ZKPDL: A Language-Based System for Zero-Knowledge Proofs and Electronic Cash

Sarah Meiklejohn (UC San Diego)

C. Chris Erway (Brown University)

Alptekin Küpcü (Brown University)

Theodora Hinkle (UW Madison)

Anna Lysyanskaya (Brown University)

Bridging the gap

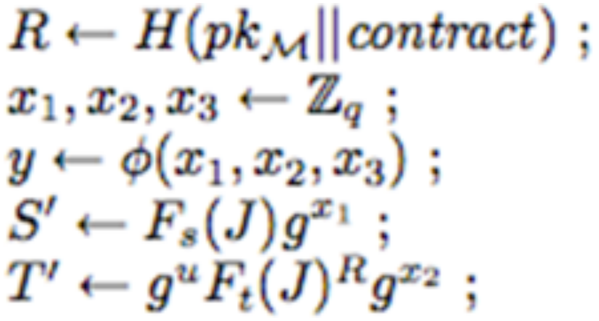
Crypto

Systems

Bridging the gap

Crypto

Systems



$R \leftarrow H(pk_{\mathcal{M}} || contract)$;

$x_1, x_2, x_3 \leftarrow \mathbb{Z}_q$;

$y \leftarrow \phi(x_1, x_2, x_3)$;

$S' \leftarrow F_s(J) g^{x_1}$;

$T' \leftarrow g^u F_t(J)^R g^{x_2}$;

Bridging the gap

Crypto

$$
\begin{aligned}
&R \leftarrow H(pk_{\mathcal{M}} || contract)\ ;\\
&x_1, x_2, x_3 \leftarrow \mathbb{Z}_q\ ;\\
&y \leftarrow \phi(x_1, x_2, x_3)\ ;\\
&S' \leftarrow F_s(J)g^{x_1}\ ;\\
&T' \leftarrow g^u F_t(J)^R g^{x_2}\ ;
\end{aligned}
$$

Systems

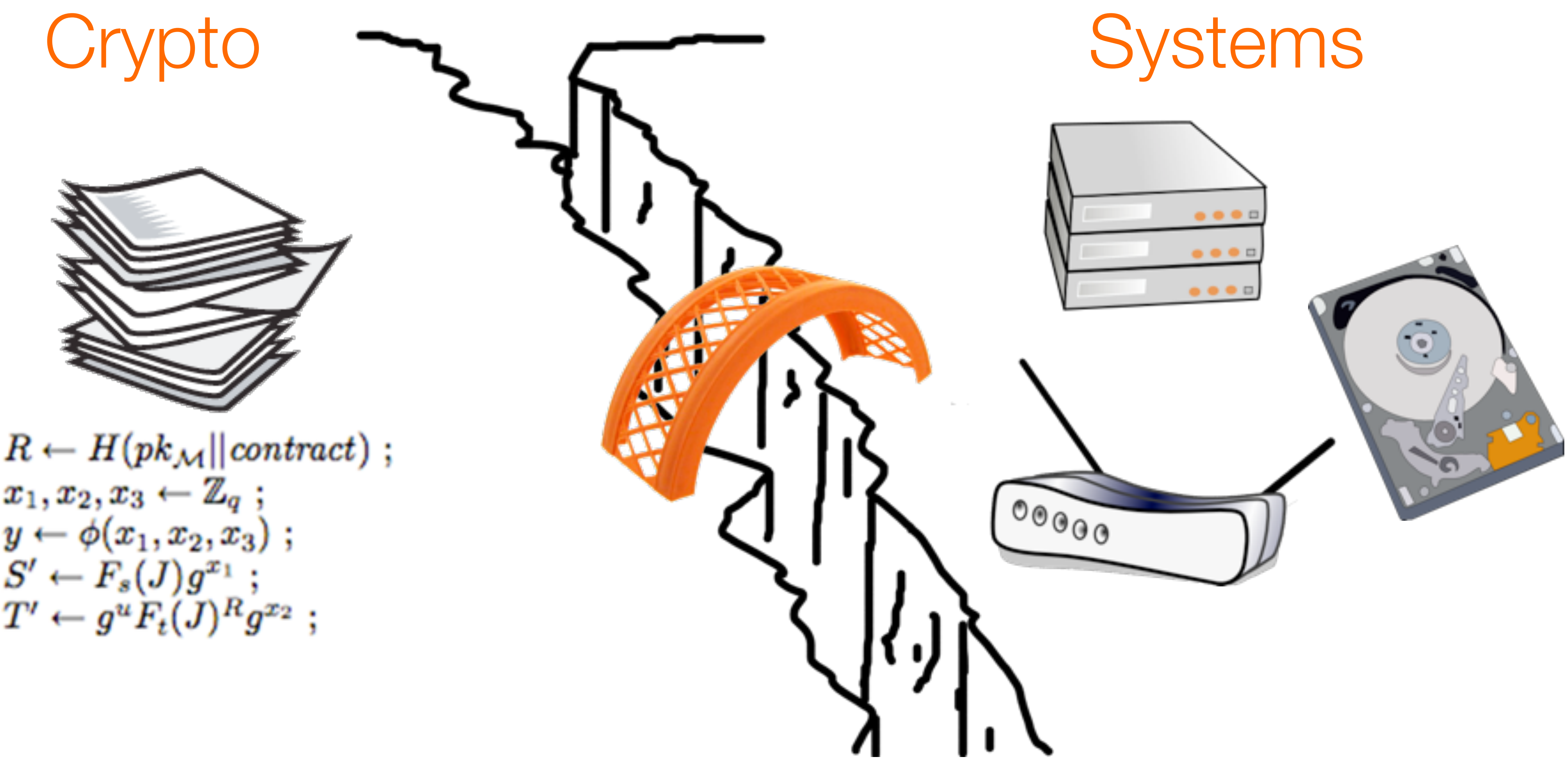
Bridging the gap

Crypto

$$R \leftarrow H(pk_{\mathcal{M}}||contract)\ ;$$
$$x_1, x_2, x_3 \leftarrow \mathbb{Z}_q\ ;$$
$$y \leftarrow \phi(x_1, x_2, x_3)\ ;$$
$$S' \leftarrow F_s(J)g^{x_1}\ ;$$
$$T' \leftarrow g^u F_t(J)^R g^{x_2}\ ;$$

