



Assume but Verify: Deductive Verification of Leaked Information in Concurrent Applications

Toby Murray, Mukesh Tiwari, Gidon Ernst and David A. Naumann

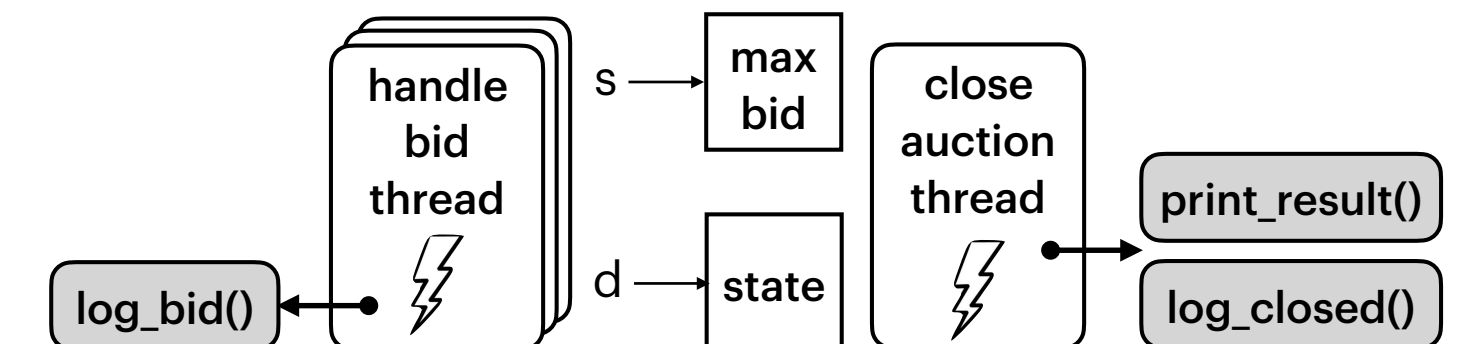
ACM CCS 2023, November 28, 2023

tl;dr

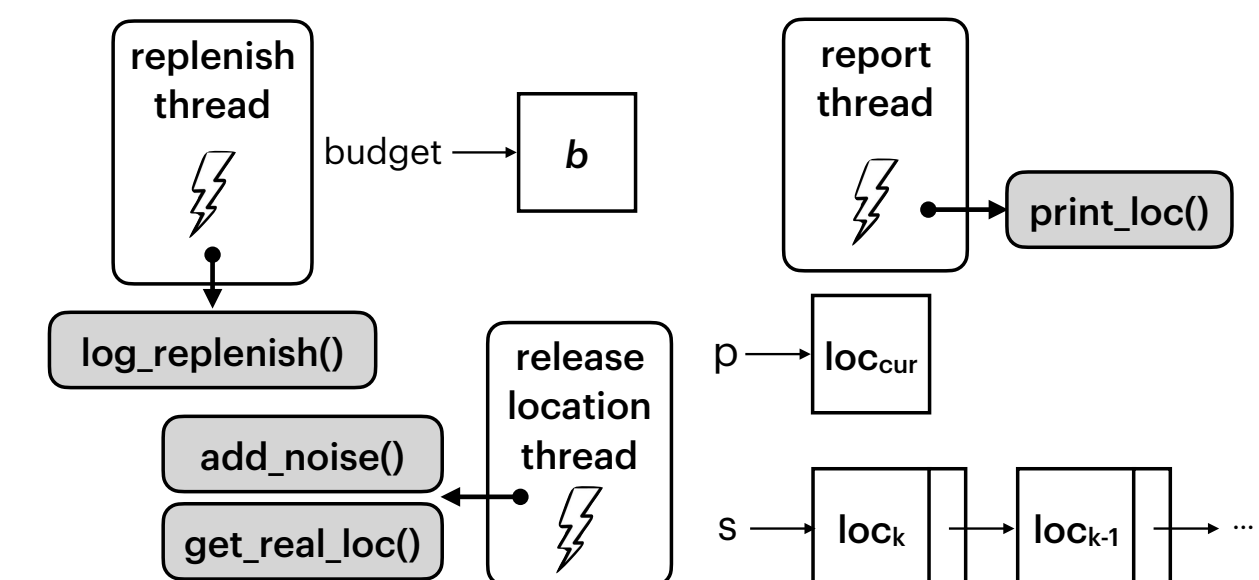
Auto-Active Verification for Concurrent C Programs against Declarative Dynamic Declassification Security Policies via Security Concurrent Separation Logic reasoning against a Constant Time threat model

Case Study	Proof Ratio	Verified SLOC	Unverified SLOC	Effort (pw)
Location Service	1.9	210	124	3
Auction Server	4.3	187	79	3.5
Wordle	5.9	47	81	0.3
Private Learning	2.5	315	114	6

