$ST^2 = \widetilde{\Omega}(\epsilon N)$$

Theorem. [Our paper]

Every generic dlog algorithm with preprocessing that:

- uses S bits of group-specific advice,
- uses T online time, and
- succeeds with probability ϵ,

must satisfy:

$$ST^2 = \widetilde{\Omega}(\epsilon N)$$

Theorem. [Our paper]

Furthermore, the preprocessing time P must satisfy

$$PT + T^2 = \Omega(\epsilon N)$$

Theorem. [Our paper]

Every generic dlog algorithm with preprocessing that:

- uses S bits of group-specific advice,
- uses T online time, and
- succeeds with probability ϵ,

must satisfy:

$$ST^2 = \widetilde{\Omega}(\epsilon N)$$

Online time $N^{1/3}$ implies $\Omega(N^{2/3})$ preprocessing

Theorem. [Our paper]

Furthermore, the preprocessing time P must satisfy

$$PT + T^2 = \Omega(\epsilon N)$$

Theorem. [Our paper]

Every generic dlog algorithm with preprocessing that:

- uses S bits of group-specific advice,
- uses T online time, and
- succeeds with probability ϵ,

must satisfy:

$$ST^2 = \widetilde{\Omega}(\epsilon N)$$

Theorem. [Our paper]

Furthermore, the preprocessing time P must satisfy

$$PT + T^2 = \Omega(\epsilon N)$$

Reminder: Generic-group model

- A group is defined by an injective **"labeling" function**

$$\sigma: \ \mathbb{Z}_N \to \{0,1\}^*$$

- Algorithm has access to a **group-operation oracle:**

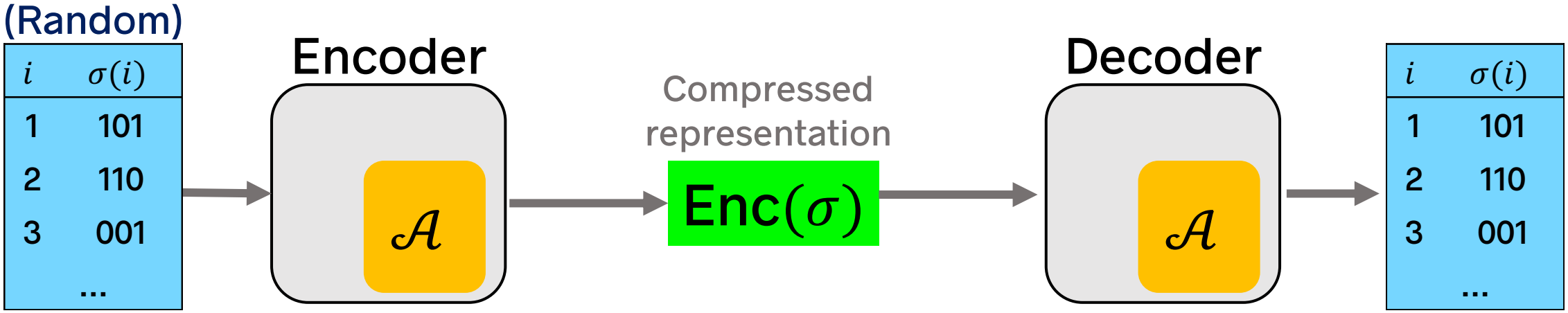
$$\mathcal{O}_\sigma\big(\sigma(i), \sigma(j)\big) \ \mapsto \ \sigma(i+j)$$

E.g., A dlog algorithm takes as input $\big(\sigma(1), \sigma(x)\big)$, representing (g, g^x), make queries to $\mathcal{O}_\sigma$, outputs x.

We prove the lower bound using an incompressibility argument [Yao90, GT00, DTT10, DHT12, DGK17...]

Use $\mathcal{A}$ to compress the mapping σ: $\mathbb{Z}_N \to \{0,1\}^*$ that defines the group

Similar technique used in [DHT12]

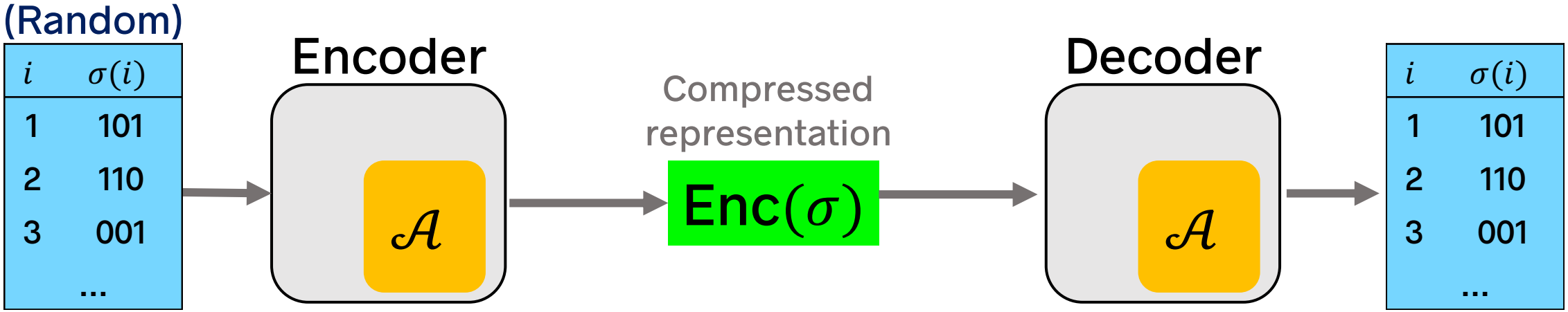


- Adv $\mathcal{A}$ uses advice S and online time T such that $ST^2 = o(N)$ ⇒ Encoder compresses well
- Random string is incompressible ⇒ Lower bound on S and T

We prove the lower bound using an incompressibility argument [Yao90, GT00, DTT10, DHT12, DGK17…]

Use $\mathcal{A}$ to compress the mapping σ: $\mathbb{Z}_N \to \{0,1\}^*$ that defines the group

Similar technique used in [DHT12]

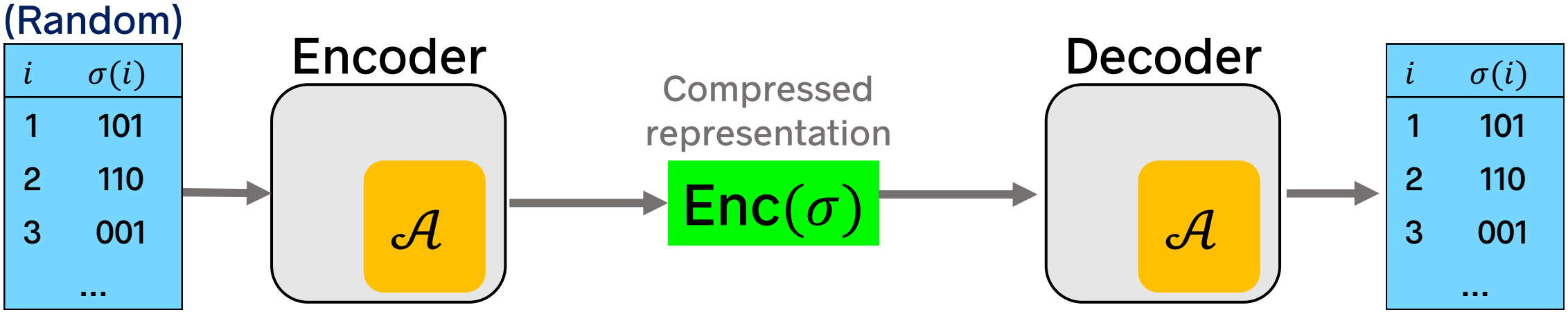


- Adv $\mathcal{A}$ uses advice S and online time T such that $ST^2 = o(N)$ ⇒ Encoder compresses well
- Random string is incompressible ⇒ Lower bound on S and T

We prove the lower bound using an incompressibility argument [Yao90, GT00, DTT10, DHT12, DGK17...]

Use $\mathcal{A}$ to compress the mapping σ: $\mathbb{Z}_N \to \{0,1\}^*$ that defines the group

Similar technique used in [DHT12]

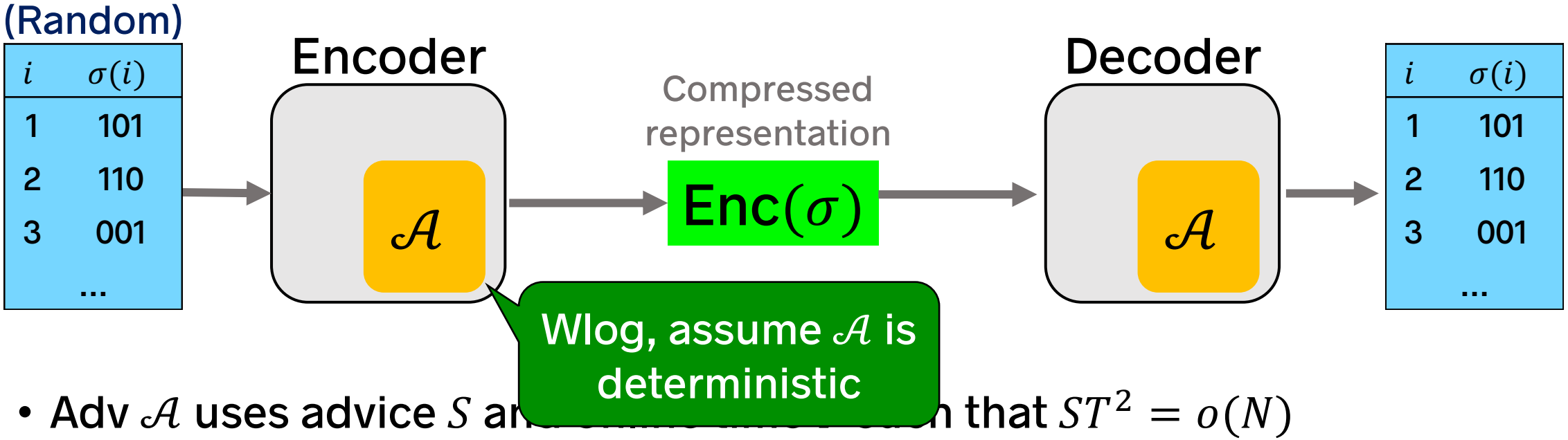


- Adv $\mathcal{A}$ uses advice S and online time T such that $ST^2 = o(N)$
  ⇒ Encoder compresses well
- Random string is incompressible ⇒ Lower bound on S and T

We prove the lower bound using an incompressibility argument [Yao90, GT00, DTT10, DHT12, DGK17...]

Use $\mathcal{A}$ to compress the mapping σ: $\mathbb{Z}_N \to \{0,1\}^*$ that defines the group

Similar technique used in [DHT12]

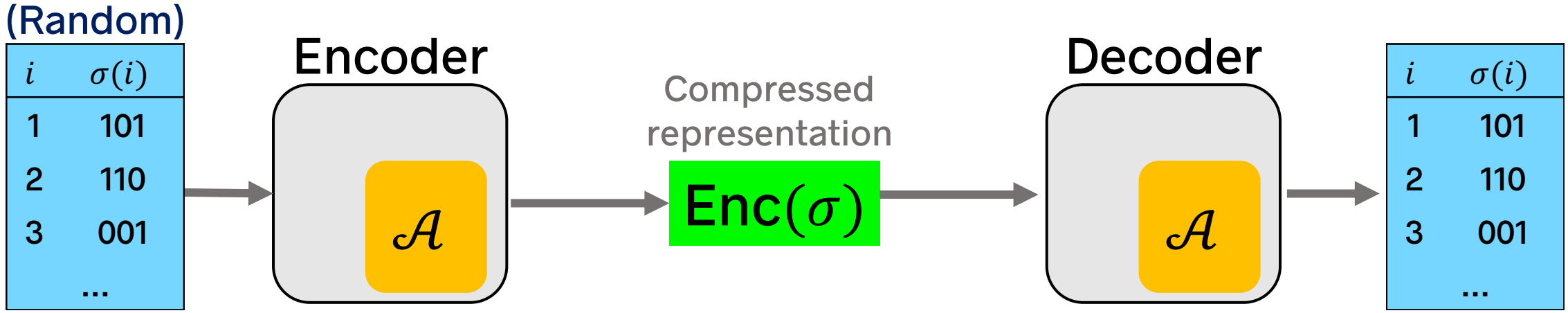


- Adv $\mathcal{A}$ uses advice S and time T such that $ST^2 = o(N)$
  ⇒ Encoder compresses well
- Random string is incompressible
  ⇒ Lower bound on S and T

We prove the lower bound using an incompressibility argument [Yao90, GT00, DTT10, DHT12, DGK17...]

Use $\mathcal{A}$ to compress the mapping σ: $\mathbb{Z}_N \to \{0,1\}^*$ that defines the group

Similar technique used in [DHT12]



- Adv $\mathcal{A}$ uses advice S and online time T such that $ST^2 = o(N)$
  ⇒ Encoder compresses well
- Random string is incompressible ⇒ Lower bound on S and T

Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

Encoder

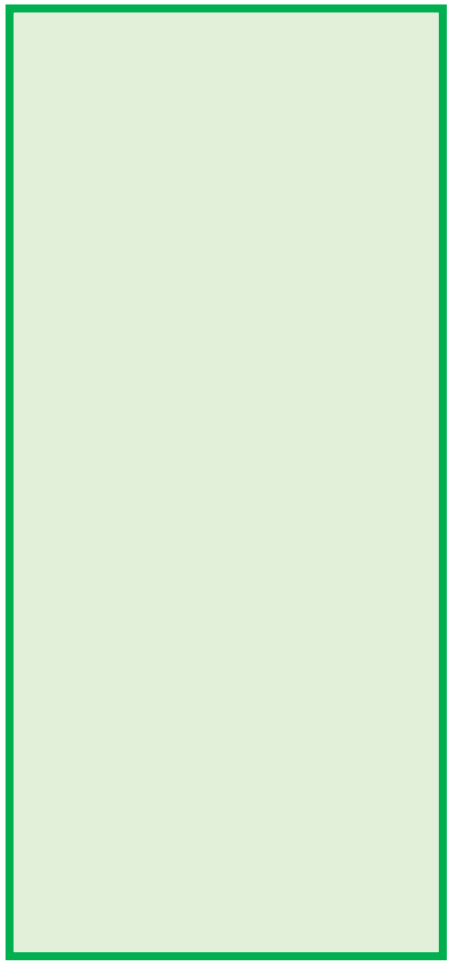
i	$\sigma(i)$
1	101
2	110
3	001
4	000
5	1111
	...

Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

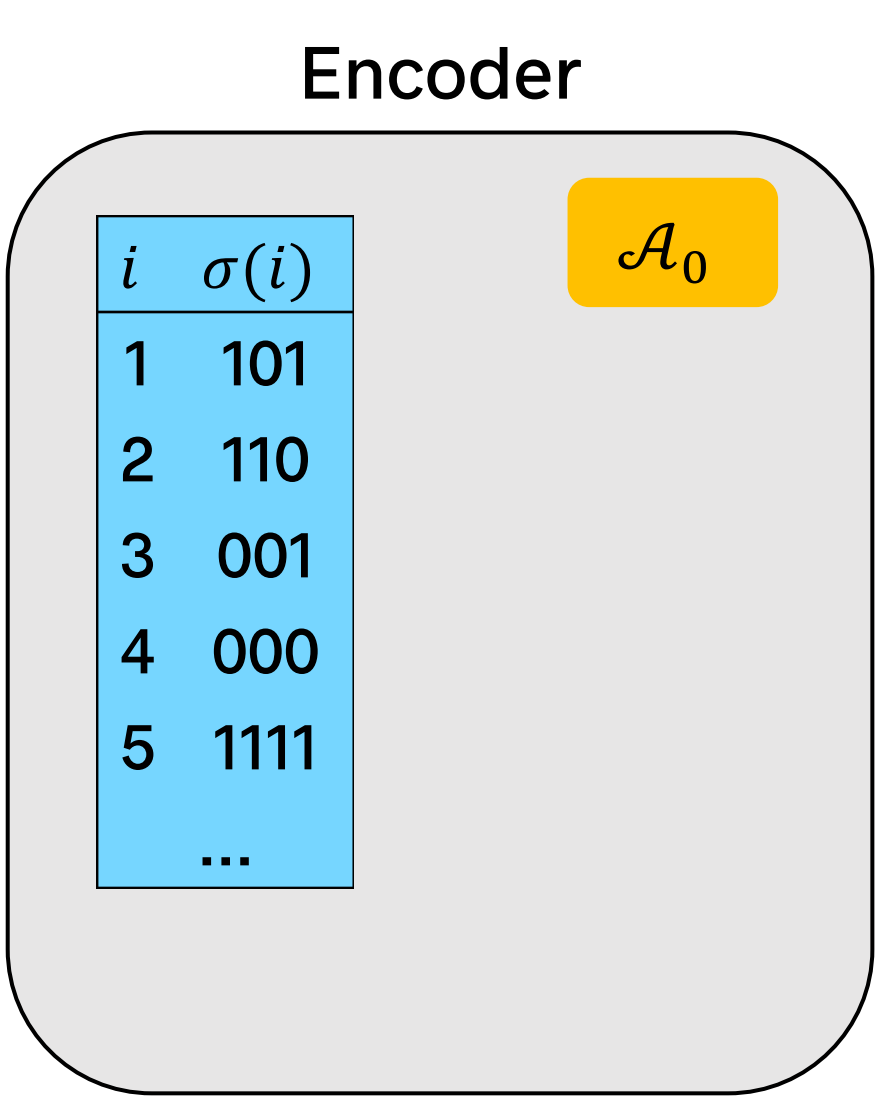
Encoder

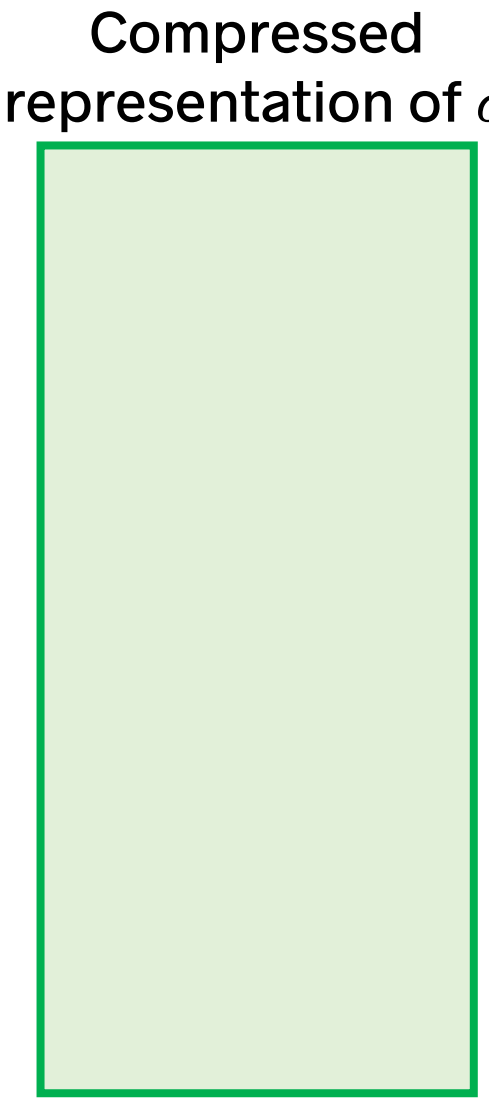
i	$\sigma(i)$
1	101
2	110
3	001
4	000
5	1111
...	