Systems

Bridging the gap



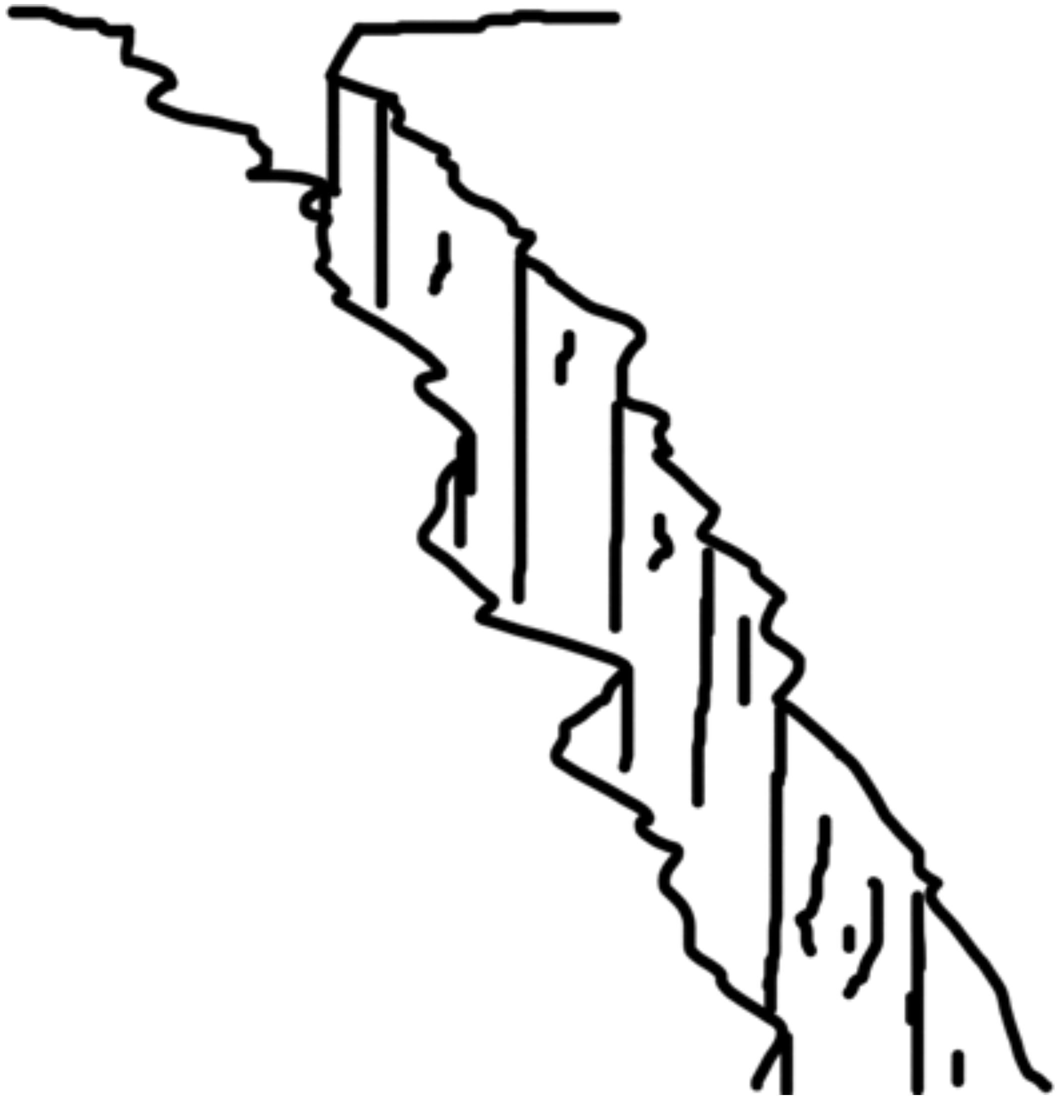
Crypto

$$
\begin{aligned}
&R \leftarrow H(pk_{\mathcal{M}} \| contract)\ ;\\
&x_1, x_2, x_3 \leftarrow \mathbb{Z}_q\ ;\\
&y \leftarrow \phi(x_1, x_2, x_3)\ ;\\
&S' \leftarrow F_s(J) g^{x_1}\ ;\\
&T' \leftarrow g^u F_t(J)^R g^{x_2}\ ;
\end{aligned}
$$