Sealed Bid Auction Server



Private Location Service

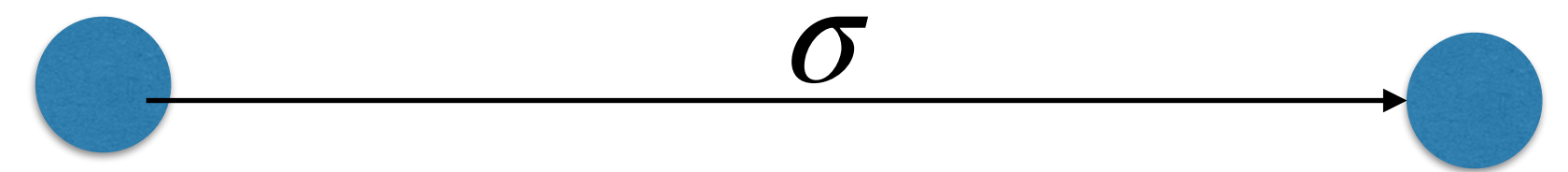


WORDLE
Server

Private Learning

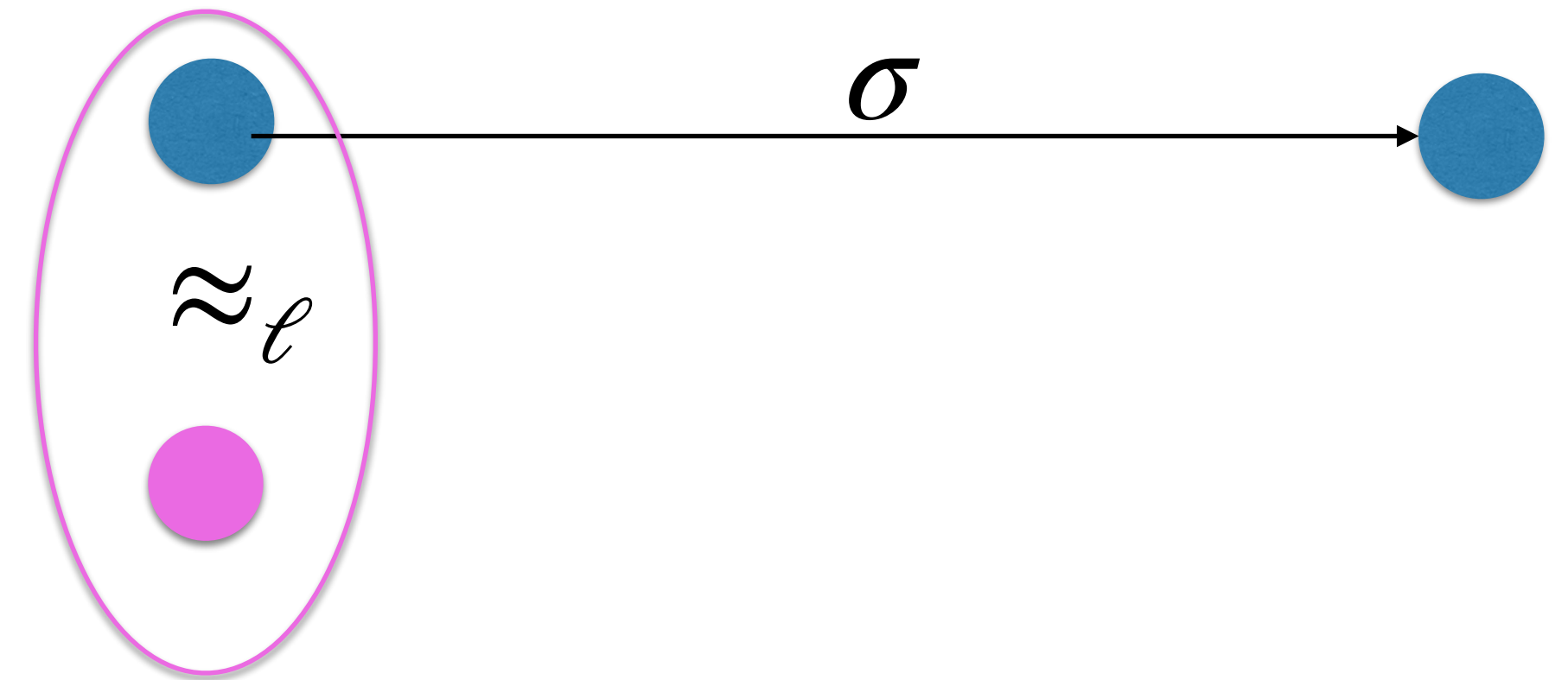
Auto-Active Reasoning about Noninterference

```
int check_pin(int pin, int guess)
// security policy: is allowed to
// reveal (only) whether pin == guess
_(requires (pin == guess) :: low)
// attacker knows their own guess
_(requires guess :: low)
{
    if (pin == guess) {
        return 0; // match
    } else if (guess > pin) {
        return -1; // guess is too high
    } else {
        return -2; // guess is too low
    }
}
```



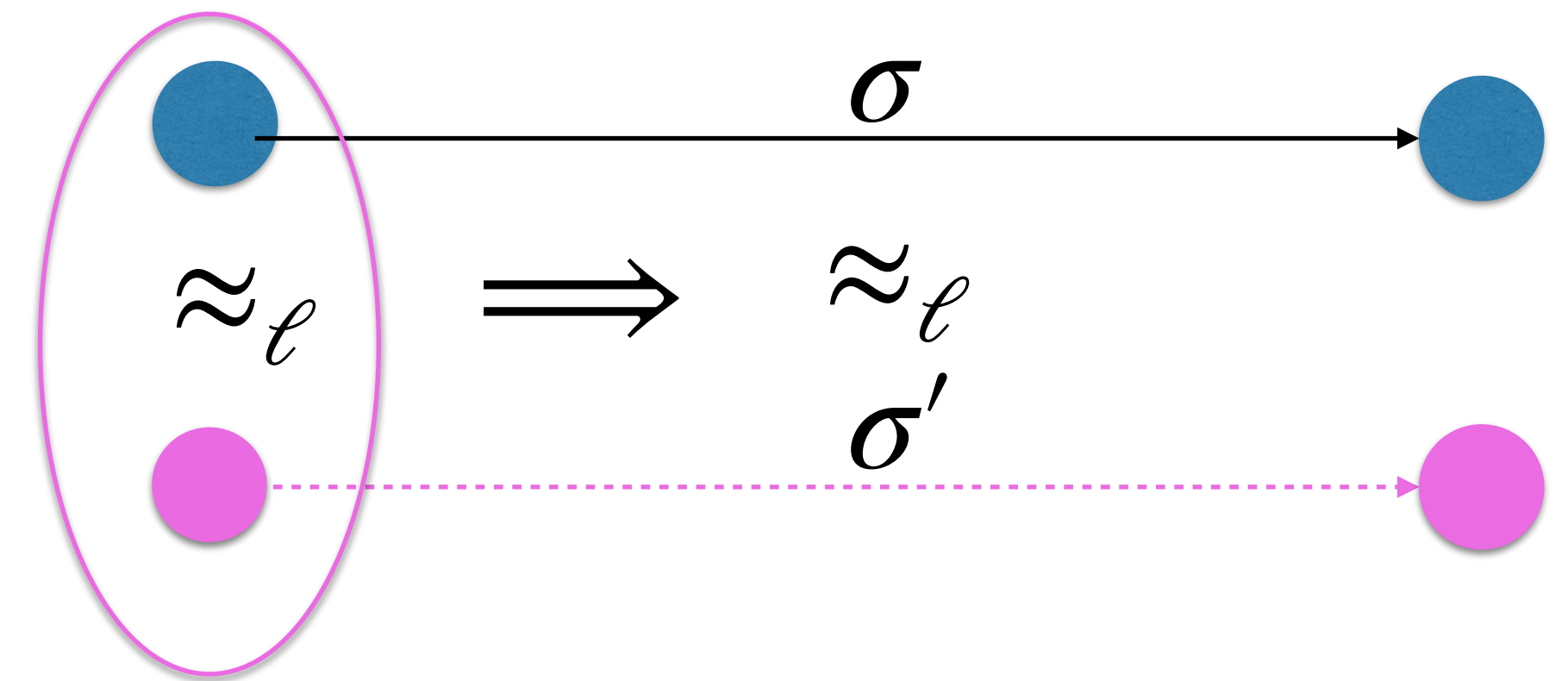
Auto-Active Reasoning about Noninterference

```
int check_pin(int pin, int guess)
// security policy: is allowed to
// reveal (only) whether pin == guess
_(requires (pin == guess) :: low)
// attacker knows their own guess
_(requires guess :: low)
{
    if (pin == guess) {
        return 0; // match
    } else if (guess > pin) {
        return -1; // guess is too high
    } else {
        return -2; // guess is too low
    }
}
```



