Batching Base Oblivious Transfers

Ian McQuoid, Mike Rosulek, Lance Roy

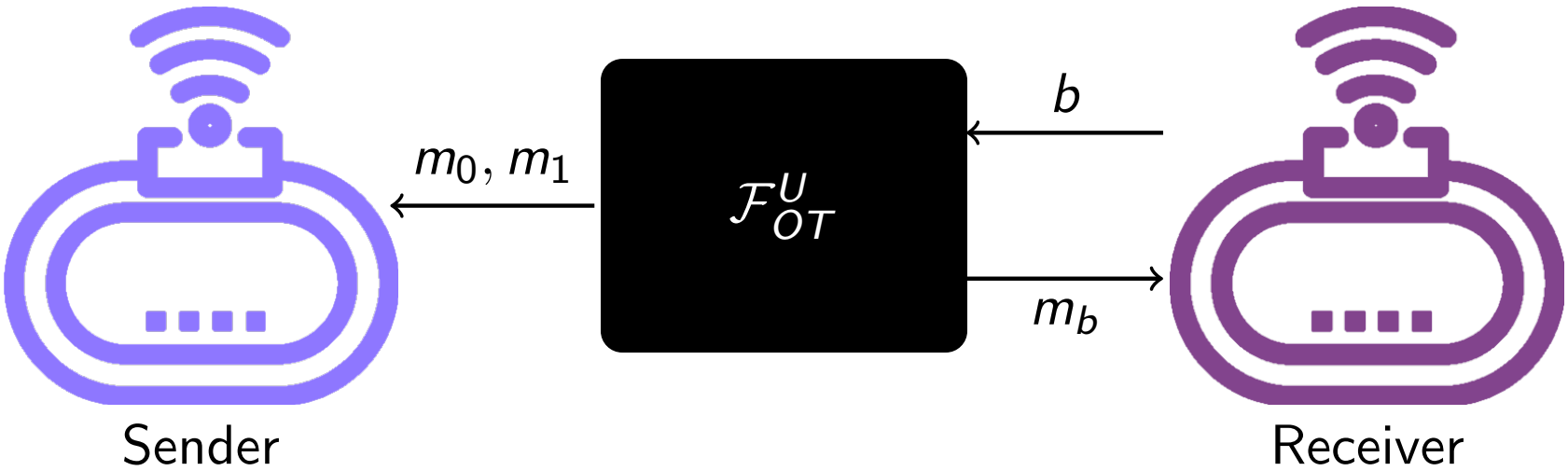
Oregon State University

{*mcquoidi,rosulekm,royl*}*@oregonstate.edu*

December 5, 2021

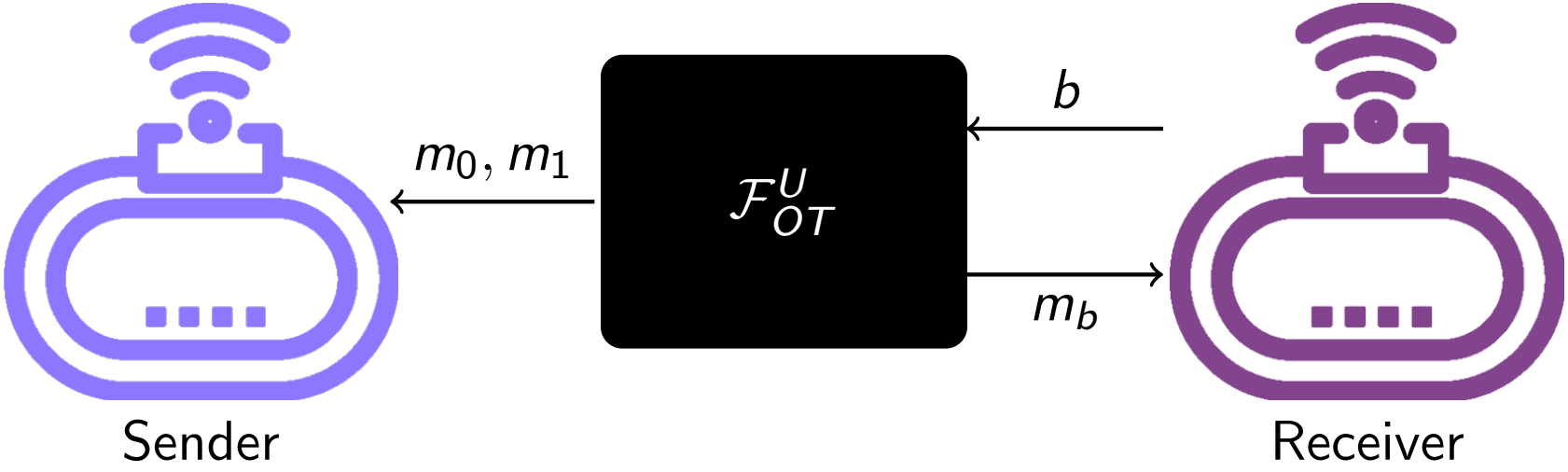
Oblivious Transfer (OT)

Uniform Oblivious Transfer



Oblivious Transfer (OT)

Uniform Oblivious Transfer



$b \approx b' \leftarrow_{\$} \{0,1\}$
$m_0, m_1 \leftarrow \$$

No information about m_{1-b}

OT Batching and Extension

Where is OT used?

1. Garbled Circuits [GoldreichMicaliWigderson87]
2. Private Set Intersection e.g. [PinkasSchneiderZohner14]

OT Batching and Extension

Where is OT used?

1. Garbled Circuits [GoldreichMicaliWigderson87]
2. Private Set Intersection e.g. [PinkasSchneiderZohner14]

Can require millions of OTs; necessarily requiring expensive assymetric operations!

Notation

Key Agreement (KA) notation:

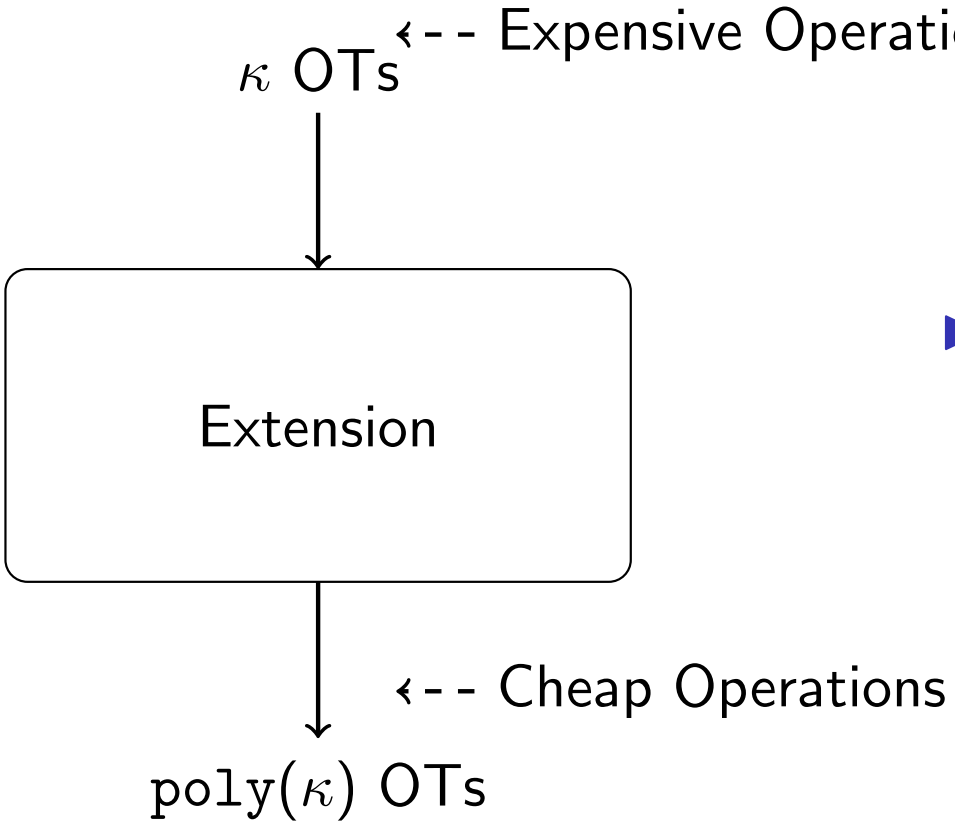
- $A := \mathsf{KA}.\mathsf{msg}_1(a)$
- $B := \mathsf{KA}.\mathsf{msg}_2(A, b)$
- $\mathsf{KA}.\mathsf{key}_i(\{a, b\}, \{A, B\})$

Elliptic Curve Diffie-Hellman KA:

- $A := a \cdot g$
- $B := b \cdot g$
- $a \cdot B, b \cdot A$

OT Batching and Extension

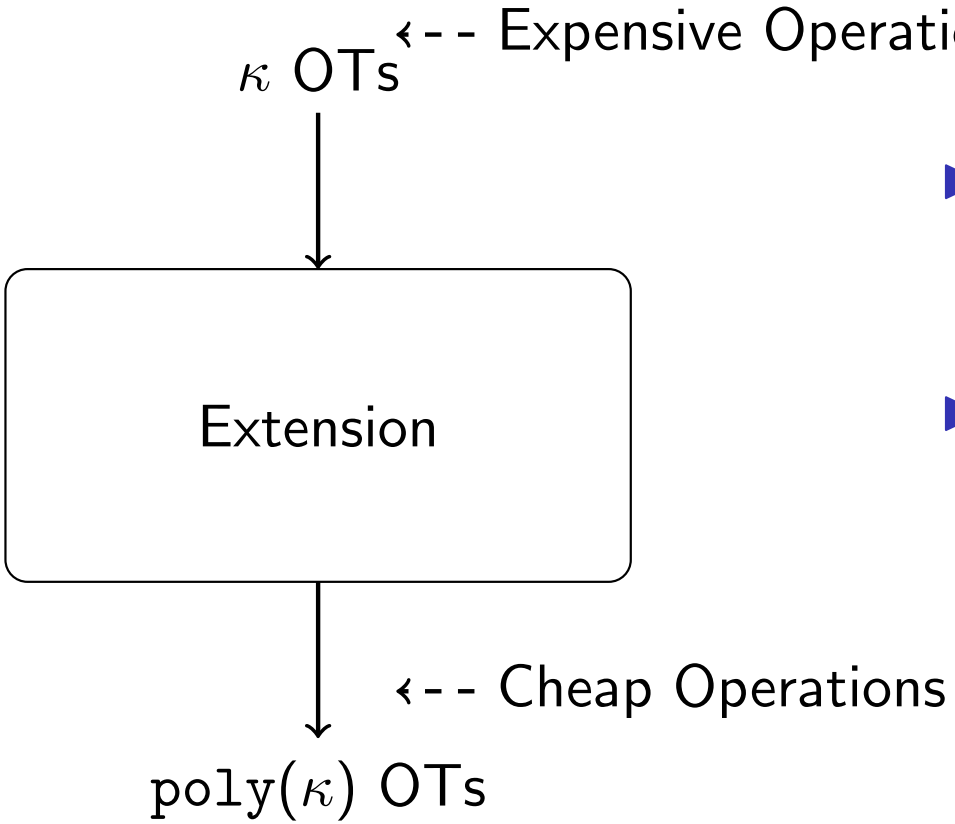
OT Extension [Beaver96]



- Transform a small number of *base* OTs into a large number of *realized* OTs.