Systems

Bridging the gap... for zero knowledge proofs

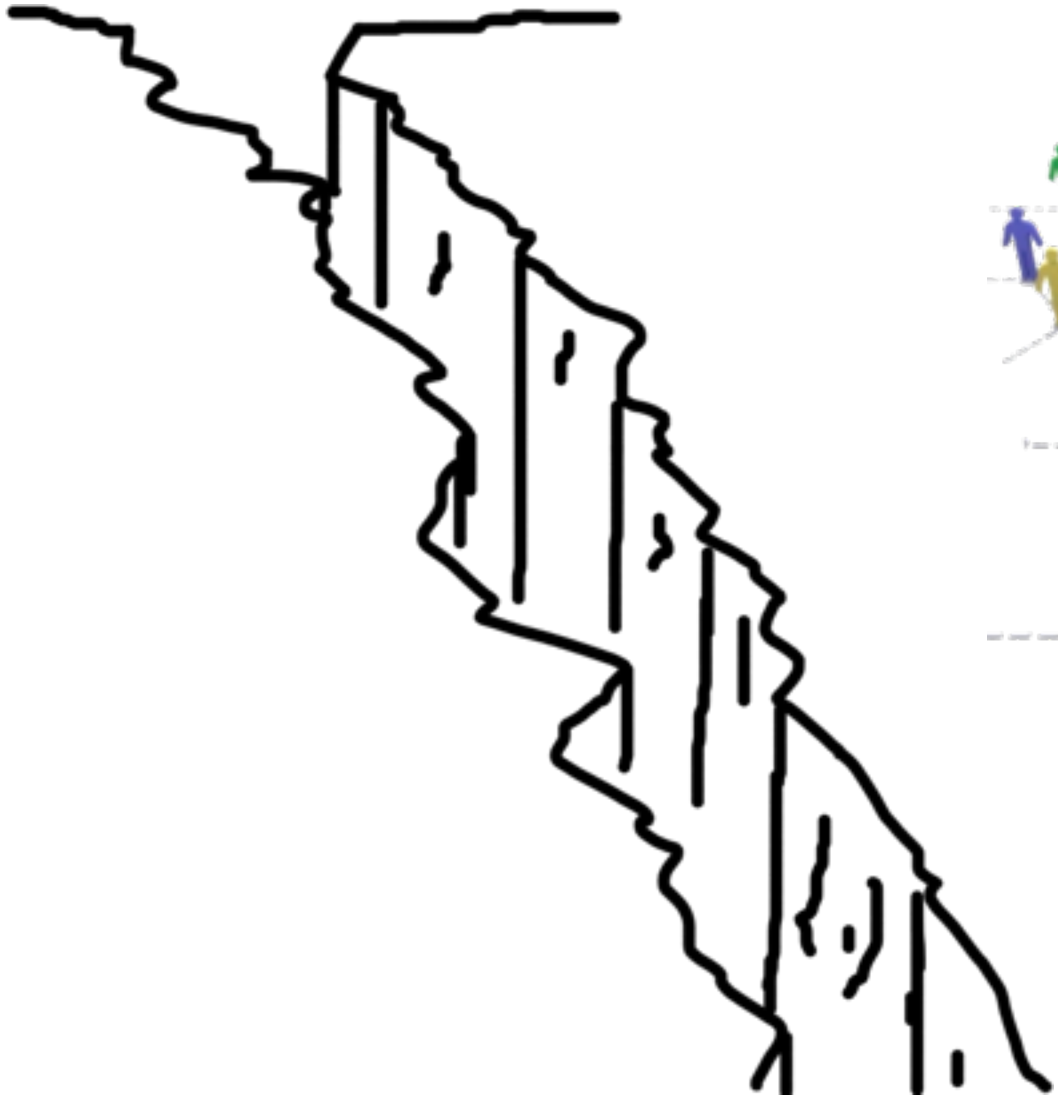
Crypto



Systems

Bridging the gap... for zero knowledge proofs

Crypto



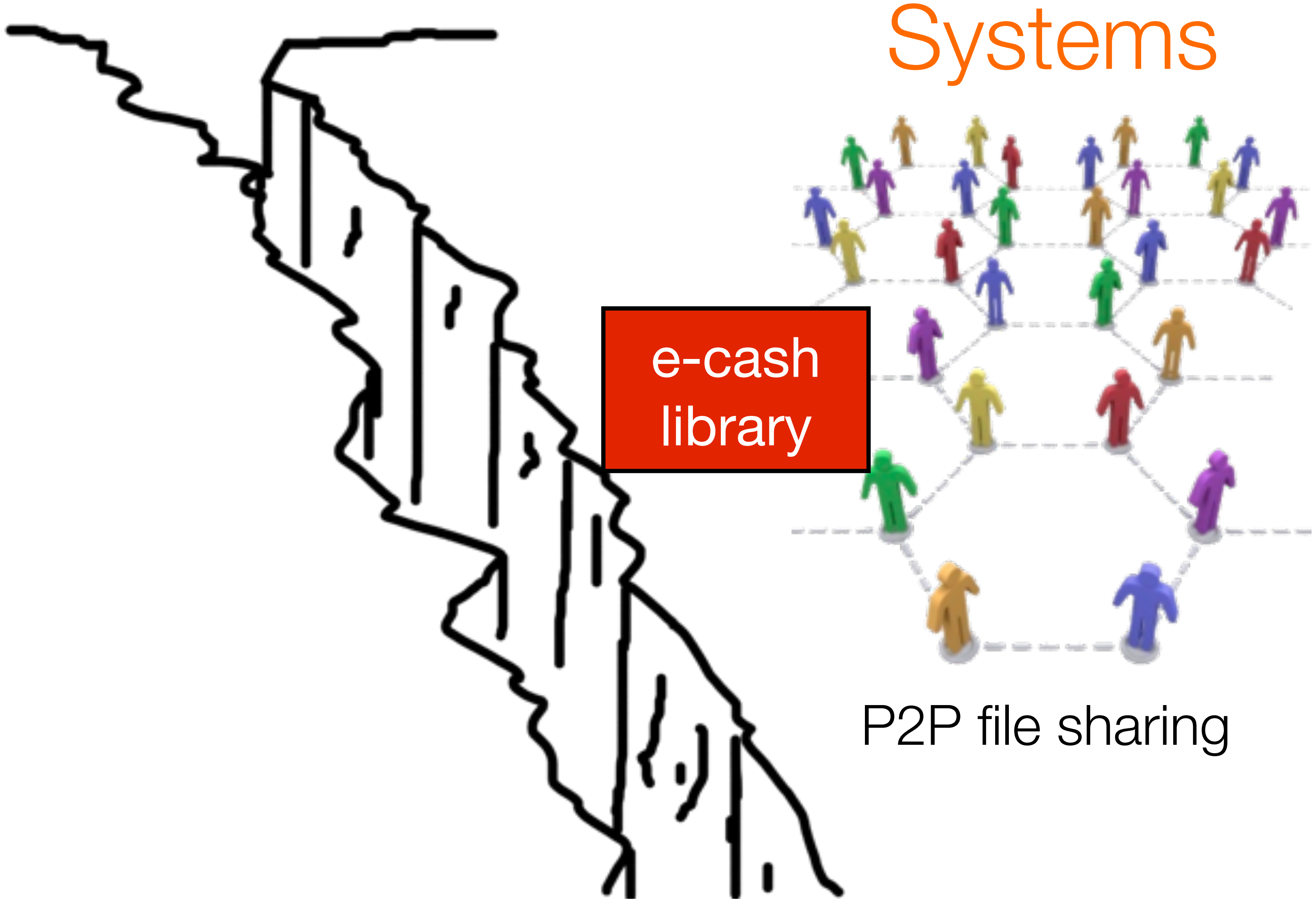
Systems

P2P file sharing

Bridging the gap... for zero knowledge proofs

Crypto

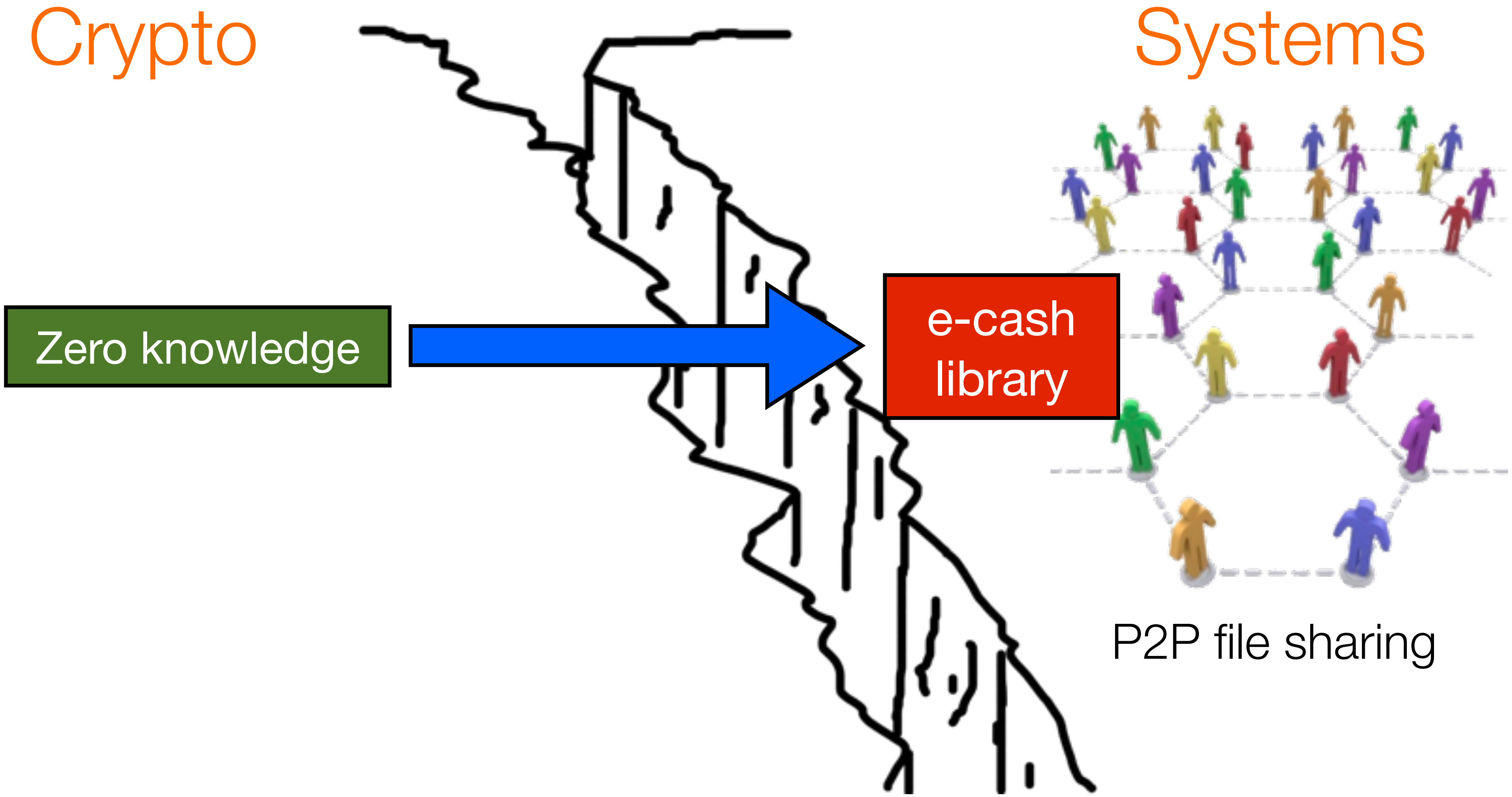