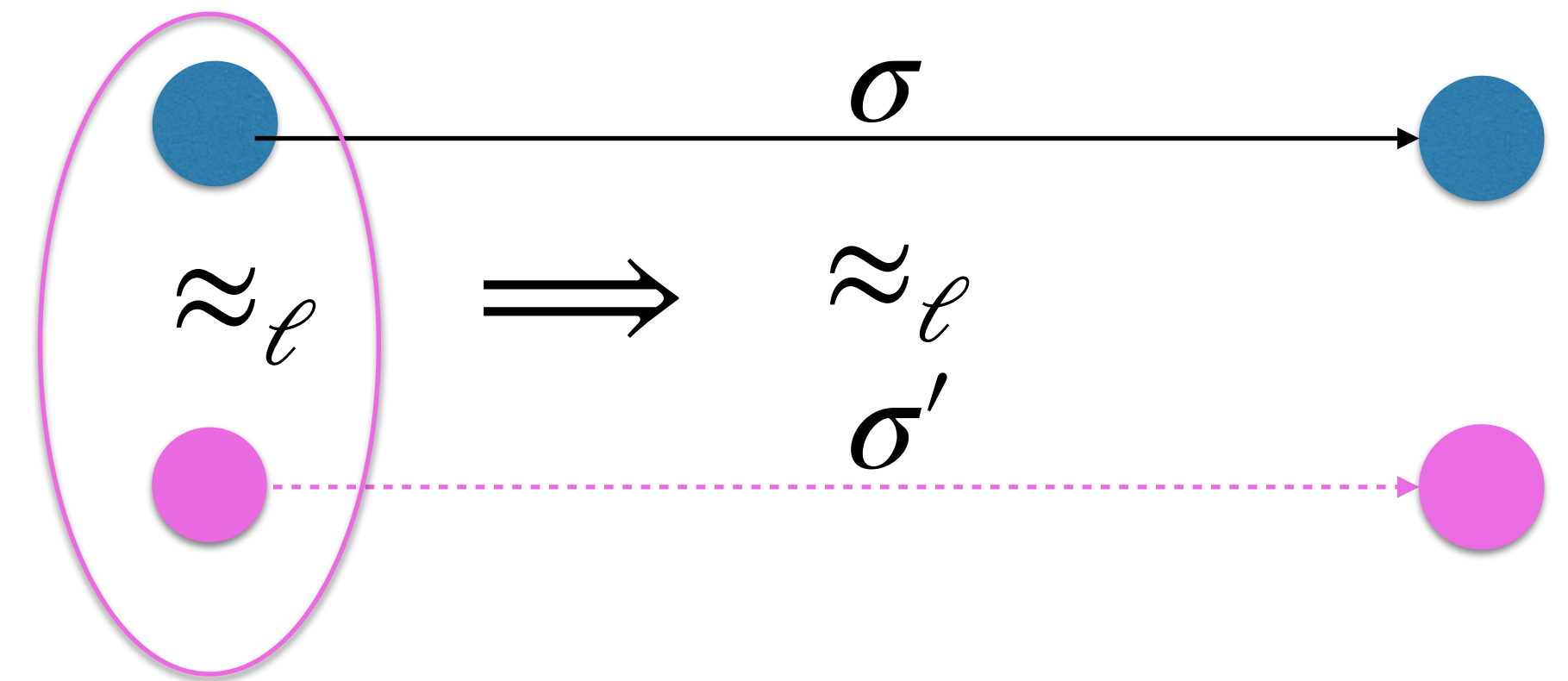
Auto-Active Reasoning about Noninterference

```
int check_pin(int pin, int guess)
// security policy: is allowed to
// reveal (only) whether pin == guess
_(requires (pin == guess) :: low)
// attacker knows their own guess
_(requires guess :: low)
{
    if (pin == guess) {
        return 0; // match
    } else if (guess > pin) {
        return -1; // guess is too high
    } else {
        return -2; // guess is too low
    }
}
```



Auto-Active Reasoning about Noninterference

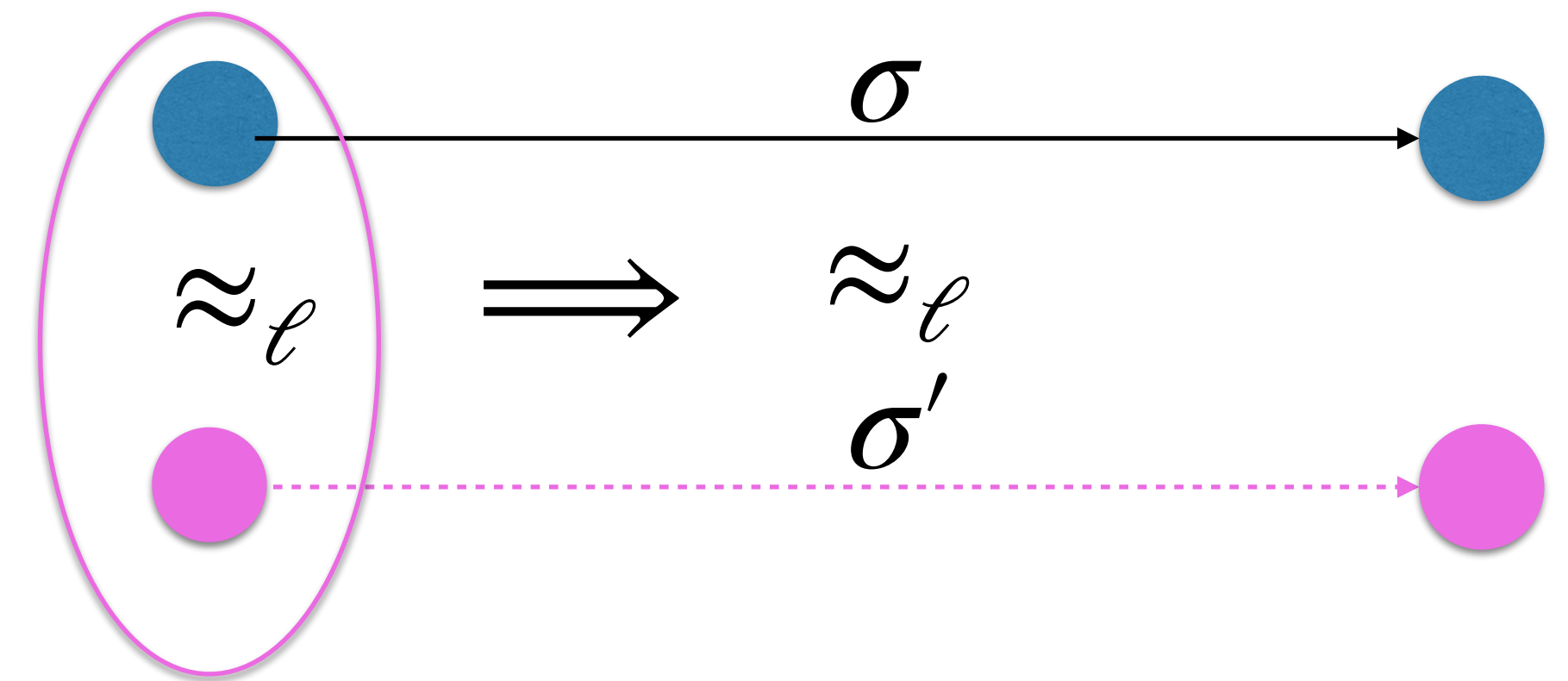
```
int check_pin(int pin, int guess)
// security policy: is allowed to
// reveal (only) whether pin == guess
_(requires (pin == guess) :: low)
// attacker knows their own guess
_(requires guess :: low)
{
    if (pin == guess) {
        return 0; // match
    } else if (guess > pin) {
        return -1; // guess is too high
    } else {
        return -2; // guess is too low
    }
}
```