OT Batching and Extension

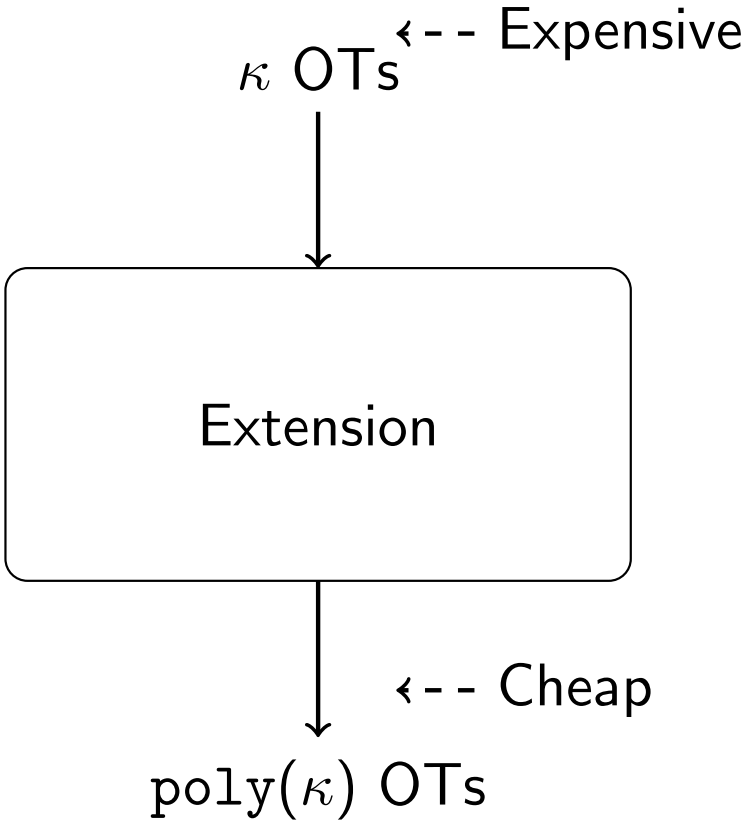
OT Extension [Beaver96]



- Transform a small number of *base* OTs into a large number of *realized* OTs.
- Optimize the base OTs.
  - We can send all of them in a *batch* — This is the batch setting.

OT Batching and Extension

OT Extension [Beaver96]



- Transform a small number of *base* OTs into a large number of *realized* OTs.
- Optimize the base OTs.
  - We can send all of them in a *batch* — This is the batch setting.
- The natural way to optimize batching lacked a principled treatment.

Results

- Provide a treatment of optimizing OTs in the batch setting.
- Expand the known OT constructions from [McQuoidRosulekRoy20].
- Optimize the resulting OT construction to the batch setting.

Roadmap

MRR20 Recap

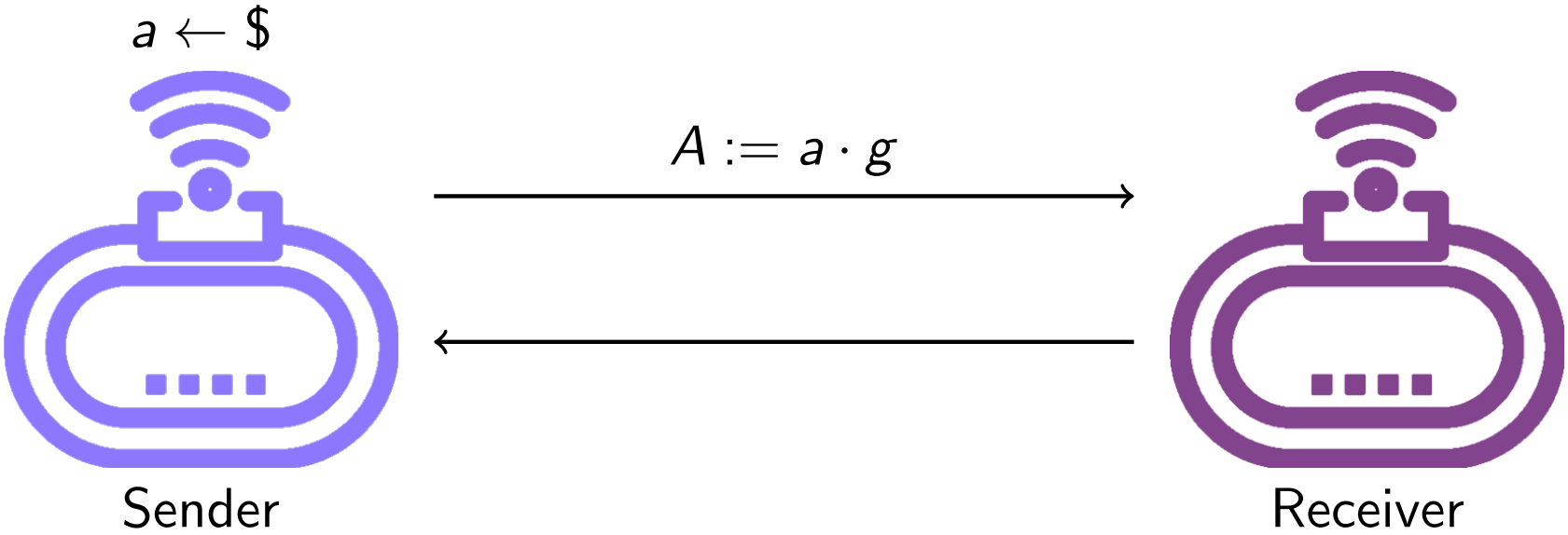
1. The [MRR20] OT protocol
2. Programmable-Once Public Functions

The Problem with Batching

1. What's the issue?
2. What's the fix?

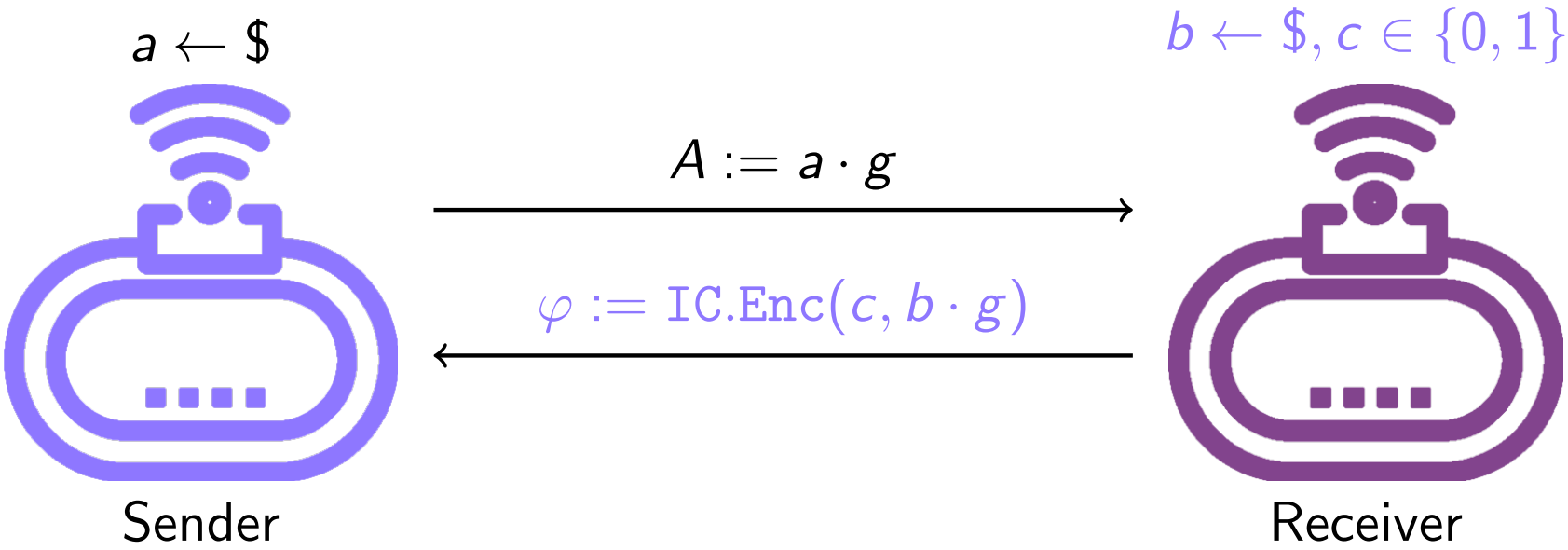
A Simple OT Protocol

As motivation [MRR20]:



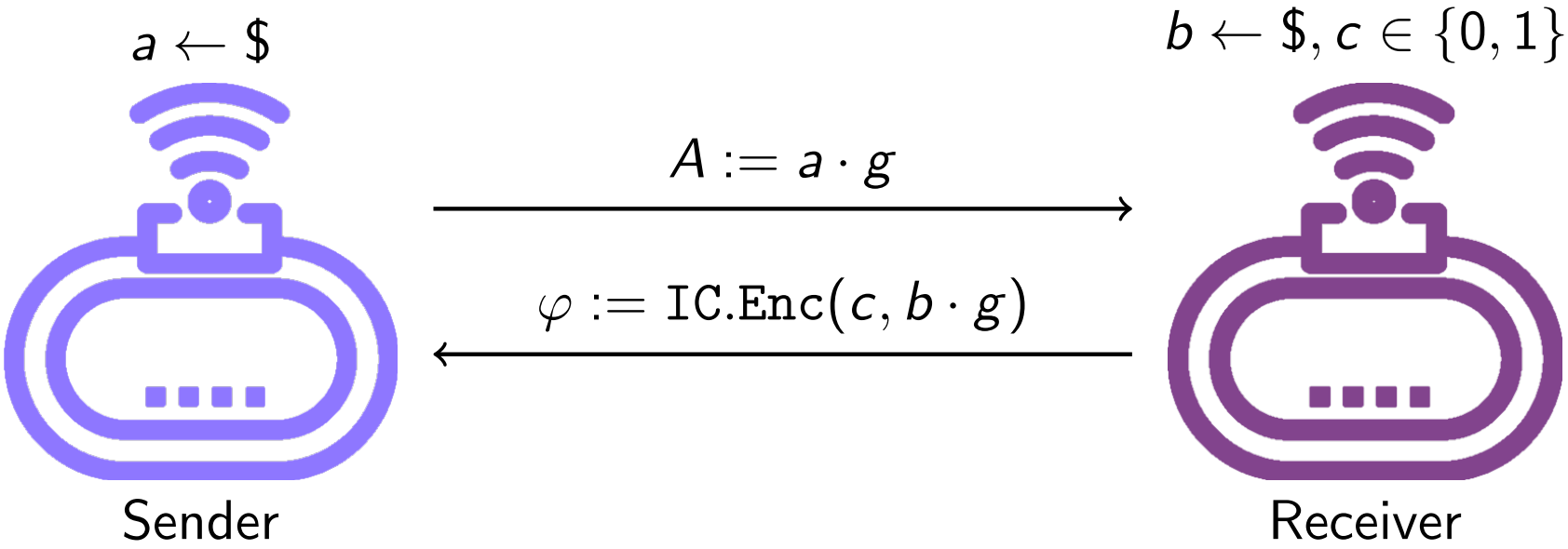
A Simple OT Protocol

As motivation [MRR20]:



A Simple OT Protocol

As motivation [MRR20]:

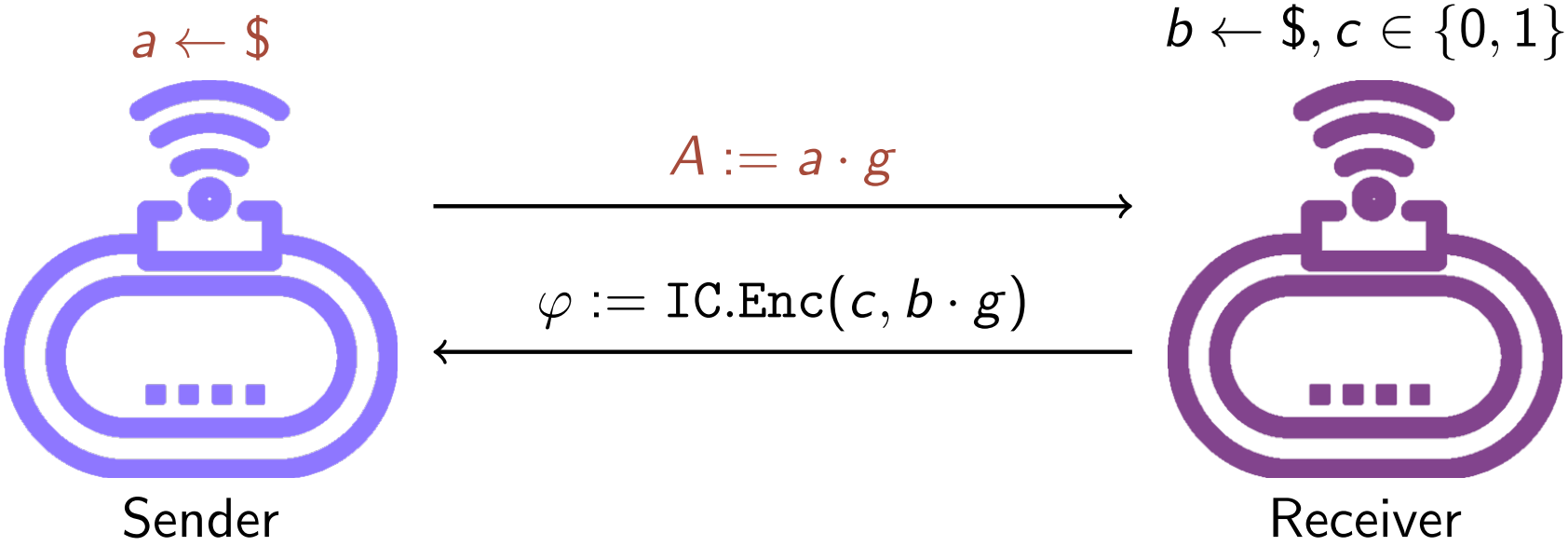


Output
$a\mathtt{IC.Dec}(0, \varphi)$
$a\mathtt{IC.Dec}(1, \varphi)$

Output
$b \cdot A = a \cdot \mathtt{IC.Dec}(c, \varphi)$

A Simple OT Protocol

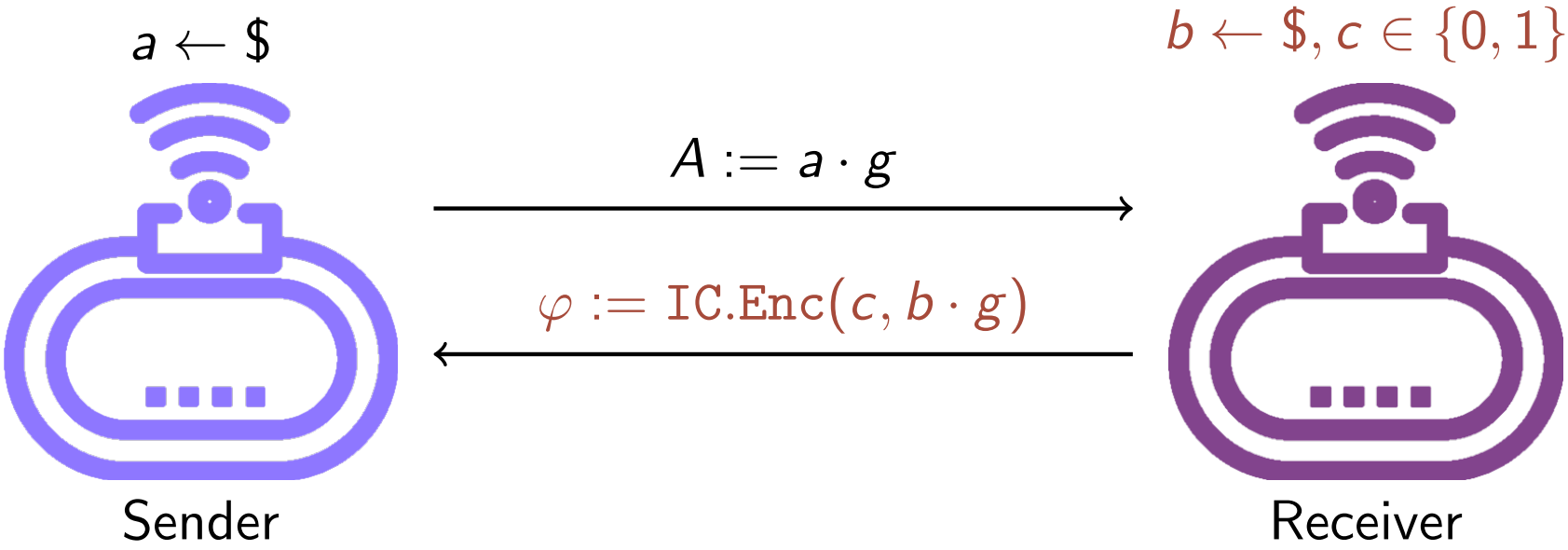
What's going on?



- The Sender sends a KA message.

A Simple OT Protocol

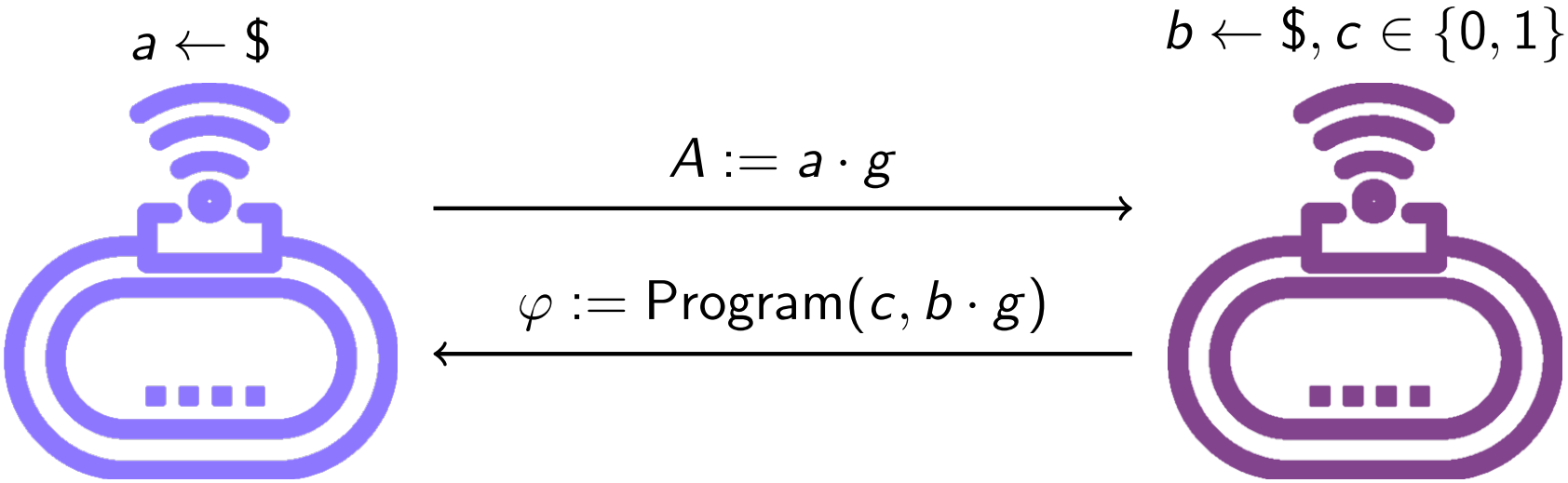
What's going on?



- The Sender sends a KA message.
- The Receiver sends back a wrapped KA message dependent on their choice bit.