Systems



e-cash library

P2P file sharing

Bridging the gap... for zero knowledge proofs



Crypto

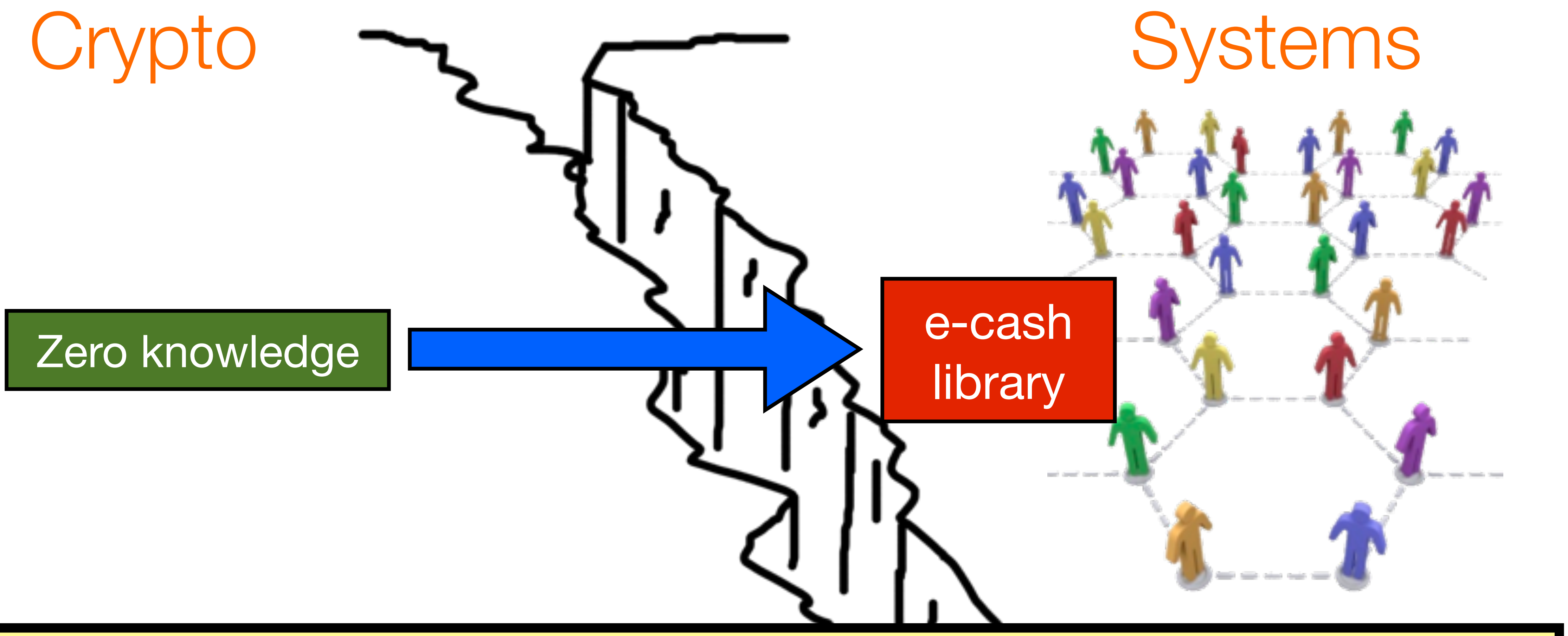
Systems

Zero knowledge

e-cash library

P2P file sharing

Bridging the gap... for zero knowledge proofs



Crypto

Systems

Zero knowledge

e-cash library