$\langle p = 234, g = 100 \rangle$

Auto-Active Reasoning about Noninterference

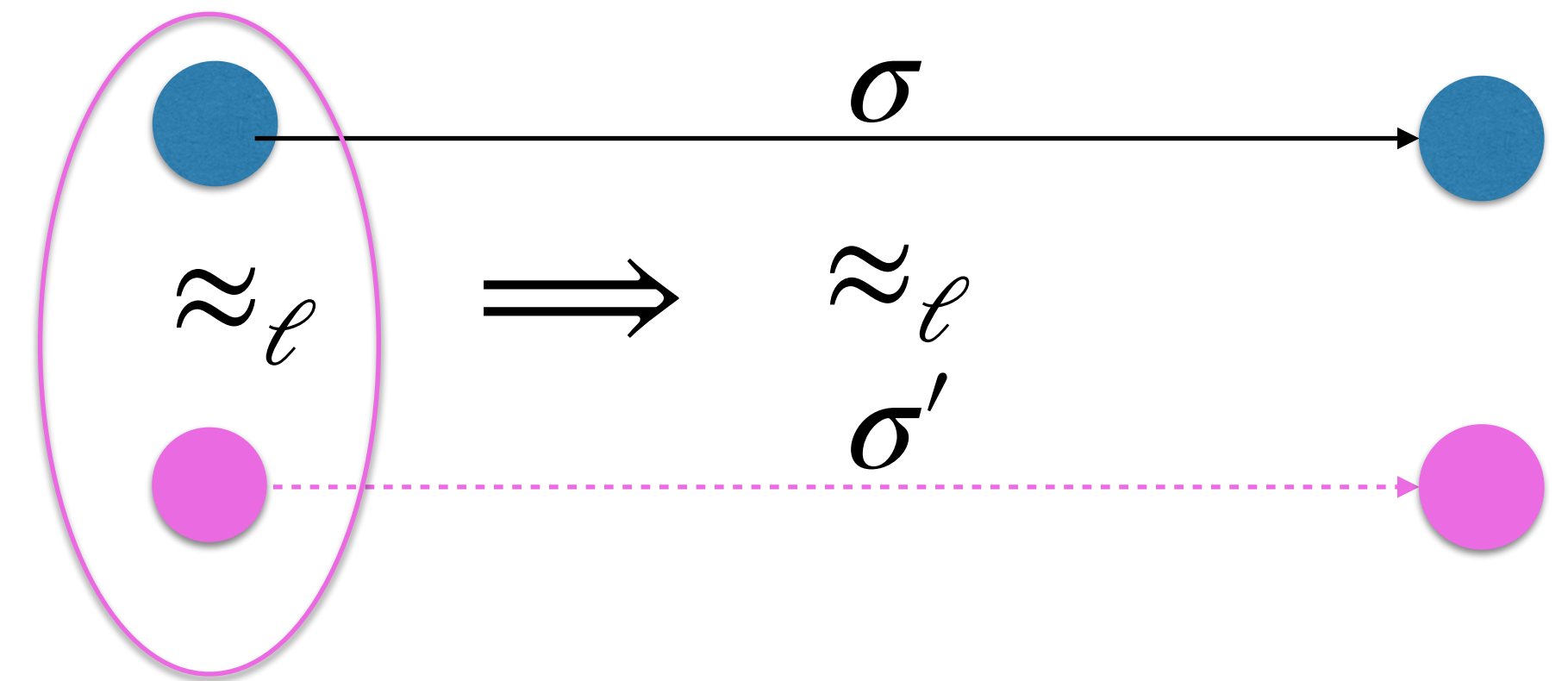
```
int check_pin(int pin, int guess)
// security policy: is allowed to
// reveal (only) whether pin == guess
_(requires (pin == guess) :: low)
// attacker knows their own guess
_(requires guess :: low)
{
    if (pin == guess) {
        return 0; // match
    } else if (guess > pin) {
        return -1; // guess is too high
    } else {
        return -2; // guess is too low
    }
}
```



$\langle p = 234, g = 100 \rangle \longrightarrow \text{return } -2;$

Auto-Active Reasoning about Noninterference

```
int check_pin(int pin, int guess)
// security policy: is allowed to
// reveal (only) whether pin == guess
_(requires (pin == guess) :: low)
// attacker knows their own guess
_(requires guess :: low)
{
    if (pin == guess) {
        return 0; // match
    } else if (guess > pin) {
        return -1; // guess is too high
    } else {
        return -2; // guess is too low
    }
}
```

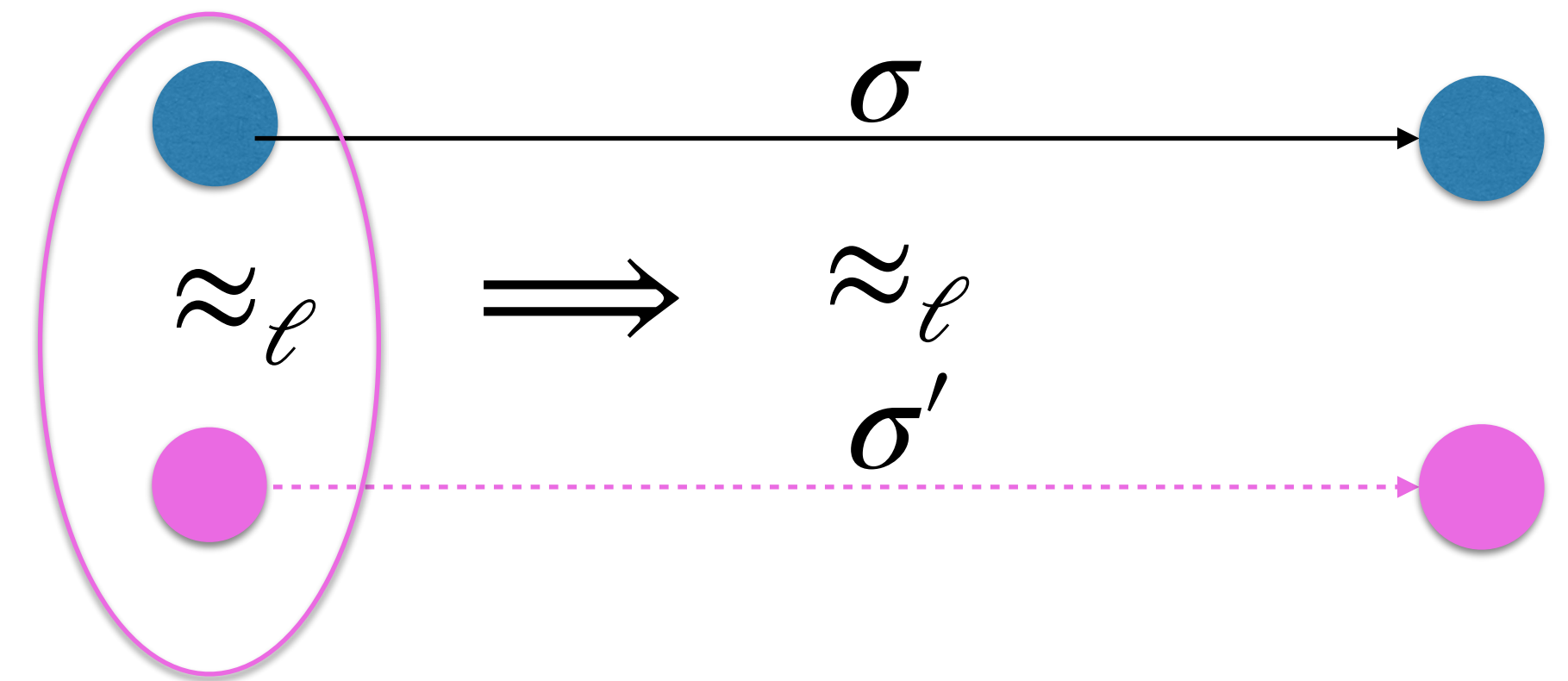


$\langle p = 234, g = 100 \rangle \longrightarrow \text{return } -2;$

$\langle p = 23, g = 100 \rangle$

Auto-Active Reasoning about Noninterference

```
int check_pin(int pin, int guess)
// security policy: is allowed to
// reveal (only) whether pin == guess
_(requires (pin == guess) :: low)
// attacker knows their own guess
_(requires guess :: low)
{
    if (pin == guess) {
        return 0; // match
    } else if (guess > pin) {
        return -1; // guess is too high
    } else {
        return -2; // guess is too low
    }
}
```



