

Cryptographic Protocol Security

through Effect Handlers and Probabilistic Relational Separation Logic

Simon Oddershede Gregersen

Randomness is an essential tool

Randomness is an essential tool

Algorithms and data structures

Simpler and faster algorithms: Bloom filters, skip lists, Miller-Rabin, ...

Randomness is an essential tool

Algorithms and data structures

Simpler and faster algorithms: Bloom filters, skip lists, Miller-Rabin, ...

Differential privacy

Adding noise: Laplace & Gaussian mechanisms, randomized response, ...

Randomness is an essential tool

Algorithms and data structures

Simpler and faster algorithms: Bloom filters, skip lists, Miller-Rabin, ...

Machine learning

Stochastic gradient descent, Markov chain Monte Carlo, ...

Differential privacy

Adding noise: Laplace & Gaussian mechanisms, randomized response, ...

Randomness is an essential tool

Algorithms and data structures

Simpler and faster algorithms: Bloom filters, skip lists, Miller-Rabin, ...

Differential privacy

Adding noise: Laplace & Gaussian mechanisms, randomized response, ...

Machine learning

Stochastic gradient descent, Markov chain Monte Carlo, ...

Cryptography

Keys, nonces, modeling hashes, pseudorandom functions, ...

Randomness is an essential tool

Algorithms and data structures

Simpler and faster algorithms: Bloom filters, skip lists, Miller-Rabin, ...

Machine learning

Stochastic gradient descent, Markov chain Monte Carlo, ...

Differential privacy

Adding noise: Laplace & Gaussian mechanisms, randomized response, ...

Cryptography

Keys, nonces, modeling hashes, pseudorandom functions, ...

Program verification techniques need to handle randomness!

Many PL successes!

Randomized algorithms

- Union-bound logic (aHL): upper bound error probabilities
- Ellora: polynomial identity testing, random walks, concentration bounds, ...
- Caesar/HeyVL: weakest pre-expectation reasoning for pGCL

Differential privacy

- apRHL: Laplace & Gaussian mechanisms, Sparse Vector, Above Threshold
- Fuzz/DFuzz/Duet/LightDP: types for DP, automated proofs

Cryptographic security

EasyCrypt/CryptHOL: OAEP, Cramer–Shoup, FDH signatures, ElGamal, constant-time MEE-CBC, HMAC, KEM-DEM, PRF-based encryption, ...

Limitations... and a divide

Limitations... and a divide

Probabilistic reasoning

- small programs
- restricted languages
- sequential
- not compositional

Limitations... and a divide

Probabilistic reasoning

- small programs
- restricted languages
- sequential
- not compositional

Separation logic at scale

- medium programs (>4k LoC)
- sophisticated languages
- concurrency/distributed
- modular reasoning

Limitations... and a divide



Probabilistic reasoning

- small programs
- restricted languages
- sequential
- not compositional

Separation logic at scale

- medium programs (>4k LoC)
- sophisticated languages
- concurrency/distributed
- modular reasoning

Can we bridge these two worlds?

CLUTCH clutch-project.org

An ongoing effort to extend modern separation logic with probabilistic reasoning.

- **Relational reasoning:** Clutch [POPL24], Approxis [POPL25], Foxtrot [PLDI26]
- **Differential privacy:** Clutch-DP [PLDI26]
- **Expected cost:** Tachis [OOPSLA24]
- **Almost-sure termination:** Caliper [ICFP24]
- **Error bounding:** Eris [ICFP24, CPP26], Coneris [ICFP25], Cont. Eris [LICS26]

All results mechanized in the Rocq prover on top of the Iris framework.



Cryptographic protocols

Cryptographic protocols

Cryptographic protocols are inherently **randomized** and **distributed**; and they also have to be secure in the presence of **adversarial** behavior.

Cryptographic protocols

Cryptographic protocols are inherently **randomized** and **distributed**; and they also have to be secure in the presence of **adversarial** behavior.

Existing verification techniques have limitations:

- **symbolic** analysis tools (e.g. Tamarin) abstract away the randomization, and
- **computational** tools (e.g. EasyCrypt) offer limited/no support for distribution.

Cryptographic protocols

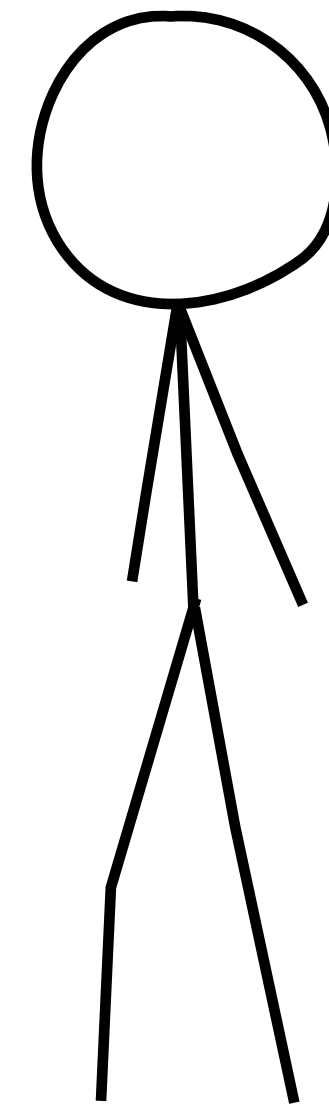
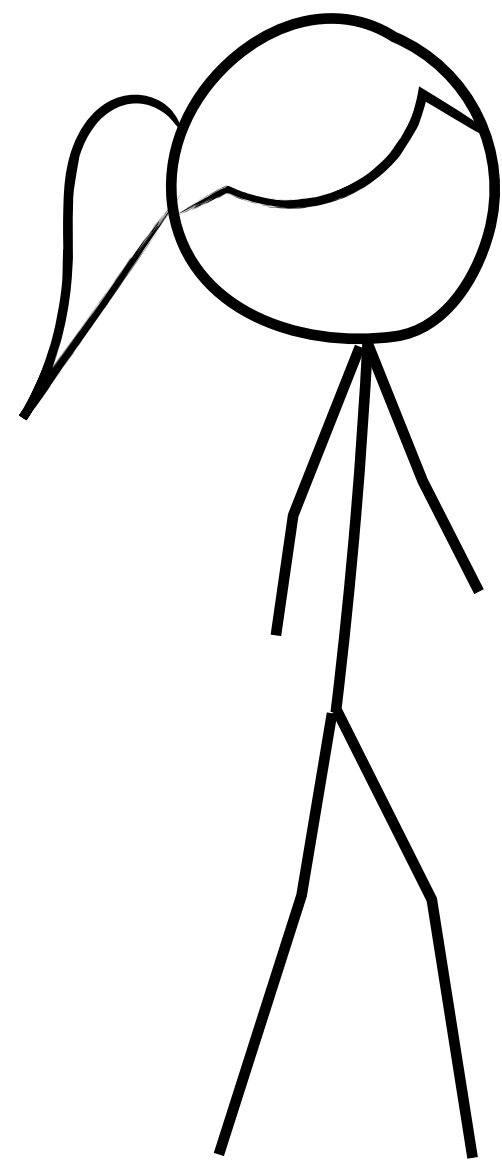
Cryptographic protocols are inherently **randomized** and **distributed**; and they also have to be secure in the presence of **adversarial** behavior.

Existing verification techniques have limitations:

- **symbolic** analysis tools (e.g. Tamarin) abstract away the randomization, and
- **computational** tools (e.g. EasyCrypt) offer limited/no support for distribution.

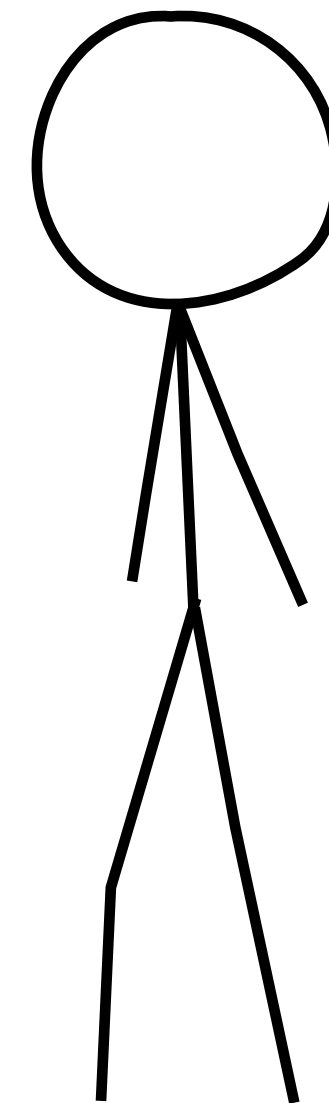
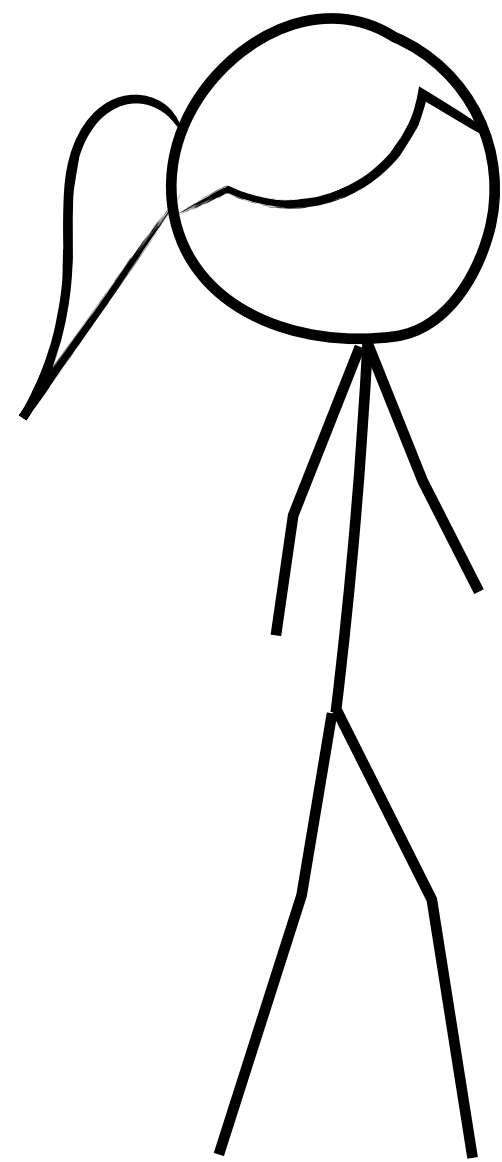
Our goal: establishing end-to-end security of cryptographic protocols.

Example: Diffie-Hellman Key Exchange



Example: Diffie-Hellman Key Exchange

Fix a finite cyclic group G with generator g .

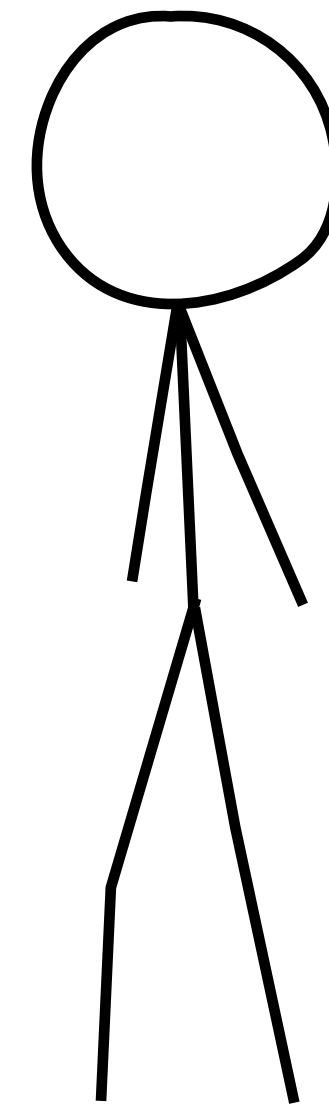
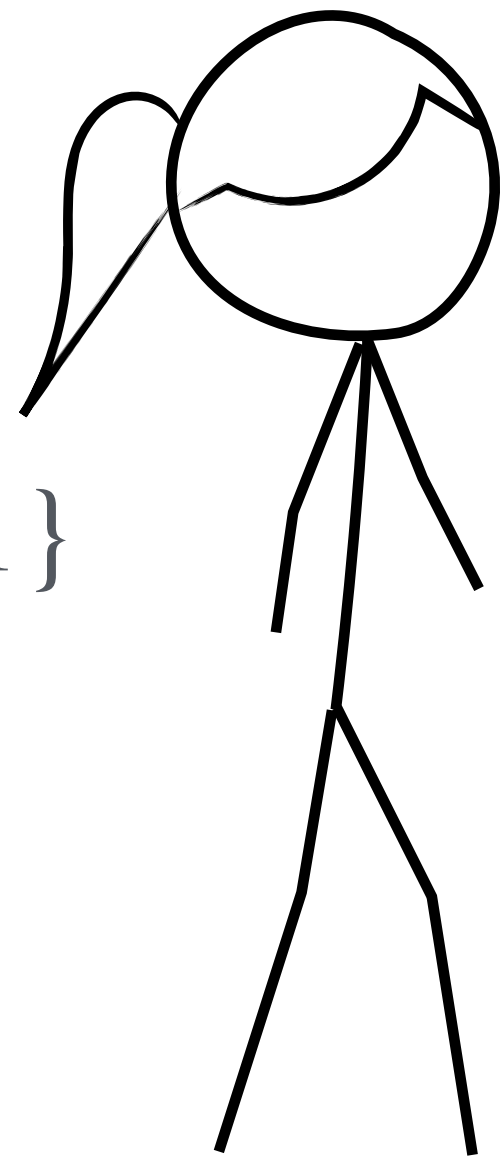


Example: Diffie-Hellman Key Exchange

Fix a finite cyclic group G with generator g .

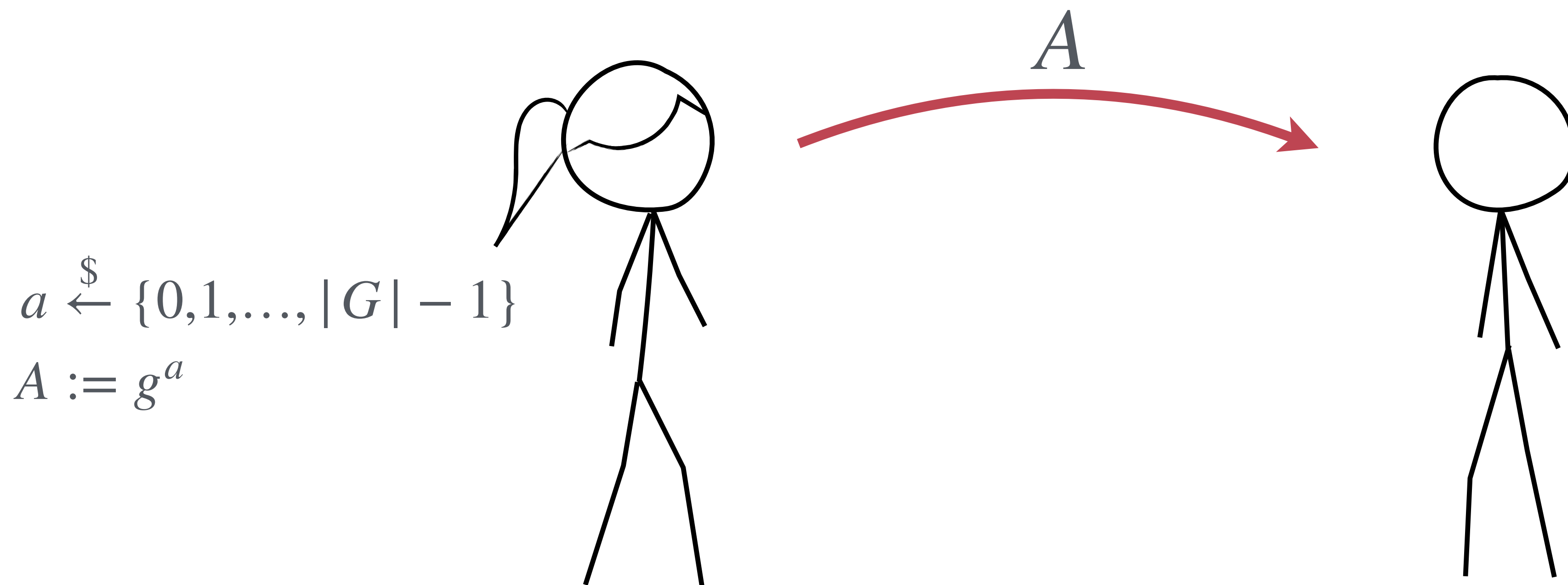
$$a \xleftarrow{\$} \{0, 1, \dots, |G| - 1\}$$

$$A := g^a$$



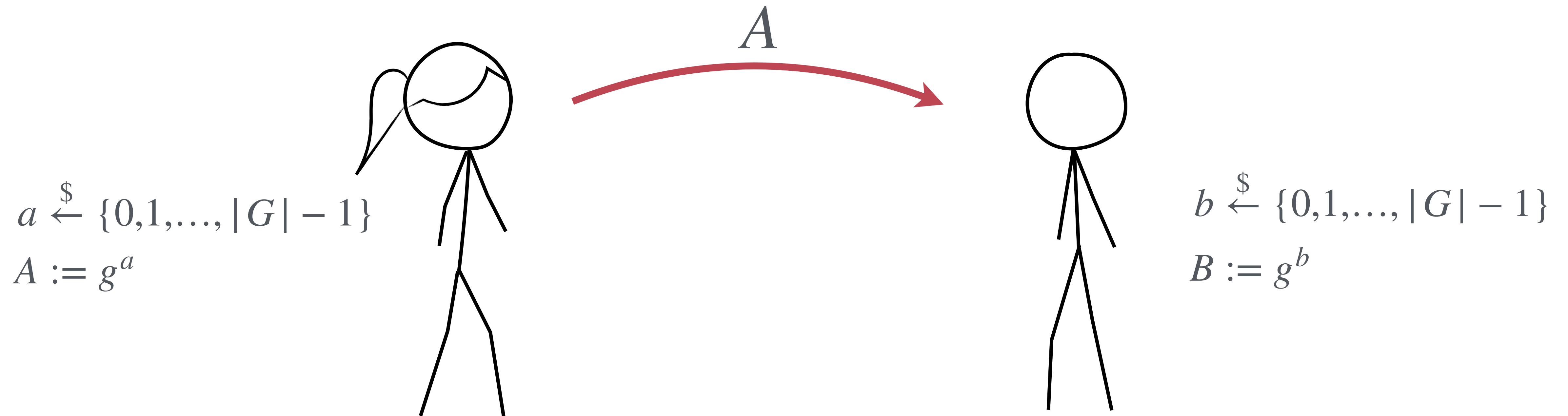
Example: Diffie-Hellman Key Exchange

Fix a finite cyclic group G with generator g .