Wrote language for zero-knowledge proofs

Removes obstacle, easy to translate from description to implementation

Wrote library for e-cash using this language/interpreter framework

Zero-knowledge proofs [GMR89,BdSMP91]

Prover

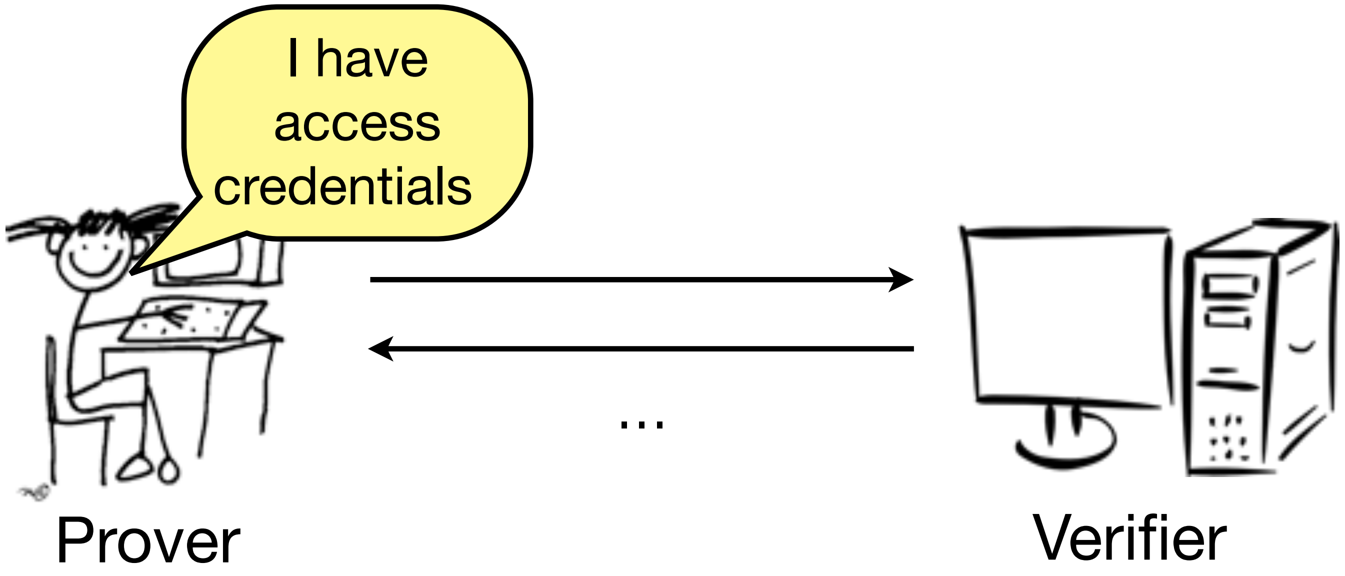
Verifier

Zero-knowledge proofs [GMR89,BdSMP91]