A Simple OT Protocol

[MRR20]



Output
$a \cdot \text{Eval}(\varphi, 0)$
$a \cdot \text{Eval}(\varphi, 1)$

Output
$b \cdot A = a \cdot \text{Eval}(\varphi, c)$

Do we need the strong properties of an Ideal Cipher? Can we use something weaker?

A Simple OT Protocol

Our Ideal Cipher

- $\varphi := \mathtt{IC.Enc}(c, b \cdot g)$
- Output: $a \cdot \mathtt{IC.Dec}(c, \varphi)$

What Weak Cipher?

-
-

A Simple Endemic OT Protocol

Our Ideal Cipher

- $\varphi := \texttt{IC.Enc}(c, b \cdot g)$
- Output: $a\texttt{IC.Dec}(c, \varphi)$

What Weak Cipher?

- $\varphi := \text{Program}(c, b \cdot g)$
-

A Simple Endemic OT Protocol

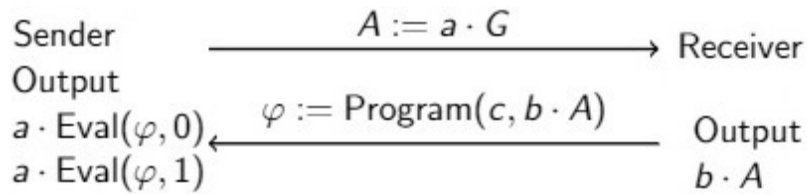
Our Ideal Cipher

- $\varphi := \mathtt{IC.Enc}(c, b \cdot g)$
- Output: $a\mathtt{IC.Dec}(c, \varphi)$

What Weak Cipher?

- $\varphi := \mathsf{Program}(c, b \cdot g)$
- Output: $a \cdot \mathsf{Eval}(\varphi, 1)$

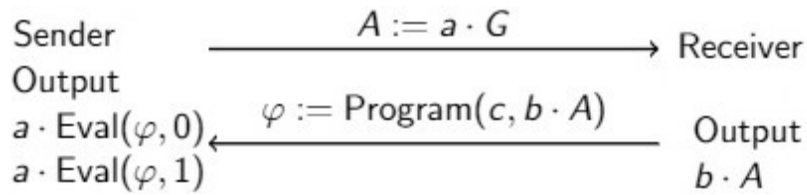
Programmable-Once Public Functions (POPFs)



For our proof, we need to: [MRR20]

1. Hide the receiver's choice bit.
2. Hide the non-chosen messages from the receiver.
3. Extract the adversary's choice bit.
4. Have a backdoor so we can program on BOTH choice bits.

Programmable-Once Public Functions (POPFs)



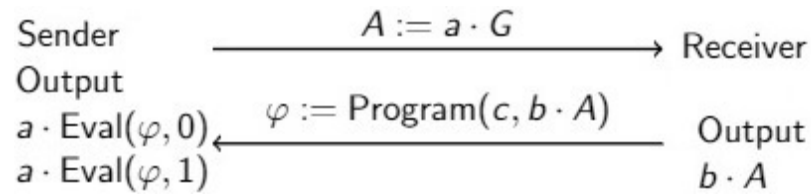
For simulatability, we need to: [MRR20]

1. Hide the receiver's choice bit.
2. Hide the non-chosen messages from the receiver.
3. Extract the adversary's choice bit.
4. Have a backdoor so we can program on BOTH choice bits.

$\Downarrow$

1. $\mathrm{Eval}(\mathrm{Program}(c, \$))$ looks like a uniform function.

Programmable-Once Public Functions (POPFs)



For simulatability, we need to: [MRR20]

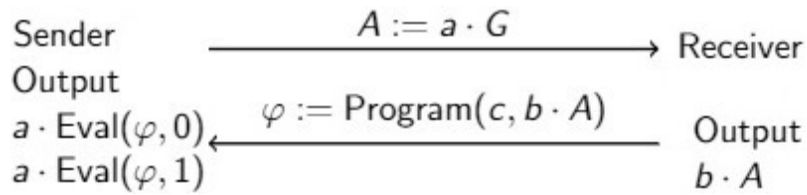
1. Hide the receiver's choice bit.
2. Hide the non-chosen messages from the receiver.
3. Extract the adversary's choice bit.
4. Have a backdoor so we can program on BOTH choice bits.

$\Downarrow$

1. $\mathsf{Eval}(\mathsf{Program}(c, \$), \cdot)$ looks like a uniform function.
2. Given $\varphi \leftarrow \mathsf{Program}(c, \star)$, $\mathsf{Eval}(\varphi, 1 - c)$ is uniform after passing through a weak random function.

Programmable-Once Public Functions (POPFs)

For simulatability, we need to: [MRR20]



1. Hide the receiver's choice bit.
2. Hide the non-chosen messages from the receiver.
3. Extract the adversary's choice bit.
4. Have a backdoor so we can program on BOTH choice bits.

$$\Downarrow$$

1. $\mathrm{Eval}(\mathrm{Program}(c, \$))$ looks like a uniform function.
2. Given $\varphi \leftarrow \mathrm{Program}(c, \star)$, $\mathrm{Eval}(\varphi, 1 - c)$ is uniform after passing through a weak random function.
3. We need the usual simulation properties e.g. from a random oracle or ideal cipher.

Programmable-Once Public Functions (POPFs)

Why do we call it a Programmable-Once Public Function?

1. Programmable-Once

1. A party can **program** the output of the function for **exactly one** input.

Programmable-Once Public Functions (POPFs)

Why do we call it a Programmable-Once Public Function?

1. Programmable-Once
2. Public Function

1. A party can **program** the output of the function for **exactly one** input.
2. A party then sends a representation which **can be evaluated by anyone** as a function.

Key Agreement (KA) Restrictions

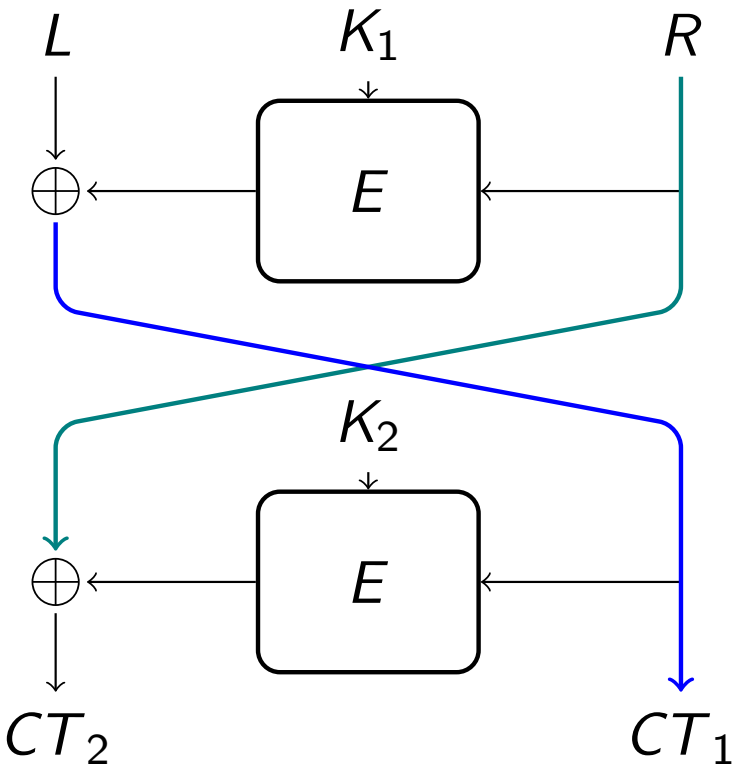
- Eval(Program(c, \$), $\cdot$) looks like a uniform function.

$$\Downarrow$$

- Key agreement messages we wrap should be uniform so we can hide the choice bit.
  - Even if subsequent messages are dependent on previous ones (certainly true for ECDHKA).

Feistel POPF

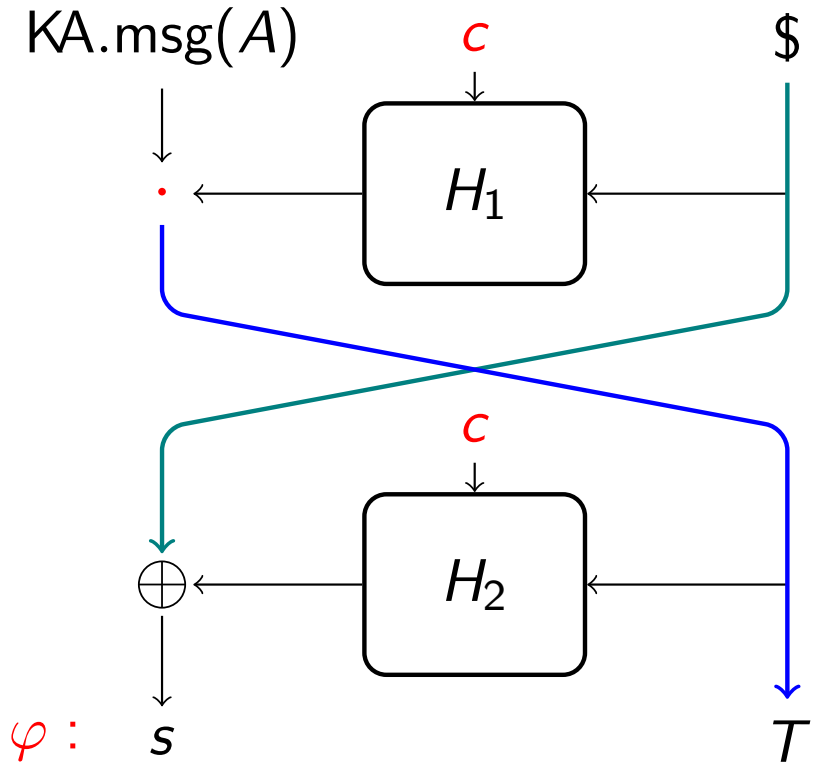
Constructing the POPF
[MRR20]



- Familiar Feistel cipher.
- Known to realize an ideal cipher at 8 rounds (with loss).