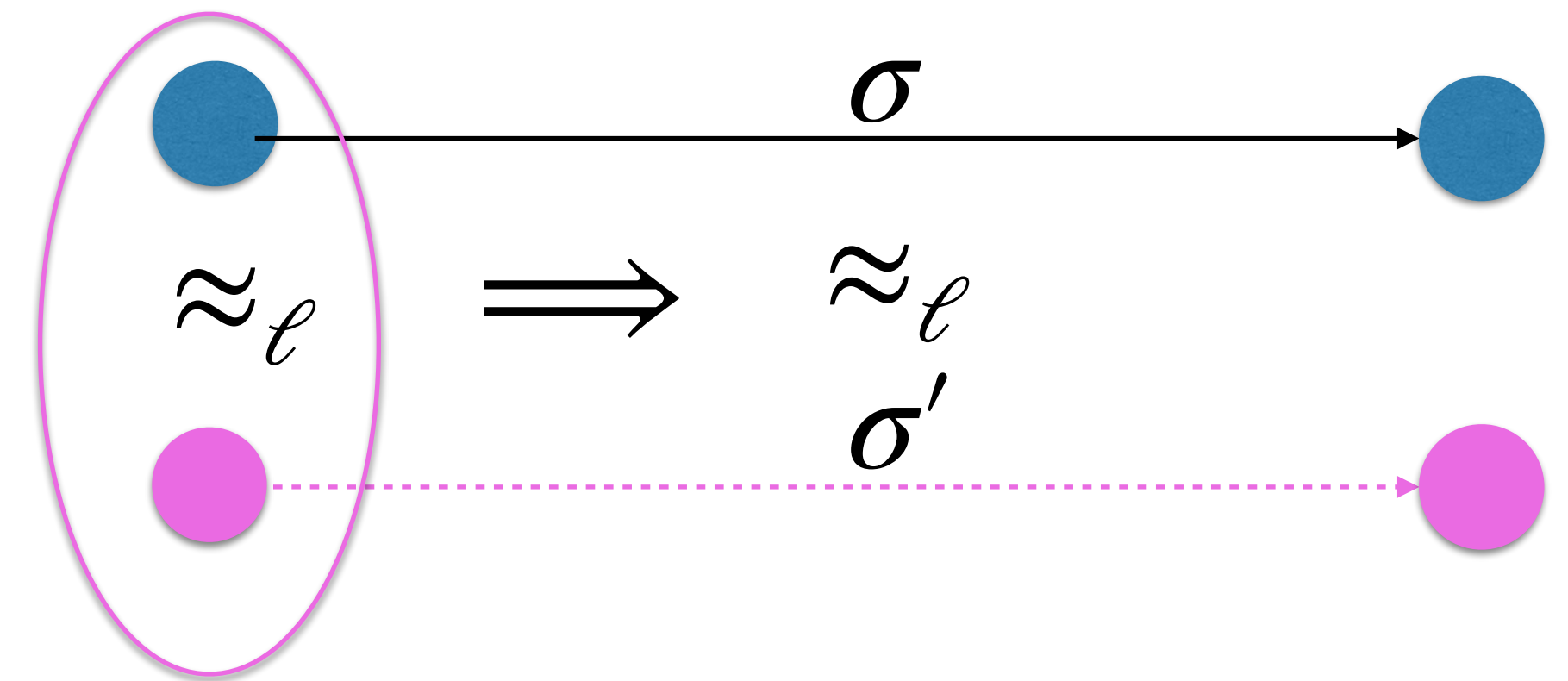
$\langle p = 234, g = 100 \rangle \longrightarrow \boxed{\text{return } -2;}$

\approx_ℓ

$\langle p = 23, g = 100 \rangle$

Auto-Active Reasoning about Noninterference

```
int check_pin(int pin, int guess)
// security policy: is allowed to
// reveal (only) whether pin == guess
_(requires (pin == guess) :: low)
// attacker knows their own guess
_(requires guess :: low)
{
    if (pin == guess) {
        return 0; // match
    } else if (guess > pin) {
        return -1; // guess is too high
    } else {
        return -2; // guess is too low
    }
}
```



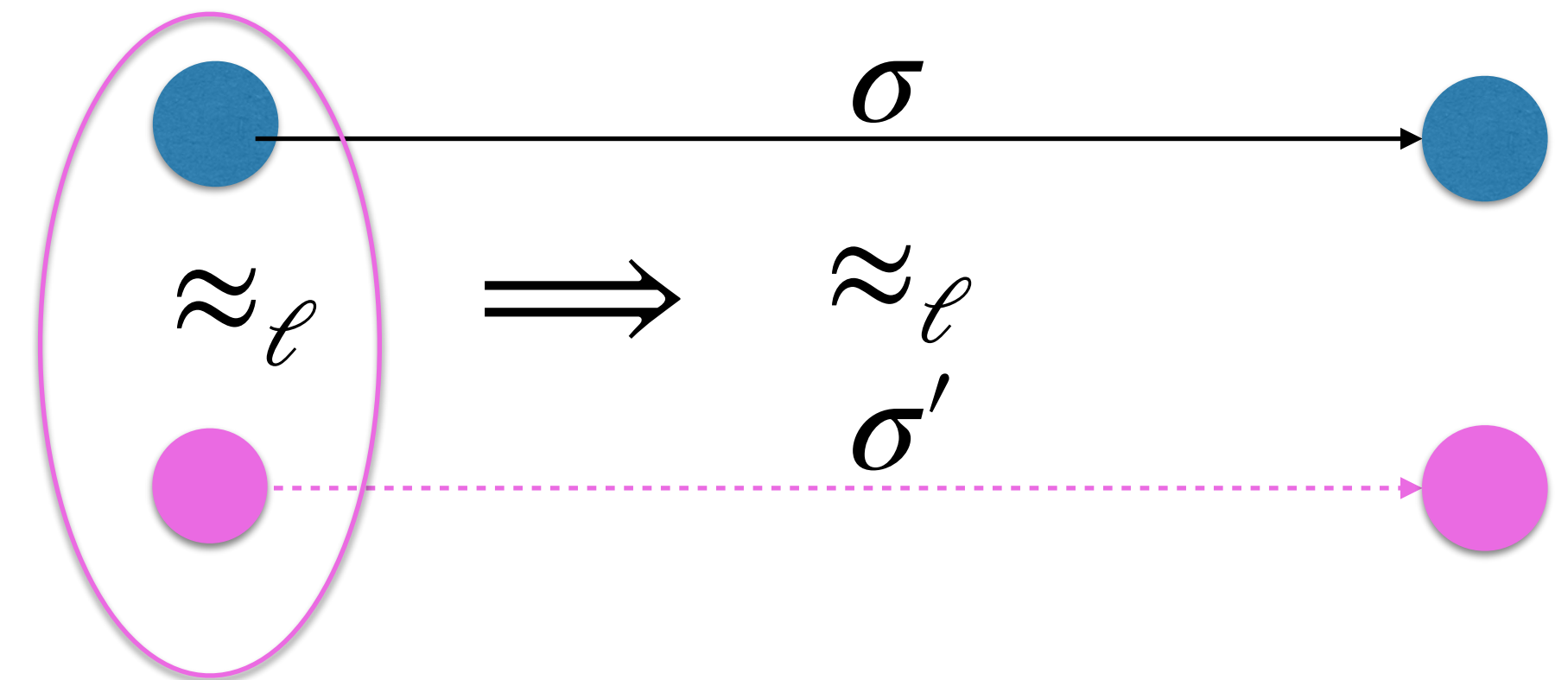
$\langle p = 234, g = 100 \rangle \longrightarrow \text{return } -2;$