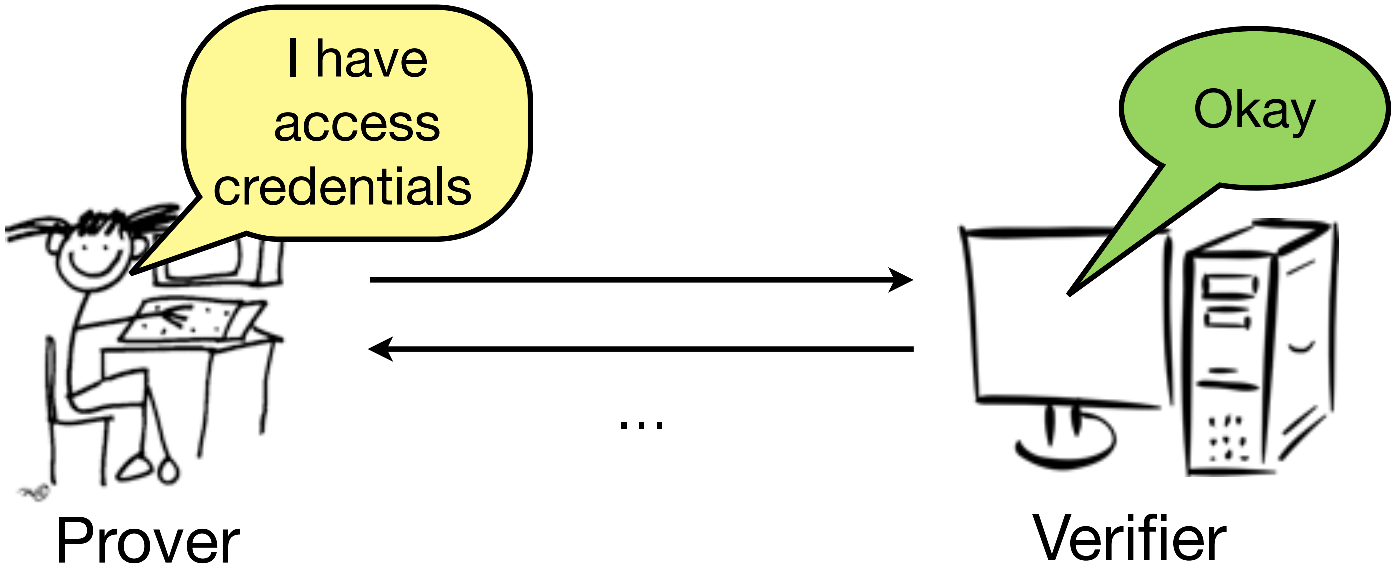
Prover

Verifier

Zero-knowledge proofs [GMR89,BdSMP91]



Zero-knowledge proofs [GMR89,BdSMP91]