Feistel POPF

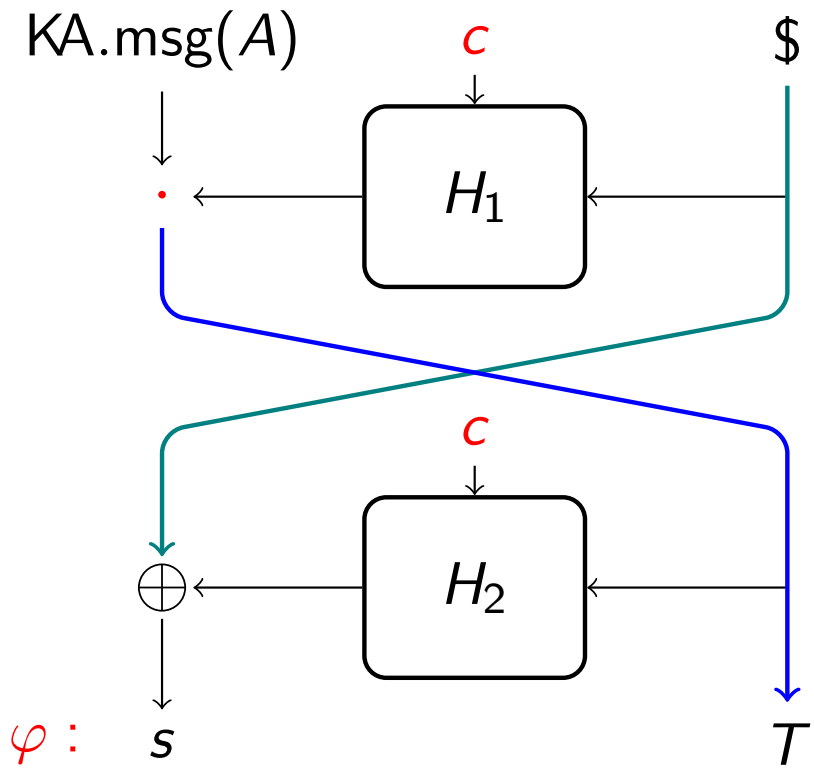
Constructing the POPF
[MRR20]



- The familiar Feistel cipher.
- Known to realize an ideal cipher at 8 rounds (with loss).
- Now in POPF form.
  - Replacing the first XOR with a group operation.

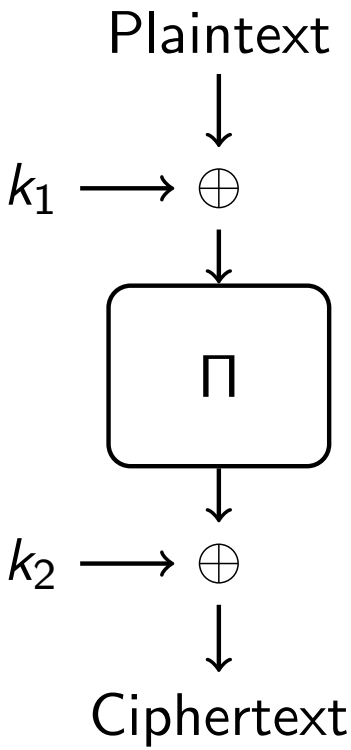
Feistel POPF

Constructing the POPF
[MRR20]



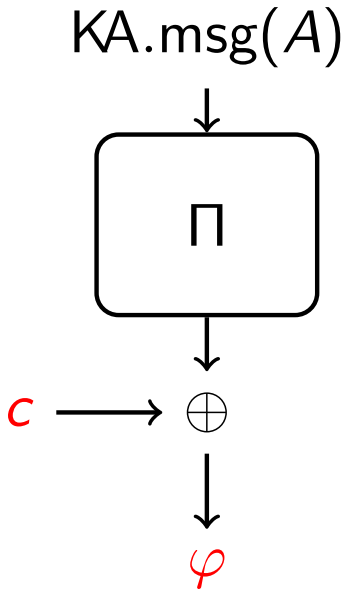
- The familiar Feistel cipher.
- Known to realize an ideal cipher at 8 rounds (with loss).
- Now in POPF form.
  - Replacing the first XOR with a group operation.
- We showed that this construction can be optimized for the small domain situation by replacing H_2 with an injection into a finite field.

Even–Mansour POPF



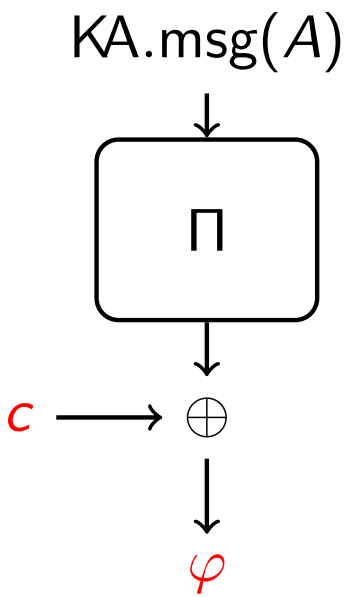
- The familiar Even-Mansour XOR cipher.
- Instantiated with a Ideal Permutation.

Even–Mansour POPF



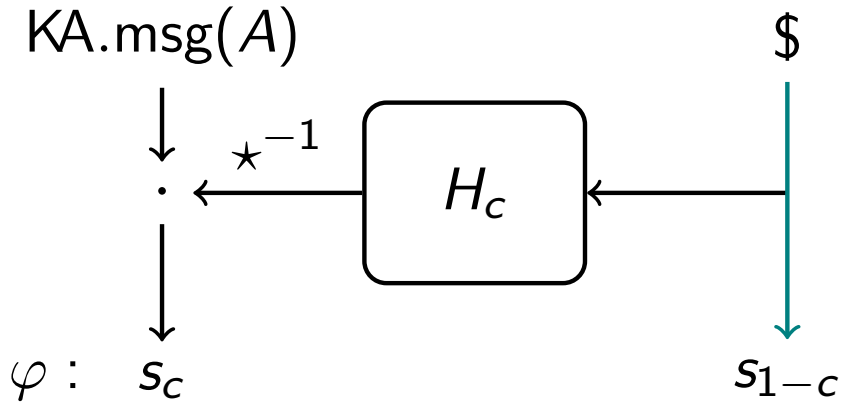
- The familiar Even-Mansour XOR cipher.
- Instantiated with a Ideal Permutation.
- Now in POPF form.
  - Dropping the first XOR.

Even–Mansour POPF



- The familiar Even-Mansour XOR cipher.
- Instantiated with a Ideal Permutation.
- Now in POPF form.
  - Dropping the first XOR.
- We showed this construction was a POPF.

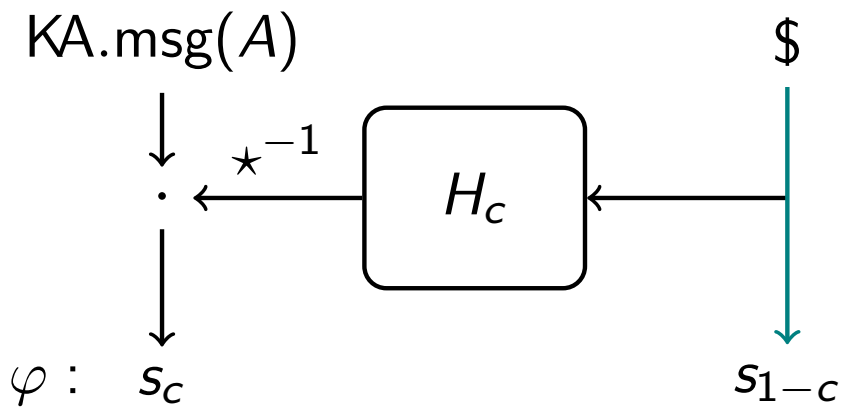
Masny-Rindal POPF



- Looks like a one round Feistel Cipher.

Masny-Rindal POPF

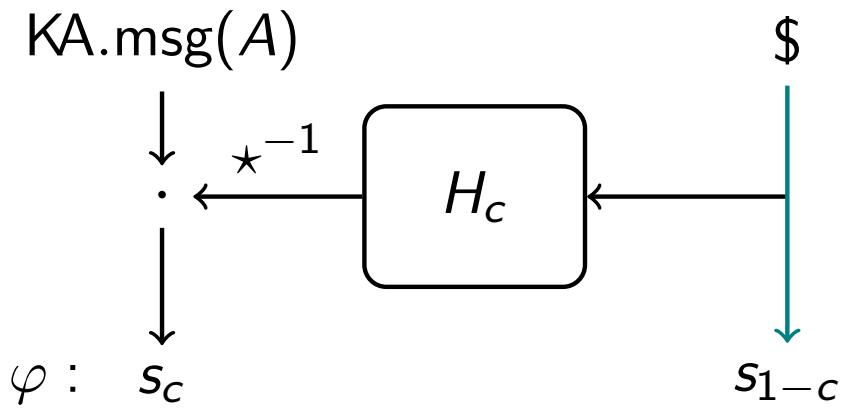
[MasnyRindal2019]



- Looks like a one round Feistel Cipher.
  - Moving to 1-of-N OT extends differently than the Feistel construction.
- Doesn't efficiently extend to exponential N.