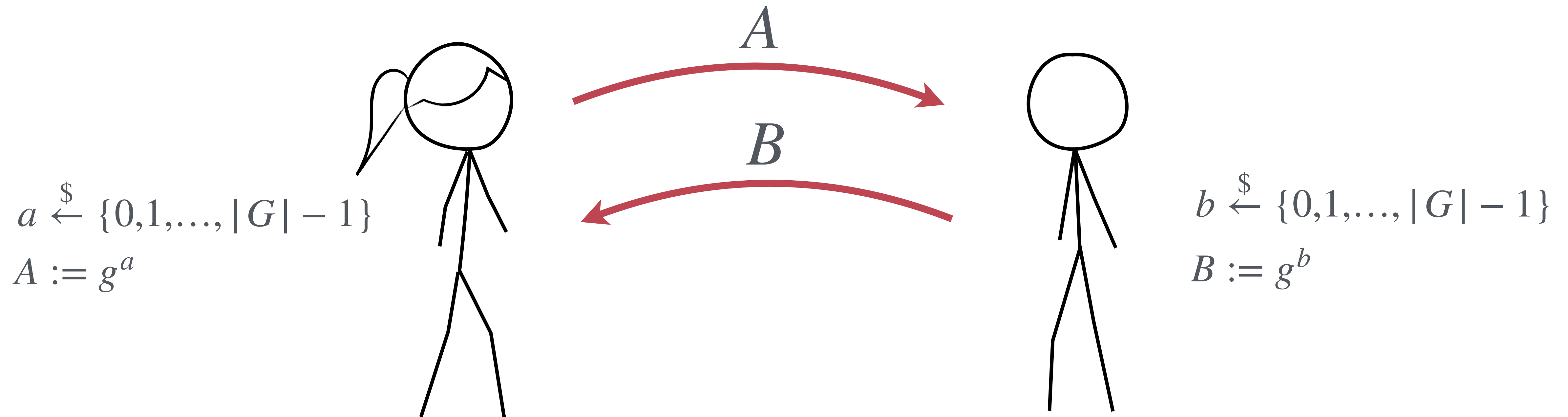
Example: Diffie-Hellman Key Exchange

Fix a finite cyclic group G with generator g .



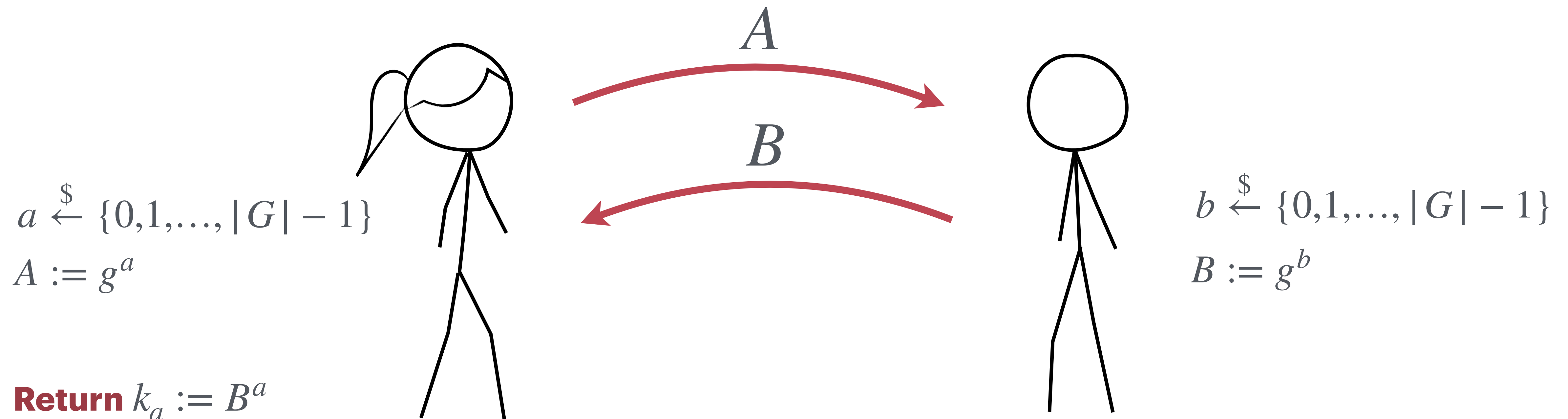
Example: Diffie-Hellman Key Exchange

Fix a finite cyclic group G with generator g .



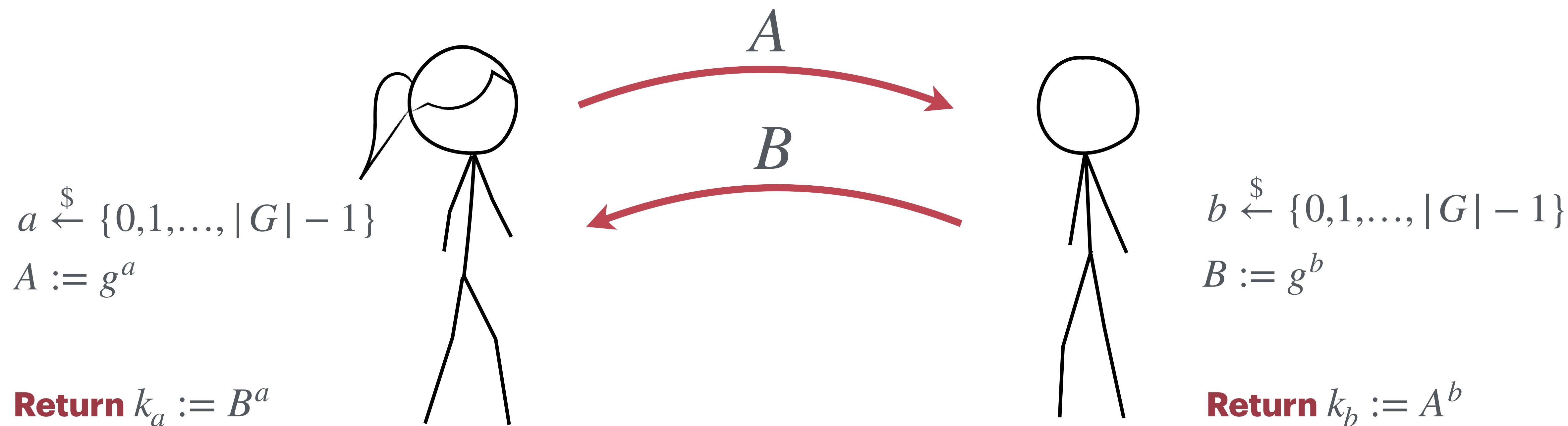
Example: Diffie-Hellman Key Exchange

Fix a finite cyclic group G with generator g .



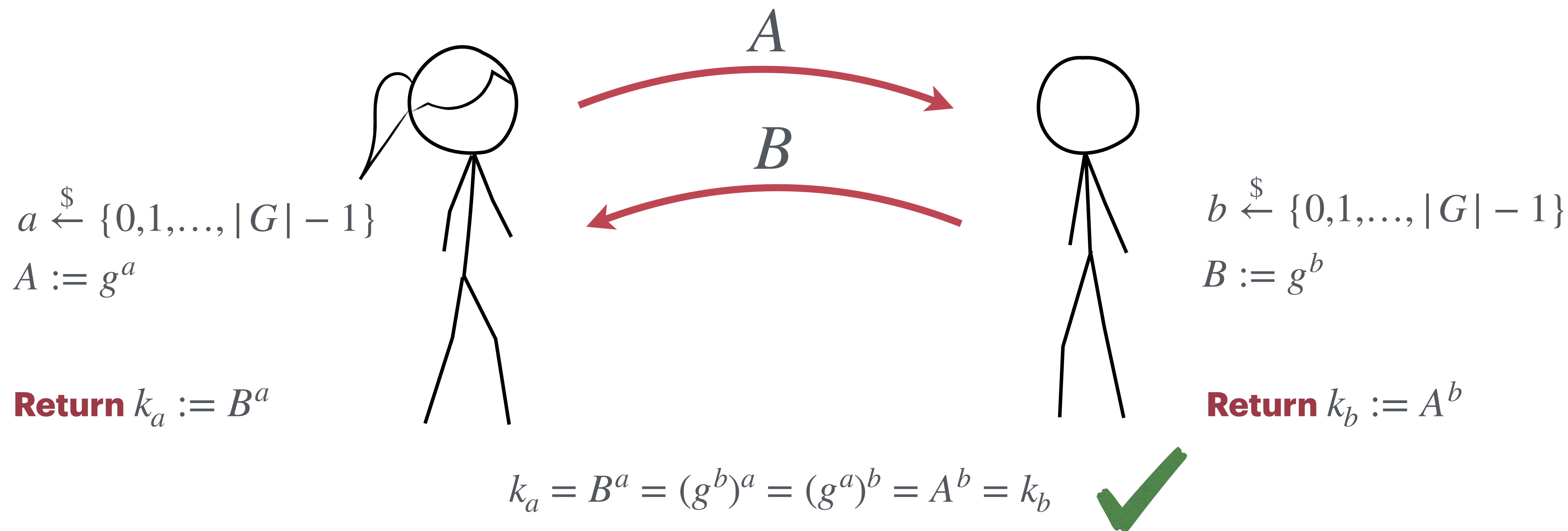
Example: Diffie-Hellman Key Exchange

Fix a finite cyclic group G with generator g .



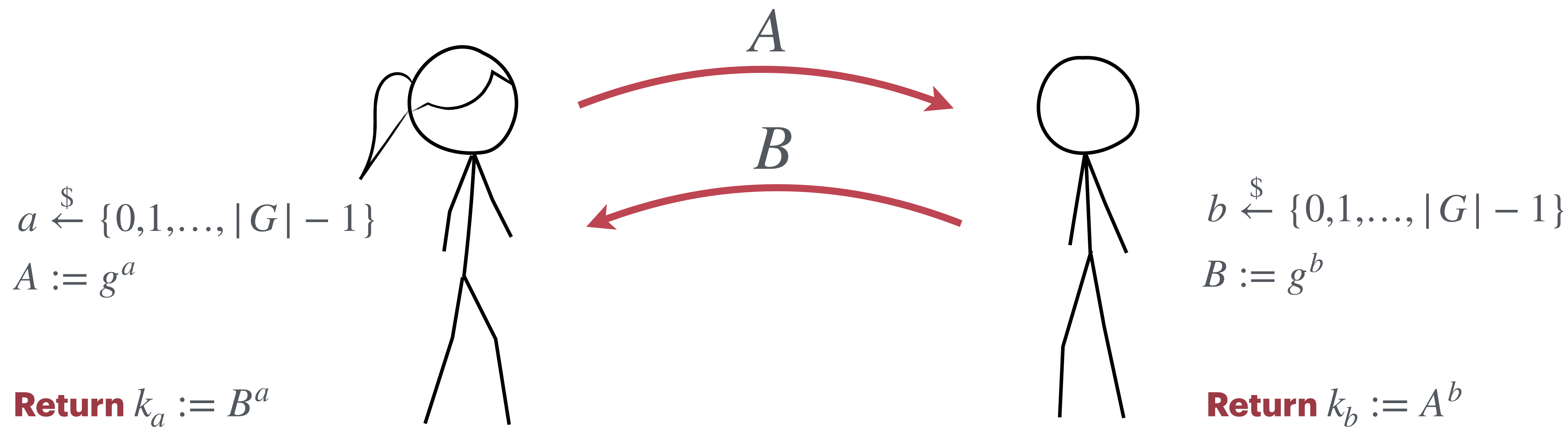
Example: Diffie-Hellman Key Exchange

Fix a finite cyclic group G with generator g .



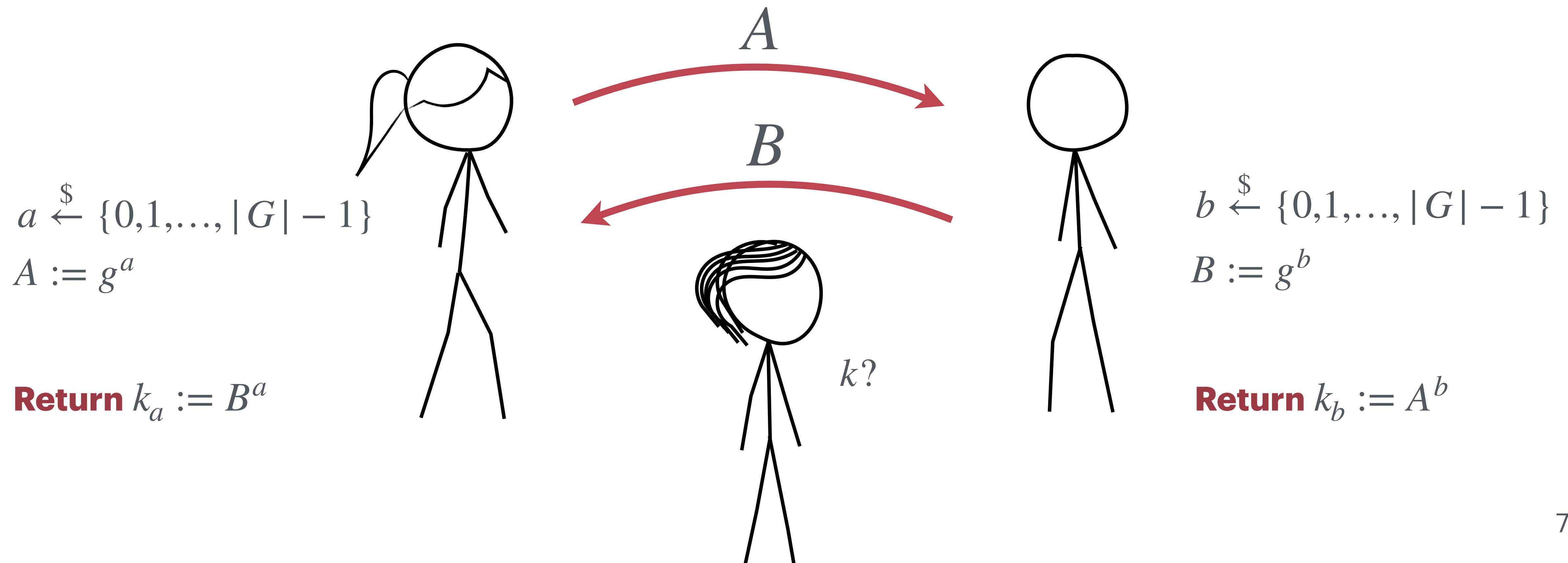
Example: Diffie-Hellman Key Exchange

Fix a finite cyclic group G with generator g .



Example: Diffie-Hellman Key Exchange

Fix a finite cyclic group G with generator g .



Decisional Diffie-Hellman assumption

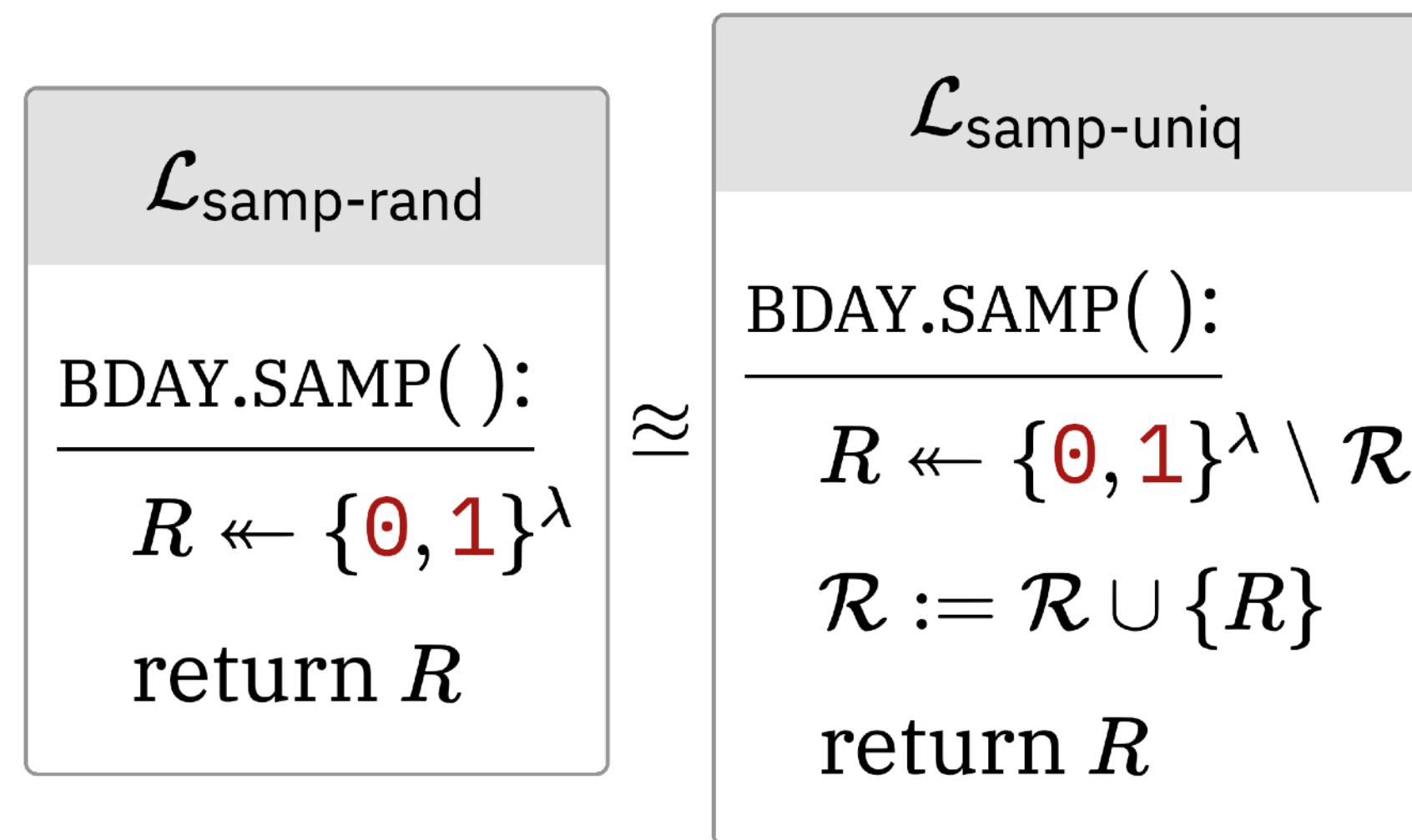
Fix a finite cyclic group G with generator g . The two distributions

- (g^a, g^b, g^{ab}) for uniformly and independently chosen $a, b \in \{0, 1, \dots, |G| - 1\}$
- (g^a, g^b, g^c) for uniformly and independently chosen $a, b, c \in \{0, 1, \dots, |G| - 1\}$

are **computationally indistinguishable** in the security parameter $n = \log |G|$.