Compressed representation of σ

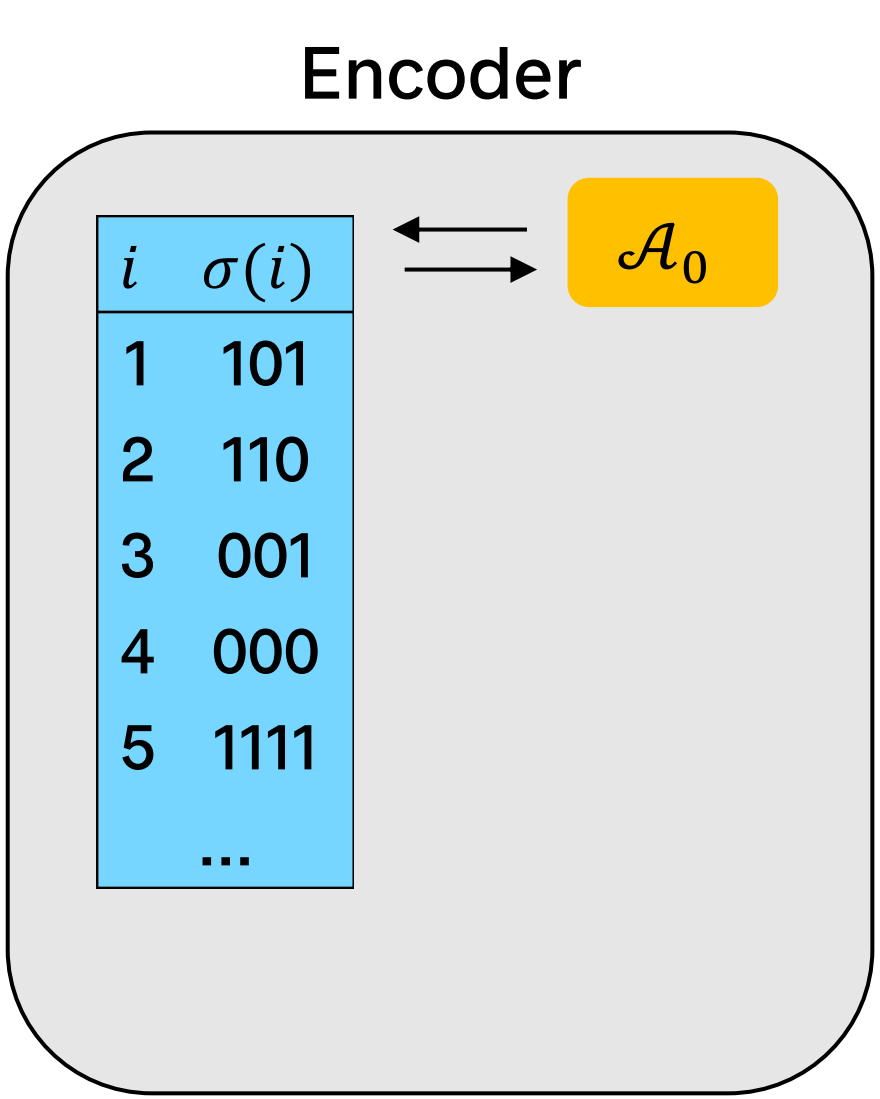


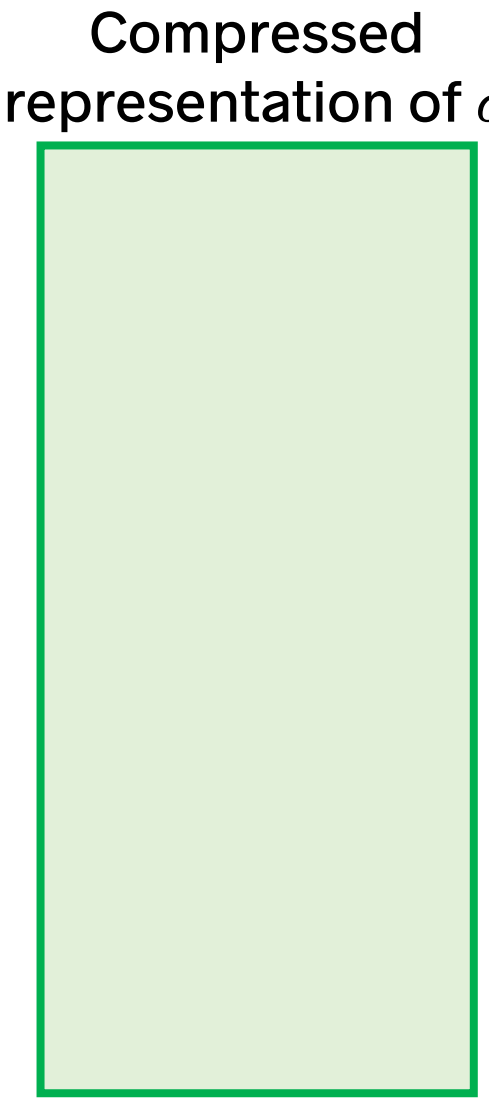
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]



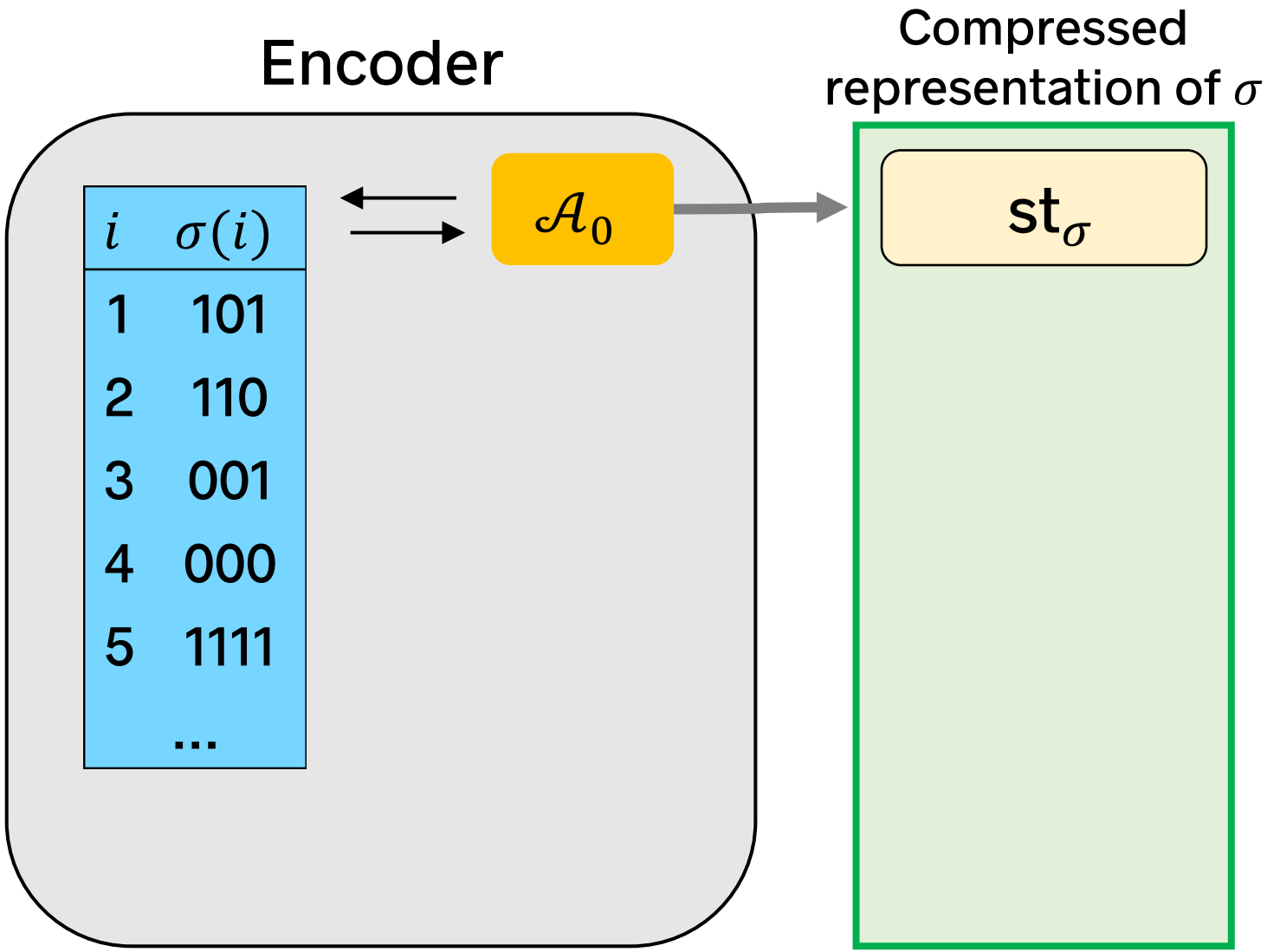


Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

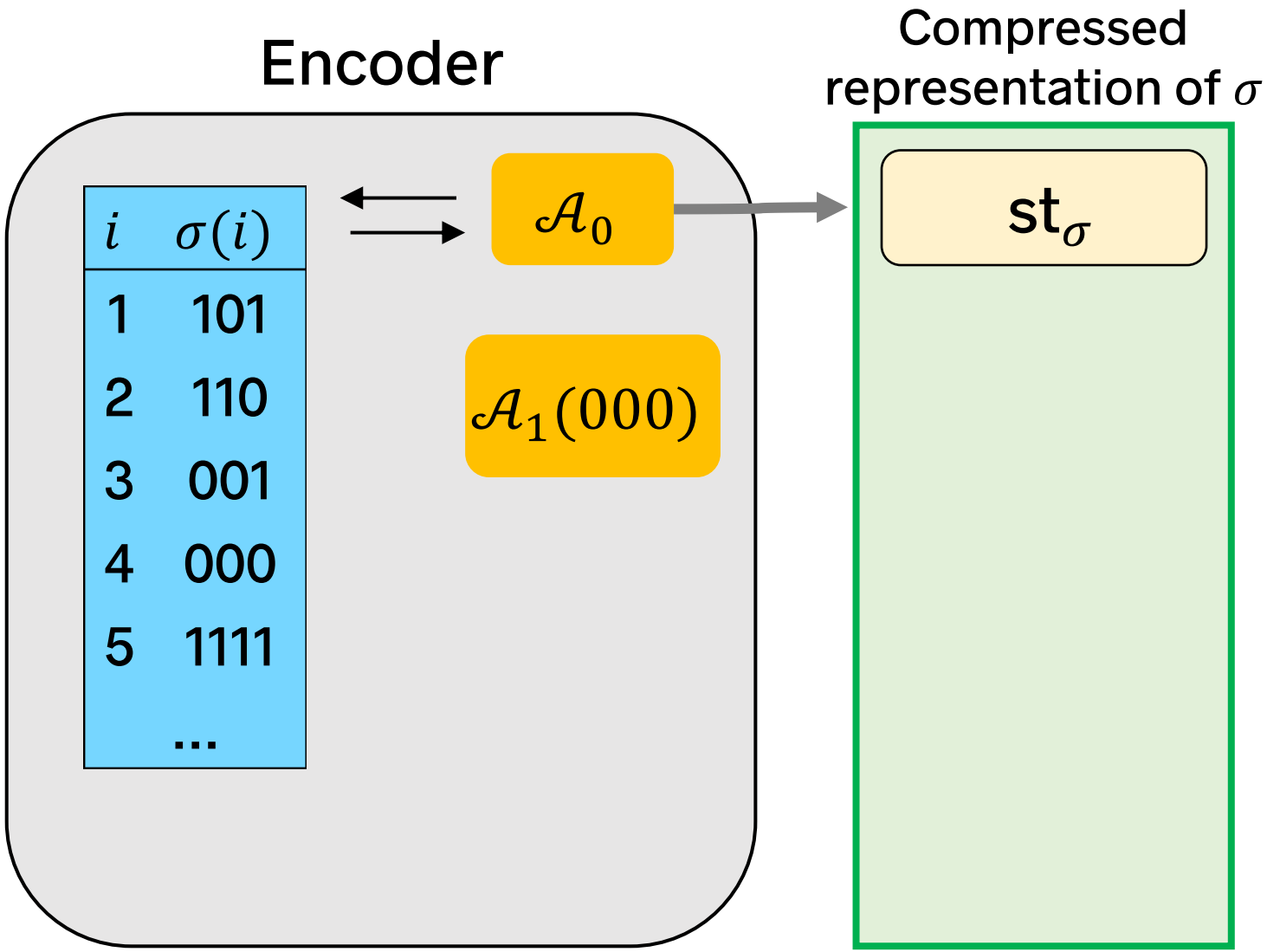




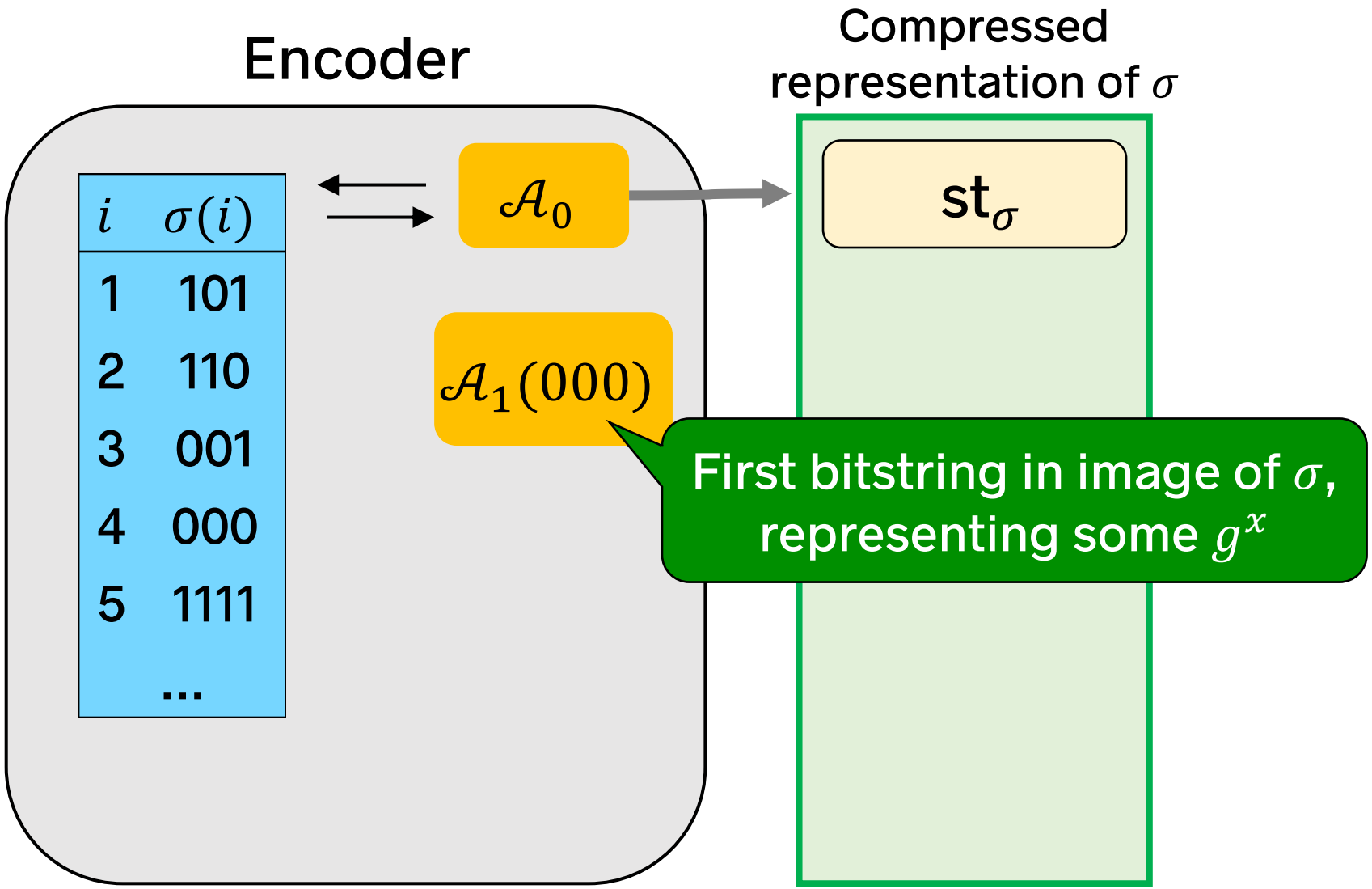
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]



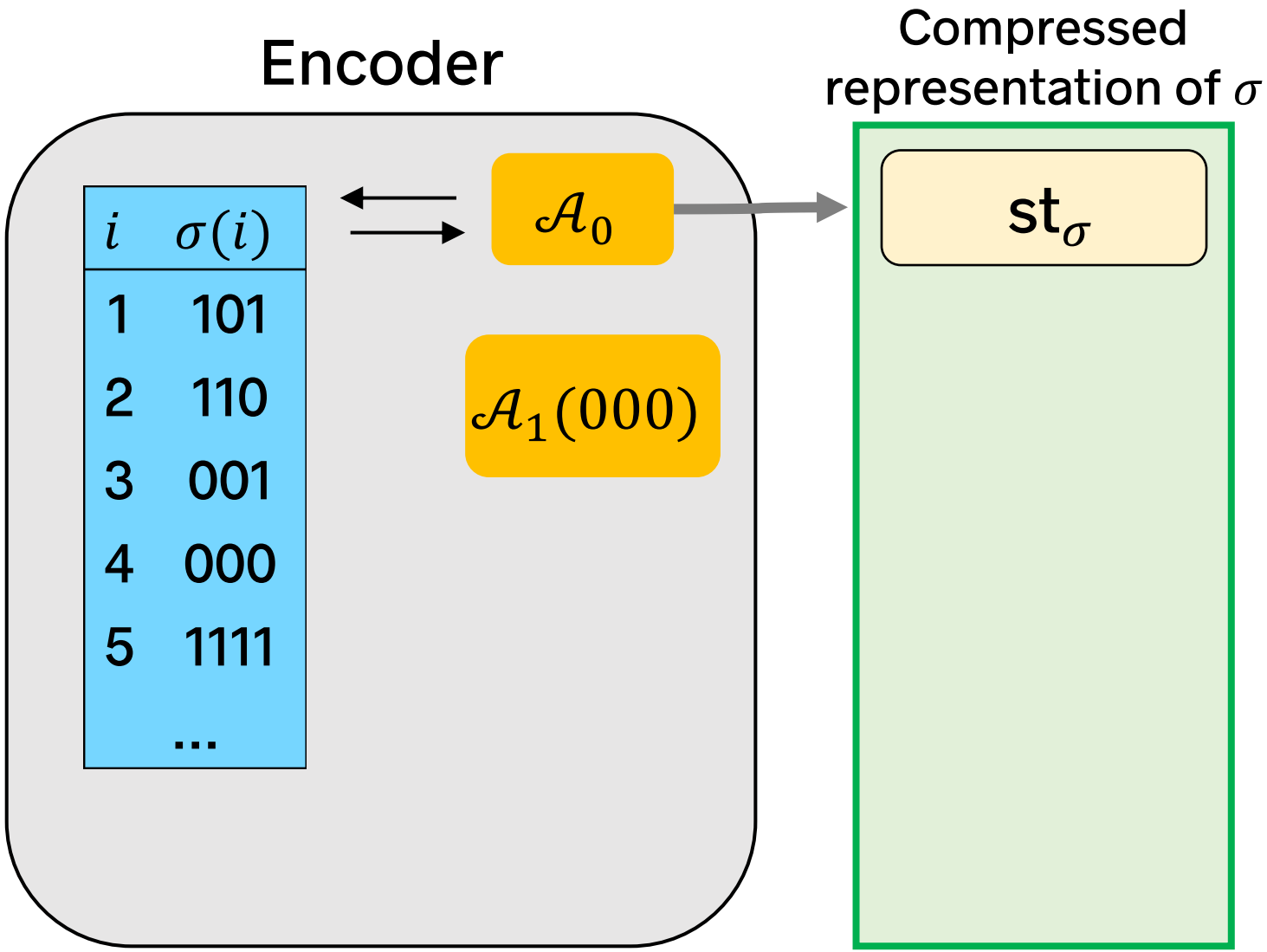
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]