Masny-Rindal POPF

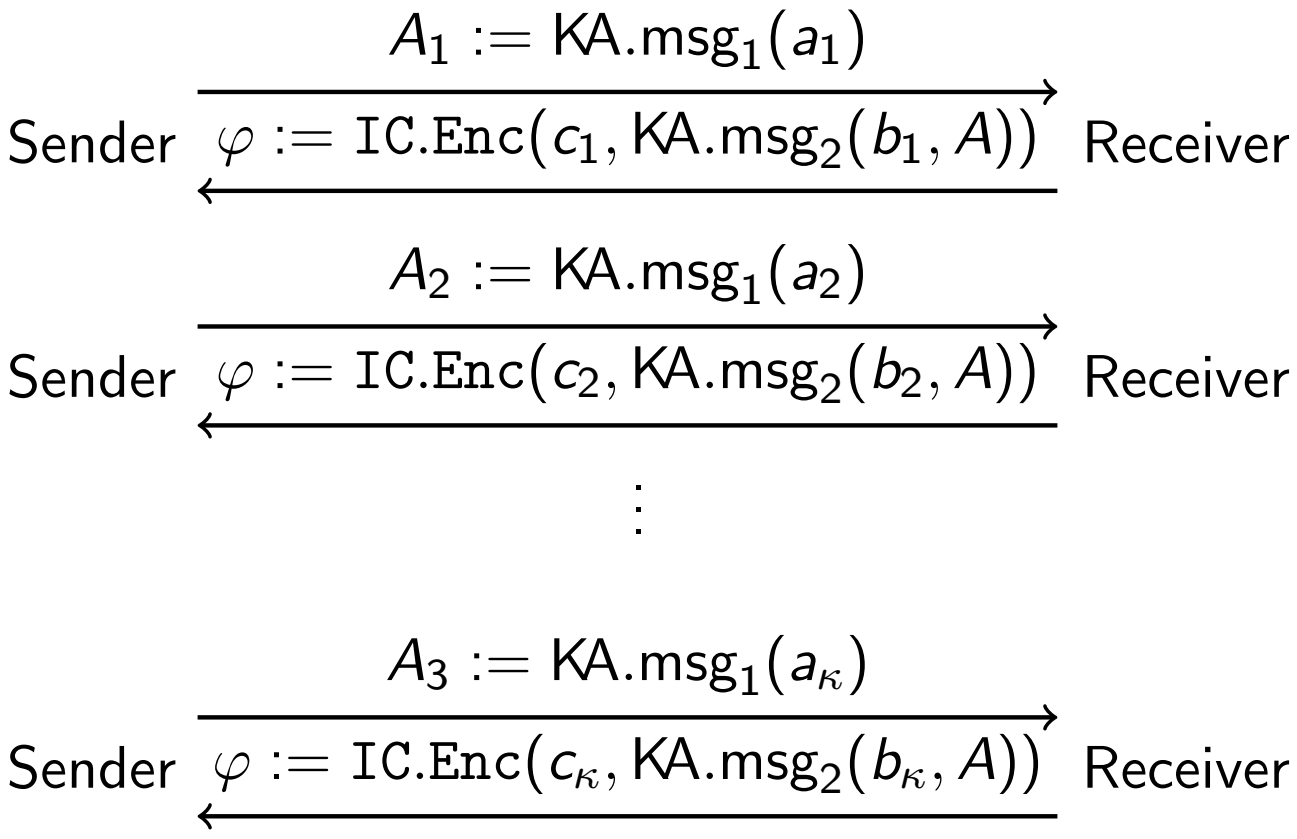
[MasnyRindal2019]



- Looks like a one round Feistel Cipher.
  - Moving to 1-of-N OT extends differently than the Feistel construction.
- Doesn't efficiently extend to exponential N.
- We showed that the Masny-Rindal protocol was a special case of the MRR20 protocol.

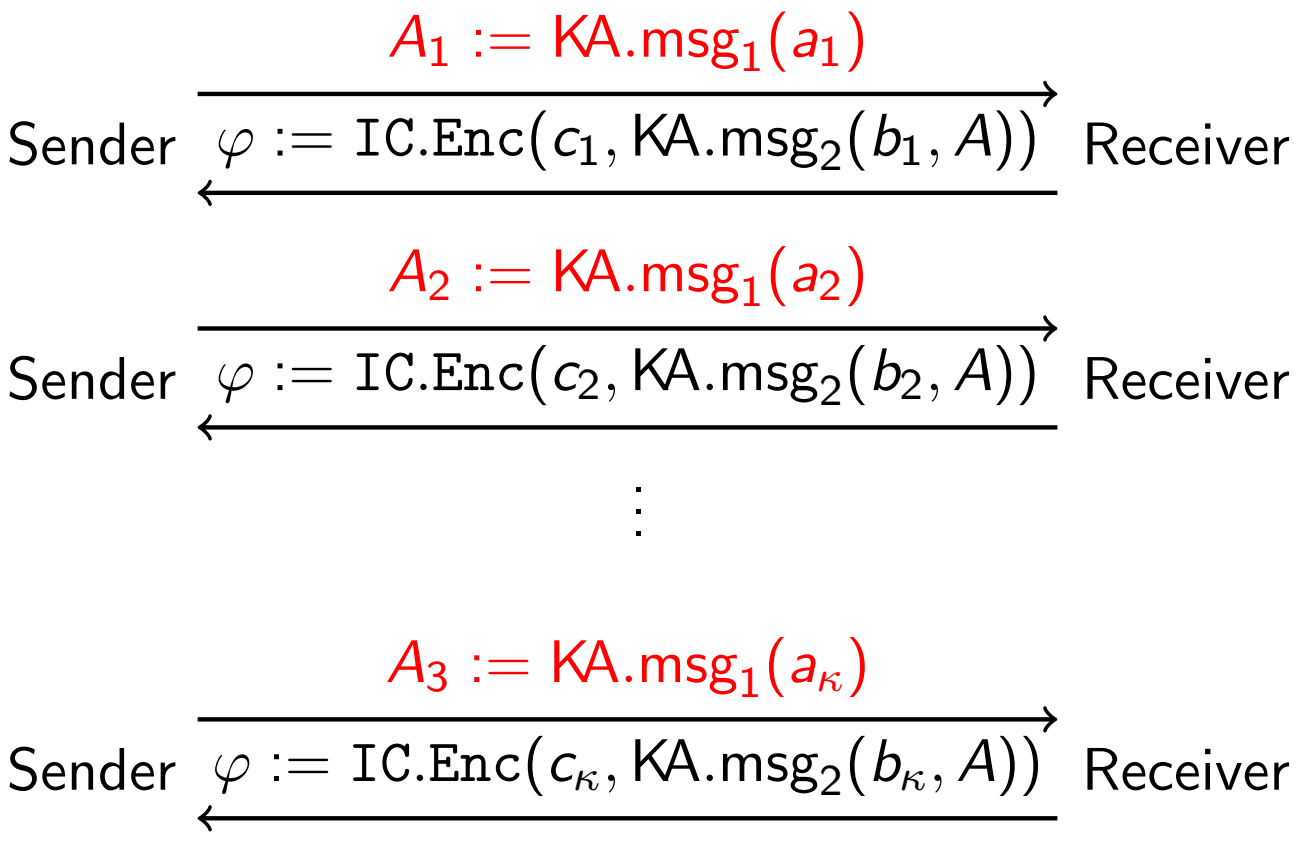
Naïve Batching

How would we naturally batch our running protocol?



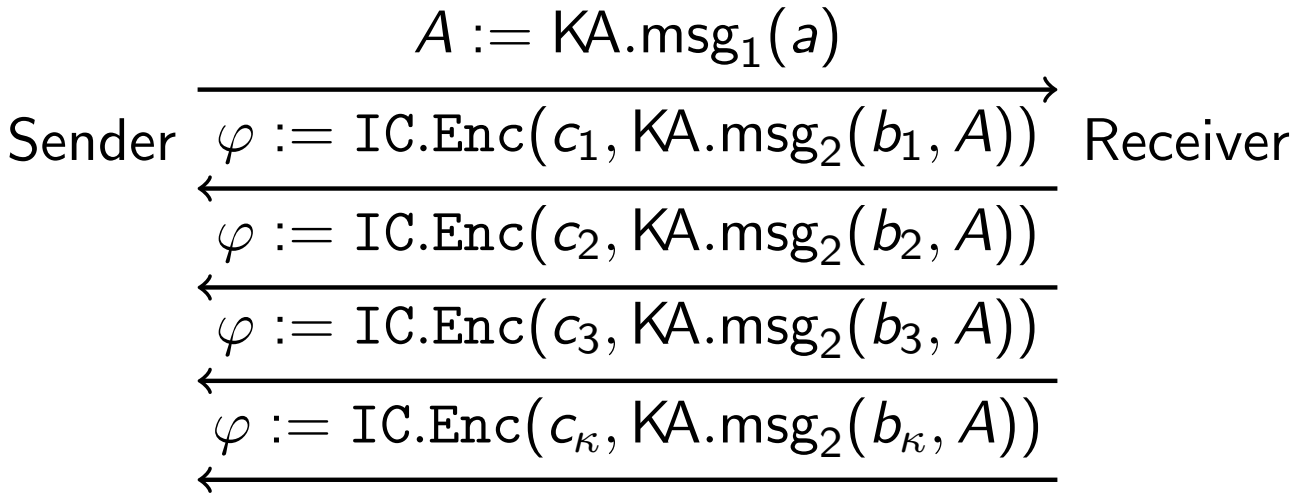
Naïve Batching

How would we naturally batch our running protocol?



Naïve Batching

How would we naturally batch our running protocol?



The Problem with Naïve Batching

1. Sender gives $A := \mathsf{KA.msg}_1(a)$ to Receiver.

The Problem with Naïve Batching

1. Sender gives $A := \mathsf{KA.msg}_1(a)$ to Receiver.
2. Receiver generates $B_1 := \mathtt{IC.Enc}(\mathsf{KA.msg}_2(b, A), 0)$, $B_2 := \mathtt{IC.Enc}(\mathsf{KA.msg}_2(b, A), 1)$.

The Problem with Naïve Batching

1. Sender gives $A := \mathsf{KA.msg}_1(a)$ to Receiver.
2. Receiver generates $B_1 := \mathtt{IC.Enc}(\mathsf{KA.msg}_2(b, A), 0)$, $B_2 := \mathtt{IC.Enc}(\mathsf{KA.msg}_2(b, A), 1)$.
3. Receiver gives $B_1, B_1, B_1, B_1, \ldots$ to Sender.

The Problem with Naïve Batching

1. Sender gives $A := \mathsf{KA.msg}_1(a)$ to Receiver.
2. Receiver generates $B_1 := \mathtt{IC.Enc}(\mathsf{KA.msg}_2(b, A), 0)$, $B_2 := \mathtt{IC.Enc}(\mathsf{KA.msg}_2(b, A), 1)$.
3. Receiver gives $B_1, B_1, B_1, B_1, \ldots$ to Sender.
4. Receiver causes all strings across the batch to be equal!
   - Or could induce complex correlations.