Statistical reasoning

Many cryptographic constructions rely on “bad events” happening only with a small probability, such as **key** or **nonce collisions**.



(relies on, e.g., a birthday bound like $\varepsilon \leq \frac{q(q-1)}{2 \cdot 2^\lambda}$)

Requirements

Requirements

1. A **language** and **semantics** for expressing randomized behavior and distributed communication.

Requirements

1. A **language** and **semantics** for expressing randomized behavior and distributed communication.
2. Support for stating **protocol assumptions**.

Requirements

1. A **language** and **semantics** for expressing randomized behavior and distributed communication.
2. Support for stating **protocol assumptions**.
3. A **security statement**, including a **realistic attacker model**, e.g., adversary-controlled network and corrupted participants.

Requirements

1. A **language** and **semantics** for expressing randomized behavior and distributed communication.
2. Support for stating **protocol assumptions**.
3. A **security statement**, including a **realistic attacker model**, e.g., adversary-controlled network and corrupted participants.
4. A proof technique for reasoning about such systems, including

Requirements

1. A **language** and **semantics** for expressing randomized behavior and distributed communication.
2. Support for stating **protocol assumptions**.
3. A **security statement**, including a **realistic attacker model**, e.g., adversary-controlled network and corrupted participants.
4. A proof technique for reasoning about such systems, including
 - i. reductions to computational hardness assumptions like DDH, and

Requirements

1. A **language** and **semantics** for expressing randomized behavior and distributed communication.
2. Support for stating **protocol assumptions**.
3. A **security statement**, including a **realistic attacker model**, e.g., adversary-controlled network and corrupted participants.
4. A proof technique for reasoning about such systems, including
 - i. reductions to computational hardness assumptions like DDH, and
 - ii. statistical/approximative equivalences.

Pen-and-paper security proofs



Ran Canetti, BU

The **Universal Composability** (UC) framework by Ran Canetti is the de-facto gold standard for specifying protocols and security.

- Protocols remain secure under “arbitrary composition”.
- Protocols are analyzed by **decomposing** them into simpler sub-protocols.
- Follows the **real/ideal paradigm**.

$$\text{ENV}(\text{PROTOCOL}) \approx \text{ENV}(\text{SIMULATOR}(\text{IDEAL}))$$

This work

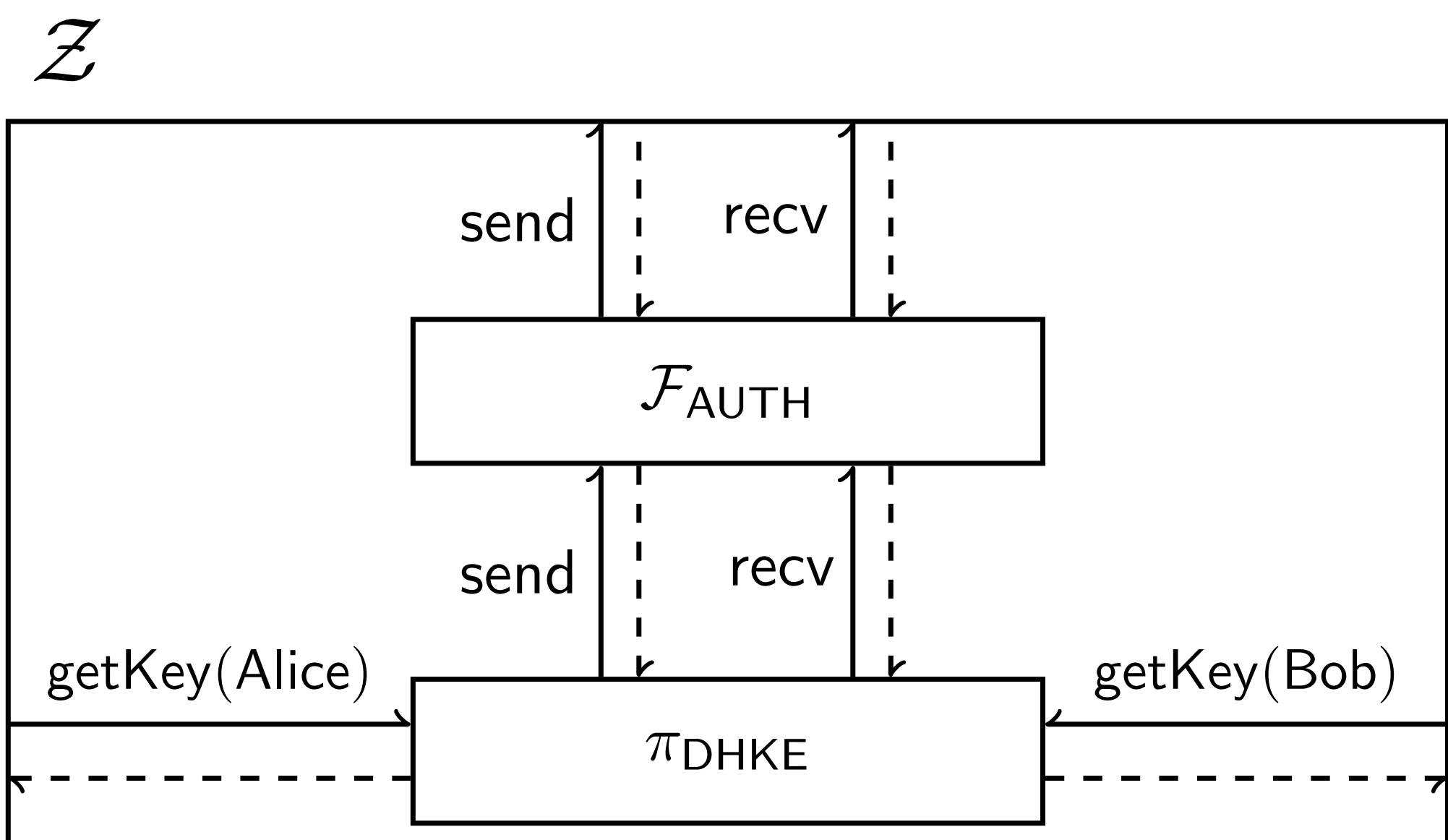
- Effect handlers allow us to capture UC-style functionalities programmatically.
- Security is an (approximate) equivalence of effect-handler programs.
- The adversary-controlled network becomes an unknown program.

We know how to reason about all that stuff!