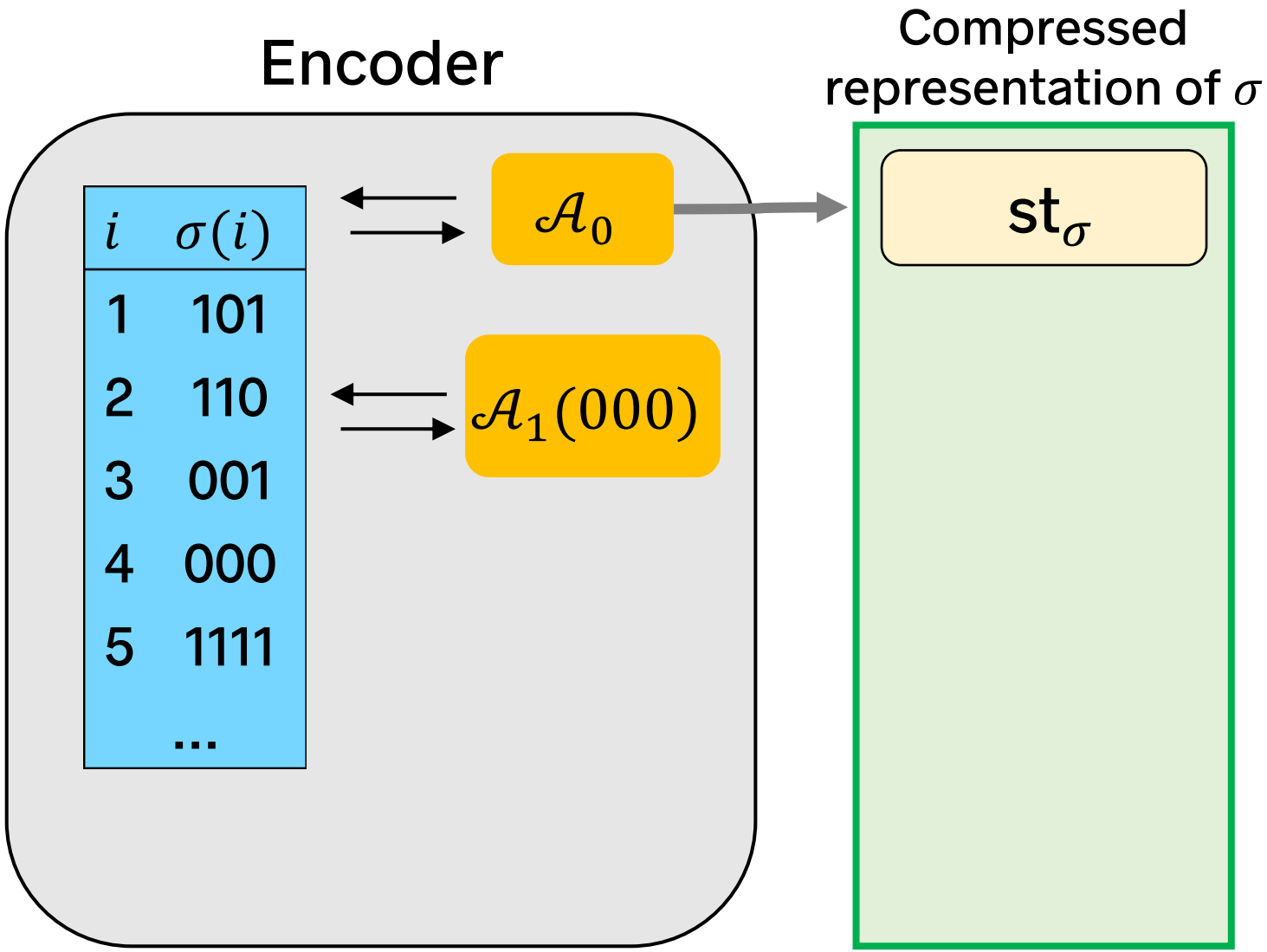
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]



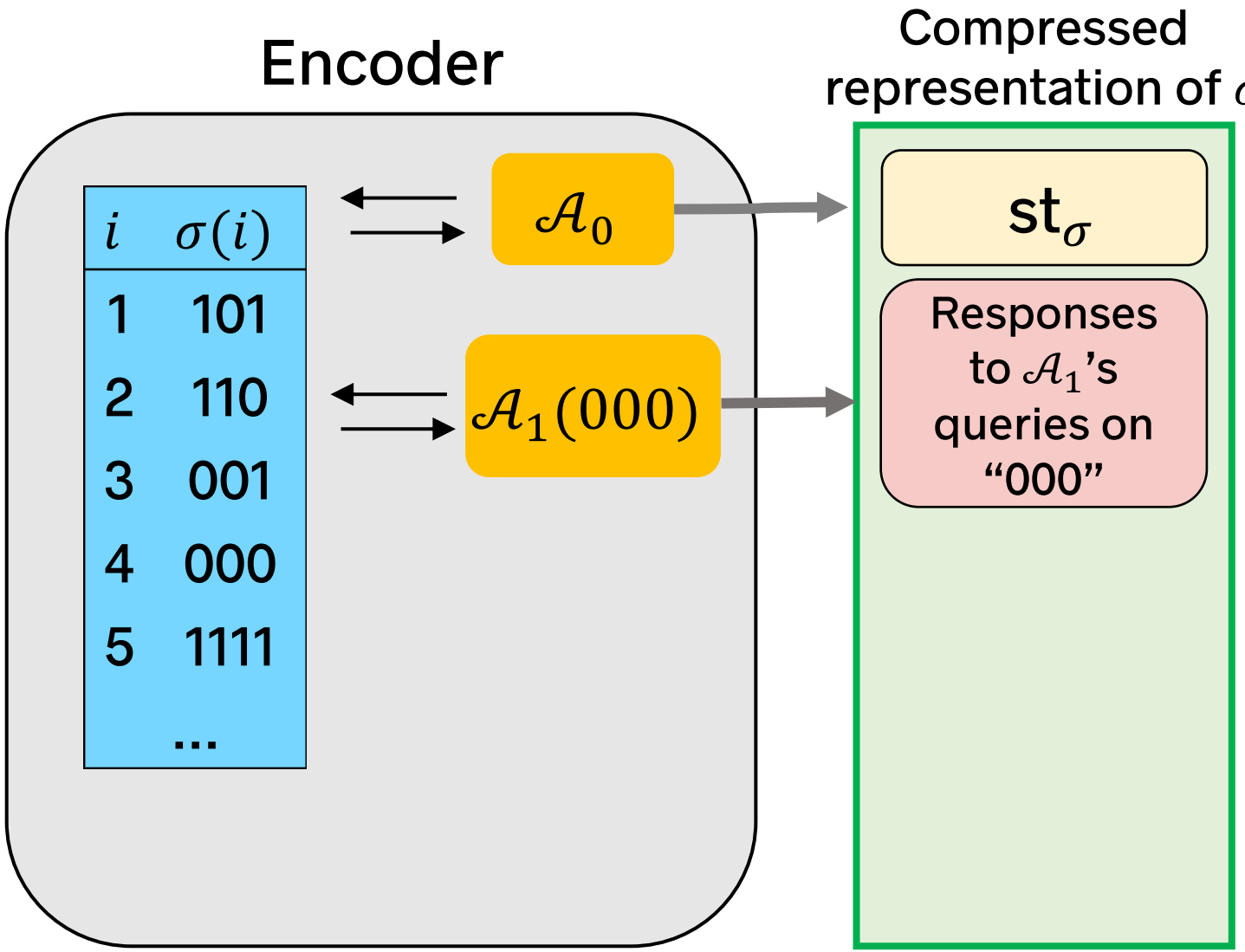
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]



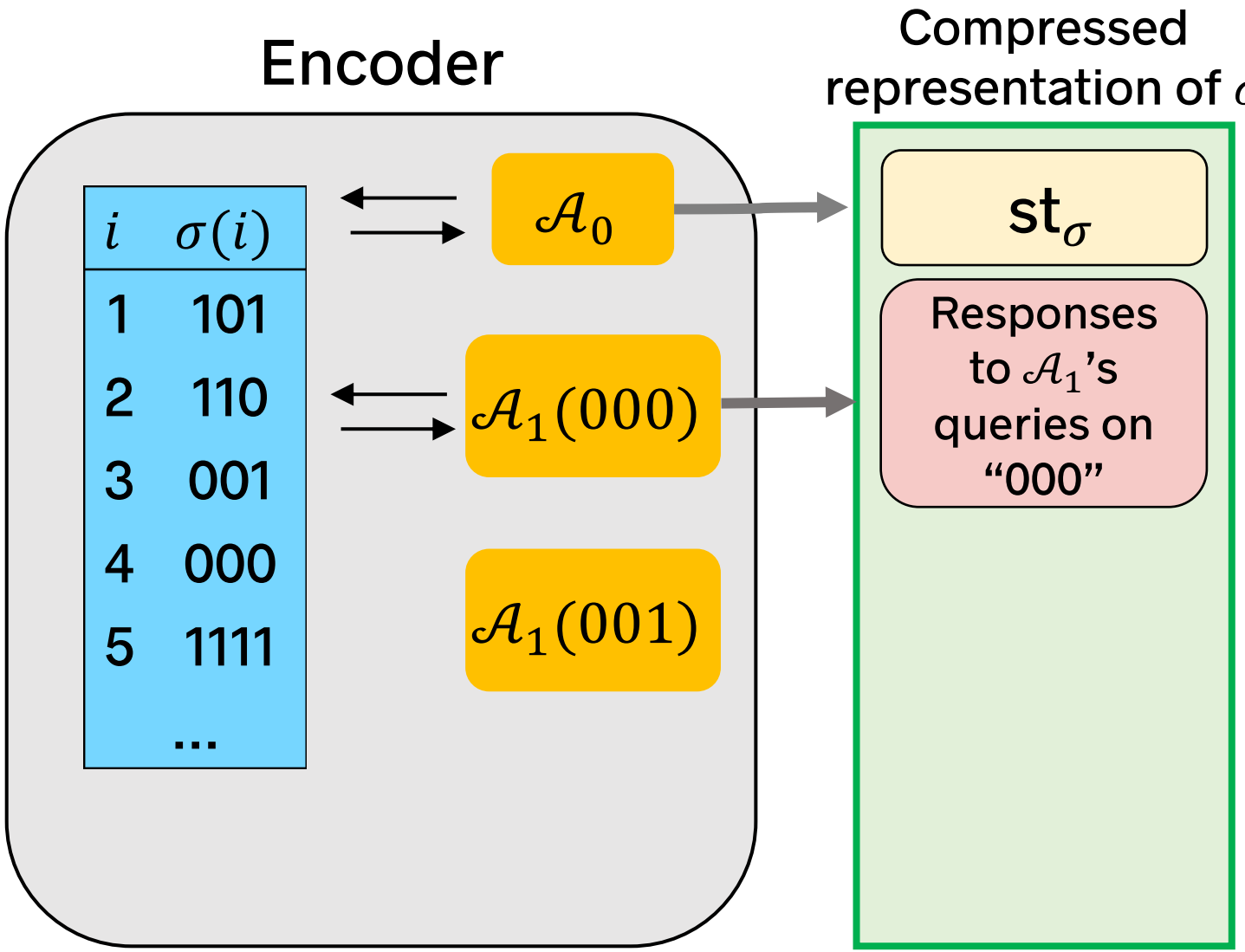
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]



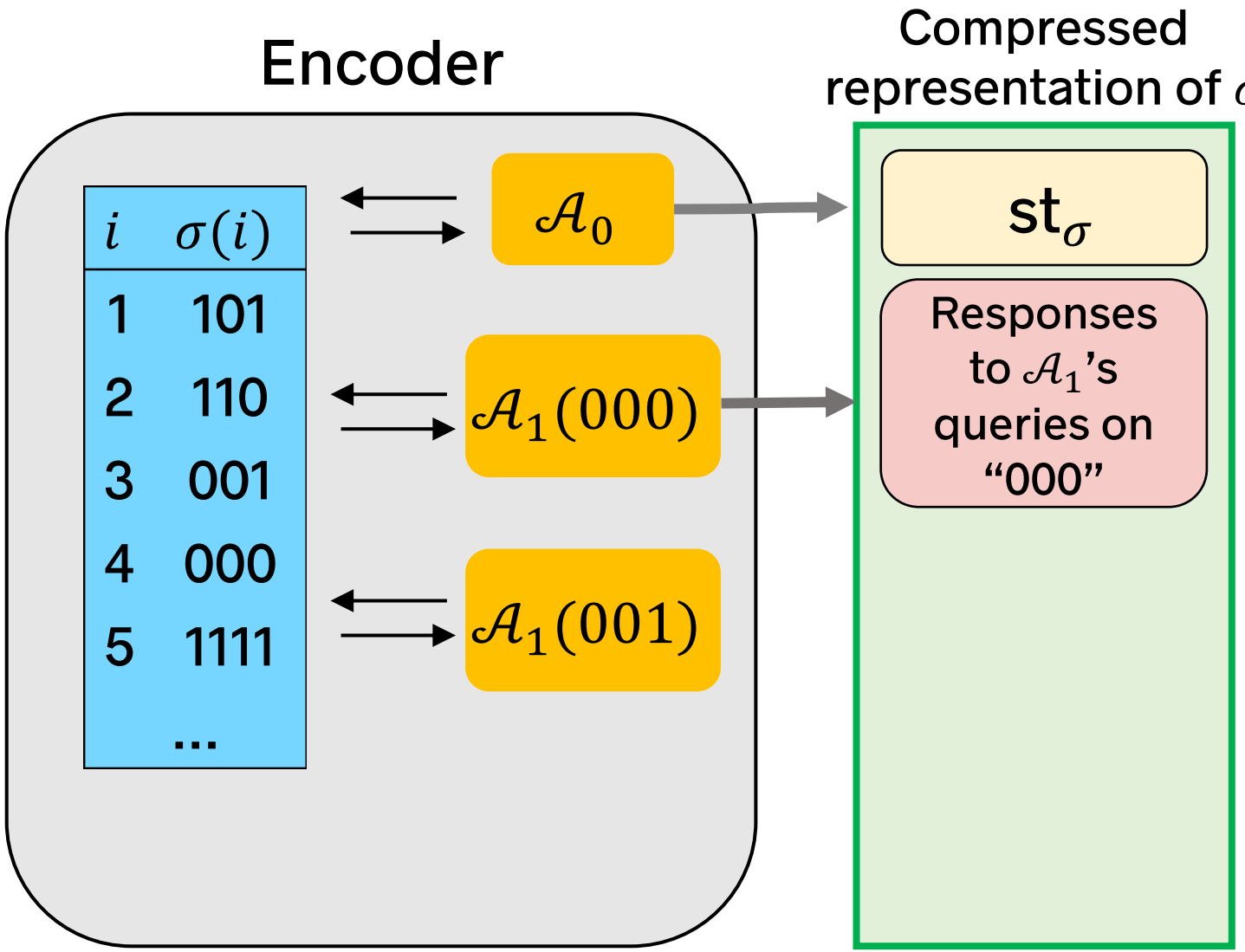
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]



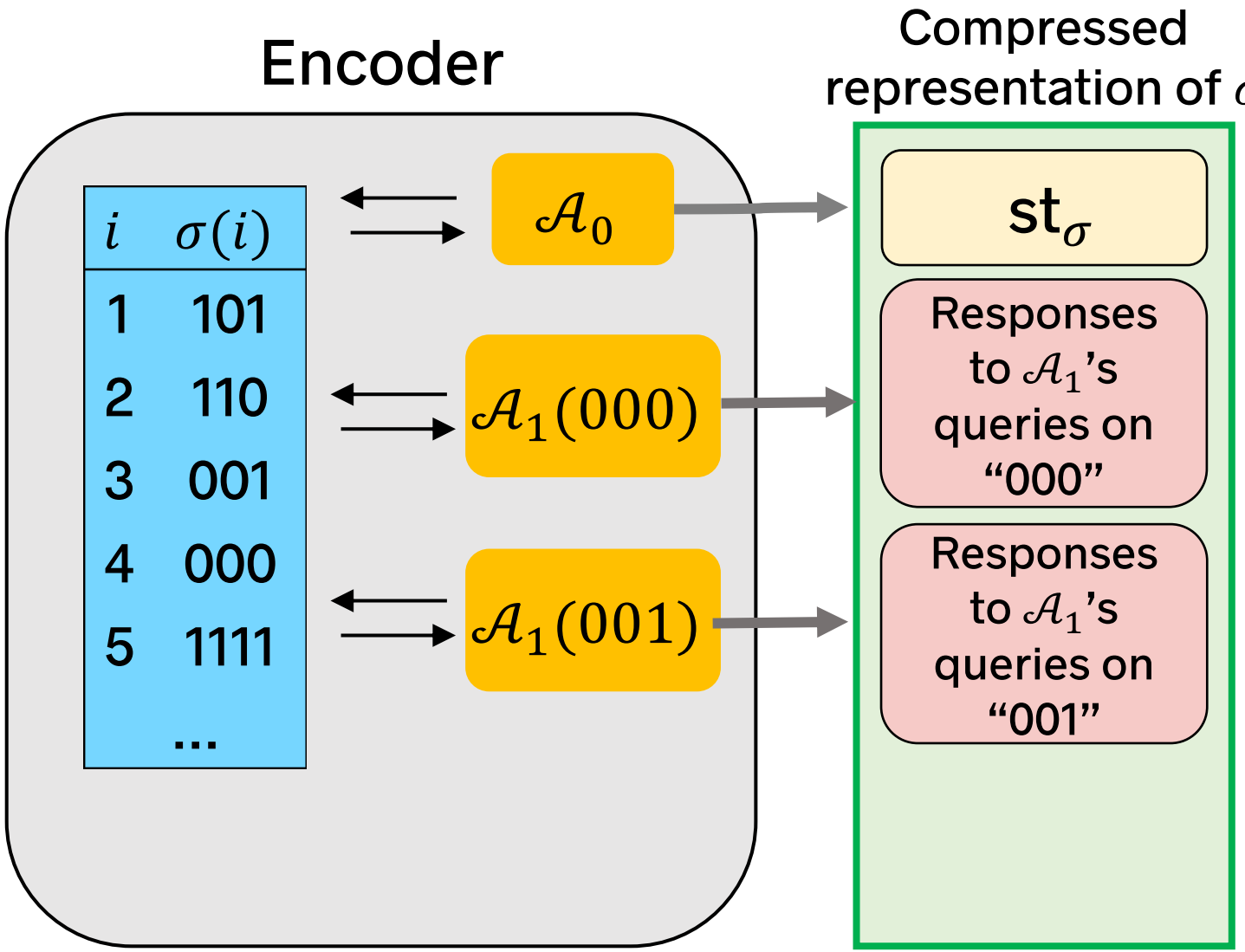
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]