\approx_ℓ

$\langle p = 23, g = 100 \rangle \longrightarrow \text{return } -1;$

Auto-Active Reasoning about Noninterference

```
int check_pin(int pin, int guess)
// security policy: is allowed to
// reveal (only) whether pin == guess
_(requires (pin == guess) :: low)
// attacker knows their own guess
_(requires guess :: low)
{
    if (pin == guess) {
        return 0; // match
    } else if (guess > pin) {
        return -1; // guess is too high
    } else {
        return -2; // guess is too low
    }
}
```



$\langle p = 234, g = 100 \rangle \longrightarrow \text{return } -2;$

\approx_ℓ

$\sigma \approx_\ell \sigma'$

$\langle p = 23, g = 100 \rangle \longrightarrow \text{return } -1;$

Declassification via Assume

```
int get_balance(int pin, int guess)
_(requires (pin == guess) :: low)
_(requires guess :: low)
_(ensures result :: low)
{
    if (pin == guess) {
        // pin is correct, get balance
        int b = balance();
        // declassify balance
        _(assume b :: low)
        return b;
    } else {
        return -1; // pin incorrect
    }
}
```


$$\frac{}{\vdash_{\ell} \{\text{emp}\} \text{ assume } \rho \{\rho\}}.$$

Declassification via Assume

```
int get_balance(int pin, int guess)
_(requires (pin == guess) :: low)
_(requires guess :: low)
_(ensures result :: low)
{
    if (pin == guess) {
        // pin is correct, get balance
        int b = balance();
        // declassify balance
        _(assume b :: low)
        return b;
    } else {
        return -1; // pin incorrect
    }
}
```




This is magic



$$\vdash_{\ell} \{\text{emp}\} \text{ assume } \rho \{\rho\}$$

Declassification via Assume

```
int get_balance(int pin, int guess)
_(requires (pin == guess) :: low)
_(requires guess :: low)
_(ensures result :: low)
{
    if (pin == guess) {
        // pin is correct, get balance
        int b = balance();
        // declassify balance
        _(assume b :: low)
        return b;
    } else {
        return -1; // pin incorrect
    }
}
```



This is magic



$$\vdash_{\ell} \{\text{emp}\} \text{ assume } \rho \{\rho\}$$

Contribution 1:

Semantic
characterisation of
soundness of this rule

Assume considered dangerous

```
int check_pin(int pin, int guess)
  _(requires (pin == guess) :: low)
  _(requires guess :: low)
{
  _(assume pin :: low)
  if (pin == guess) {
    int b = balance();
    _(assume b :: low)
    return b;
  } else if (guess > pin) {
    return -1; // guess is too high
  } else {
    return -2; // guess is too low
  }
}
```

Assume considered dangerous

```
int check_pin(int pin, int guess)
  _(requires (pin == guess) :: low)
  _(requires guess :: low)
{
  _(assume pin :: low)
  if (pin == guess) {
    int b = balance();
    _(assume b :: low)
    return b;
  } else if (guess > pin) {
    return -1; // guess is too high
  } else {
    return -2; // guess is too low
  }
}
```

**Insecure
Declassification**



Assume considered dangerous

```
int check_pin(int pin, int guess)
  _(requires (pin == guess) :: low)
  _(requires guess :: low)
{
  _(assume pin :: low)
  if (pin == guess) {
    int b = balance();
    _(assume b :: low)
    return b;
  } else if (guess > pin) {
    return -1; // guess is too high
  } else {
    return -2; // guess is too low
  }
}
```

**Insecure
Declassification**



Contribution 2:

Justify
declassifications
against a
declarative policy

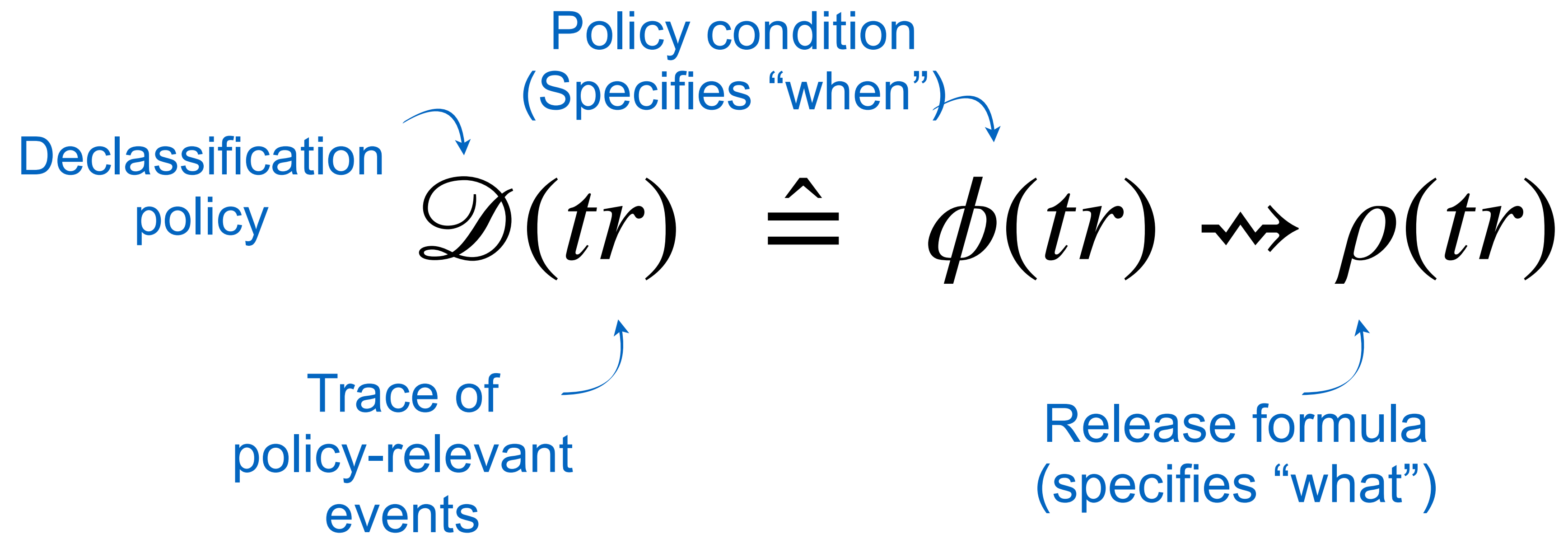
Declarative Security Policies

Specify **when** declassifications are allowed to occur and **what** information is allowed to be declassified when they do occur

$$\mathcal{D}(tr) \hat{=} \phi(tr) \rightsquigarrow \rho(tr)$$

Declarative Security Policies

Specify **when** declassifications are allowed to occur and **what** information is allowed to be declassified when they do occur