DHKE in UC

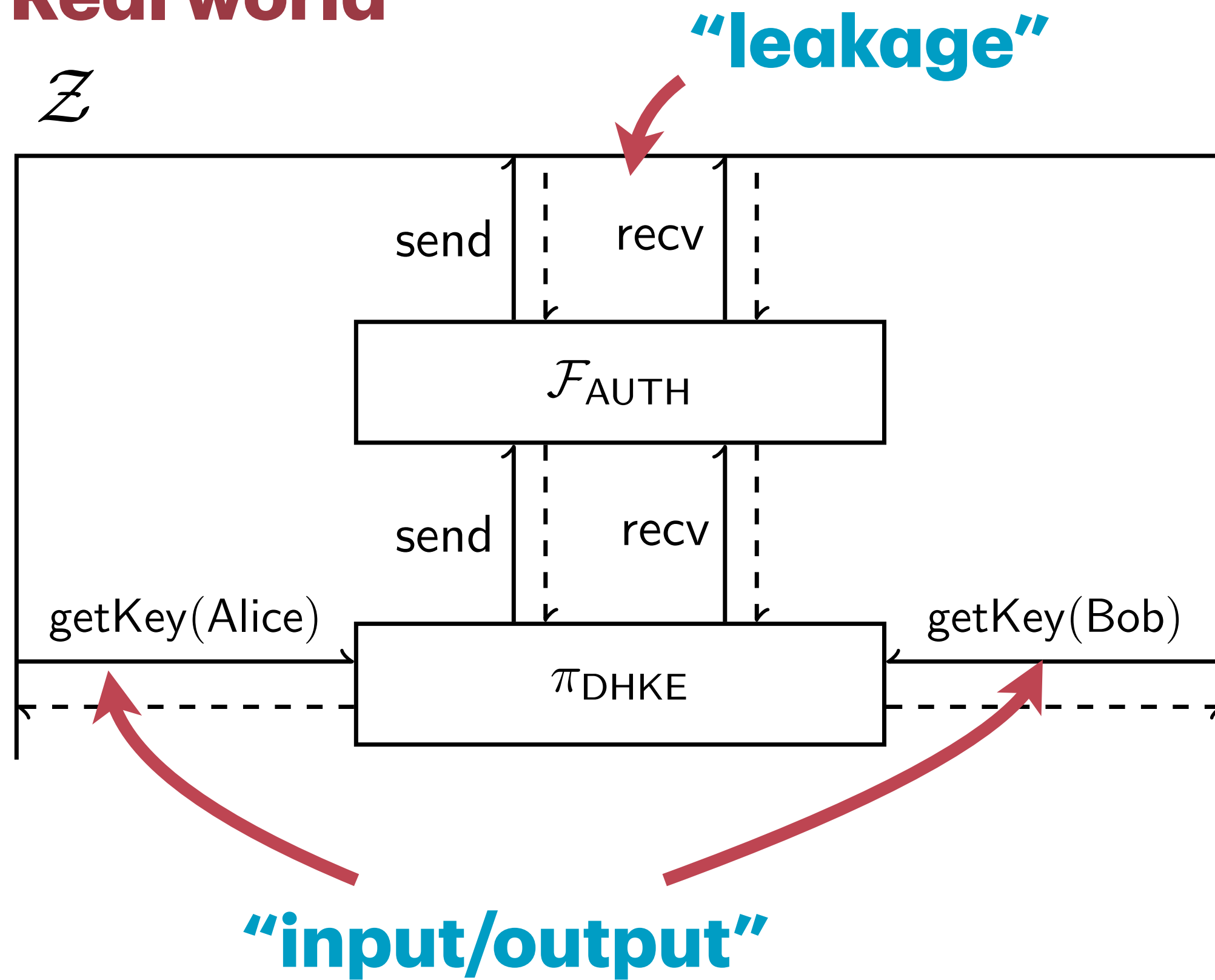
DHKE in UC

Real world



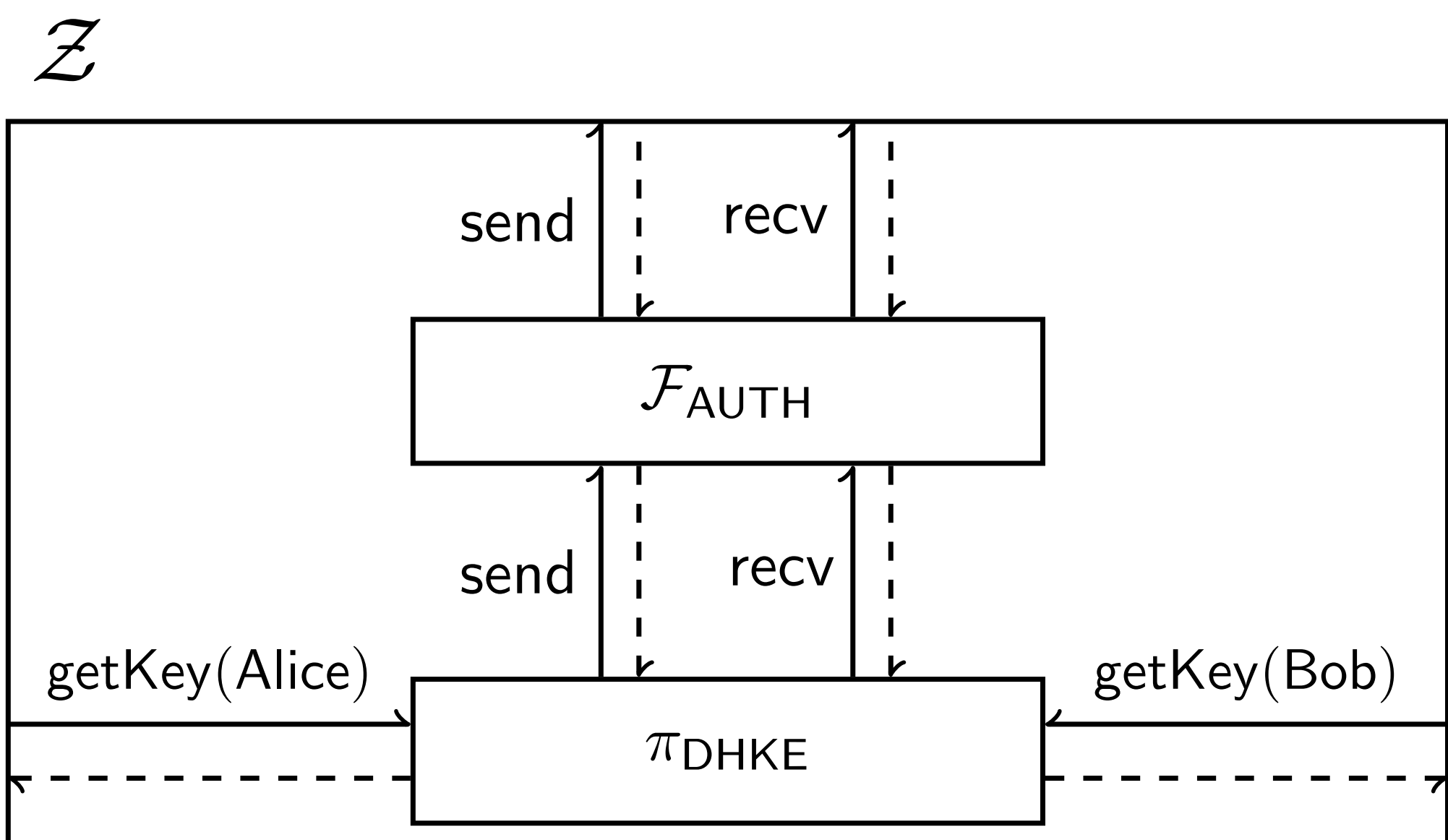
DHKE in UC

Real world



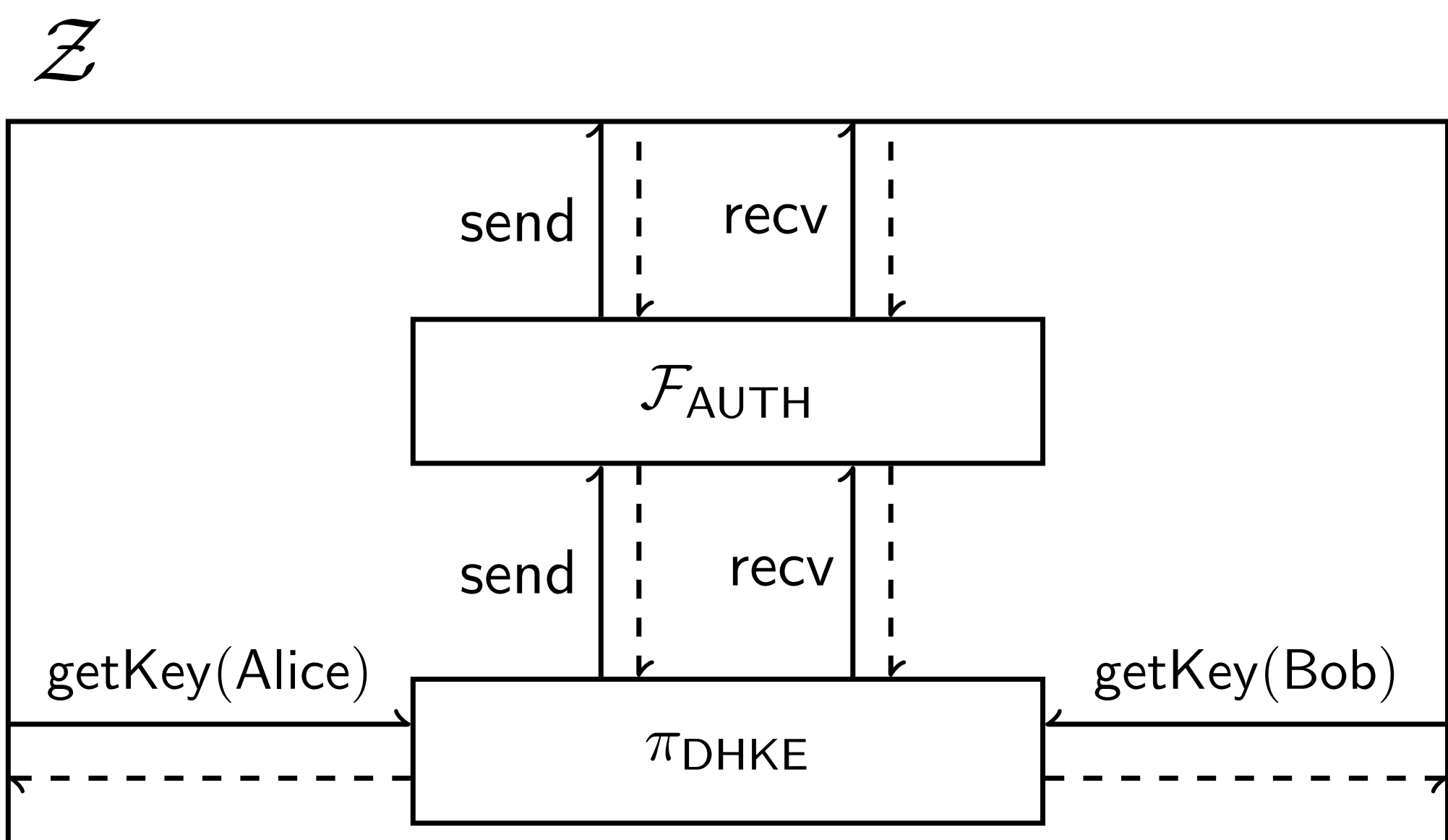
DHKE in UC

Real world

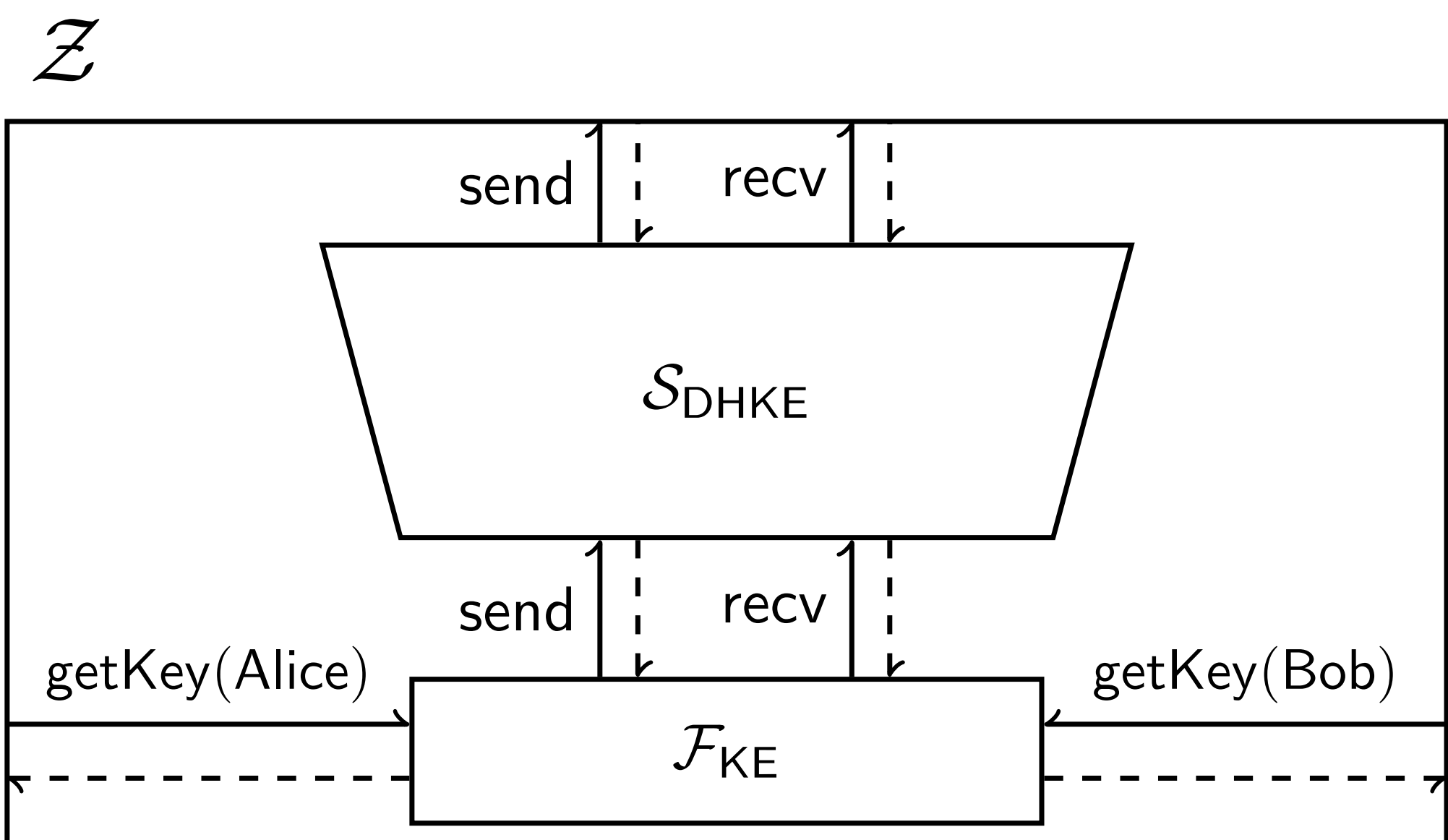


DHKE in UC

Real world

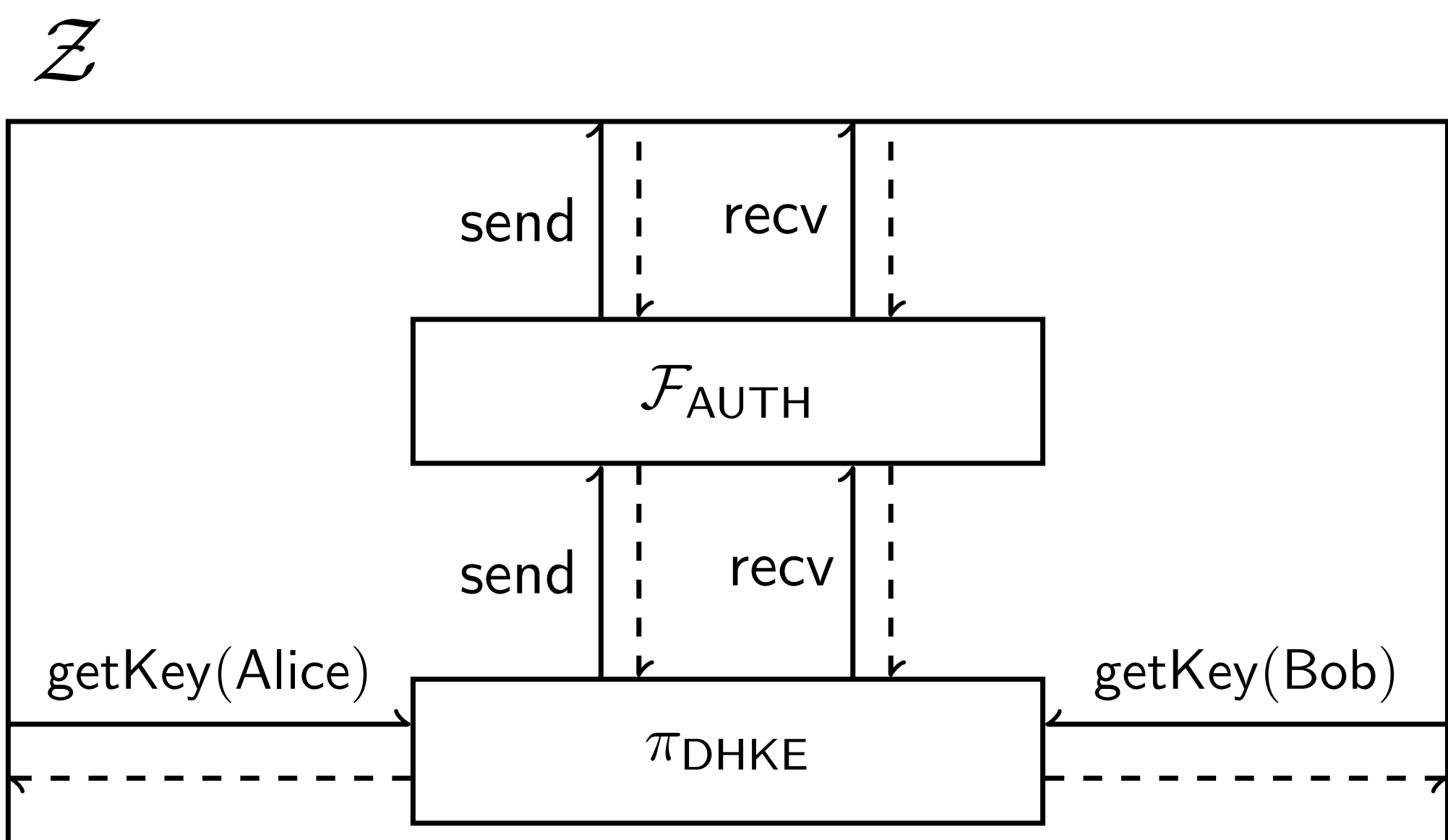


Ideal world

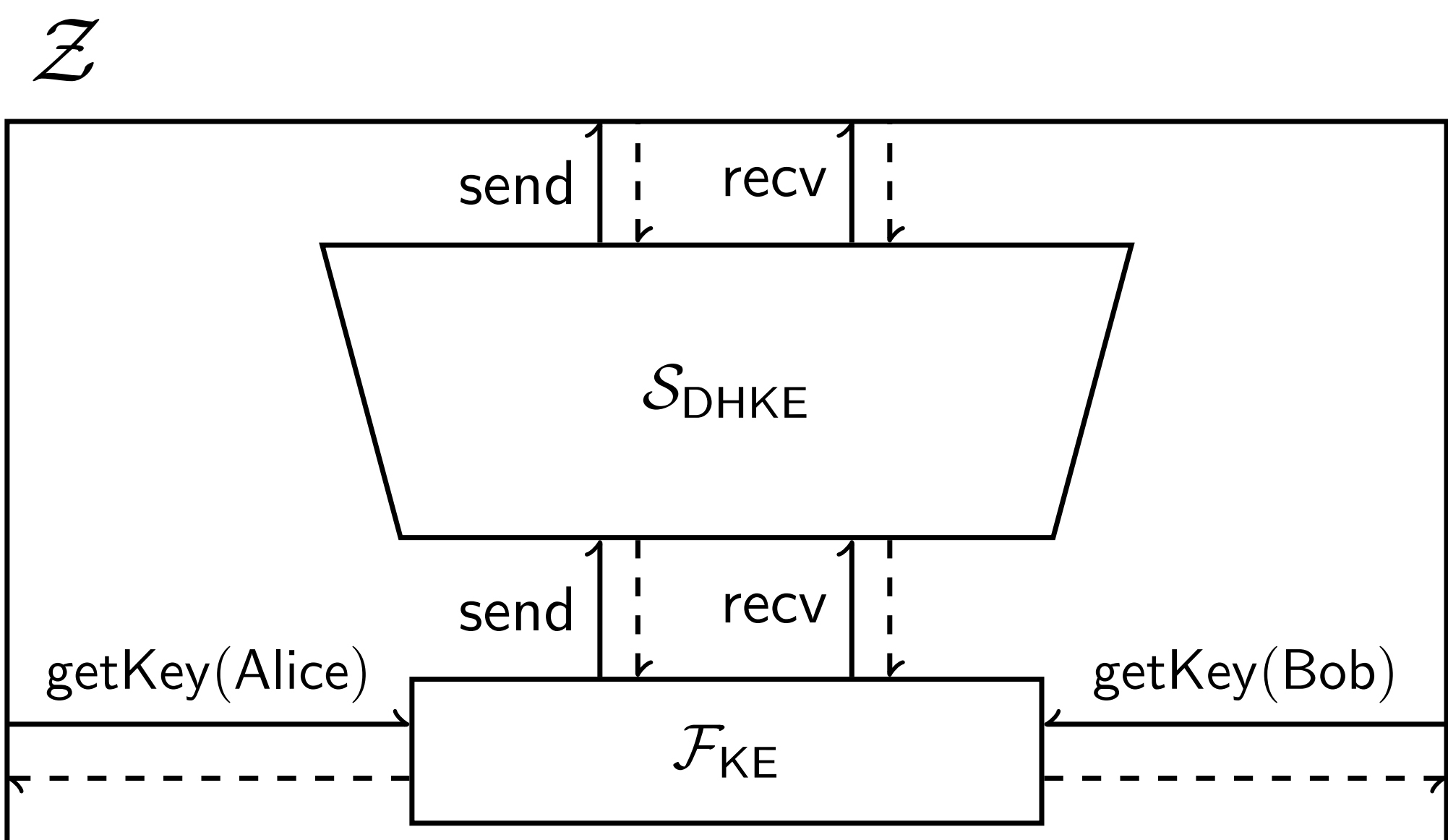


DHKE in UC

Real world



Ideal world



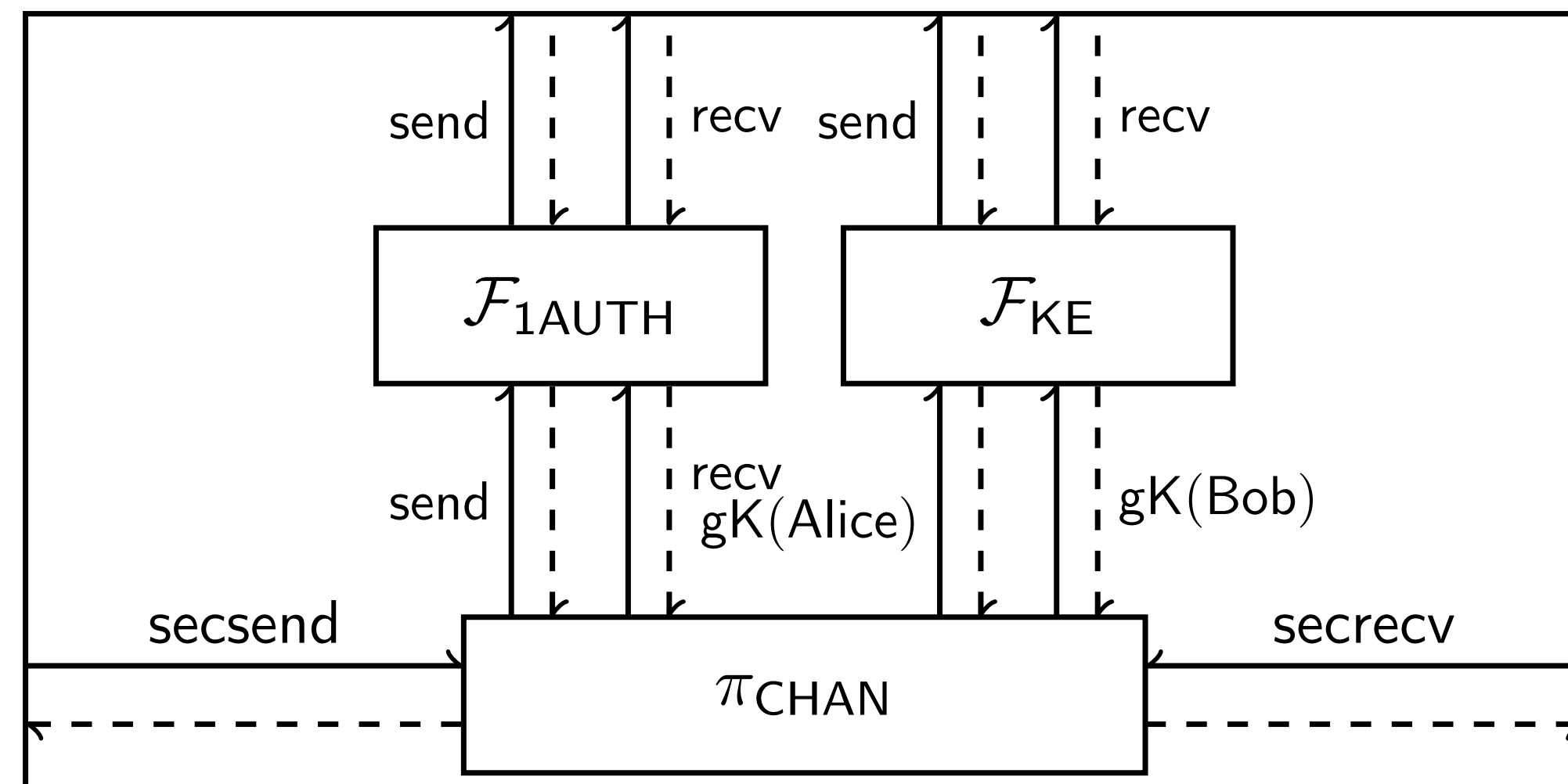
$\approx$

π realizes $\mathcal{F}$ when $\exists \mathcal{S}$ such that $\forall \mathcal{Z}. \mathcal{Z}(\pi) \approx \mathcal{Z}(\mathcal{S} \circ \mathcal{F})$