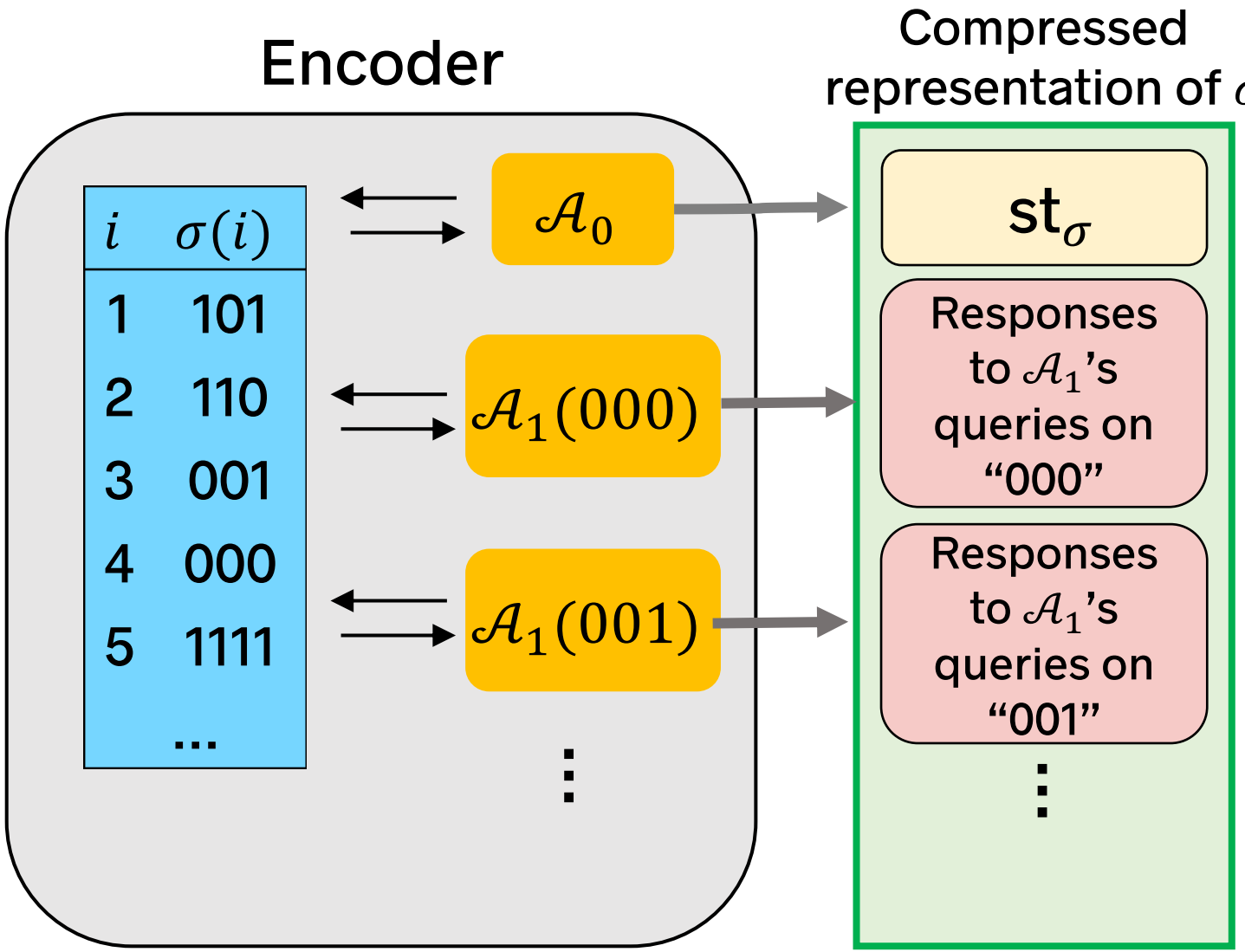
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]



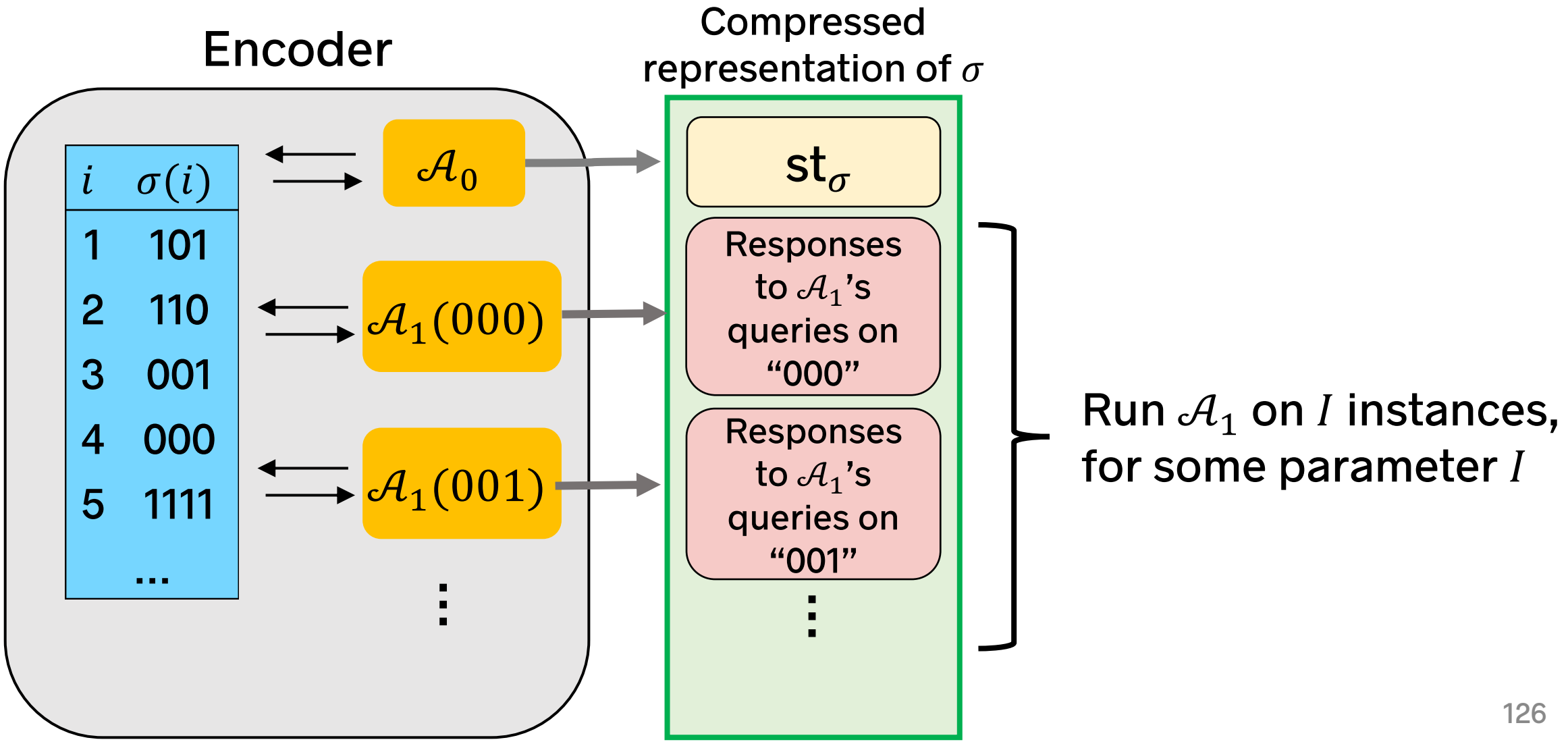
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]



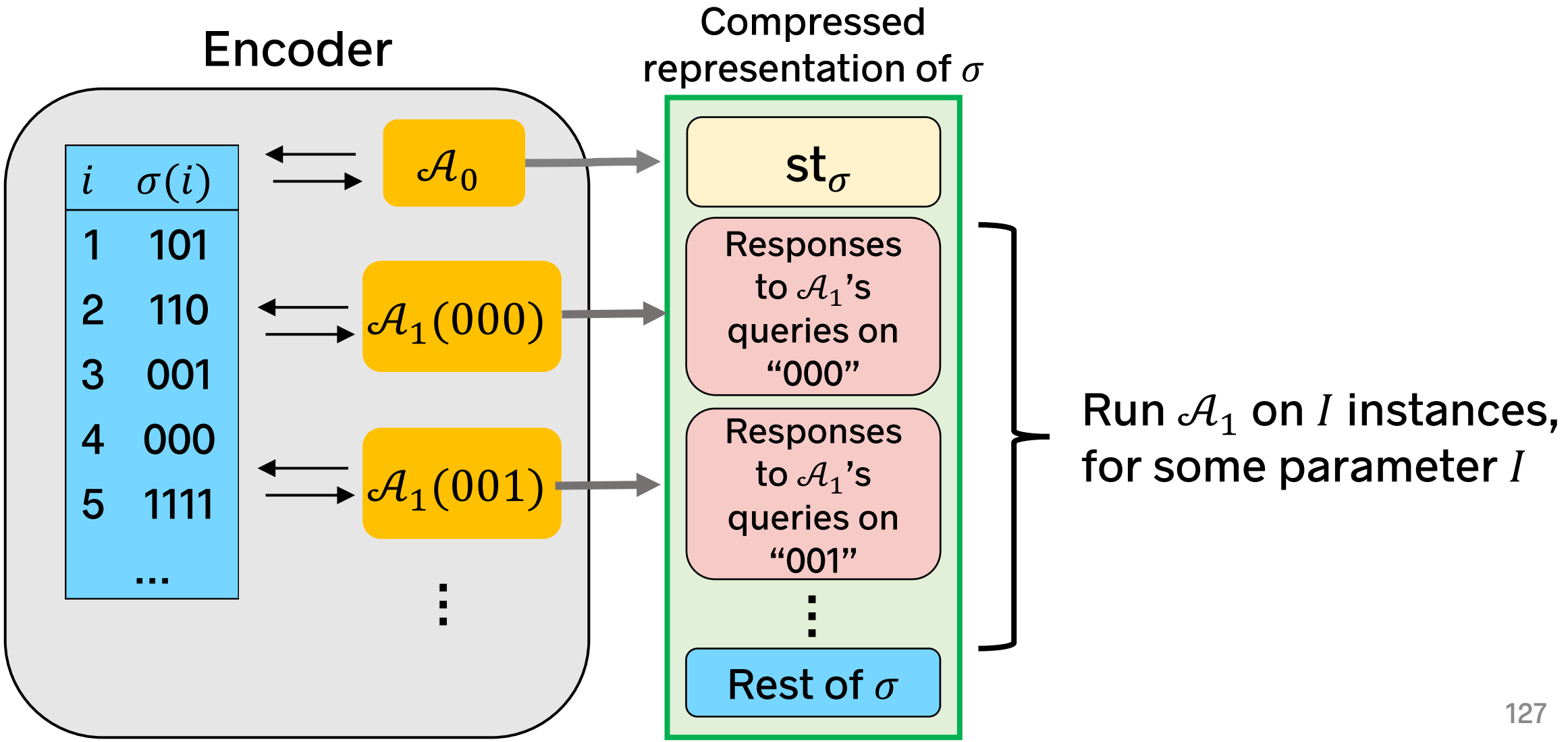
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]



Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

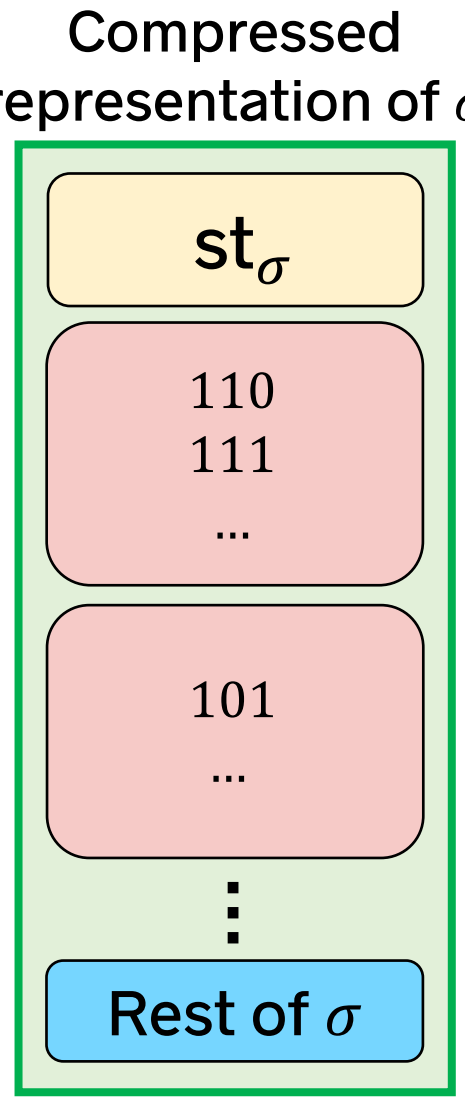


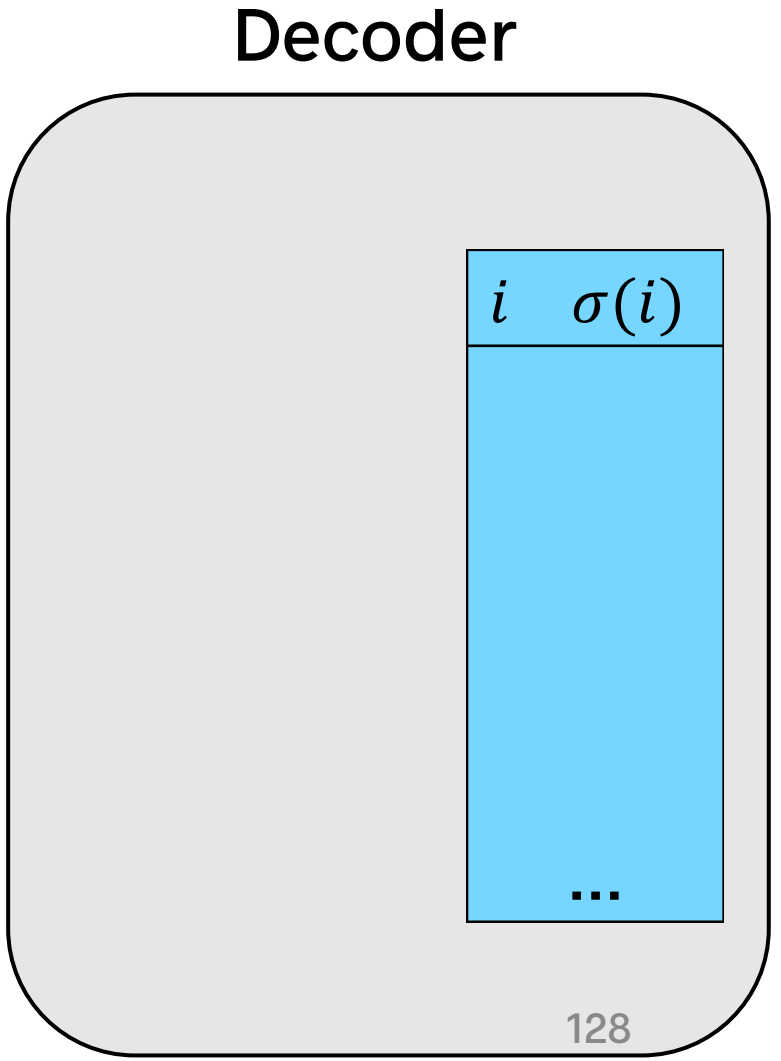
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]



Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

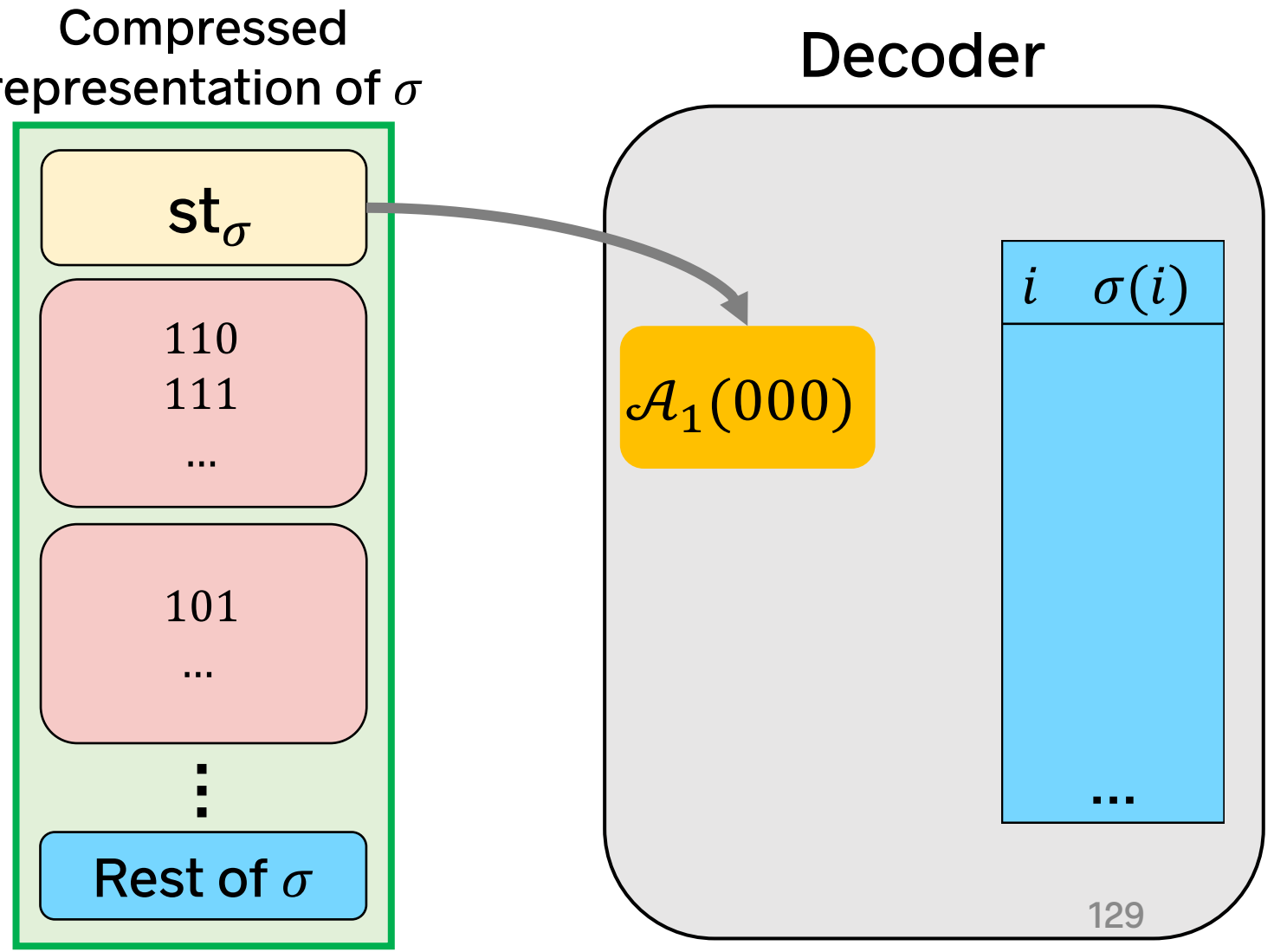
- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"