I have access credentials

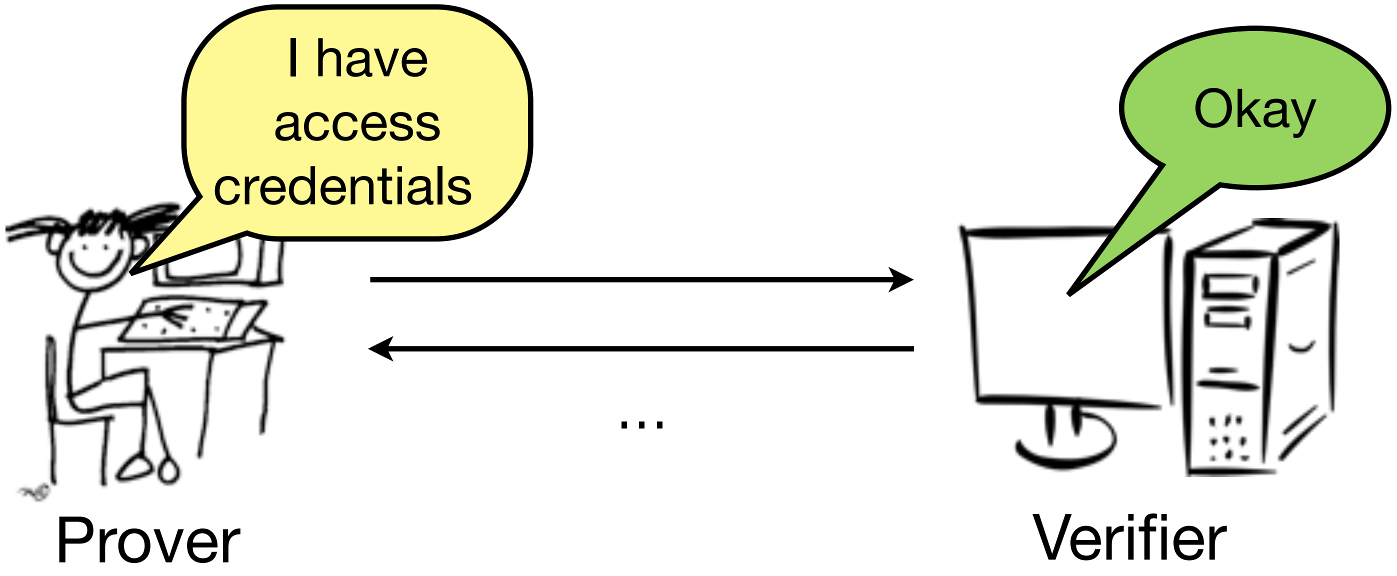
Okay

…

Prover

Verifier

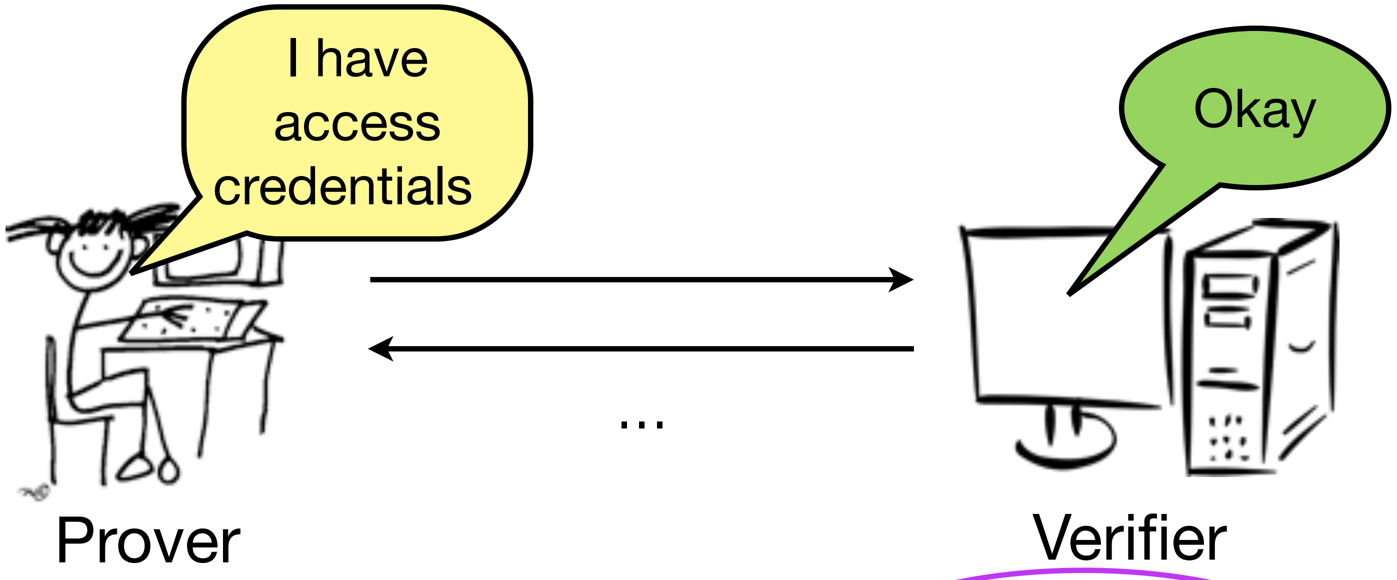
Zero-knowledge proofs [GMR89,BdSMP91]



Soundness: system won’t accept incorrect proof

Zero-knowledge: system won’t learn anything it didn’t already know

Zero-knowledge proofs [GMR89,BdSMP91]