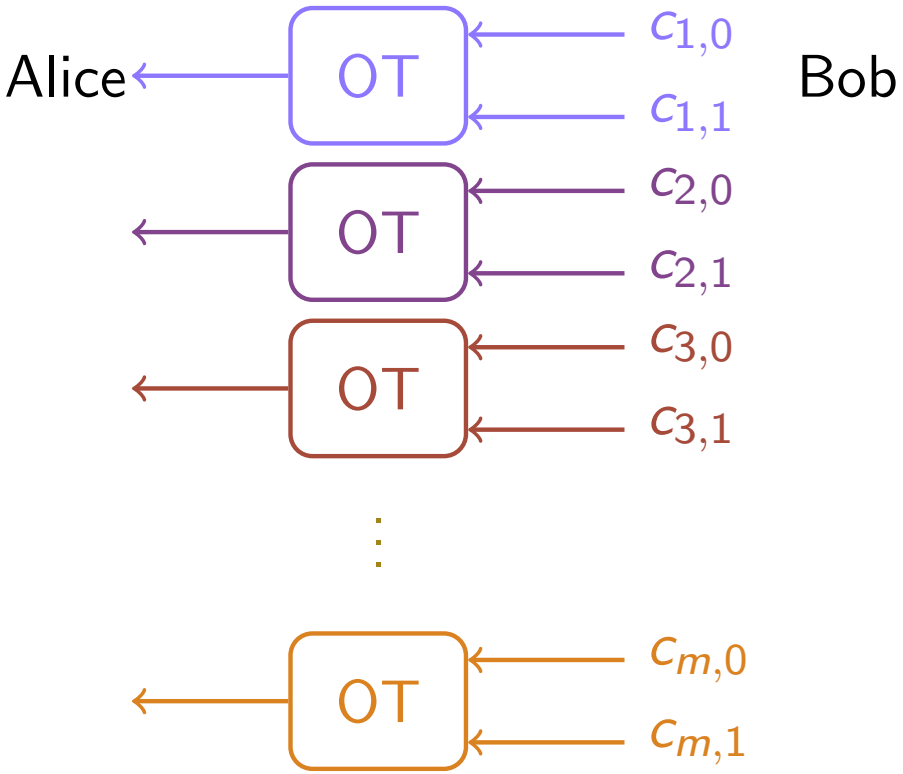
The Problem with Naïve Batching

1. Sender gives $A := \mathsf{KA.msg}_1(a)$ to Receiver.
2. Receiver generates $B_1 := \mathtt{IC.Enc}(\mathsf{KA.msg}_2(b, A), 0)$, $B_2 := \mathtt{IC.Enc}(\mathsf{KA.msg}_2(b, A), 1)$.
3. Receiver gives $B_1, B_1, B_1, B_1, \ldots$ to Sender.
4. Receiver causes all strings across the batch to be equal!
   - Or could induce complex correlations.

Affects OT Extension protocols in a devastating way. . .

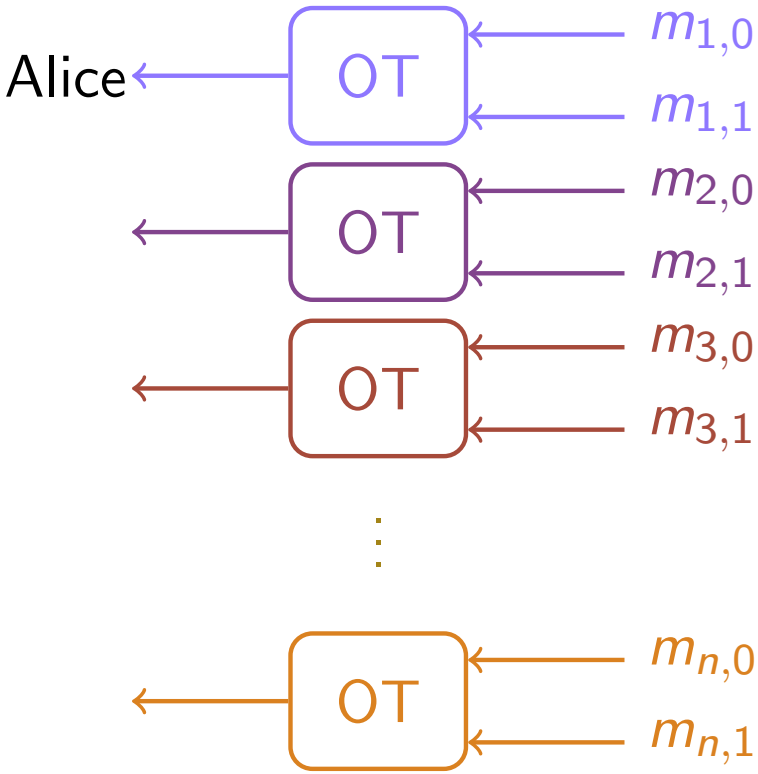
Naïve Batching in OT Extension

OOS OT Extension [OrrùOrsiniScholl17]



Naïve Batching in OT Extension

OOS OT Extension [OrrùOrsiniScholl17]



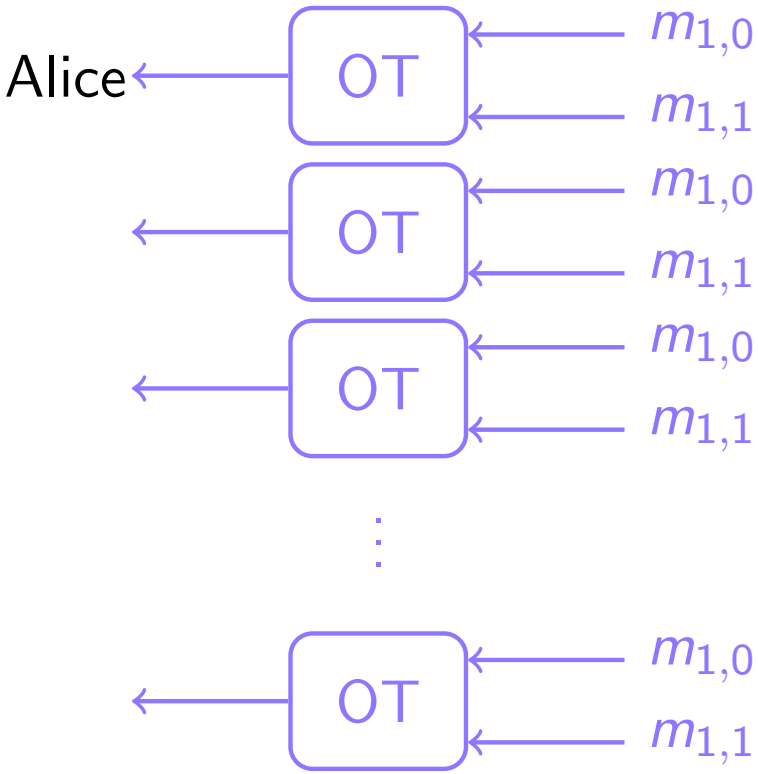
Bob

$K_{\star,i} = \mathrm{PRG}(m_{i,0}) \oplus \mathrm{PRG}(m_{i,1})$

$k_{1,1}$	$k_{1,2}$	$k_{1,3}$		$k_{1,m}$
$k_{2,1}$	$k_{2,2}$	$k_{2,3}$		$k_{2,m}$
$k_{3,1}$	$k_{3,2}$	$k_{3,3}$	$\cdots$	$k_{3,m}$
$\vdots$	$\vdots$	$\vdots$		$\vdots$
$k_{N,1}$	$k_{N,2}$	$k_{N,3}$		$k_{N,m}$

Naïve Batching in OT Extension

OOS OT Extension [OrrùOrsiniScholl17]



Bob $K_{\star,i} = \mathrm{PRG}(m_{i,0}) \oplus \mathrm{PRG}(m_{i,1})$

$k_{1,1}$	$k_{1,2}$	$k_{1,3}$		$k_{1,m}$
$k_{2,1}$	$k_{2,2}$	$k_{2,3}$		$k_{2,m}$
$k_{3,1}$	$k_{3,2}$	$k_{3,3}$	$\cdots$	$k_{3,m}$
$\vdots$	$\vdots$	$\vdots$		$\vdots$
$k_{N,1}$	$k_{N,2}$	$k_{N,3}$		$k_{N,m}$

Naïve Batching in OT Extension

OOS OT Extension [OrrùOrsiniScholl17]

$K_{i,\star} \in \{0^m, 1^m\}$

$$\begin{array}{|ccccc|}
\hline
0 & 0 & 0 & \cdots & 0 \\
1 & 1 & 1 & \cdots & 1 \\
1 & 1 & 1 & \cdots & 1 \\
 & & \vdots & & \\
0 & 0 & 0 & \cdots & 0 \\
\hline
\end{array}$$

$R_{j,\star} = C(c_j)$

$$\begin{array}{|c|}
\hline
C(c_1) \\
C(c_2) \\
C(c_3) \\
\vdots \\
C(c_N) \\
\hline
\end{array}$$

$K \oplus R$ is sent to Alice

$$\begin{array}{|c|}
\hline
C(c_1) \\
C(c_2)^c \\
C(c_3)^c \\
\vdots \\
C(c_N) \\
\hline
\end{array}$$

Naïve Batching in OT Extension

OOS OT Extension [OrrùOrsiniScholl17]

If $C(0) \neq C(1)^c$, we can determine each c_i

$$\begin{bmatrix} C(c_1) \\ C(c_2)^c \\ C(c_3)^c \\ \vdots \\ C(c_N) \end{bmatrix}$$

- Can extract all the receiver's choice bits.
- Relies on the two codewords not being complements (KOS).
- Could there be more complex correlations?

Reusing the First Message in POPF-OT

- How do we solve this problem?
  - Disallow for correlations by separating each OT instance!

Reusing the First Message in POPF-OT

- How do we solve this problem?
  - Disallow for correlations by separating each OT instance!
- Implement domain separation at the Key Agreement level with Tagged KA.

$$\mathsf{KA.key}_{\{1,2\}}(\cdot,\cdot) \qquad \mathsf{KA.key}_{\{1,2\}}(\cdot,\cdot,\tau)$$

Argument of Security

1. $a, b \leftarrow \$$
2. $A := \mathsf{KA.msg}_1(a), B := \mathsf{KA.msg}_2(b, A)$
3. $K := \mathsf{KA.key}_1(a, B, \tau) = \mathsf{KA.key}_2(b, A, \tau)$

1. $a, b \leftarrow \$$
2. $A := a \cdot g, B := b \cdot G$
3. $K := H(a \cdot B, \tau) = H(b \cdot A, \tau)$

Argument of Security

1. $a, b \leftarrow \$$
2. $A := \mathsf{KA.msg}_1(a)$,
   $B := \mathsf{KA.msg}_2(b, A)$
3. $K := \mathsf{KA.key}_1(a, B, \tau)$
   $B = \mathsf{KA.key}_2(b, A, \tau)$

1. $a, b \leftarrow \$$
2. $A := a \cdot g, B := b \cdot G$
3. $K := H(a \cdot B, \tau) = H(b \cdot A, \tau)$

- Let τ be the OT index in a batch.