A secure channel in UC

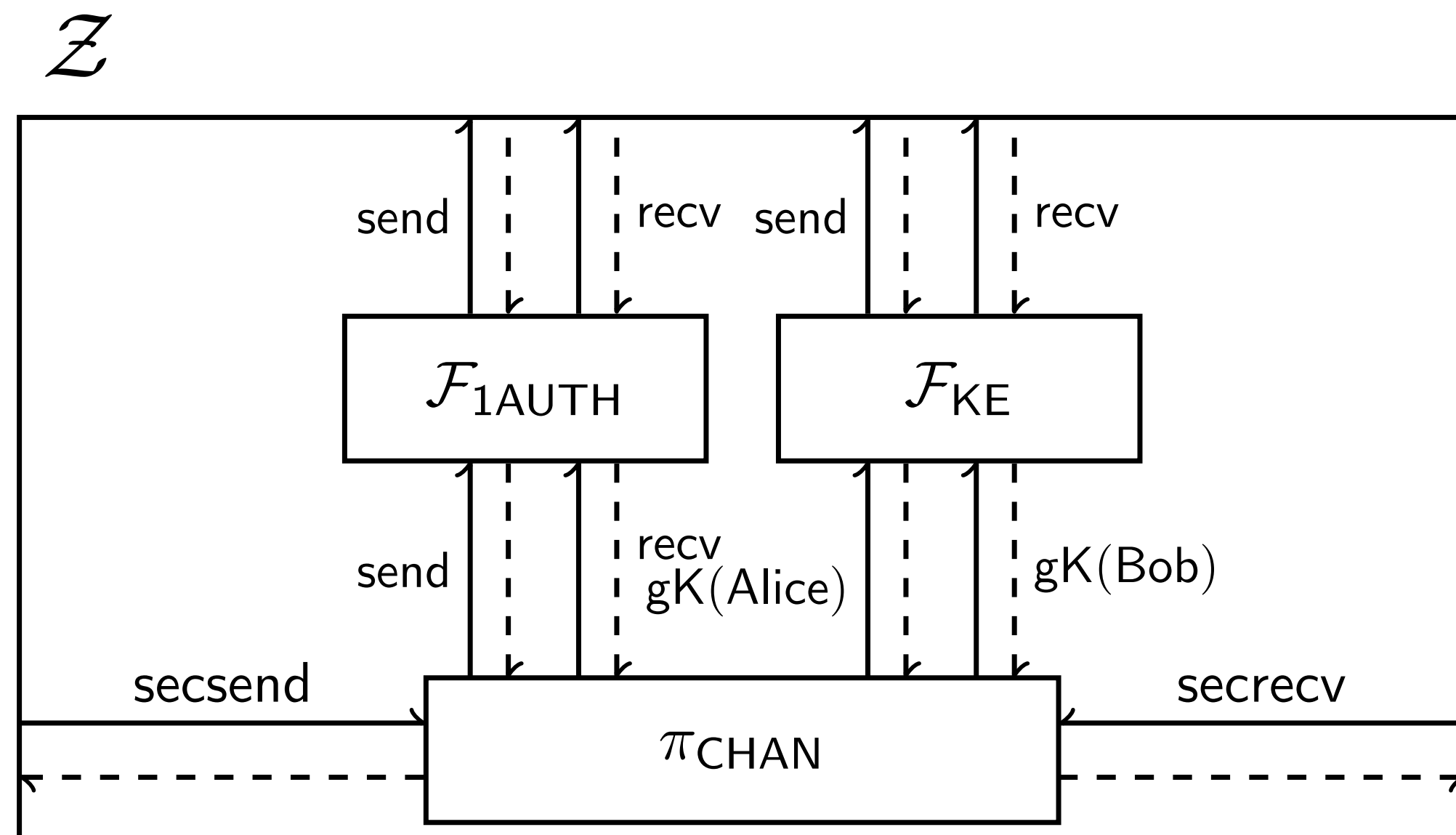
Real world

$\mathcal{Z}$

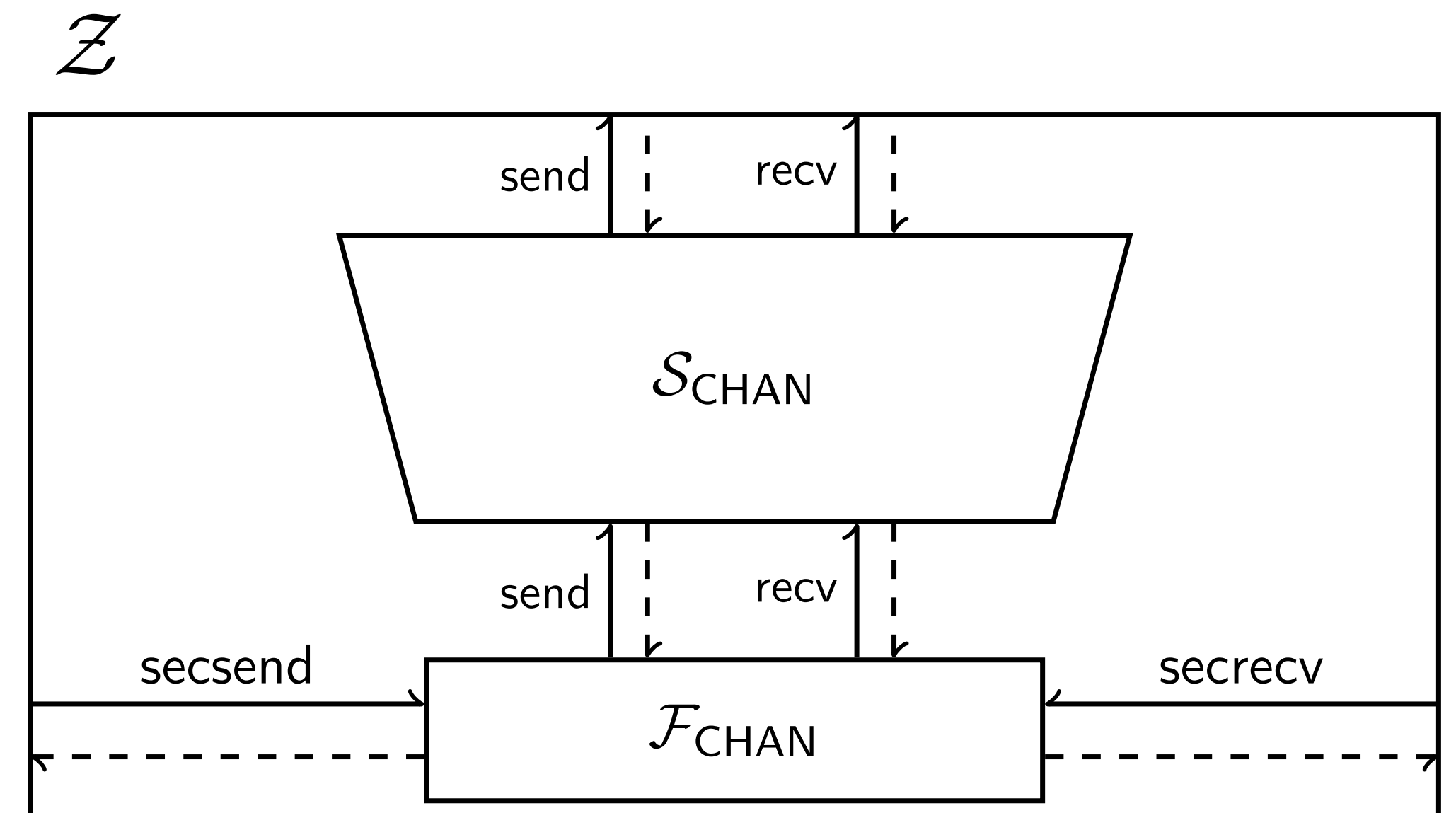


A secure channel in UC

Real world



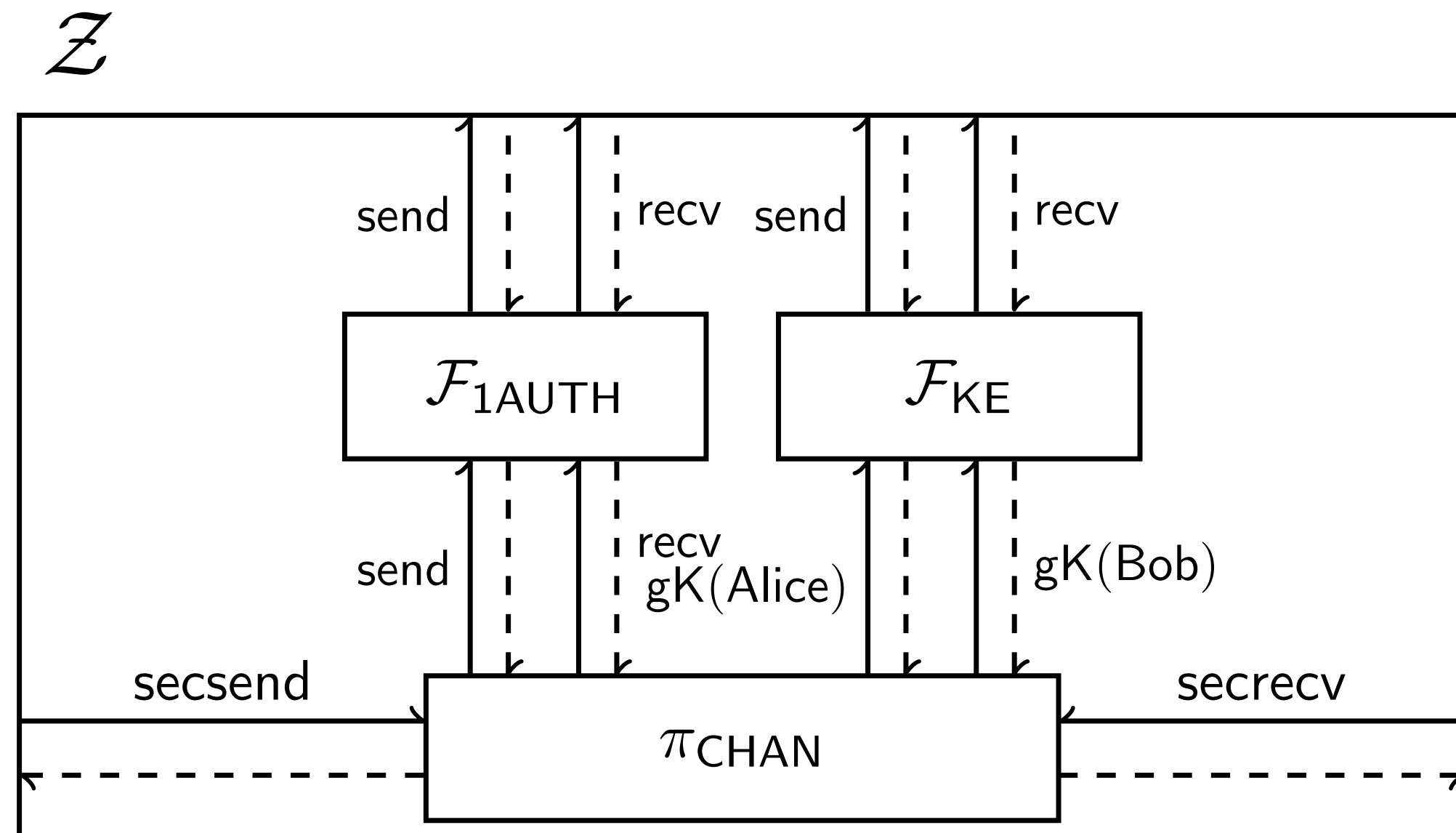
Ideal world



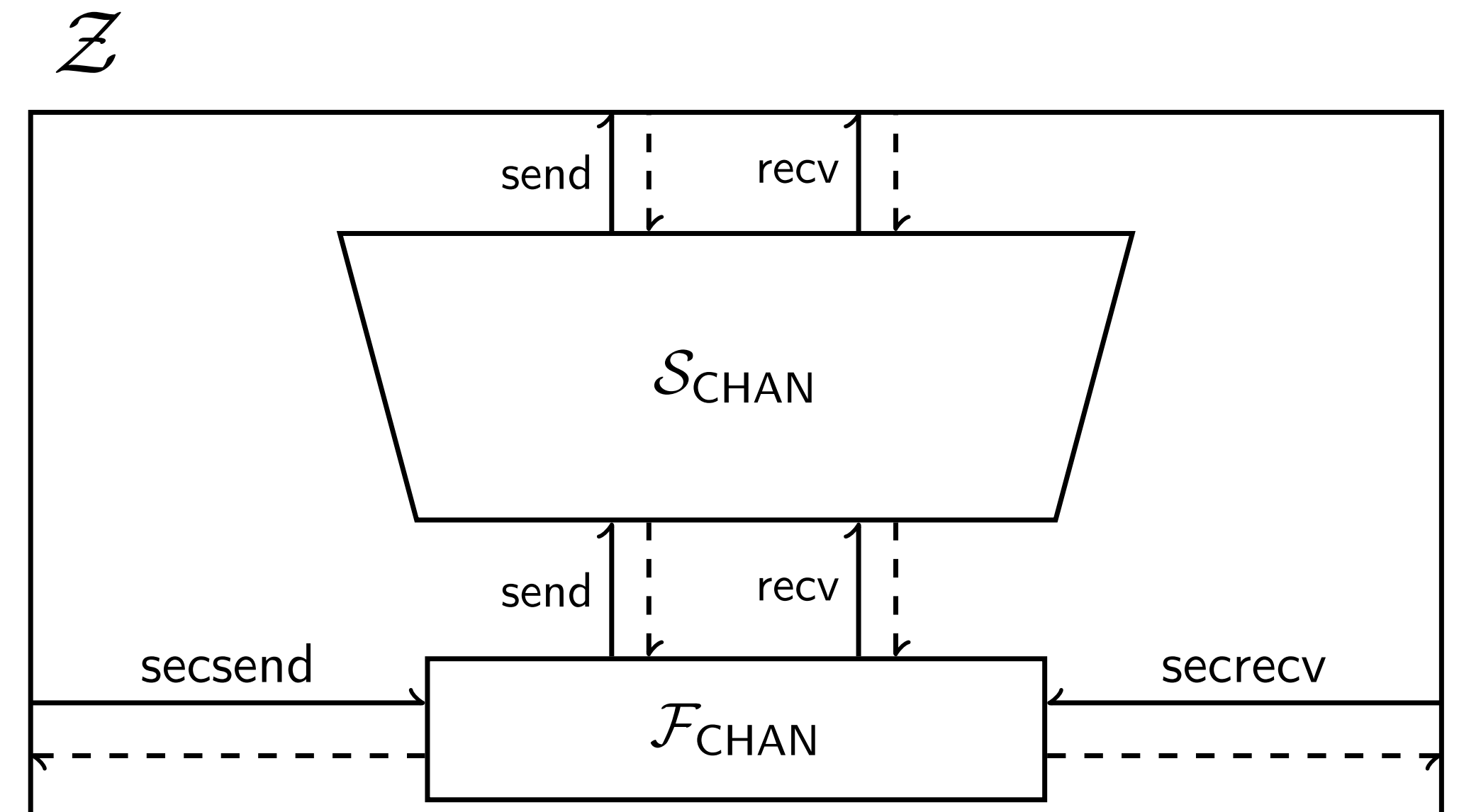
$\approx$

A secure channel in UC

Real world



Ideal world



$\approx$

By the UC composition theorem, $\mathcal{F}_{1\text{AUTH}} \circ \pi_{\text{DHKE}} \circ \pi_{\text{CHAN}}$ is secure.

Key observation

The interaction between the different UC entities **can be captured using effects!**

Effects may correspond to entire sub-protocols, but we will we **interpret** them in different ways when stating protocol security.

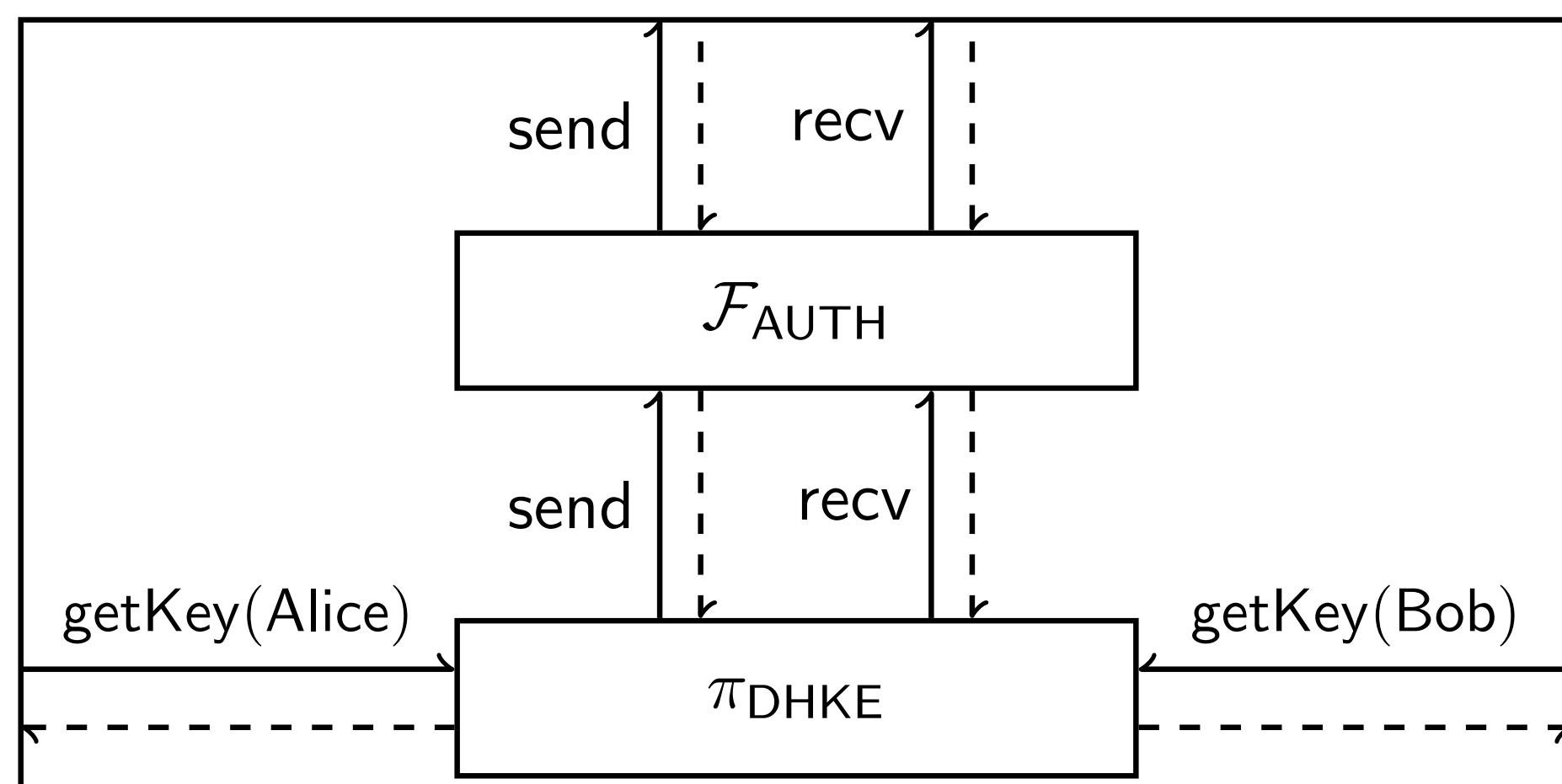
Key observation

The interaction between the different UC entities **can be captured using effects!**

Effects may correspond to entire sub-protocols, but we will **interpret** them in different ways when stating protocol security.