Example Policy

```
int check_pin(int pin, int guess)
  _(requires (pin == guess) :: low)
  _(requires guess :: low)
{
  _(assume pin :: low)
  if (pin == guess) {
    int b = balance();
    _(assume b :: low)
    return b;
  } else if (guess > pin) {
    return -1; // too high
  } else {
    return -2; // too low
  }
}
```

$$\mathcal{D}(tr) \hat{=} \phi(tr) \rightsquigarrow \rho(tr)$$

$$\phi(tr) \hat{=} \text{pin} = \text{guess} \wedge \text{last}(tr) = \mathbf{Bal}(b)$$

$$\rho(tr) \hat{=} b :: \text{low}$$

Example Policy

```
int check_pin(int pin, int guess)
  _(requires (pin == guess) :: low)
  _(requires guess :: low)
  {
    _ (assume pin :: low)
    if (pin == guess) {
      int b = balance();
      _ (assume b :: low)
      return b;
    } else if (guess > pin) {
      return -1; // too high
    } else {
      return -2; // too low
    }
  }
}
```

$$\mathcal{D}(tr) \hat{=} \phi(tr) \rightsquigarrow \rho(tr)$$

$$\phi(tr) \hat{=} \text{pin} = \text{guess} \wedge \text{last}(tr) = \mathbf{Bal}(b)$$

$$\rho(tr) \hat{=} b :: \text{low}$$

Example Policy

```
int check_pin(int pin, int guess)
  _(requires (pin == guess) :: low)
  _(requires guess :: low)
  {
    _ (assume pin :: low)
    if (pin == guess) {
      int b = balance();
      _ (assume b :: low)
      return b;
    } else if (guess > pin) {
      return -1; // too high
    } else {
      return -2; // too low
    }
  }
}
```

$$\mathcal{D}(tr) \hat{=} \phi(tr) \rightsquigarrow \rho(tr)$$

$$\phi(tr) \hat{=} \text{pin} = \text{guess} \wedge \text{last}(tr) = \mathbf{Bal}(b)$$

$$\rho(tr) \hat{=} b :: \text{low}$$

Example Policy

```
int check_pin(int pin, int guess)
  _(requires (pin == guess) :: low)
  _(requires guess :: low)
  {
    _(assume pin :: low)
    if (pin == guess) {
      int b = balance();
      _(assume b :: low)
      return b;
    } else if (guess > pin) {
      return -1; // too high
    } else {
      return -2; // too low
    }
  }
}
```

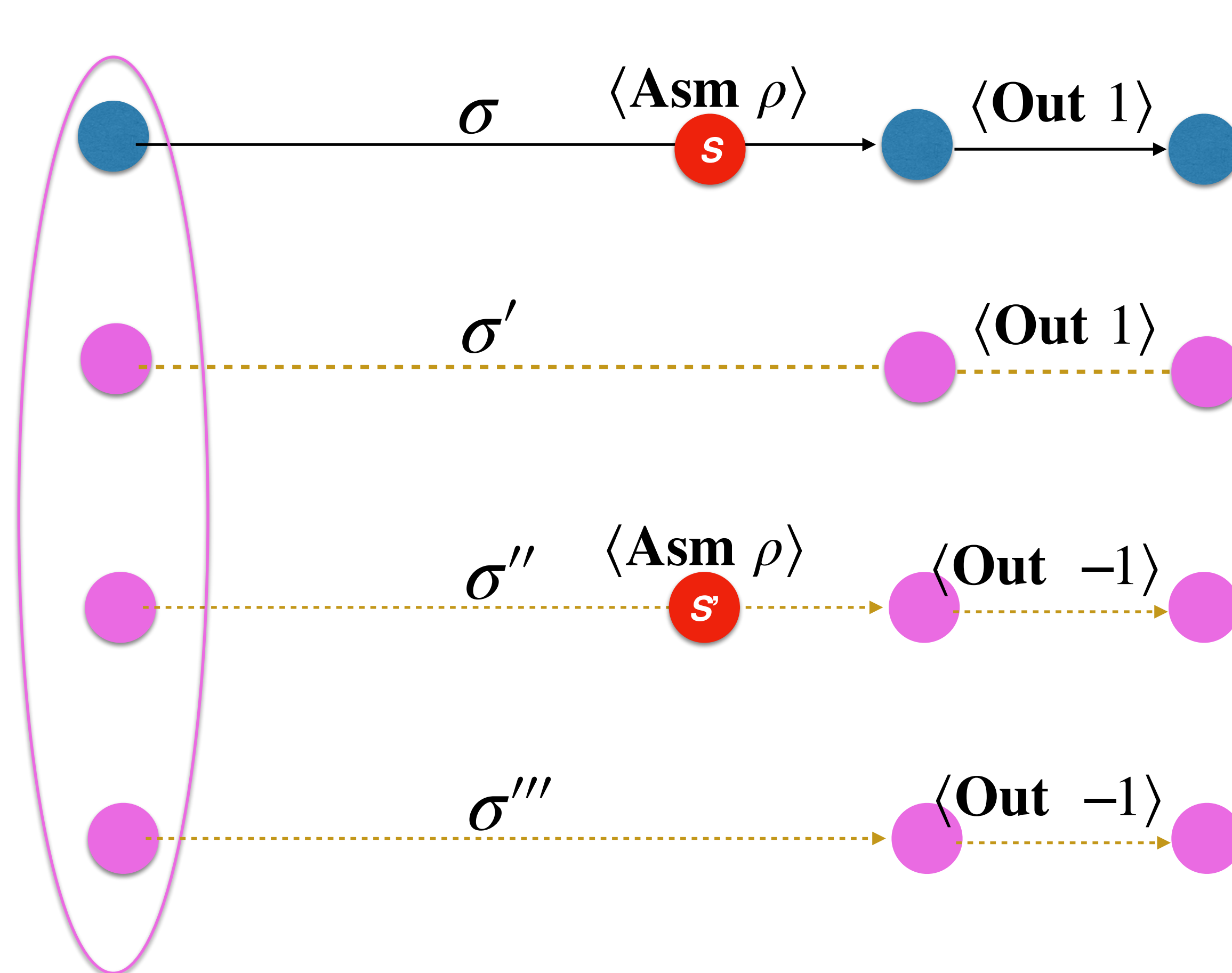
$$\mathcal{D}(tr) \hat{=} \phi(tr) \rightsquigarrow \rho(tr)$$

$$\phi(tr) \hat{=} \text{pin} = \text{guess} \wedge \text{last}(tr) = \mathbf{Bal}(b)$$

$$\rho(tr) \hat{=} b :: \text{low}$$

What does Assume mean?

Policy-Agnostic Guarantee: Each step is allowed to leak some information, specifically **whether an (earlier) assumption has been violated**

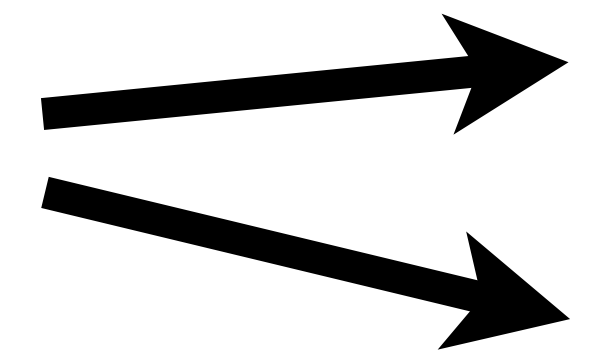


Theorem: Soundness

secure output



failed assumption,
allowed by policy?