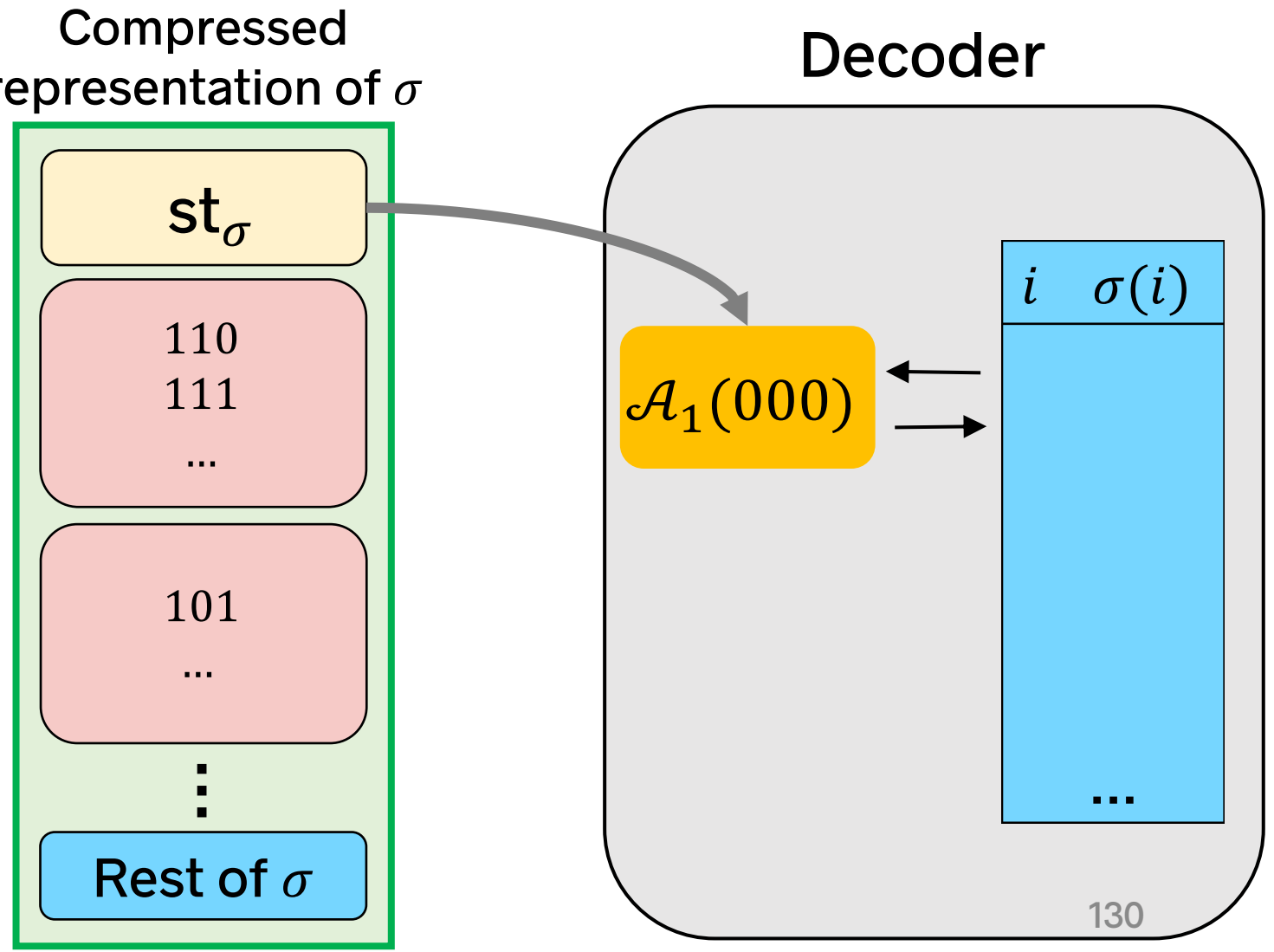
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"



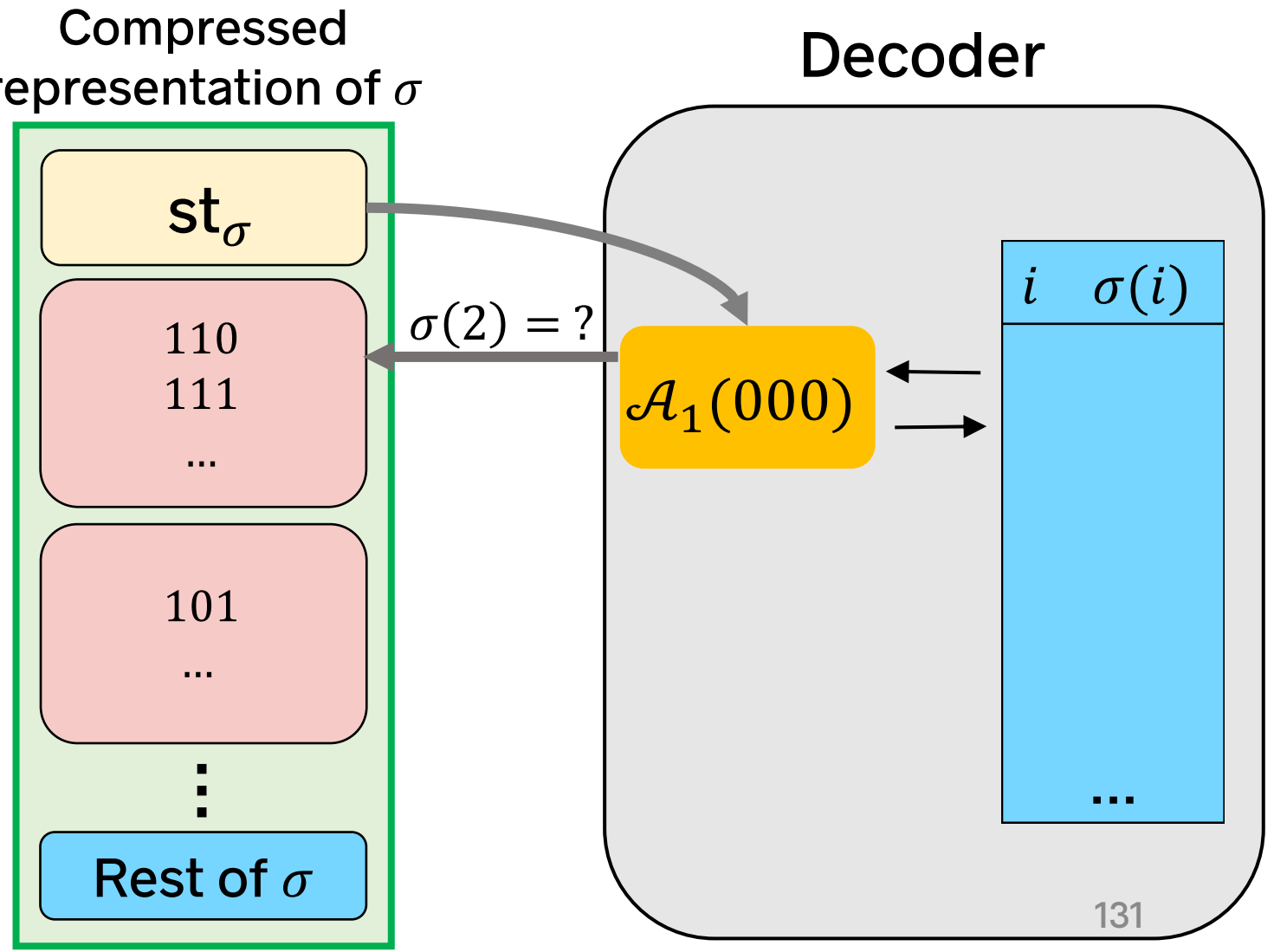
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"



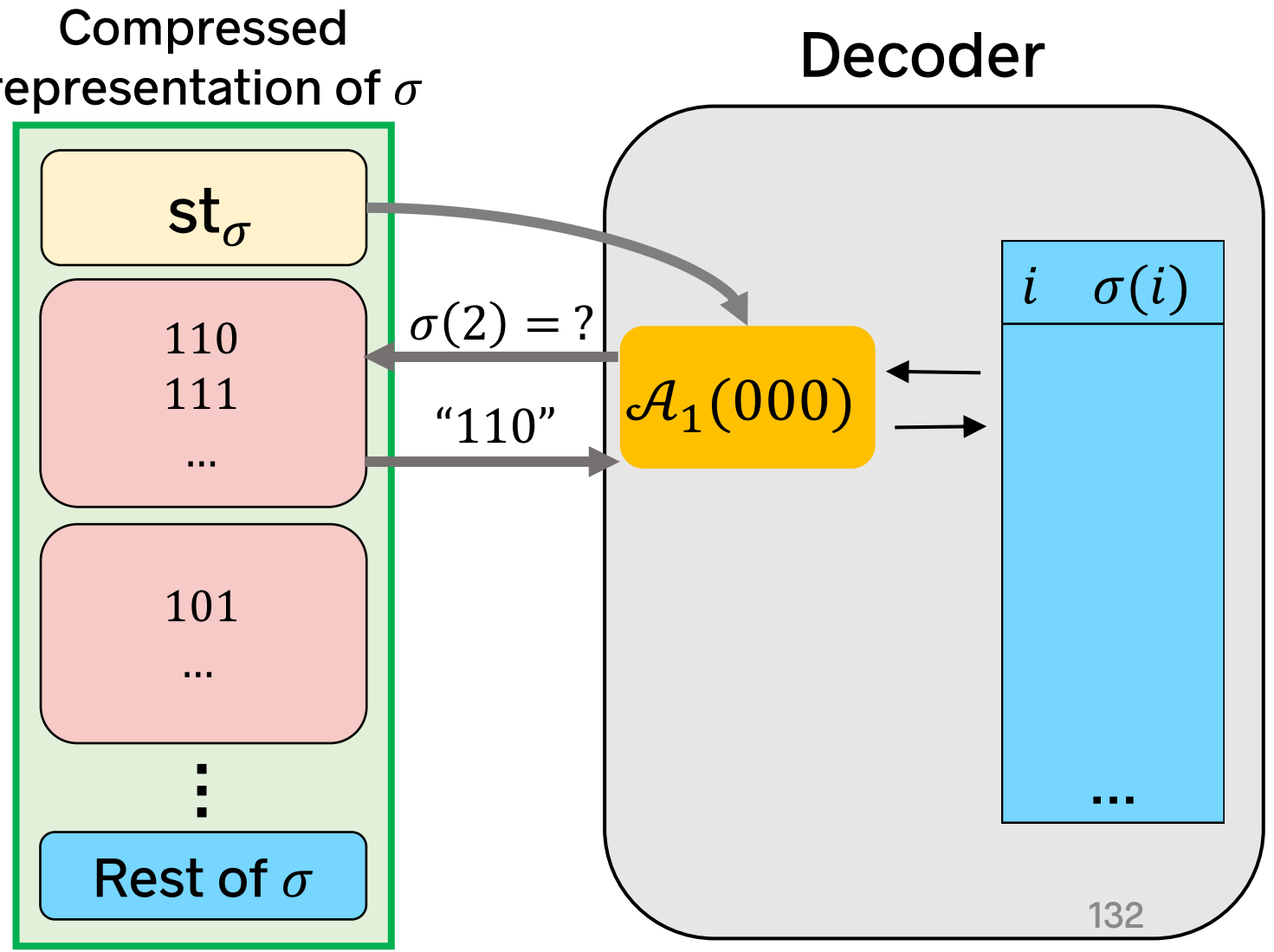
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"



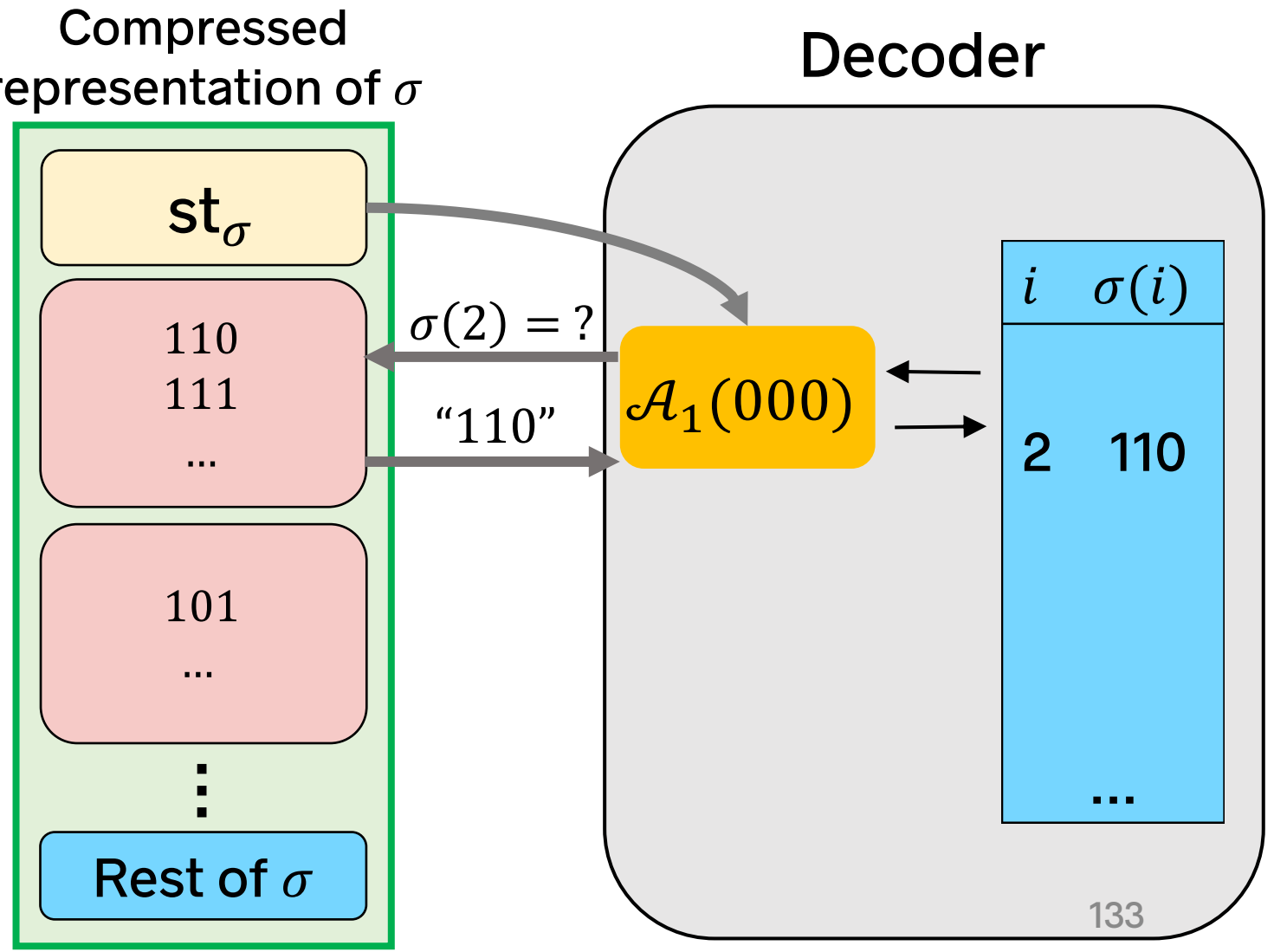
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"



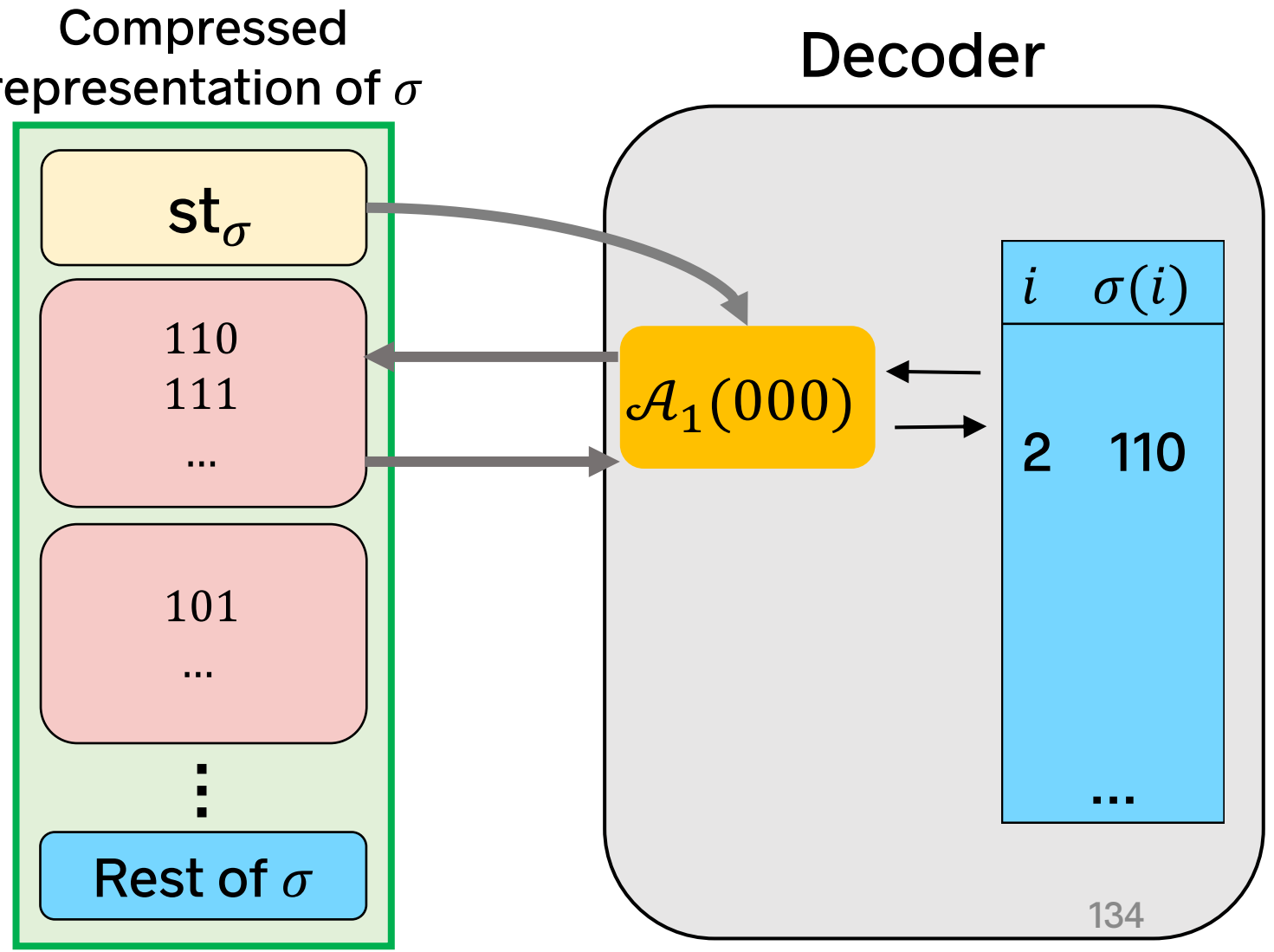
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"