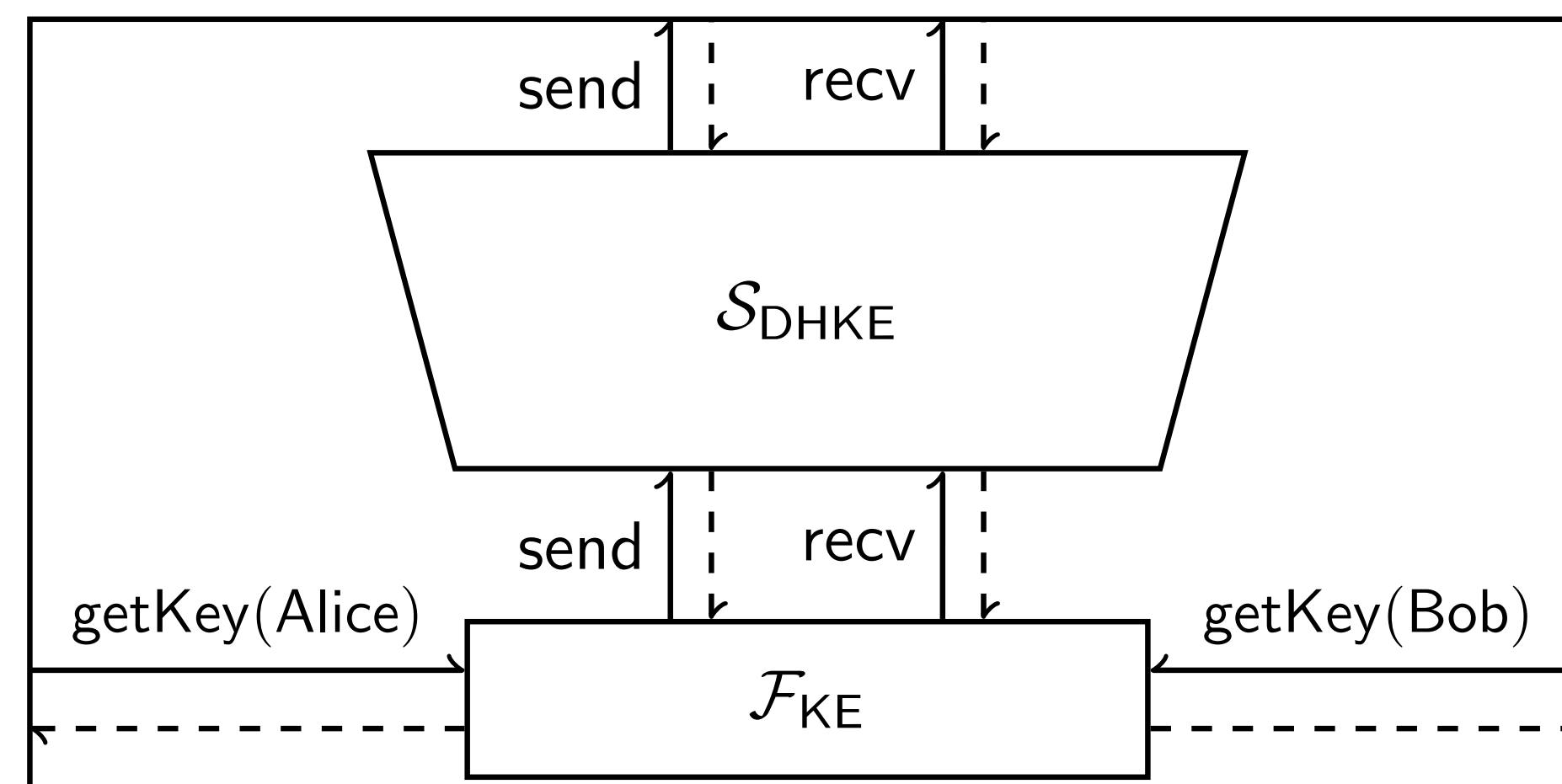
Real

$\mathcal{Z}$



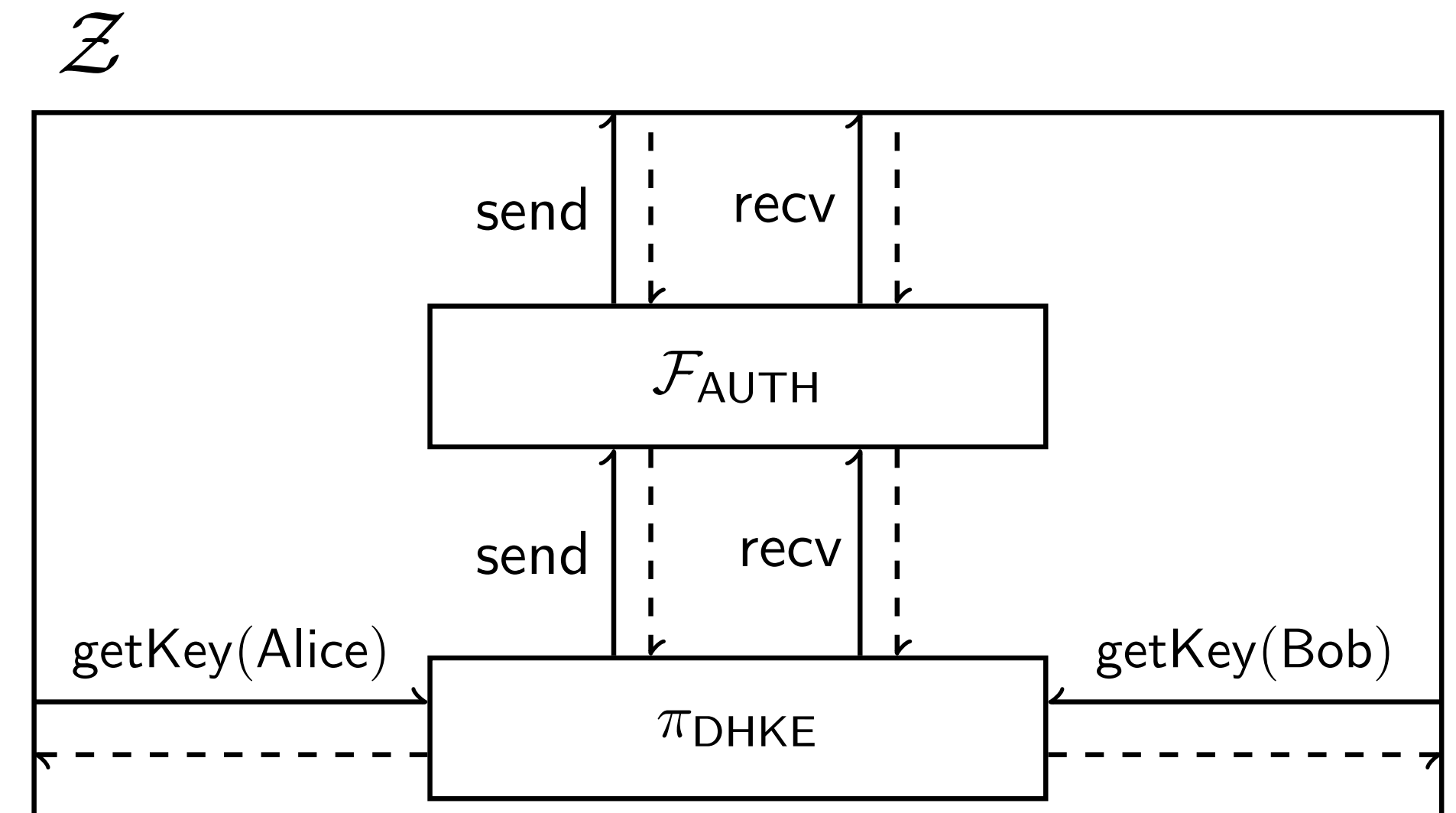
Ideal

$\mathcal{Z}$



Informal concept mapping

- Protocols and functionalities become **effect handling** contexts.
- Functionalities are invoked using **effects**.
- Leakage correspond to **effects that propagate to the environment**.



Effects and handlers 101

```
handle
  (* ... *)
  let x = do eff 41 in
  (* ... *)
with
| effect eff v, k  $\Rightarrow$ 
  (* ... *)
  k (v + 1);
  (* ... *)
```

Effects and handlers 101

```
handle
  (* ... *)
  let x = do eff 41 in
  (* ... *)
  with
  | effect eff v, k =>
    (* ... *)
    k (v + 1);
    (* ... *)
```



Effects and handlers 101

```
handle
  (* ... *)
  let x = do eff 41 in
  (* ... *)
with
| effect eff v, k =>
  (* ... *)
  k (v + 1);
  (* ... *)
```



Effects and handlers 101

```
handle
  (* ... *)
  let x = 42 in
  (* ... *)
with
| effect eff v, k  $\Rightarrow$ 
  (* ... *)
  k (v + 1);
  (* ... *)
```

Effects and handlers 101

```
handle
  (* ... *)
  let x = 42 in
  (* ... *)
with
| effect eff v, k ⇒
  (* ... *)
  k (v + 1);
  (* ... *)
```

For the sake of this talk:

- Deep handlers
- One-shot continuations

DHKE using effects and handlers

```
 $\pi_{\text{DHKE}} f \triangleq$   
  let la = ref None in  
  let lb = ref None in  
  handle f () with  
  | effect getKey Alice, k  $\Rightarrow$   
    let a = sample_or_read la in  
    let A = exp g a in  
    do send (A, Bob);  
    let r = do recv Bob in  
    match r with  
    | None  $\Rightarrow$  k None  
    | Some B  $\Rightarrow$   
      let key = exp B a in  
      k (Some key)  
  (* ... *)
```

DHKE using effects and handlers

```
 $\pi_{\text{DHKE}} f \triangleq$   
let la = ref None in  
let lb = ref None in  
handle f () with  
| effect getKey Alice, k  $\Rightarrow$   
  let a = sample_or_read la in  
  let A = exp g a in  
  do send (A, Bob);  
  let r = do recv Bob in  
  match r with  
  | None  $\Rightarrow$  k None  
  | Some B  $\Rightarrow$   
    let key = exp B a in  
    k (Some key)  
(* ... *)
```

DHKE using effects and handlers

```
 $\pi_{\text{DHKE}} f \triangleq$   
  let la = ref None in  
  let lb = ref None in  
  handle f () with  
  | effect getKey Alice, k  $\Rightarrow$   
    let a = sample_or_read la in  
    let A = exp g a in  
    do send (A, Bob);  
    let r = do recv Bob in  
    match r with  
    | None  $\Rightarrow$  k None  
    | Some B  $\Rightarrow$   
      let key = exp B a in  
      k (Some key)  
  (* ... *)
```

DHKE using effects and handlers

```
 $\pi_{\text{DHKE}} f \triangleq$   
  let la = ref None in  
  let lb = ref None in  
  handle f () with  
  | effect getKey Alice, k  $\Rightarrow$   
    let a = sample_or_read la in  
    let A = exp g a in  
    do send (A, Bob);  
    let r = do recv Bob in  
    match r with  
    | None  $\Rightarrow$  k None  
    | Some B  $\Rightarrow$   
      let key = exp B a in  
      k (Some key)  
(* ... *)
```

```
sample_or_read l  $\triangleq$   
  match !l with  
  | None  $\Rightarrow$  let c = rand |G| in  
    l := Some c; c  
  | Some c  $\Rightarrow$  c
```

DHKE using effects and handlers

```
 $\pi_{\text{DHKE}} f \triangleq$   
  let la = ref None in  
  let lb = ref None in  
  handle f () with  
  | effect getKey Alice, k  $\Rightarrow$   
    let a = sample_or_read la in  
    let A = exp g a in  
    do send (A, Bob);  
    let r = do recv Bob in  
    match r with  
    | None  $\Rightarrow$  k None  
    | Some B  $\Rightarrow$   
      let key = exp B a in  
      k (Some key)  
  (* ... *)
```

```
(* ... *)  
| effect getKey Bob, k  $\Rightarrow$   
  let r = do recv Alice in  
  match r with  
  | None  $\Rightarrow$  k None  
  | Some A  $\Rightarrow$   
    let b = sample_or_read lb in  
    let B = exp g b in  
    do send (B, Alice);  
    let key = exp A b in  
    k (Some key)
```

```
sample_or_read l  $\triangleq$   
  match !l with  
  | None  $\Rightarrow$  let c = rand |G| in  
    l := Some c; c  
  | Some c  $\Rightarrow$  c
```

```

 $\mathcal{F}_{\text{AUTH}} \ f \triangleq$ 
  let m1 = ref None in
  let m2 = ref None in
  handle f () with
  | effect send (m, Alice), k  $\Rightarrow$ 
    match !m1 with
    | None  $\Rightarrow$ 
      m1 := Some m;
      do send (m, Alice); k ()
    | Some _  $\Rightarrow$  k ()
  (* ... *)
  | effect recv from, k  $\Rightarrow$ 
    let r = do recv from in
    match r with
    | None  $\Rightarrow$  k None
    | Some _  $\Rightarrow$ 
      match from with
      | Alice  $\Rightarrow$  k (!m2)
      | Bob  $\Rightarrow$  k (!m1)

```

```

 $\mathcal{F}_{\text{AUTH}} \ f \triangleq$ 
  let m1 = ref None in
  let m2 = ref None in
  handle f () with
  | effect send (m, Alice), k  $\Rightarrow$ 
    match !m1 with
    | None  $\Rightarrow$ 
      m1 := Some m;
      do send (m, Alice); k ()
    | Some _  $\Rightarrow$  k ()
  (* ... *)
  | effect recv from, k  $\Rightarrow$ 
    let r = do recv from in
    match r with
    | None  $\Rightarrow$  k None
    | Some _  $\Rightarrow$ 
      match from with
      | Alice  $\Rightarrow$  k (!m2)
      | Bob  $\Rightarrow$  k (!m1)

```

```

 $\mathcal{F}_{\text{AUTH}} \ f \triangleq$ 
  let m1 = ref None in
  let m2 = ref None in
  handle f () with
  | effect send (m, Alice), k  $\Rightarrow$ 
    match !m1 with
    | None  $\Rightarrow$ 
      m1 := Some m;
      do send (m, Alice); k ()
    | Some _  $\Rightarrow$  k ()
  (* ... *)
  | effect recv from, k  $\Rightarrow$ 
    let r = do recv from in
    match r with
    | None  $\Rightarrow$  k None
    | Some _  $\Rightarrow$ 
      match from with
      | Alice  $\Rightarrow$  k (!m2)
      | Bob  $\Rightarrow$  k (!m1)

```

```

 $\mathcal{F}_{KE} \ f \triangleq$ 
  let l = ref None in
  handle f () with
  | effect getKey Alice, k  $\Rightarrow$ 
    do send (0, Bob);
    let c = sample_or_read l in
    let key = exp g c in
    let r = do recv Bob in
    match r with
    | None  $\Rightarrow$  k None
    | Some _  $\Rightarrow$  k (Some key)
  | effect getKey Bob, k  $\Rightarrow$ 
    let r = do recv Alice in
    match r with
    | None  $\Rightarrow$  k None
    | Some _  $\Rightarrow$ 
      do send (0, Alice);
      match !l with
      | None  $\Rightarrow$  k None
      | Some c  $\Rightarrow$ 
        let key = exp g c in
        k (Some key)

```

$$\mathcal{F}_{\text{KE}} f \triangleq$$

```

let l = ref None in
handle f () with
| effect getKey Alice, k  $\Rightarrow$ 
  do send (0, Bob);
  let c = sample_or_read l in
  let key = exp g c in
  let r = do recv Bob in
  match r with
  | None  $\Rightarrow$  k None
  | Some _  $\Rightarrow$  k (Some key)
| effect getKey Bob, k  $\Rightarrow$ 
  let r = do recv Alice in
  match r with
  | None  $\Rightarrow$  k None
  | Some _  $\Rightarrow$ 
    do send (0, Alice);
    match !l with
    | None  $\Rightarrow$  k None
    | Some c  $\Rightarrow$ 
      let key = exp g c in
      k (Some key)

```

```

 $\mathcal{F}_{KE} \ f \triangleq$ 
  let l = ref None in
  handle f () with
  | effect getKey Alice, k  $\Rightarrow$ 
    do send (0, Bob);
    let c = sample_or_read l in
    let key = exp g c in
    let r = do recv Bob in
    match r with
    | None  $\Rightarrow$  k None
    | Some _  $\Rightarrow$  k (Some key)
  | effect getKey Bob, k  $\Rightarrow$ 
    let r = do recv Alice in
    match r with
    | None  $\Rightarrow$  k None
    | Some _  $\Rightarrow$ 
      do send (0, Alice);
      match !l with
      | None  $\Rightarrow$  k None
      | Some c  $\Rightarrow$ 
        let key = exp g c in
        k (Some key)

```

$$\mathcal{F}_{\text{KE}} f \triangleq$$

```

let l = ref None in
handle f () with
| effect getKey Alice, k  $\Rightarrow$ 
  do send (0, Bob);
  let c = sample_or_read l in
  let key = exp g c in
  let r = do recv Bob in
  match r with
  | None  $\Rightarrow$  k None
  | Some _  $\Rightarrow$  k (Some key)
| effect getKey Bob, k  $\Rightarrow$ 
  let r = do recv Alice in
  match r with
  | None  $\Rightarrow$  k None
  | Some _  $\Rightarrow$ 
    do send (0, Alice);
    match !l with
    | None  $\Rightarrow$  k None
    | Some c  $\Rightarrow$ 
      let key = exp g c in
      k (Some key)

```

$$\mathcal{S}_{\text{DHKE}} f \triangleq$$

```

let la = ref None in
let lb = ref None in
handle f () with
| effect send (_, Alice), k  $\Rightarrow$ 
  match !la with
  | Some _  $\Rightarrow$  k ()
  | None  $\Rightarrow$ 
    let c = sample_or_read la in
    let C = exp g c in
    do send (C, Alice);
    k ()
| effect send (_, Bob), k  $\Rightarrow$ 
  match !lb with
  | Some _  $\Rightarrow$  k ()
  | None  $\Rightarrow$ 
    let c = sample_or_read lb in
    let C = exp g c in
    do send (C, Bob);
    k ()
| effect recv from, k  $\Rightarrow$ 
  let r = do recv from in k r

```

$$\mathcal{F}_{\text{KE}} f \triangleq$$

```

let l = ref None in
handle f () with
| effect getKey Alice, k  $\Rightarrow$ 
  do send (0, Bob);
  let c = sample_or_read l in
  let key = exp g c in
  let r = do recv Bob in
  match r with
  | None  $\Rightarrow$  k None
  | Some _  $\Rightarrow$  k (Some key)
| effect getKey Bob, k  $\Rightarrow$ 
  let r = do recv Alice in
  match r with
  | None  $\Rightarrow$  k None
  | Some _  $\Rightarrow$ 
    do send (0, Alice);
    match !l with
    | None  $\Rightarrow$  k None
    | Some c  $\Rightarrow$ 
      let key = exp g c in
      k (Some key)

```

$$\mathcal{S}_{\text{DHKE}} f \triangleq$$

```

let la = ref None in
let lb = ref None in
handle f () with
| effect send (_, Alice), k  $\Rightarrow$ 
  match !la with
  | Some _  $\Rightarrow$  k ()
  | None  $\Rightarrow$ 
    let c = sample_or_read la in
    let C = exp g c in
    do send (C, Alice);
    k ()
| effect send (_, Bob), k  $\Rightarrow$ 
  match !lb with
  | Some _  $\Rightarrow$  k ()
  | None  $\Rightarrow$ 
    let c = sample_or_read lb in
    let C = exp g c in
    do send (C, Bob);
    k ()
| effect recv from, k  $\Rightarrow$ 
  let r = do recv from in k r

```

$$\mathcal{F}_{\text{KE}} f \triangleq$$

```

let l = ref None in
handle f () with
| effect getKey Alice, k  $\Rightarrow$ 
  do send (0, Bob);
  let c = sample_or_read l in
  let key = exp g c in
  let r = do recv Bob in
  match r with
  | None  $\Rightarrow$  k None
  | Some _  $\Rightarrow$  k (Some key)
| effect getKey Bob, k  $\Rightarrow$ 
  let r = do recv Alice in
  match r with
  | None  $\Rightarrow$  k None
  | Some _  $\Rightarrow$ 
    do send (0, Alice);
    match !l with
    | None  $\Rightarrow$  k None
    | Some c  $\Rightarrow$ 
      let key = exp g c in
      k (Some key)

```

$$\mathcal{S}_{\text{DHKE}} f \triangleq$$

```

let la = ref None in
let lb = ref None in
handle f () with
| effect send (_, Alice), k  $\Rightarrow$ 
  match !la with
  | Some _  $\Rightarrow$  k ()
  | None  $\Rightarrow$ 
    let c = sample_or_read la in
    let C = exp g c in
    do send (C, Alice);
    k ()
| effect send (_, Bob), k  $\Rightarrow$ 
  match !lb with
  | Some _  $\Rightarrow$  k ()
  | None  $\Rightarrow$ 
    let c = sample_or_read lb in
    let C = exp g c in
    do send (C, Bob);
    k ()
| effect recv from, k  $\Rightarrow$ 
  let r = do recv from in k r

```

DHKE using effects and handlers

We now have all the program components to state the DHKE security theorem.