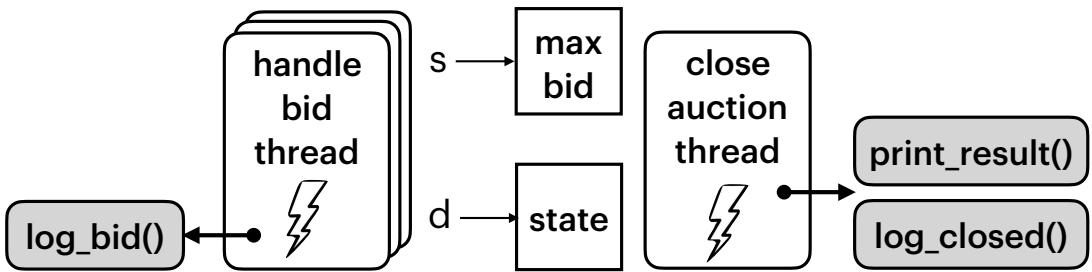
insecure output
is detected



Case Studies

$$\mathcal{D}(tr) \hat{=} \phi(tr) \rightsquigarrow \rho(tr)$$

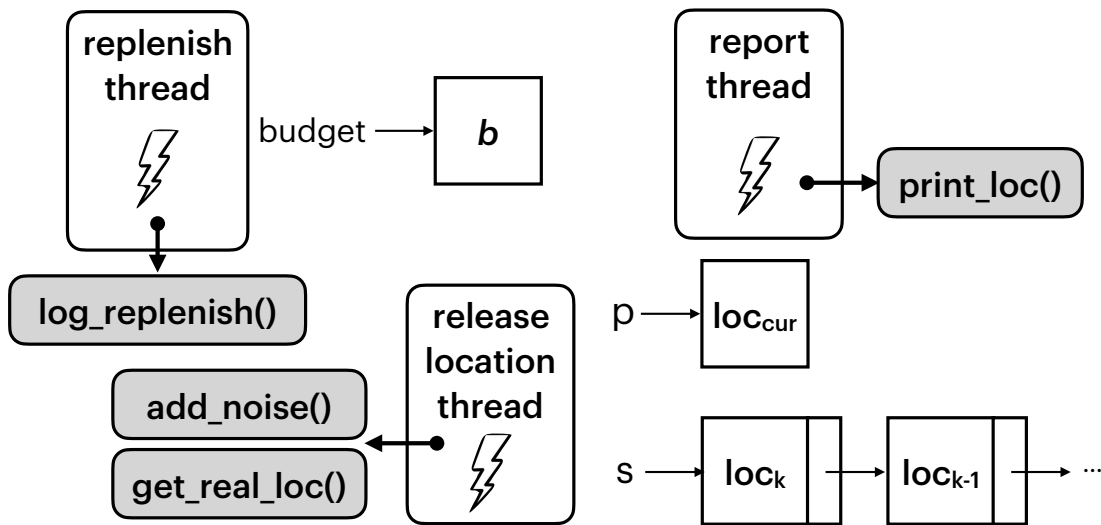
Sealed Bid Auction Server



When: auction is finished

What: winning bid

Private Location Service



When: sufficient privacy budget remains

What: a noisy location point (to ensure differential privacy)

WORDLE
Server

<https://verse.systems/blog/>

Private Learning

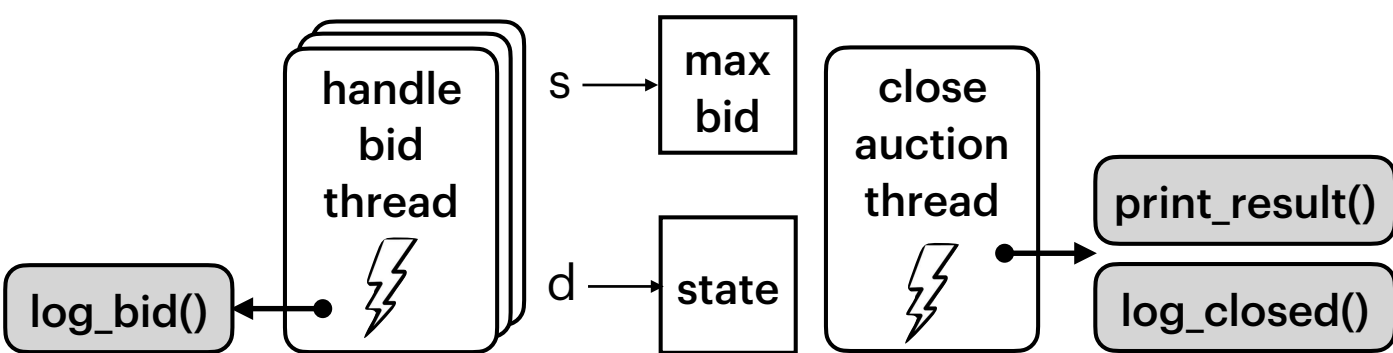
Thank You



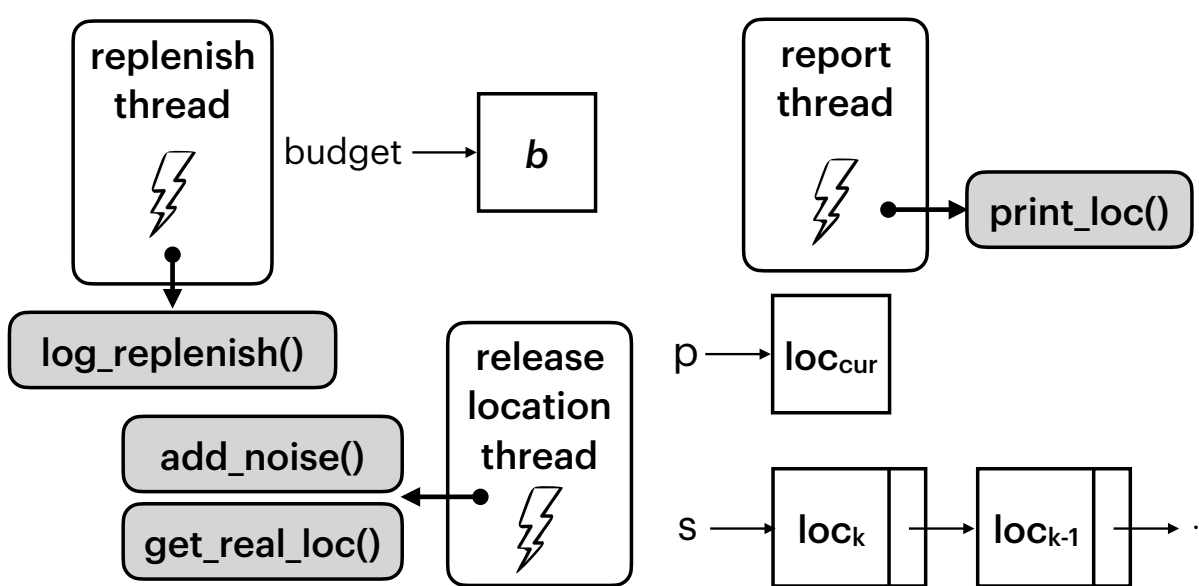
<https://covern.org/>

Case Study	Proof Ratio	Verified SLOC	Unverified SLOC	Effort (pw)
Location Service	1.9	210	124	3
Auction Server	4.3	187	79	3.5
Wordle	5.9	47	81	0.3
Private Learning	2.5	315	114	6

Sealed Bid Auction Server



Private Location Service



WORDLE
Server

Private Learning