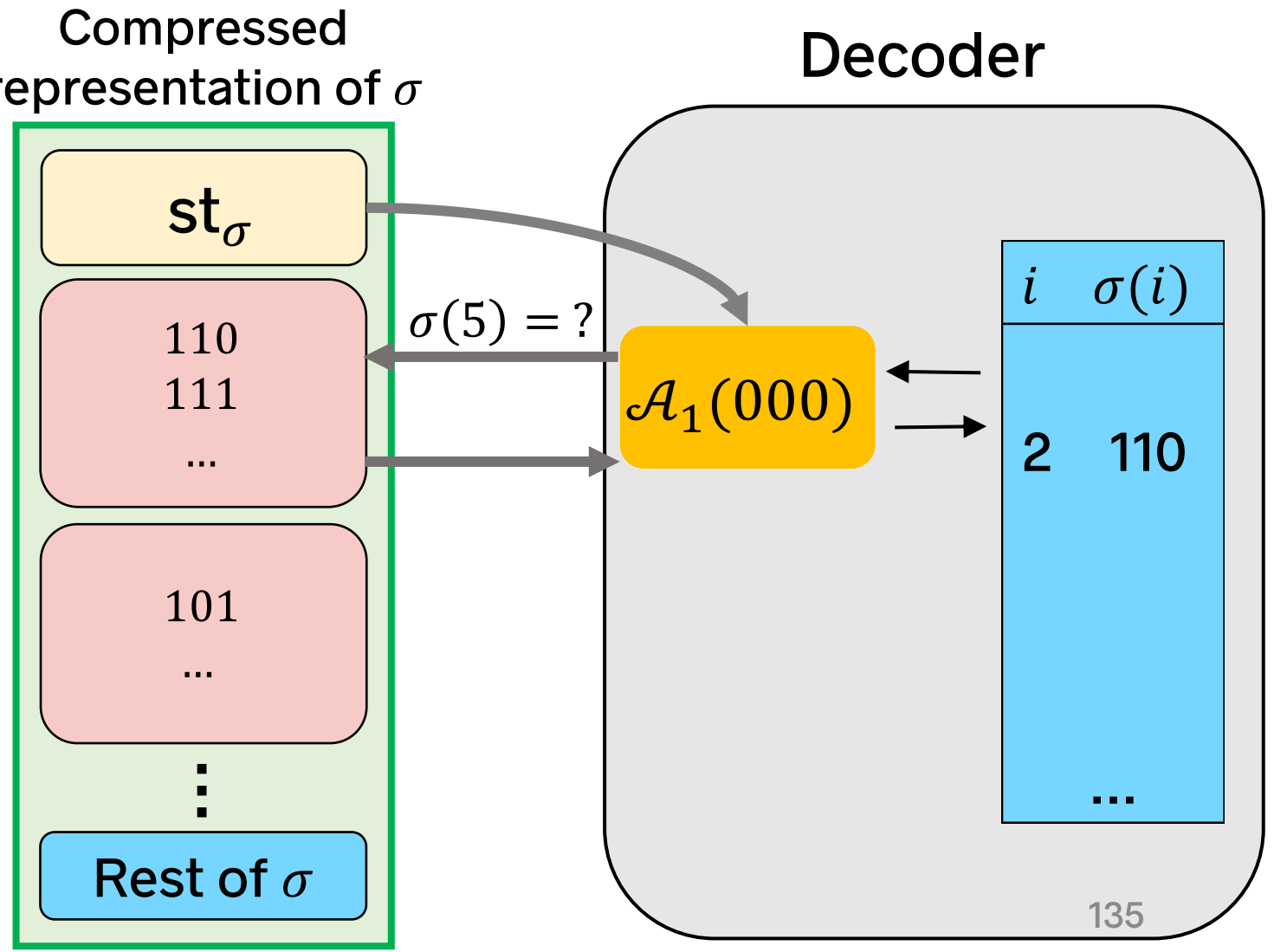
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"



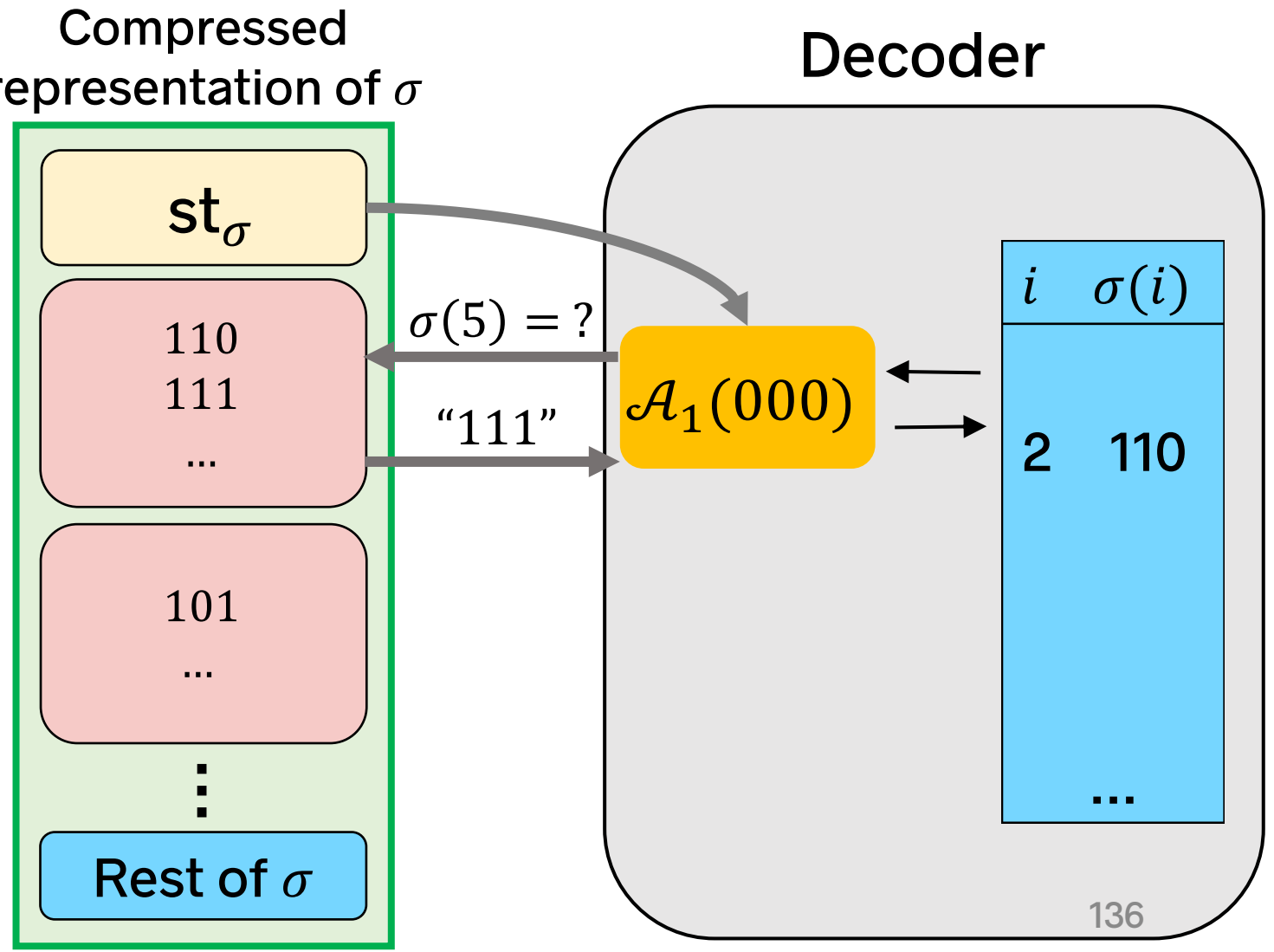
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"



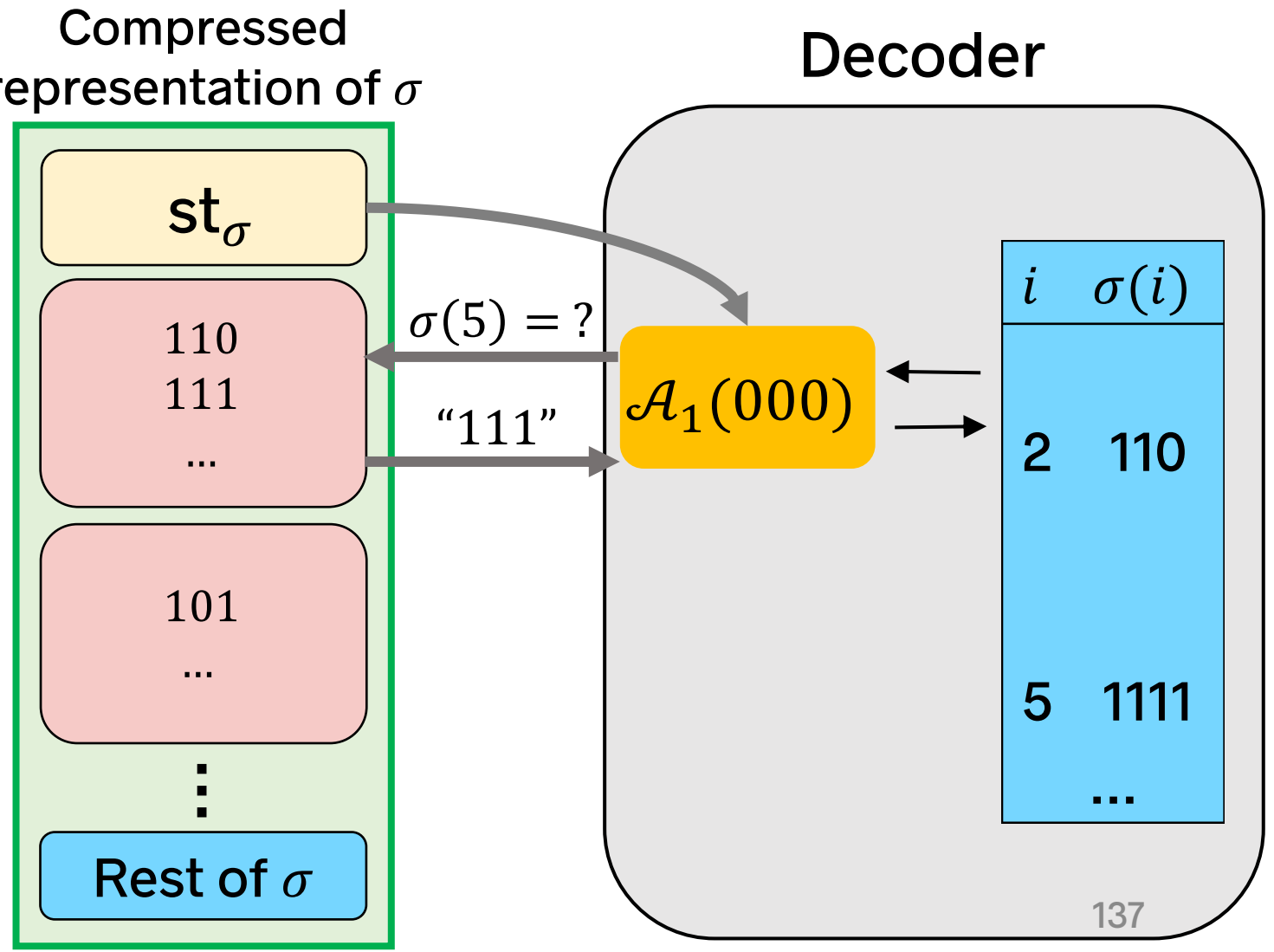
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"



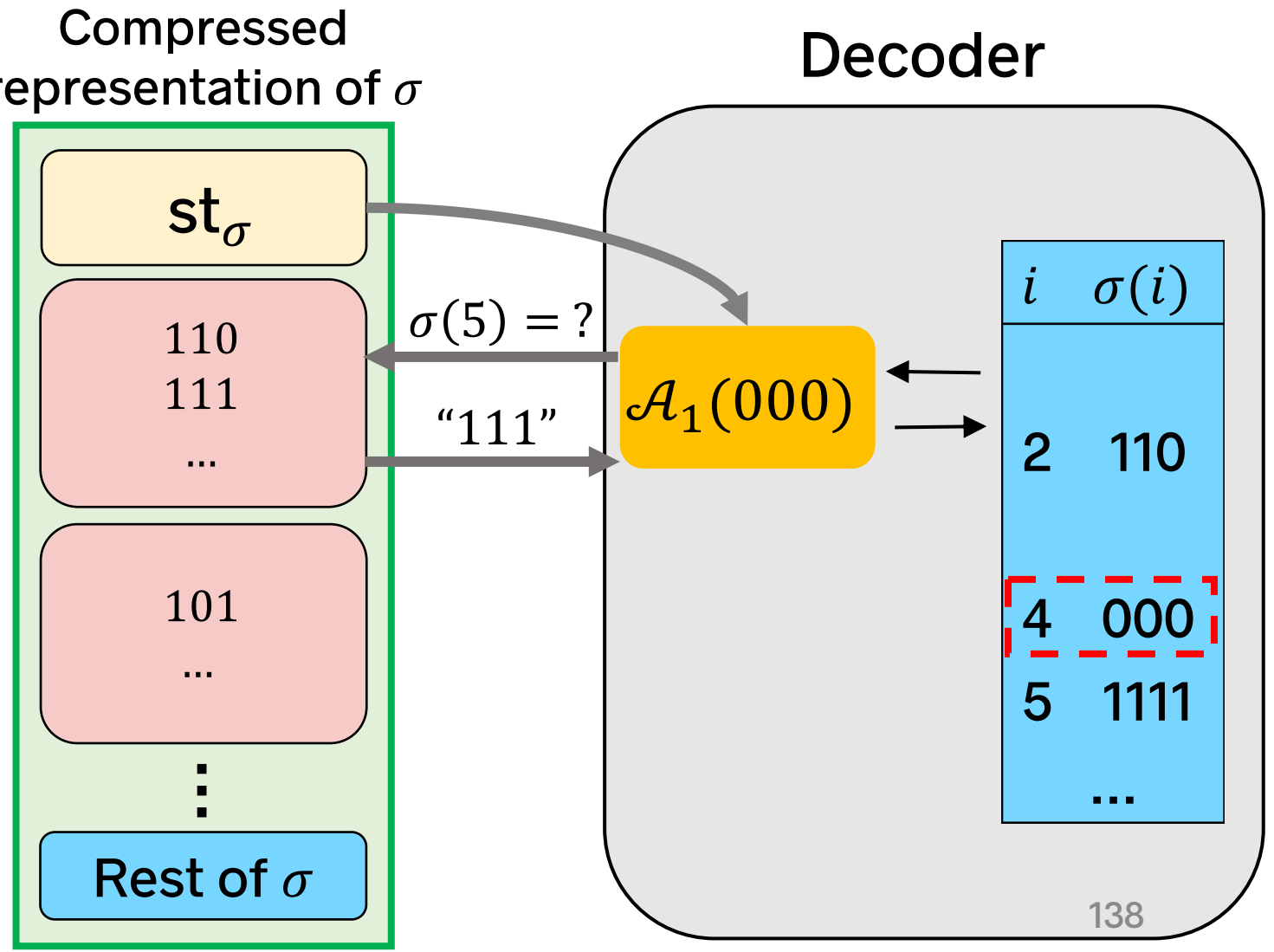
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"



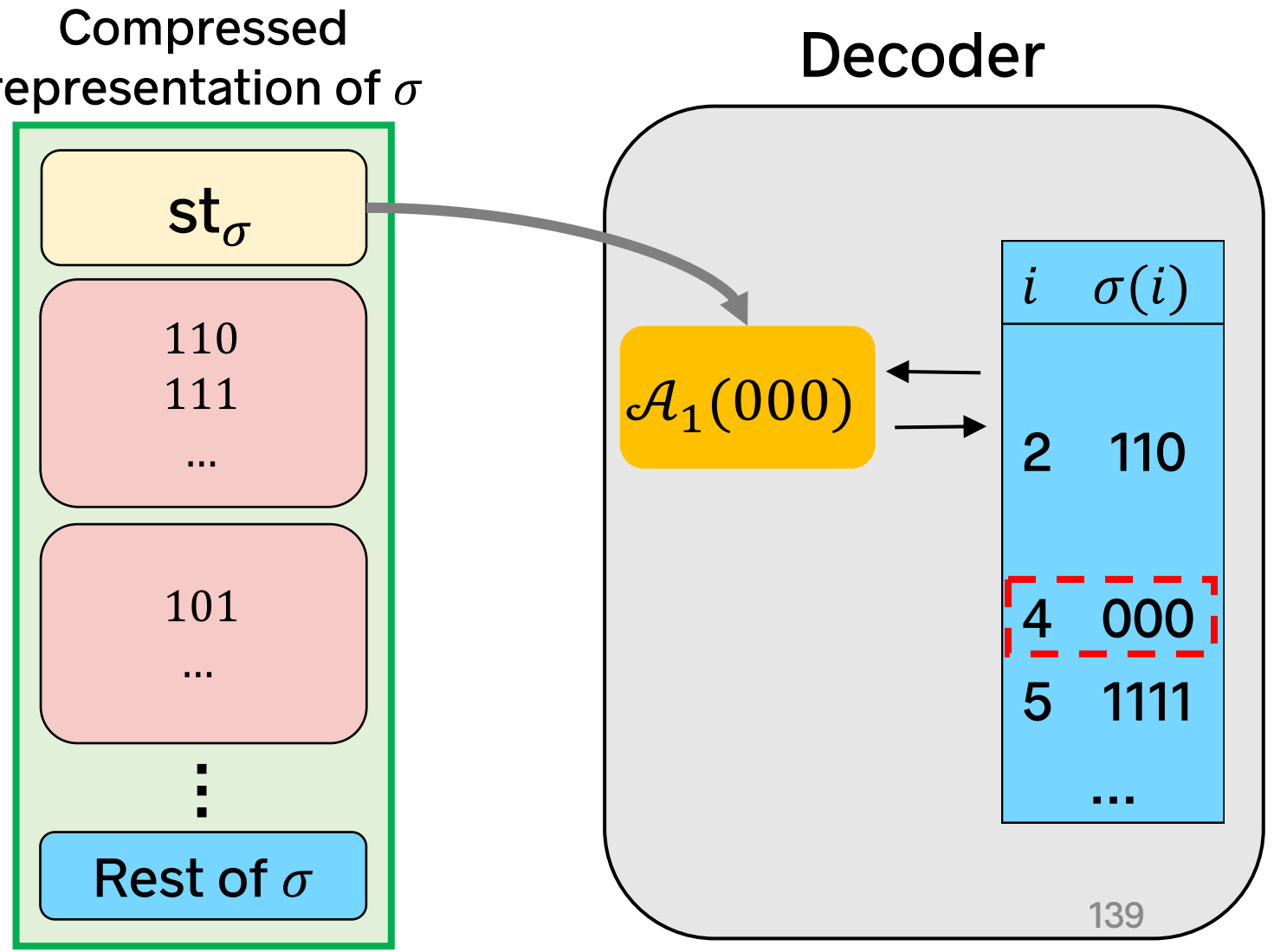
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"