Formally, we show that the **distinguishing advantage**

$$\text{Adv}(\mathcal{Z}, \mathcal{F}_{\text{AUTH}} \circ \pi_{\text{DHKE}}, \mathcal{S}_{\text{DHKE}} \circ \mathcal{F}_{\text{KE}})$$

is small for any computationally bounded $\mathcal{Z}$.

DHKE using effects and handlers

We now have all the program components to state the DHKE security theorem.

Formally, we show that the **distinguishing advantage**

$$\text{Adv}(\mathcal{Z}, \mathcal{F}_{\text{AUTH}} \circ \pi_{\text{DHKE}}, \mathcal{S}_{\text{DHKE}} \circ \mathcal{F}_{\text{KE}})$$

is small for any computationally bounded $\mathcal{Z}$.

$$\text{Adv}(\mathcal{Z}, e, e') \triangleq |\text{Pr}_{\text{exec}}(\mathcal{Z}(e))[\text{true}] - \text{Pr}_{\text{exec}}(\mathcal{Z}(e'))[\text{true}]|$$

Security of DHKE

Security of DHKE

We need to show that

$$\text{Adv}(\mathcal{Z}, \mathcal{R}_{\text{DHKE}}, \mathcal{I}_{\text{DHKE}})$$

is small. We do by a **reduction** to the DDH assumption:

Security of DHKE

We need to show that

$$\text{Adv}(\mathcal{Z}, \mathcal{R}_{\text{DHKE}}, \mathcal{I}_{\text{DHKE}})$$

is small. We do by a **reduction** to the DDH assumption:

Security of DHKE

We need to show that

$$\text{Adv}(\mathcal{Z}, \mathcal{R}_{\text{DHKE}}, \mathcal{I}_{\text{DHKE}})$$

is small. We do by a **reduction** to the DDH assumption:

```
 $DH_{\text{real}}() \triangleq$   
  let  $a = \text{rand } |G|$  in  
  let  $b = \text{rand } |G|$  in  
  ( $\text{exp } g \ a, \text{exp } g \ b, \text{exp } g \ (a * b)$ )
```

```
 $DH_{\text{ideal}}() \triangleq$   
  let  $a = \text{rand } |G|$  in  
  let  $b = \text{rand } |G|$  in  
  let  $c = \text{rand } |G|$  in  
  ( $\text{exp } g \ a, \text{exp } g \ b, \text{exp } g \ c$ )
```

Security of DHKE

To do so, it suffices to show

$$\mathcal{R}_{\text{DHKE}} \simeq_{\text{ctx}} C[DH_{\text{real}}] \quad \text{and} \quad \mathcal{I}_{\text{DHKE}} \simeq_{\text{ctx}} C[DH_{\text{ideal}}]$$

Security of DHKE

To do so, it suffices to show

$$\mathcal{R}_{\text{DHKE}} \simeq_{\text{ctx}} C[DH_{\text{real}}] \quad \text{and} \quad \mathcal{I}_{\text{DHKE}} \simeq_{\text{ctx}} C[DH_{\text{ideal}}]$$

To complete the proof:

- Assume there exists an adversary $\mathcal{A}$ that can distinguish our REAL/IDEAL.
- Using the two equations, $\mathcal{A}$ can then distinguish $C[DH_{\text{real}}]$ and $C[DH_{\text{ideal}}]$
- But that means $\mathcal{A} \circ C$ can distinguish the DDH games! (assuming $\mathcal{A} \circ C$ is PPT)

Reduction

```
Red DH f  $\triangleq$ 
  let (A, B, C) = DH () in
  let flag = ref false in
  handle f () with
  | effect getKey Alice, k  $\Rightarrow$ 
    flag := true;
    do send (A, Bob);
    let r = do recv Bob in
    match r with
    | None  $\Rightarrow$  k None
    | Some _  $\Rightarrow$  k (Some C)
    end
  | effect getKey Bob, k  $\Rightarrow$ 
    let r = do recv Alice in
    match r with
    | None  $\Rightarrow$  k None
    | Some _  $\Rightarrow$ 
      do send (B, Alice);
      if !flag then k (Some C)
      else k None
```

Reduction

```
Red DH f  $\triangleq$ 
  let (A, B, C) = DH () in
  let flag = ref false in
  handle f () with
  | effect getKey Alice, k  $\Rightarrow$ 
    flag := true;
    do send (A, Bob);
    let r = do recv Bob in
    match r with
    | None  $\Rightarrow$  k None
    | Some _  $\Rightarrow$  k (Some C)
    end
  | effect getKey Bob, k  $\Rightarrow$ 
    let r = do recv Alice in
    match r with
    | None  $\Rightarrow$  k None
    | Some _  $\Rightarrow$ 
      do send (B, Alice);
      if !flag then k (Some C)
      else k None
```

“the hole”



Key takeaways

- Effect handlers allow us to model UC-style functionalities programmatically.
- We turn protocol security into a question of (approximate) equivalence of well-understood programming constructs.
- The (adversary-controlled) network is “just” an unknown program!

We know how to reason about all that stuff!

Plan

1. Define **RandMLEff**, an ML-like language with effect and handlers, and a probabilistic uniform sampling construct.
2. Develop a **probabilistic relational separation logic** for reasoning about RandMLEff programs.

To reason about “unknown” code:

3. Equip RandMLEff with a **type-and-effect system**.
4. Define a **logical-relations model** of the RandMLEff type systems and show the fundamental theorem of logical relations.

The RandMLEff language

The RandMLEff language

To define the semantics of RandMLEff, we define a single-step distribution, eg

$$\text{step}(\text{rand } N, \sigma)(n, \sigma) \triangleq \begin{cases} \frac{1}{N+1} & \text{for } n \in \{0, 1, \dots, N\}, \\ 0 & \text{otherwise.} \end{cases}$$

The RandMLEff language

To define the semantics of RandMLEff, we define a single-step distribution, eg

$$\text{step}(\text{rand } N, \sigma)(n, \sigma) \triangleq \begin{cases} \frac{1}{N+1} & \text{for } n \in \{0, 1, \dots, N\}, \\ 0 & \text{otherwise.} \end{cases}$$

Next, we define n steps of execution and take the limit.

$$\text{exec}_0(e, \sigma)(v) \triangleq \begin{cases} 1 & \text{if } e \in \text{Val} \wedge e = v, \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{exec}_{m+1}(e, \sigma)(v) \triangleq \begin{cases} 1 & \text{if } e \in \text{Val} \text{ and } e = v, \\ \sum_{c \in \text{Cfg}} \text{step}(e, \sigma)(c) \cdot \text{exec}_m(c)(v) & \text{otherwise.} \end{cases}$$

$$\text{exec}(c)(v) \triangleq \lim_{n \rightarrow \infty} \text{exec}_n(c)(v)$$

ClutchEff

To reason about RandMLEff programs, we define **ClutchEff**, a probabilistic relational separation logic.

$$\begin{aligned} P, Q \in iProp ::= & \text{True} \mid \text{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \forall x. P \mid \exists x. P \mid P * Q \mid \\ & P \multimap Q \mid \boxed{P} \mid \ell \mapsto v \mid \ell \mapsto_s v \mid \kappa \hookrightarrow (N, \vec{n}) \mid \kappa \hookrightarrow_s (N, \vec{n}) \mid \\ & \text{!}(\varepsilon) \mid e_1 \lesssim e_2 \langle \mathcal{L} \rangle \{v_1, v_2. P\} \mid \dots \end{aligned}$$

ClutchEff

To reason about RandMLEff programs, we define **ClutchEff**, a probabilistic relational separation logic.

invariants

$$\begin{aligned} P, Q \in iProp ::= & \text{True} \mid \text{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \forall x. P \mid \exists x. P \mid P * Q \mid \\ & P \multimap Q \mid \boxed{P} \mid \ell \mapsto v \mid \ell \mapsto_s v \mid \kappa \hookrightarrow (N, \vec{n}) \mid \kappa \hookrightarrow_s (N, \vec{n}) \mid \\ & \text{!}(\varepsilon) \mid e_1 \lesssim e_2 \langle \mathcal{L} \rangle \{v_1, v_2. P\} \mid \dots \end{aligned}$$

ClutchEff

To reason about RandMLEff programs, we define **ClutchEff**, a probabilistic relational separation logic.

$$\begin{array}{c} \text{invariants} \quad \text{LHS points-to} \\ P, Q \in iProp ::= \text{True} \mid \text{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \forall x. P \mid \exists x. P \mid P * Q \mid \\ P \multimap Q \mid \boxed{P} \mid \ell \mapsto v \mid \ell \mapsto_s v \mid \kappa \hookrightarrow (N, \vec{n}) \mid \kappa \hookrightarrow_s (N, \vec{n}) \mid \\ \text{!}(\varepsilon) \mid e_1 \lesssim e_2 \langle \mathcal{L} \rangle \{v_1, v_2. P\} \mid \dots \end{array}$$

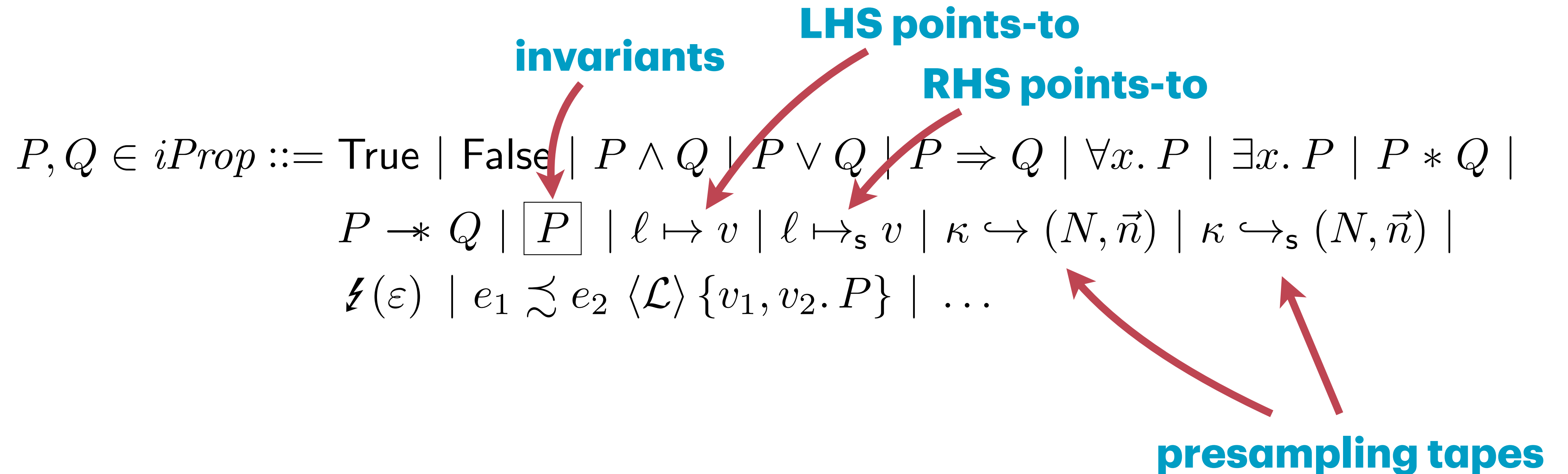
ClutchEff

To reason about RandMLEff programs, we define **ClutchEff**, a probabilistic relational separation logic.

$$\begin{array}{c} \text{invariants} \quad \text{LHS points-to} \quad \text{RHS points-to} \\ P, Q \in iProp ::= \text{True} \mid \text{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \forall x. P \mid \exists x. P \mid P * Q \mid \\ P \multimap Q \mid \boxed{P} \mid \ell \mapsto v \mid \ell \mapsto_s v \mid \kappa \hookrightarrow (N, \vec{n}) \mid \kappa \hookrightarrow_s (N, \vec{n}) \mid \\ \text{!}(\varepsilon) \mid e_1 \lesssim e_2 \mid \langle \mathcal{L} \rangle \{v_1, v_2. P\} \mid \dots \end{array}$$

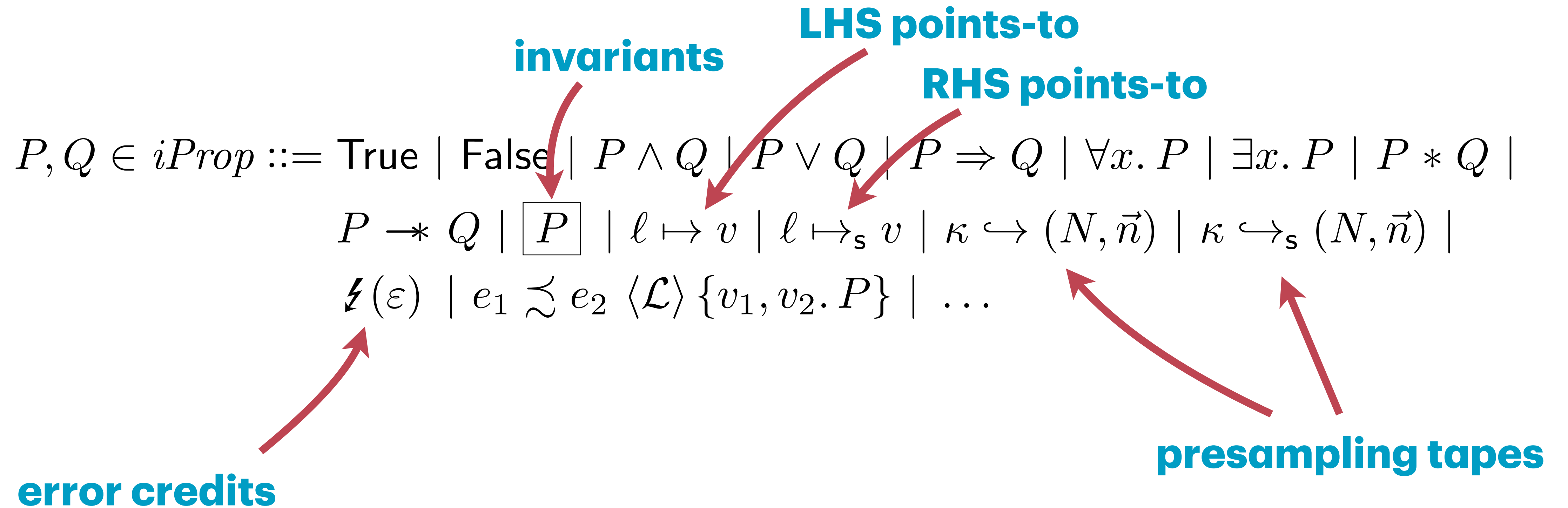
ClutchEff

To reason about RandMLEff programs, we define **ClutchEff**, a probabilistic relational separation logic.



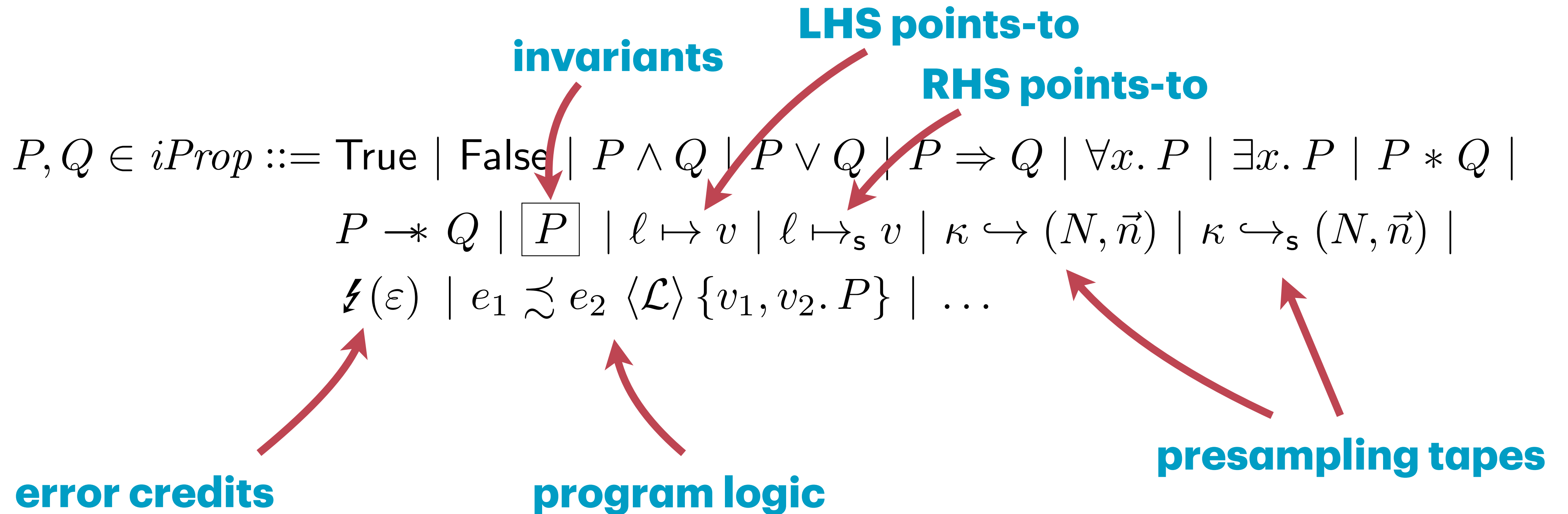
ClutchEff

To reason about RandMLEff programs, we define **ClutchEff**, a probabilistic relational separation logic.



ClutchEff

To reason about RandMLEff programs, we define **ClutchEff**, a probabilistic relational separation logic.



ClutchEff (cont'd)

Theorem (Adequacy)

Let $\varphi \subseteq Val \times Val$ and $\varepsilon \in [0, 1]$. If

$$\text{⚡}(\varepsilon) \vdash e_1 \lesssim e_2 \langle \perp \rangle \{ \varphi \}$$

then $\text{exec}(e_1, \sigma_1) \lesssim_\varepsilon \text{exec}(e_2, \sigma_2) : \varphi$ for all σ_1 and σ_2 .

Equivalence implies **the total variation distance is bounded by ε** if $\varphi \triangleq (=)$

ClutchEff (cont'd)

The relational program logic

$$e_1 \lesssim e_2 \langle \mathcal{L} \rangle \{ \Phi \}$$

consists of three kinds of reasoning rules:

1. rules for relating **higher-order imperative programs** (ReLoC),
2. rules for relating **effects and handlers** (blaze [de Vilhena et al. POPL'26]),
3. rules for relating **probabilistic sampling** (Clutch/Approxis).