Argument of Security

1. $a, b \leftarrow \$$
2. $A := \mathsf{KA.msg}_1(a), B := \mathsf{KA.msg}_2(b, A)$
3. $K := \mathsf{KA.key}_1(a, B, \tau) = \mathsf{KA.key}_2(b, A, \tau)$

1. $a, b \leftarrow \$$
2. $A := a \cdot g, B := b \cdot G$
3. $K := H(a \cdot B, \tau) = H(b \cdot A, \tau)$

- Let τ be the OT index in a batch.
- The simulator can program each output separately to maintain separation.

KA In POPF-OT

- Now that we know how to batch properly, how can we further optimize the process?

KA In POPF-OT

- Now that we know how to batch properly, how can we further optimize the process?
- MRR20 required a KA with uniform messages for both parties in a PAKE.
  - This was unsatisfied by stock Elliptic Curve Diffie-Hellman KA.

KA In POPF-OT

- Now that we know how to batch properly, how can we further optimize the process?
- MRR20 required POPF-compatible key agreements for both parties in a PAKE.
- This was unsatisfied by stock Elliptic Curve Diffie-Hellman KA.
  - Ideal Cipher Compatible — Uniform Bitstrings — Elligator.
  - Random Oracle Compatible — Hash to Curve.

KA In POPF-OT

- Now that we know how to batch properly, how can we further optimize the process?
- MRR20 required POPF-compatible key agreements for both parties in a PAKE.
- This was unsatisfied by stock Elliptic Curve Diffie-Hellman KA.
  - Ideal Cipher Compatible — Uniform Bitstrings — Elligator.
  - Random Oracle Compatible — Hash to Curve.
- OT requires the property for **the receiver only**. The sender's message is outside a POPF.

KA In POPF-OT

- Now that we know how to batch properly, how can we further optimize the process?
- MRR20 required POPF-compatible key agreements for both parties in a PAKE.
- This was unsatisfied by stock Elliptic Curve Diffie-Hellman KA.
  - Ideal Cipher Compatible — Uniform Bitstrings — Elligator.
  - Random Oracle Compatible — Hash to Curve.
- OT requires the property for **the receiver only**. The sender's message is outside a POPF.
  - Uniform Bitstrings for One Party — Möller's Trick [Möller04]

Möller's KA

[Möller04]

- Elliptic curve elements don't look like uniform bitstrings naturally.
  - Even the X-coordinates don't all lie on the curve.

Möller's KA

[Möller04]

- Elliptic curve elements don't look like uniform bitstrings naturally.
  - Even the X-coordinates don't all lie on the curve.
  - But where do all the other X-coordinates lie?

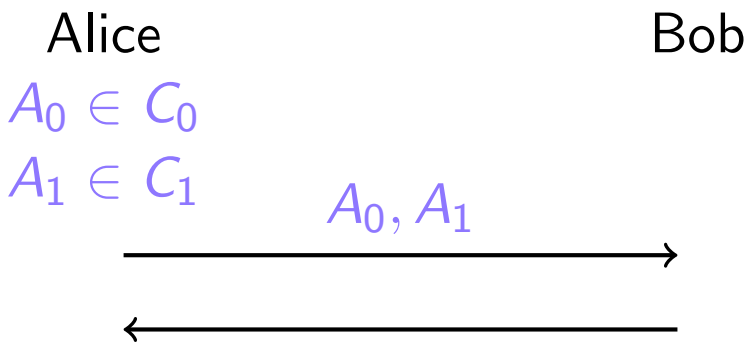
Möller's KA

[Möller04]

- Elliptic curve elements don't look like uniform bitstrings naturally.
  - Even the X-coordinates don't all lie on the curve.
  - But where do all the other X-coordinates lie?
- All the other X-coordinates lie on the curve's twist!
- If both curves are secure and about the same size, we can use uniform messages!

Möller's KA

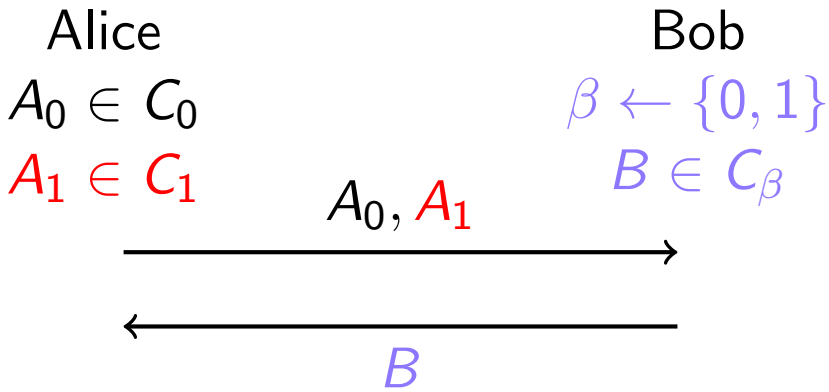
[Möller04]



1. Alice sends KA messages for both curves.

Möller's KA

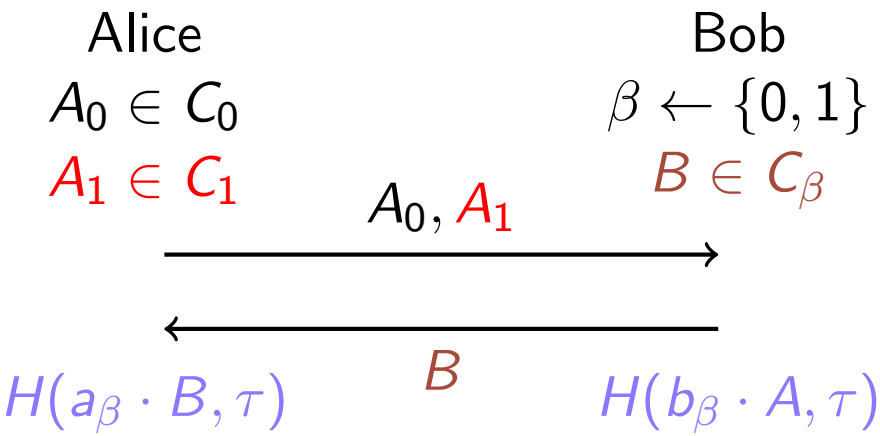
[Möller04]



1. Alice sends KA messages for both curves — This cost is amortized over each KA in a batch.
2. Bob samples a bit and sends a KA message for one of the two curves.

Möller's KA

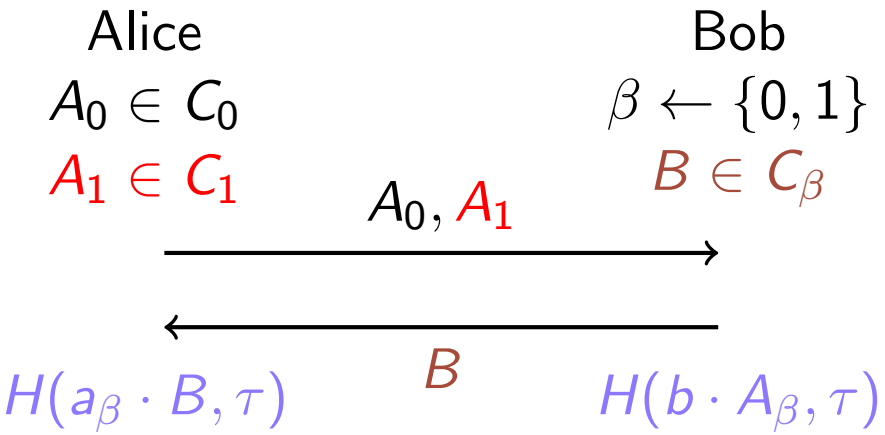
[Möller04]



1. Alice sends KA messages for both curves — This cost is amortized over each KA in a batch.
2. Bob samples a bit and sends a KA message for one of the two curves.
3. Alice/Bob then compute the corresponding shared key.

Möller's KA

[Möller04]



1. Alice sends KA messages for both curves — This cost is amortized over each KA in a batch.
2. Bob samples a bit and sends a KA message for one of the two curves.
3. Alice/Bob then compute the corresponding shared key.
4. B is now uniformly distributed in $\mathbb{F}_n$

Performance Evaluation

Batch of 128 OTs

Protocol	Sender (ms)	Receiver (ms)
0.1ms latency, 10000Mbps bandwidth cap (LAN)		
Simplest OT (Sender-reuse)	35	17
Naor-Pinkas OT (Sender-reuse)	43	34
Endemic OT (No reuse)	79	42
Endemic OT (Sender-reuse)	62	37
Ours (Feistel POPF)	82	40
Ours (Feistel POPF — Möller)	49	26
Ours (MR POPF — Möller)	48	27
Ours (EKE POPF — Möller)	50	25

Performance Evaluation

Batch of 128 OTs

Protocol	Sender (ms)	Receiver (ms)
30ms latency, 100Mbps bandwidth cap (WAN)		
Simplest OT (Sender-reuse)	105	111
Naor-Pinkas OT (Sender-reuse)	101	107
Endemic OT (No reuse)	161	53
Endemic OT (Sender-reuse)	137	53
Ours (Feistel POPF)	155	47
Ours (Feistel POPF — Möller)	128	44
Ours (MR POPF — Möller)	128	44
Ours (EKE POPF — Möller)	128	44

Performance Evaluation

- 18% WAN / 11% LAN performance increase in batching reusing the sender's KA message.

Performance Evaluation

- 18% WAN / 11% LAN performance increase in batching reusing the sender's KA message.
  - 126 fewer exponentiations and group elements sent from the sender.
  - All receiver exponentiations use the same base.

Performance Evaluation

- 18% WAN / 11% LAN performance increase in batching reusing the sender's KA message.
  - 126 fewer exponentiations and group elements sent from the sender.
  - All receiver exponentiations use the same base.
- 31% WAN / 12% LAN performance increase in batching moving to Möller KA.

Performance Evaluation

- 18% WAN / 11% LAN performance increase in batching reusing the sender's KA message.
  - 126 fewer exponentiations and group elements sent from the sender.
  - All receiver exponentiations use the same base.
- 31% WAN / 12% LAN performance increase in batching moving to Möller KA.
  - No expensive mapping operations.
  - Multiplication can be accomplished with Montgomery Ladders.

Open Problems

- Do any similar problems arise in other OT extensions?
- Are there any post-quantum KAs that meet our properties?
- What else can POPFs be used in?