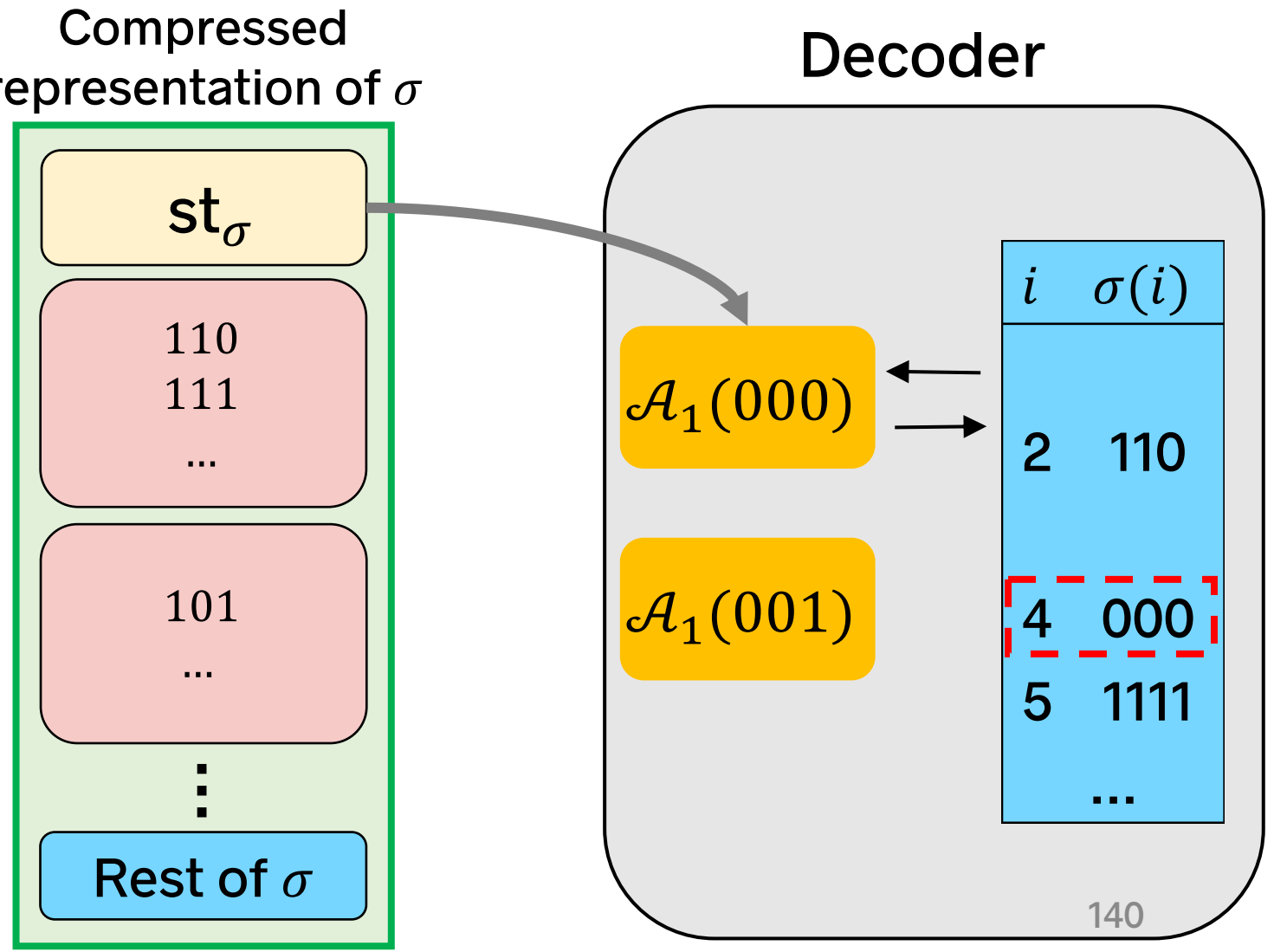
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"



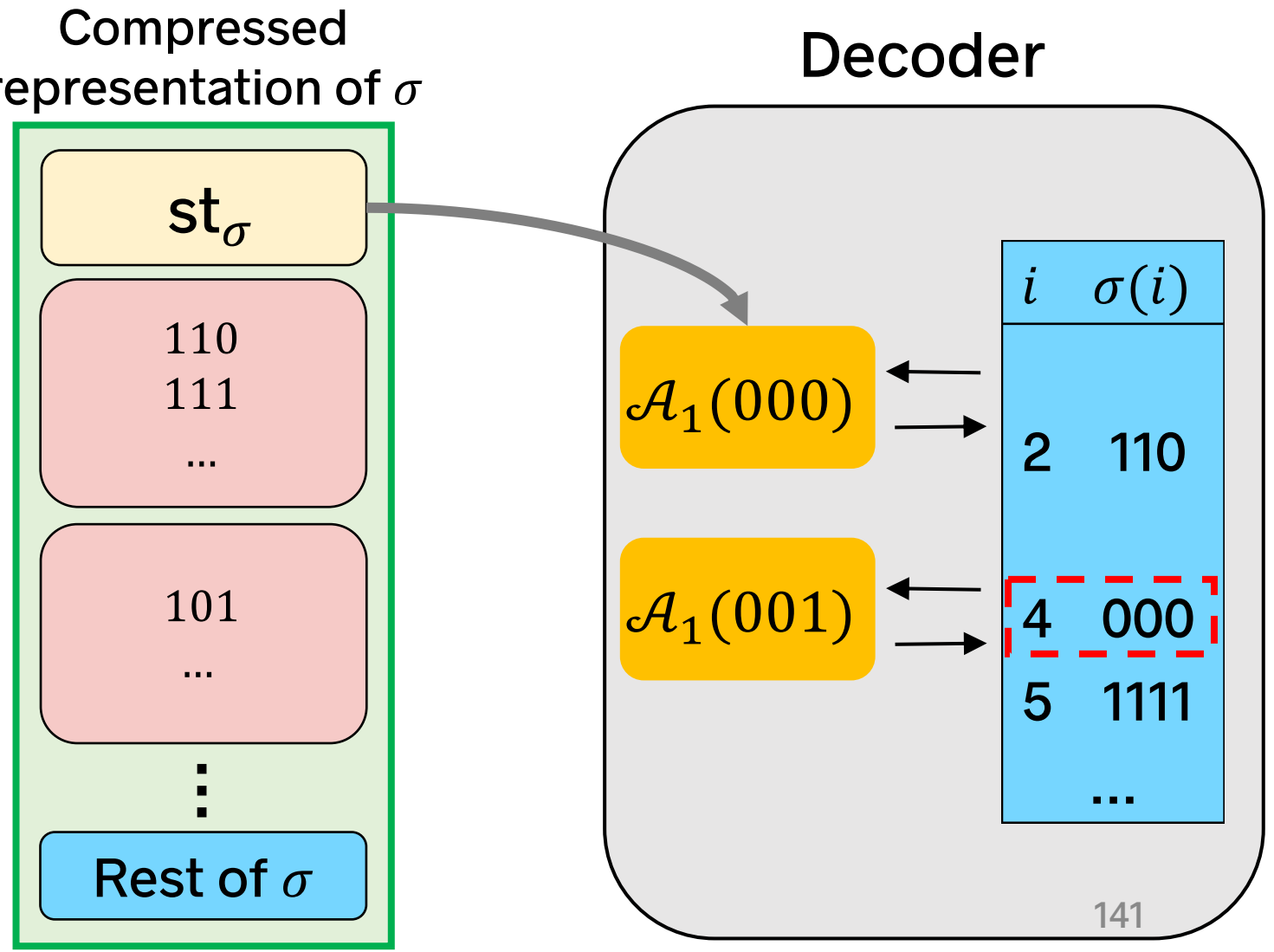
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"



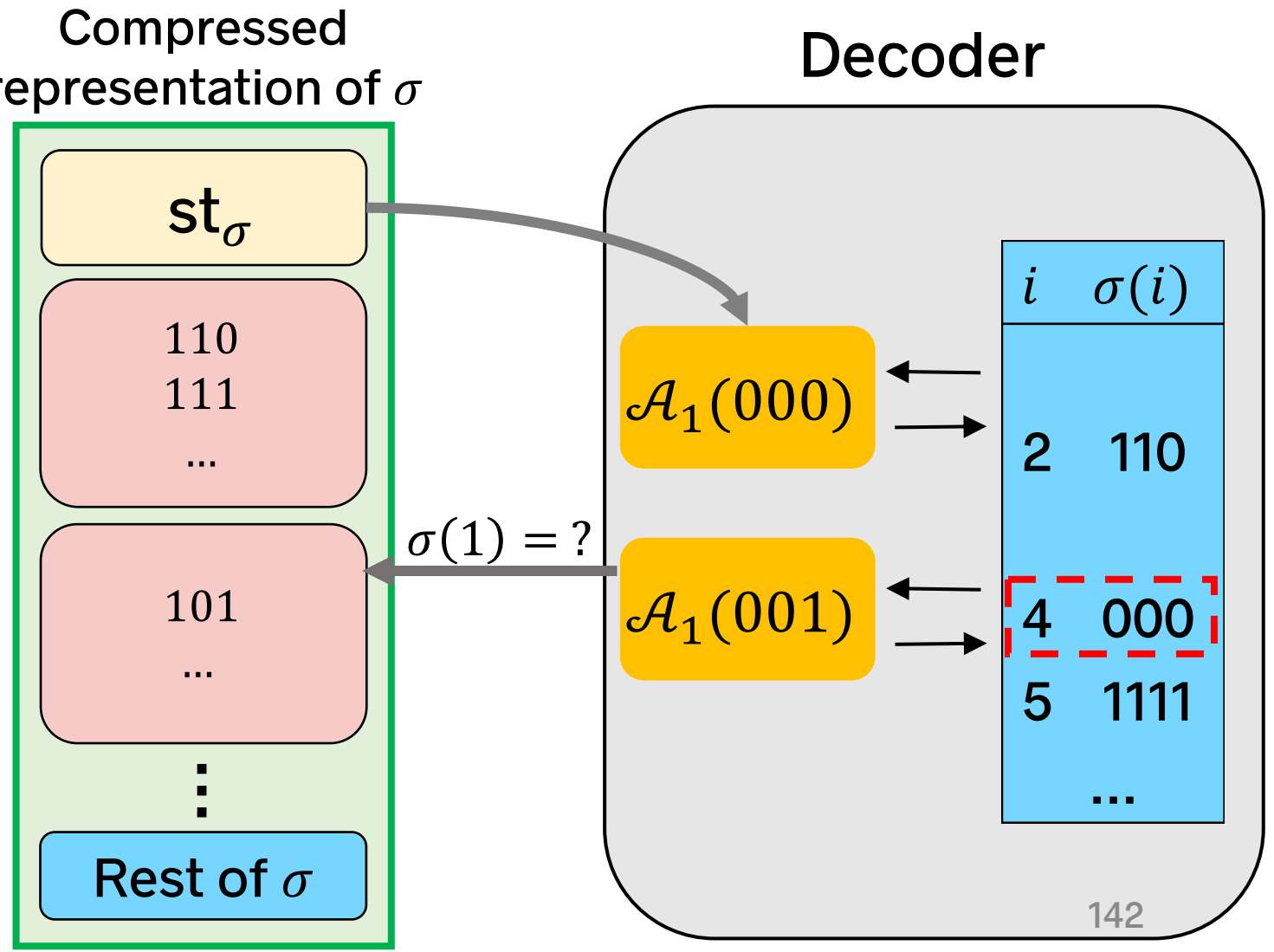
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"



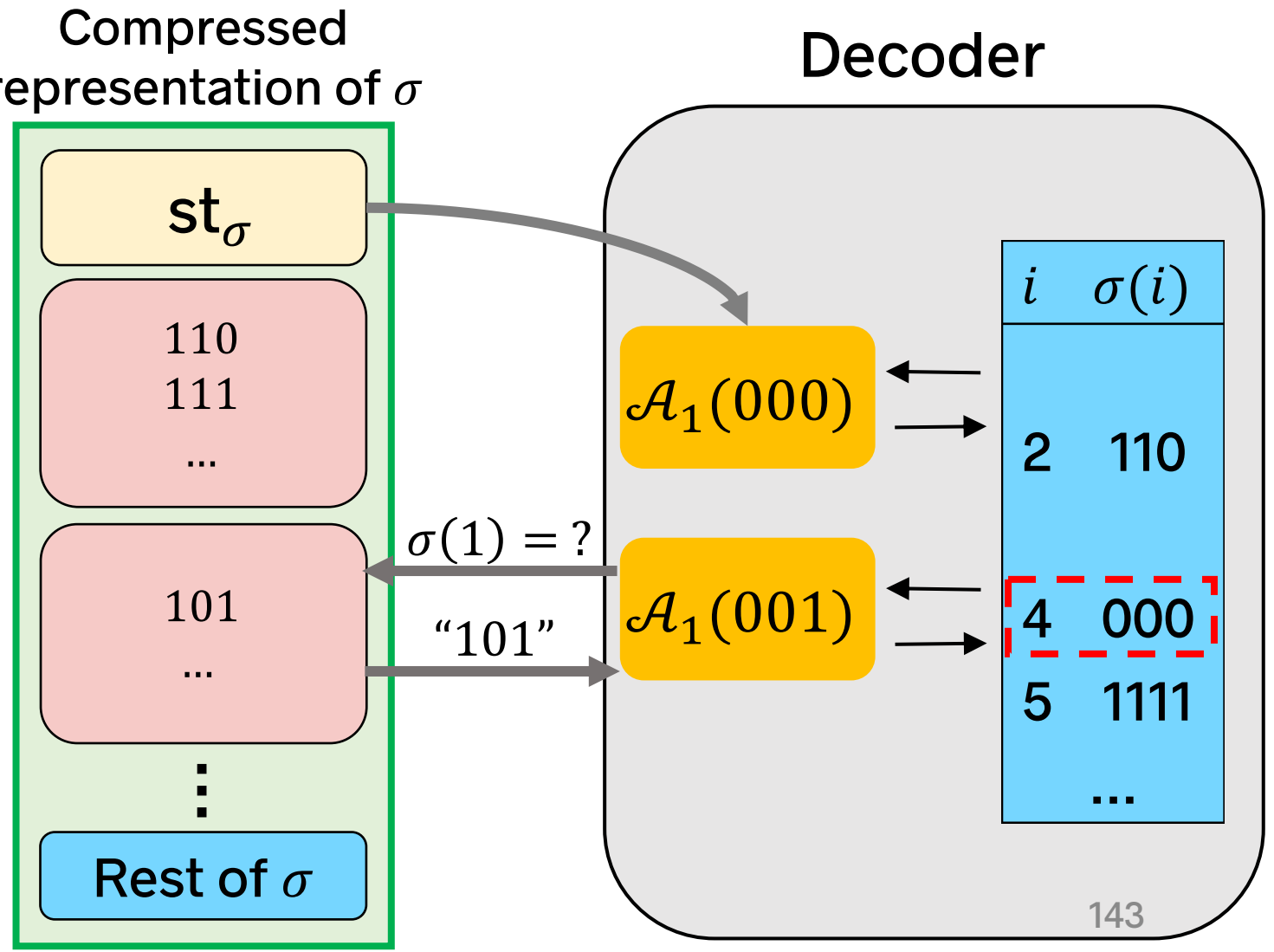
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"



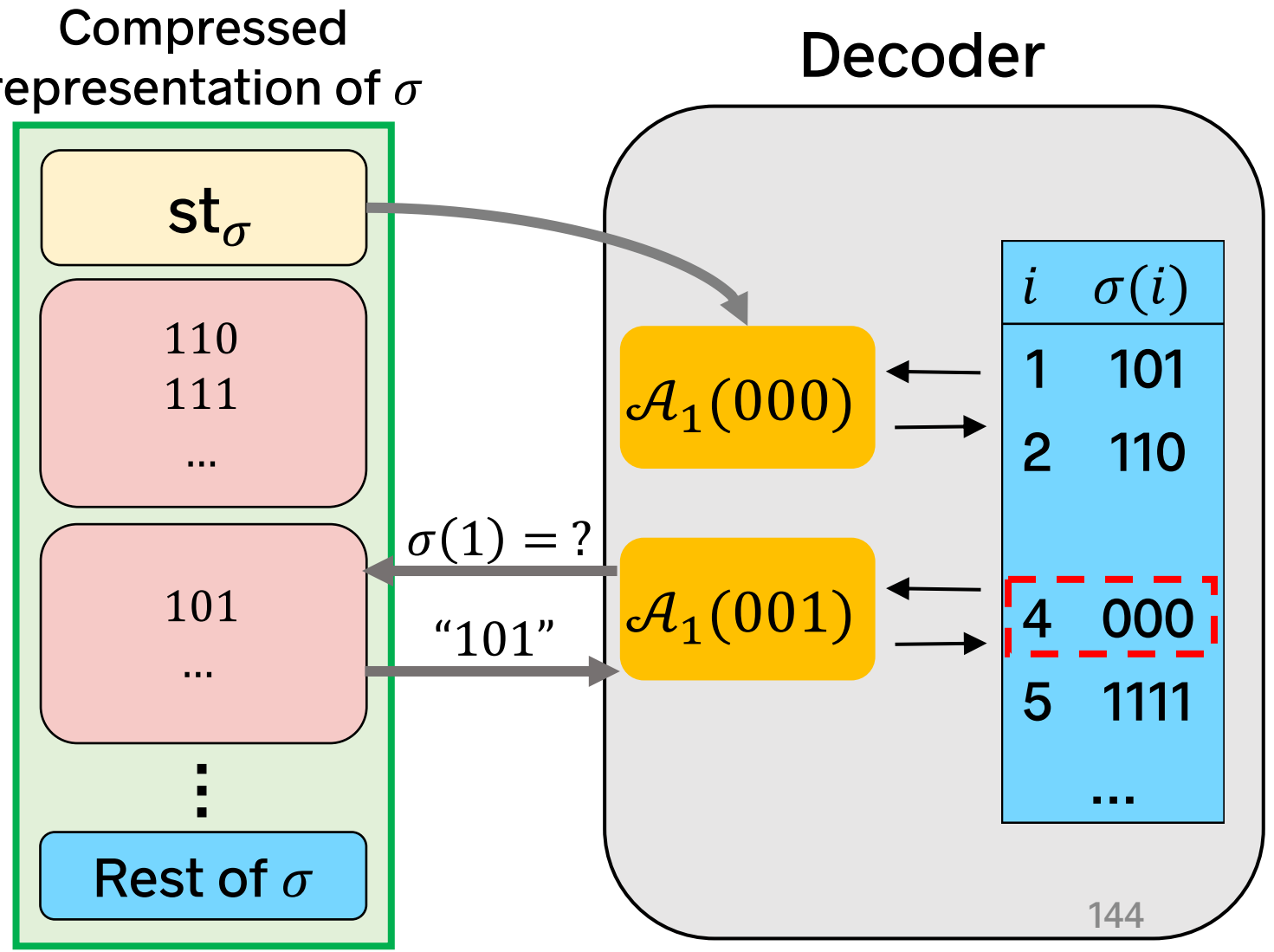
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"