ClutchEff - effects and handlers (blaze)

Intuitively,

$$e_1 \lesssim e_2 \langle \mathcal{L} \rangle \{ \Phi \}$$

says that e_1 and e_2 can be plugged into all contexts K_1 and K_2 that **validate** the relational theory $\mathcal{L}$ such that

$$K_1[e_1] \lesssim K_2[e_2] \langle \perp \rangle \{ \Phi \}$$

Morally, $\mathcal{L}$ is a set of “assumed relational specifications”.

ClutchEff - probabilistic sampling (Approxis)

$$\frac{\text{REL-COUPLE} \quad f \text{ bijection} \quad \forall n \leq N. n \precsim f(n) \langle \mathcal{L} \rangle \{\Phi\}}{\text{rand } N \precsim \text{rand } N \langle \mathcal{L} \rangle \{\Phi\}}$$

Including the corresponding presampling rules.

ClutchEff - probabilistic sampling (Approxis)

$$\frac{\text{REL-COUPLE} \quad f \text{ bijection} \quad \forall n \leq N. n \lesssim f(n) \langle \mathcal{L} \rangle \{\Phi\}}{\text{rand } N \lesssim \text{rand } N \langle \mathcal{L} \rangle \{\Phi\}}$$

$$\frac{\text{REL-COUPLE-AVOID} \quad f \text{ bijection} \quad \text{⚡} \left(\frac{\text{size}(X)}{N+1} \right) \quad \forall n \leq N. n \notin X \multimap n \lesssim f(n) \langle \mathcal{L} \rangle \{\Phi\}}{\text{rand } N \lesssim \text{rand } N \langle \mathcal{L} \rangle \{\Phi\}}$$

Including the corresponding presampling rules.

Battle plan

1. Define **RandMLEff**, an OCaml-like language with effect and handlers, and a probabilistic uniform sampling construct.
2. Develop a **probabilistic relational separation logic** for reasoning about RandMLEff programs.

To reason about “unknown” code:

3. Equip RandMLEff with a **type-and-effect system**.
4. Define a **logical-relations model** of the RandMLEff type systems and show the fundamental theorem of logical relations.

The Affect type-and-effect system

We equip RandMLEff with the **Affect** type system [van Rooij & Krebbers, POPL'25]

$$\begin{aligned} m \in Mode &::= \nu \mid \text{os} \mid \text{ms} \\ \tau \in Type &::= \alpha \mid \perp \mid \top \mid 1 \mid \mathbf{B} \mid \mathbf{N} \mid \mathbf{Z} \mid \tau \times \tau \mid \tau + \tau \mid \\ &\quad \tau \xrightarrow{\rho} \tau \mid \forall \alpha. \tau \mid \forall \theta. \tau \mid \forall \nu. \tau \mid \exists \alpha. \tau \mid \mu \alpha. \tau \mid \text{ref } \tau \mid !_m \tau \\ \sigma \in Signature &::= \text{op} : \forall \alpha. \tau \Rightarrow \tau \mid i_m \sigma \\ \rho \in Row &::= \theta \mid \langle \rangle \mid \sigma \cdot \rho \mid \rho_1 \cdot \rho_2 \mid i_m \rho \end{aligned}$$

The Affect type-and-effect system

We equip RandMLEff with the **Affect** type system [van Rooij & Krebbers, POPL'25]

affine function types, annotated with a row

$$\begin{aligned} m &\in Mode ::= \nu \mid os \mid ms \\ \tau &\in Type ::= \alpha \mid \perp \mid \top \mid \mathbf{1} \mid B \mid N \mid Z \mid \tau \times \tau \mid \tau + \tau \mid \\ &\quad \tau \xrightarrow{\rho} \tau \mid \forall \alpha. \tau \mid \forall \theta. \tau \mid \forall \nu. \tau \mid \exists \alpha. \tau \mid \mu \alpha. \tau \mid \text{ref } \tau \mid !_m \tau \\ \sigma &\in Signature ::= \text{op} : \forall \alpha. \tau \Rightarrow \tau \mid i_m \sigma \\ \rho &\in Row ::= \theta \mid \langle \rangle \mid \sigma \cdot \rho \mid \rho_1 \cdot \rho_2 \mid i_m \rho \end{aligned}$$

The Affect type-and-effect system

We equip RandMLEff with the **Affect** type system [van Rooij & Krebbers, POPL'25]

affine function types, annotated with a row

$m \in Mode ::= \nu \mid os \mid ms$
 $\tau \in Type ::= \alpha \mid \perp \mid \top \mid \mathbf{1} \mid \mathbf{B} \mid \mathbf{N} \mid \mathbf{Z} \mid \tau \times \tau \mid \tau + \tau \mid$
 $\tau \xrightarrow{\rho} \tau \mid \forall \alpha. \tau \mid \forall \theta. \tau \mid \forall \nu. \tau \mid \exists \alpha. \tau \mid \mu \alpha. \tau \mid \text{ref } \tau \mid !_m \tau$
 $\sigma \in Signature ::= \text{op} : \forall \alpha. \tau \Rightarrow \tau \mid i_m \sigma$
 $\rho \in Row ::= \theta \mid \langle \rangle \mid \sigma \cdot \rho \mid \rho_1 \cdot \rho_2 \mid i_m \rho$

“flip-bang” operator

The Affect type-and-effect system

We equip RandMLEff with the **Affect** type system [van Rooij & Krebbers, POPL'25]

affine function types, annotated with a row

$$\begin{aligned} m &\in Mode ::= \nu \mid os \mid ms \\ \tau &\in Type ::= \alpha \mid \perp \mid \top \mid \mathbf{1} \mid \mathbf{B} \mid \mathbf{N} \mid \mathbf{Z} \mid \tau \times \tau \mid \tau + \tau \mid \\ &\quad \tau \xrightarrow{\rho} \tau \mid \forall \alpha. \tau \mid \forall \theta. \tau \mid \forall \nu. \tau \mid \exists \alpha. \tau \mid \mu \alpha. \tau \mid \text{ref } \tau \mid !_m \tau \\ \sigma &\in Signature ::= \text{op} : \forall \alpha. \tau \Rightarrow \tau \mid i_m \sigma \\ \rho &\in Row ::= \theta \mid \langle \rangle \mid \sigma \cdot \rho \mid \rho_1 \cdot \rho_2 \mid i_m \rho \end{aligned}$$

“flip-bang” operator

“lists of effect signatures”

The Affect type-and-effect system

We equip RandMLEff with the **Affect** type system [van Rooij & Krebbers, POPL'25]

affine function types, annotated with a row

$m \in Mode ::= \nu \mid os \mid ms$

$\tau \in Type ::= \alpha \mid \perp \mid \top \mid 1 \mid B \mid N \mid Z \mid \tau \times \tau \mid \tau + \tau \mid$

$\tau \xrightarrow{\rho} \tau \mid \forall \alpha. \tau \mid \forall \theta. \tau \mid \forall \nu. \tau \mid \exists \alpha. \tau \mid \mu \alpha. \tau \mid \text{ref } \tau \mid !_m \tau$

$\sigma \in Signature ::= \text{op} : \forall \alpha. \tau \Rightarrow \tau \mid i_m \sigma$

$\rho \in Row ::= \theta \mid \langle \rangle \mid \sigma \cdot \rho \mid \rho_1 \cdot \rho_2 \mid i_m \rho$

“flip-bang” operator

effect signature

“lists of effect signatures”

The Affect type-and-effect system

We equip RandMLEff with the **Affect** type system [van Rooij & Krebbers, POPL'25]

affine function types, annotated with a row

$m \in Mode ::= \nu \mid os \mid ms$

$\tau \in Type ::= \alpha \mid \perp \mid \top \mid \mathbf{1} \mid \mathbf{B} \mid \mathbf{N} \mid \mathbf{Z} \mid \tau \times \tau \mid \tau + \tau \mid$

$\tau \xrightarrow{\rho} \tau \mid \forall \alpha. \tau \mid \forall \theta. \tau \mid \forall \nu. \tau \mid \exists \alpha. \tau \mid \mu \alpha. \tau \mid \text{ref } \tau \mid !_m \tau$

$\sigma \in Signature ::= \text{op} : \forall \alpha. \tau \Rightarrow \tau \mid i_m \sigma$

$\rho \in Row ::= \theta \mid \langle \rangle \mid \sigma \cdot \rho \mid \rho_1 \cdot \rho_2 \mid i_m \rho$

“flip-bang” operator

effect signature

“lists of effect signatures”

type, row, and mode polymorphism

A binary logical-relations model of Affect

A binary logical-relations model of Affect

1. Define a **binary interpretation** $\llbracket \tau \rrbracket$ using ClutchEff.

A binary logical-relations model of Affect

1. Define a **binary interpretation** $\llbracket \tau \rrbracket$ using ClutchEff.
2. Define **a logical refinement judgment** for open terms by quantifying over closing substitutions as usual.

A binary logical-relations model of Affect

1. Define a **binary interpretation** $\llbracket \tau \rrbracket$ using ClutchEff.
2. Define **a logical refinement judgment** for open terms by quantifying over closing substitutions as usual.
3. Show that all the typing rules are **compatible** with logical refinement.

A binary logical-relations model of Affect

1. Define a **binary interpretation** $\llbracket \tau \rrbracket$ using ClutchEff.
2. Define **a logical refinement judgment** for open terms by quantifying over closing substitutions as usual.
3. Show that all the typing rules are **compatible** with logical refinement.
4. Show the **fundamental theorem**: intuitively, if $\Gamma \vdash e : \tau$ then $\Gamma \models e \lesssim e : \tau$

Recap

- Effect handlers allow us to model UC-style functionalities programmatically.
- We turn protocol security into approximate equivalence of programs.
- We develop a probabilistic relational separation logic to reason about **known code**; we use a logical relation to reason about **unknown code**.
- Security is **compositional**: we also show how we can combine the DHKE with a secure channel, where the proof only relies on the ideal KE functionality.

1-out-of-2 oblivious transfer

1-out-of-2 oblivious transfer

Alice picks b and receives either m_0 or m_1 chosen by Bob.

1-out-of-2 oblivious transfer

Alice picks b and receives either m_0 or m_1 chosen by Bob.

Alice will only learn m_b and Bob will never learn b
— even when Alice or Bob are corrupted and act **maliciously**!

1-out-of-2 oblivious transfer

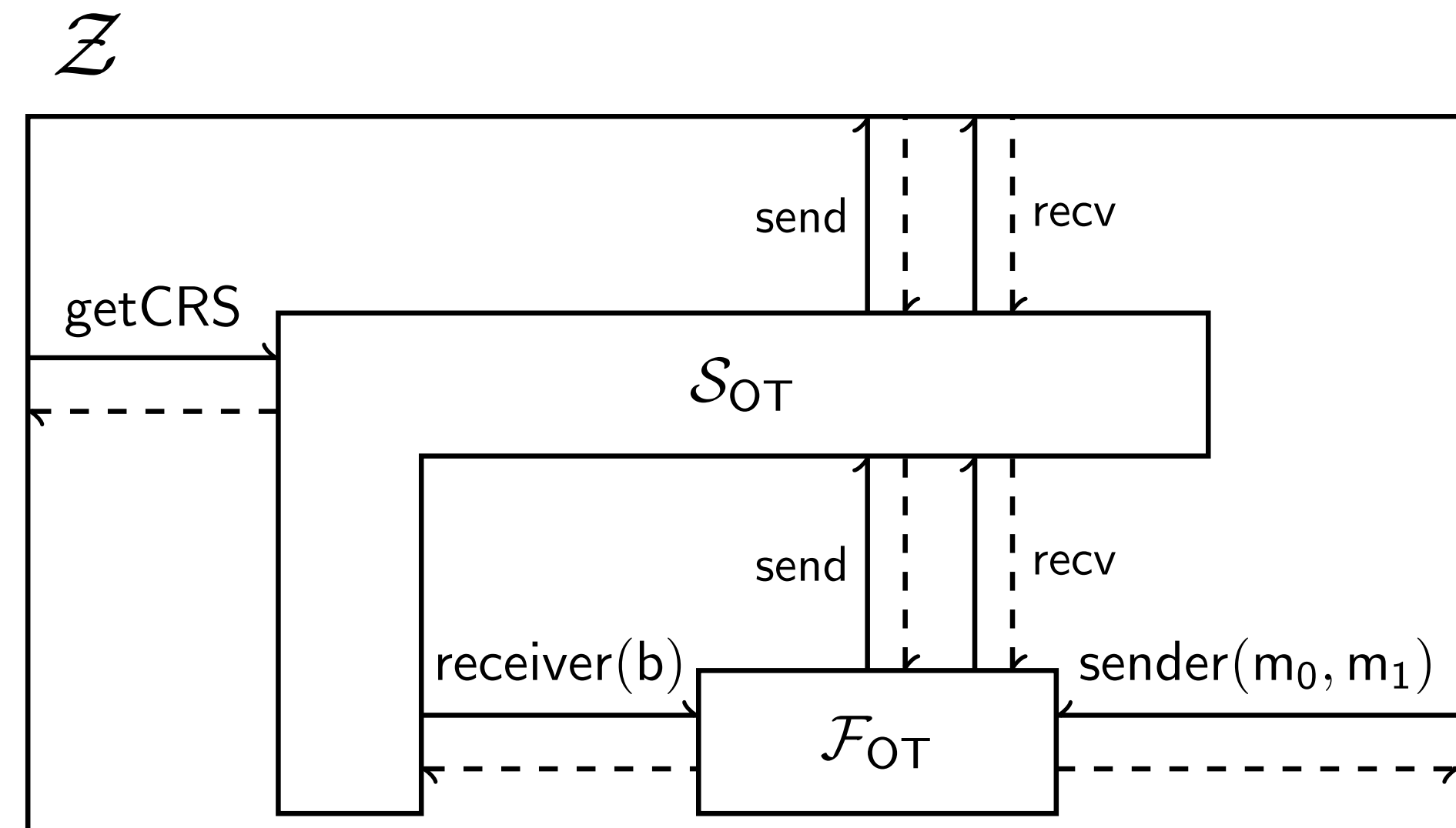
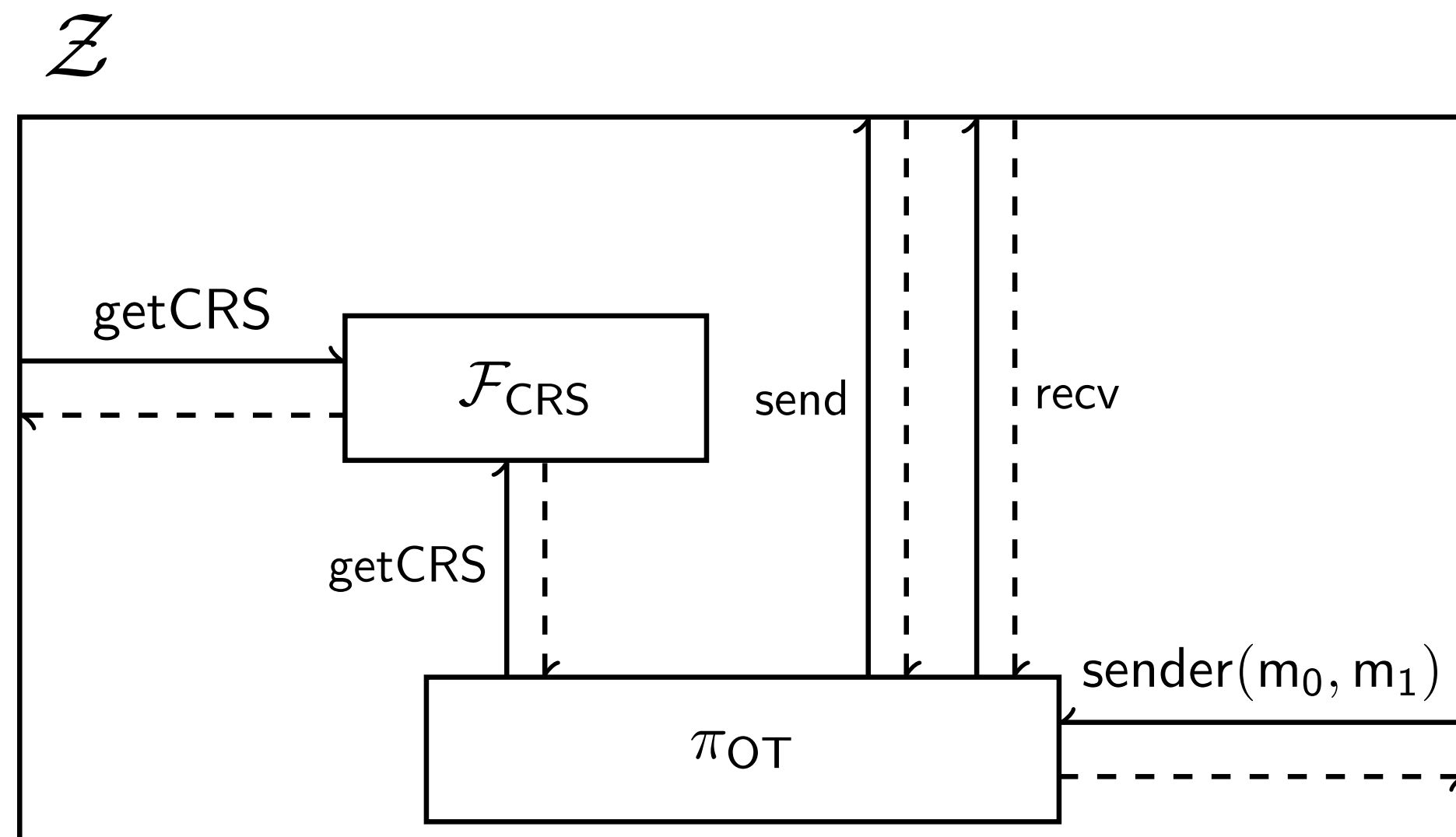
Alice picks b and receives either m_0 or m_1 chosen by Bob.

Alice will only learn m_b and Bob will never learn b
— even when Alice or Bob are corrupted and act **maliciously**!

We implement and verify the PVW OT protocol and show

$$\text{Adv}(\mathcal{Z}, \mathcal{R}_{\text{OT}}, \mathcal{I}_{\text{OT}}) \leq \frac{3}{|G|}.$$

1-out-of-2 oblivious transfer



Future work

- **Reasoning about running time** (PPT) is currently out-of-scope.
— however, this is also the case for proofs in, e.g., EasyCrypt.
- **Composition happens at the meta level**; can we internalize computational hardness assumptions? May require a complete and transitive LR...
- **Larger and more substantial case studies**, in particular a “closed” protocol.