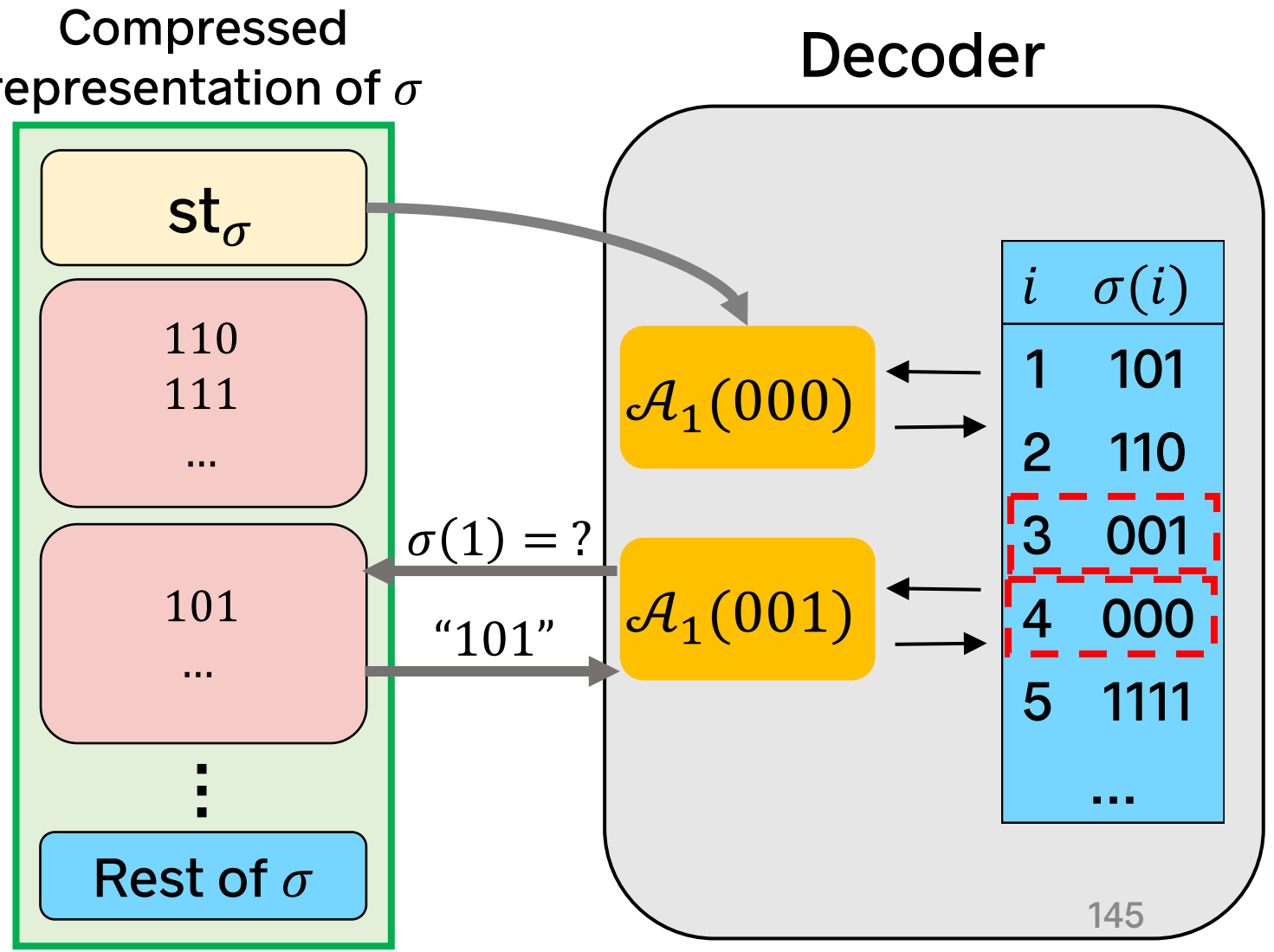
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"



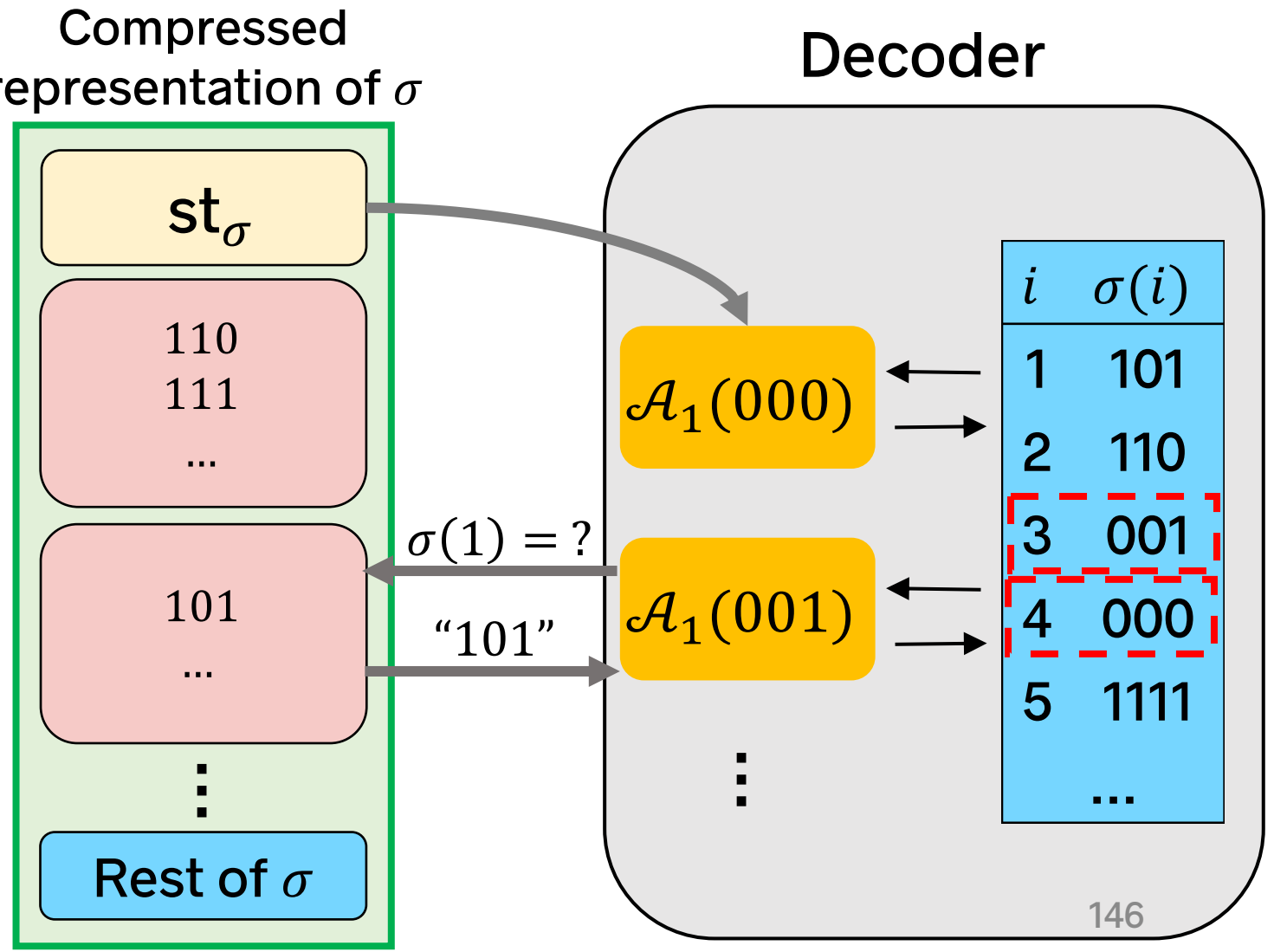
Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"



Proof idea: Use preprocessing dlog adversary $(\mathcal{A}_0, \mathcal{A}_1)$ to build a compressed representation of the mapping σ. [Yao90, GT00, DHT12]

- Run $\mathcal{A}_1$ on I instances
- Whenever $\mathcal{A}_1$ outputs a dlog, we get one value $\sigma(i)$ "for free"



Claim: Each invocation of $\mathcal{A}_1$ allows the encoder to compress σ by at least one bit.

Easy case: The response to all of $\mathcal{A}_1$'s queries are distinct

- $\mathcal{A}_1$ outputs a discrete log "for free" $\Rightarrow$ Compress by $\approx \log N$ bits

Harder case: The response to query t is the same as the response to query $t' < t$.

- A naïve encoding "pays twice" for the same value $\sigma(i) \Rightarrow$ No savings ☹
- Instead, encoder writes a pointer to query t'

  If the encoder runs $\mathcal{A}_1$ on I instances, requires $\underbrace{\log IT}_{\text{Pointer to query } t'} + \underbrace{\log T}_{\text{Index of query } t}$ bits.

[DHT12] treats a more difficult version of "hard case"

Claim: Each invocation of $\mathcal{A}_1$ allows the encoder to compress σ by at least one bit.

Easy case: The response to all of $\mathcal{A}_1$'s queries are distinct

- $\mathcal{A}_1$ outputs a discrete log "for free" ⇒ Compress by $\approx \log N$ bits

Harder case: The response to query t is the same as the response to query $t' < t$.

- A naïve encoding "pays twice" for the same value $\sigma(i)$ ⇒ No savings ☹
- Instead, encoder writes a pointer to query t'

  If the encoder runs $\mathcal{A}_1$ on I instances, requires $\underbrace{\log IT}_{\text{Pointer to query } t'} + \underbrace{\log T}_{\text{Index of query } t}$ bits.

[DHT12] treats a more difficult version of "hard case"

Claim: Each invocation of $\mathcal{A}_1$ allows the encoder to compress σ by at least one bit.

Easy case: The response to all of $\mathcal{A}_1$'s queries are distinct

- $\mathcal{A}_1$ outputs a discrete log "for free" $\Rightarrow$ Compress by $\approx \log N$ bits

Harder case: The response to query t is the same as the response to query $t' < t$.

- A naïve encoding "pays twice" for the same value $\sigma(i) \Rightarrow$ No savings ☹
- Instead, encoder writes a pointer to query t'

  If the encoder runs $\mathcal{A}_1$ on I instances, requires $\underbrace{\log IT}_{\text{Pointer to query } t'} + \underbrace{\log T}_{\text{Index of query } t}$ bits.

[DHT12] treats a more difficult version of "hard case"

Claim: Each invocation of $\mathcal{A}_1$ allows the encoder to compress σ by at least one bit.

Easy case: The response to all of $\mathcal{A}_1$'s queries are distinct

- $\mathcal{A}_1$ outputs a discrete log "for free" ⇒ Compress by $\approx \log N$ bits

Harder case: The response to query t is the same as the response to query $t' < t$.

- A naïve encoding "pays twice" for the same value $\sigma(i)$ ⇒ No savings ☹
- Instead, encoder writes a pointer to query t'

  If the encoder runs $\mathcal{A}_1$ on I instances, requires $\log IT + \log T$ bits.

  $\log IT$: Pointer to query t'; $\log T$: Index of query t

[DHT12] treats a more difficult version of "hard case"

Claim: Each invocation of $\mathcal{A}_1$ allows the encoder to compress σ by at least one bit.

Easy case: The response to all of $\mathcal{A}_1$'s queries are distinct

- $\mathcal{A}_1$ outputs a discrete log "for free" $\Rightarrow$ Compress by $\approx \log N$ bits

Harder case: The response to query t is the same as the response to query $t' < t$.

- A naïve encoding "pays twice" for the same value $\sigma(i) \Rightarrow$ No savings ☹
- Instead, encoder writes a pointer to query t'

  If the encoder runs $\mathcal{A}_1$ on I instances, requires $\underbrace{\log IT}_{\text{Pointer to query } t'} + \underbrace{\log T}_{\text{Index of query } t}$ bits.

[DHT12] treats a more difficult version of "hard case"

Claim: Each invocation of $\mathcal{A}_1$ allows the encoder to compress σ by at least one bit.

Easy case: The response to all of $\mathcal{A}_1$'s queries are distinct

- $\mathcal{A}_1$ outputs a discrete log "for free" $\Rightarrow$ Compress by $\approx \log N$ bits

Harder case: The response to query t is the same as the response to query $t' < t$.

- A naïve encoding "pays twice" for the same value $\sigma(i) \Rightarrow$ No savings ☹
- Instead, encoder writes a pointer to query t'

  If the encoder runs $\mathcal{A}_1$ on I instances, requires $\log IT + \log T$ bits.

  $\log IT$: Pointer to query t'; $\log T$: Index of query t

Each execution of $\mathcal{A}_1$ saves at least 1 bit, when:

$$\log IT^2 < \log N, \text{ or } I < N/T^2$$

[DHT12] treats a more difficult version of "hard case"

Completing the proof

- We run the adversary $\mathcal{A}_1$ on $I = N/T^2$ instances
- Each execution compresses by ≥ 1 bit
- BUT, we have to include the S-bit advice string in the encoding

$$\text{Encoding overhead} = S - \frac{N}{T^2} \geq 0 \qquad \Rightarrow \qquad ST^2 = \Omega(N) \quad \blacksquare$$

Extra complications

- Algorithms that succeed on an ϵ-fraction of group elements
  - Use the random self-reducibility of dlog
  - Hardcode a good set of random coins for $\mathcal{A}_1$ into $\mathbf{Enc}(\sigma)$

- Decisional type problems (DDH, etc.)
  - $\mathcal{A}_1$ only outputs 1 bit—prior argument fails because encoding the runtime in $\log T$ bits is too expensive
  - Run $\mathcal{A}_1$ on batches of inputs [See paper for details]

What about Decision Diffie–Hellman (DDH)?

DDH problem: Distinguish (g, g^x, g^y, g^{xy}) from (g, g^x, g^y, g^z)

	Upper bound	Lower bound	Time T
Discrete log:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon N)$	$N^{1/4}$
CDH:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon N)$	$N^{1/4}$
DDH:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon^2 N)$	$\leq N^{1/4}$ $\geq N^{1/8}$

For $\epsilon = N^{-1/4}$ $S = N^{1/4}$

What about Decision Diffie–Hellman (DDH)?

DDH problem: Distinguish (g, g^x, g^y, g^{xy}) from (g, g^x, g^y, g^z)

	Upper bound	Lower bound	Time T
Discrete log:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon N)$	$N^{1/4}$
CDH:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon N)$	$N^{1/4}$
DDH:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon^2 N)$	$\leq N^{1/4}$ $\geq N^{1/8}$

For $\epsilon = N^{-1/4}$, $S = N^{1/4}$

What about Decision Diffie-Hellman (DDH)?

DDH problem: Distinguish (g, g^x, g^y, g^{xy}) from (g, g^x, g^y, g^z)

	Upper bound	Lower bound	Time T
Discrete log:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon N)$	$N^{1/4}$
CDH:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon N)$	$N^{1/4}$
DDH:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon^2 N)$	$\leq N^{1/4}$ $\geq N^{1/8}$

For $\epsilon = N^{-1/4}$, $S = N^{1/4}$

What about Decision Diffie-Hellman (DDH)?

DDH problem: Distinguish (g, g^x, g^y, g^{xy}) from (g, g^x, g^y, g^z)

	Upper bound	Lower bound	Time T
Discrete log:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon N)$	$N^{1/4}$
CDH:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon N)$	$N^{1/4}$
DDH:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon^2 N)$	$\leq N^{1/4}$ $\geq N^{1/8}$

For $\epsilon = N^{-1/4}$ $S = N^{1/4}$

Better attack?

What about Decision Diffie-Hellman (DDH)?

DDH problem: Distinguish (g, g^x, g^y, g^{xy}) from (g, g^x, g^y, g^z)

	Upper bound	Lower bound	Time T
Discrete log:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon N)$	$N^{1/4}$
CDH:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon N)$	$N^{1/4}$
DDH:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon^2 N)$	$\leq N^{1/4}$ $\geq N^{1/8}$

For $\epsilon = N^{-1/4}$ $S = N^{1/4}$

What about Decision Diffie-Hellman (DDH)?

DDH problem: Distinguish (g, g^x, g^y, g^{xy}) from (g, g^x, g^y, g^z)

	Upper bound	Lower bound	Time T
Discrete log:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon N)$	$N^{1/4}$
CDH:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon N)$	$N^{1/4}$
DDH:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon^2 N)$	$\leq N^{1/4}$
			$\geq N^{1/8}$
sqDDH:	$ST^2 = \tilde{O}(\epsilon^2 N)$	$ST^2 = \tilde{\Omega}(\epsilon^2 N)$	$N^{1/8}$

For $\epsilon = N^{-1/4}$, $S = N^{1/4}$

What about Decision Diffie–Hellman (DDH)?

DDH problem: Distinguish (g, g^x, g^y, g^{xy}) from (g, g^x, g^y, g^z)

	Upper bound	Lower bound	Time T
Discrete log:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon N)$	$N^{1/4}$
CDH:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon N)$	$N^{1/4}$
DDH:	$ST^2 = \tilde{O}(\epsilon N)$	$ST^2 = \tilde{\Omega}(\epsilon^2 N)$	$\leq N^{1/4}$
			$\geq N^{1/8}$
sqDDH:	$ST^2 = \tilde{O}(\epsilon^2 N)$		$N^{1/8}$

For $\epsilon = N^{-1/4}$, $S = N^{1/4}$

Our new results

Definition. The sqDDH problem is to distinguish

$$(g, g^x, g^{(x^2)}) \quad \text{from} \quad (g, g^x, g^y) \qquad \text{for } x, y \leftarrow_R \mathbb{Z}_N.$$

Why it's interesting:

- For generic online-only algs, it's as hard as discrete log
- For generic preprocesssing algs, we show that it's "much easier"

→ A DDH-like problem that is easier than dlog

Definition. The sqDDH problem is to distinguish

$$(g, g^x, g^{(x^2)}) \quad \text{from} \quad (g, g^x, g^y)$$

for $x, y \leftarrow_R \mathbb{Z}_N$.

Why it's interesting:

- For generic online-only algs, it's as hard as discrete log
- For generic preprocesssing algs, we show that it's "much easier"

→ A DDH-like problem that is easier than dlog

Definition. The sqDDH problem is to distinguish

$$(g, g^x, g^{(x^2)}) \quad \text{from} \quad (g, g^x, g^y) \qquad \text{for } x, y \leftarrow_R \mathbb{Z}_N.$$

Why it's interesting:

- For generic online-only algs, it's as hard as discrete log
- For generic preprocesssing algs, we show that it's "much easier"

→ A DDH-like problem that is easier than dlog

Definition. The sqDDH problem is to distinguish

$$(g, g^x, g^{(x^2)}) \quad \text{from} \quad (g, g^x, g^y) \qquad \text{for } x, y \leftarrow_R \mathbb{Z}_N.$$

Why it's interesting:

- For generic online-only algs, it's as hard as discrete log
- For generic preprocesssing algs, we show that it's "much easier"

→ A DDH-like problem that is easier than dlog

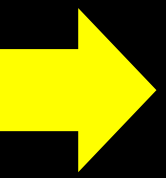
This talk

Background: Preprocessing attacks are relevant

- Preexisting $S = T = \tilde{O}(N^{1/3})$ generic attack on discrete log

Our results: Preprocessing lower-bounds and attacks

- The $\tilde{O}(N^{1/3})$ generic dlog attack is optimal
- Any such attack must use lots of preprocessing: $\Omega(N^{2/3})$
- New $\tilde{O}(N^{1/5})$ preprocessing attack on DDH-like problem



Open questions

Open questions and recent progress

- Tightness of DDH upper/lower bounds?
  - Is it as hard as dlog or as easy as sqDDH?
- Non-generic preprocessing attacks on ECDL?
  - As we have for $\mathbb{Z}_p^*$

Coretti, Dodis, and Guo (2018)

- Elegant proofs of generic-group lower bounds using "presampling" (à la Unruh, 2007)
- Prove hardness of "one-more" dlog, KEA assumptions, …

This talk

Background: Preprocessing attacks are relevant

- Preexisting $S = T = \tilde{O}(N^{1/3})$ generic attack on discrete log

Our results: Preprocessing lower-bounds and attacks

- The $\tilde{O}(N^{1/3})$ generic dlog attack is optimal
- Any such attack must use lots of preprocessing: $\Omega(N^{2/3})$
- New $\tilde{O}(N^{1/5})$ preprocessing attack on DDH-like problem

Open questions

This talk

Background: Preprocessing attacks are relevant

- Preexisting $S = T = \tilde{O}(N^{1/3})$ generic attack on discrete log

Our results: Preprocessing lower-bounds and attacks

- The $\tilde{O}(N^{1/3})$ generic dlog attack is optimal
- Any such attack must use lots of preprocessing: $\Omega(N^{2/3})$
- New $\tilde{O}(N^{1/5})$ preprocessing attack on DDH-like problem

Open questions

Henry – henrycg@cs.stanford.edu
Dima – dkogan@cs.stanford.edu
https://eprint.iacr.org/2017/1113