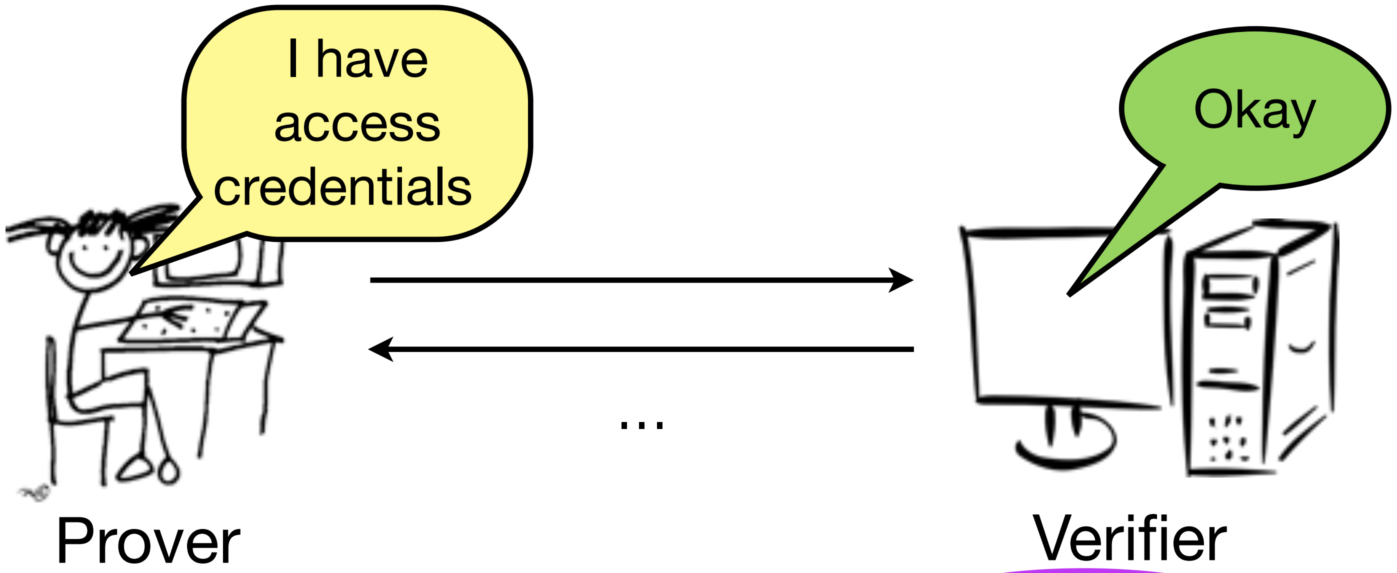


Soundness: system won't accept incorrect proof

Zero-knowledge: system won't learn anything it didn't already know

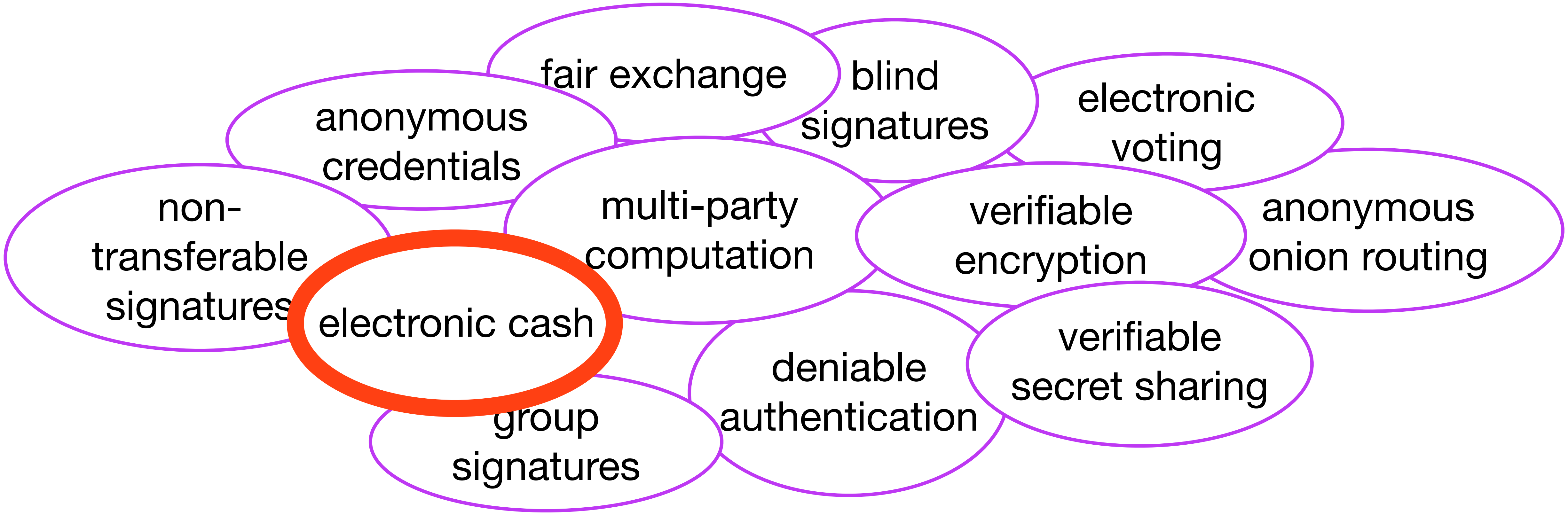
- fair exchange
- blind signatures
- electronic voting
- anonymous credentials
- multi-party computation
- verifiable encryption
- anonymous onion routing
- non-transferable signatures
- electronic cash
- deniable authentication
- verifiable secret sharing
- group signatures

Zero-knowledge proofs [GMR89,BdSMP91]



Soundness: system won't accept incorrect proof

Zero-knowledge: system won't learn anything it didn't already know



fair exchange

blind signatures

electronic voting

anonymous credentials

multi-party computation

verifiable encryption

anonymous onion routing

non-transferable signatures

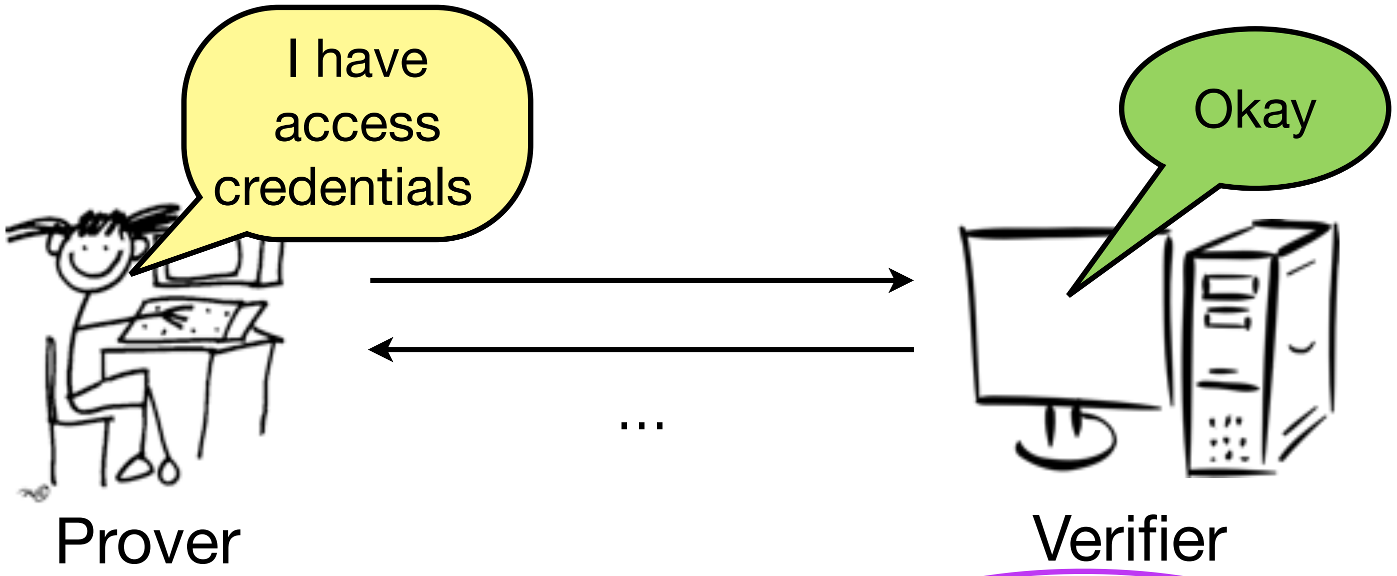
electronic cash

deniable authentication

verifiable secret sharing

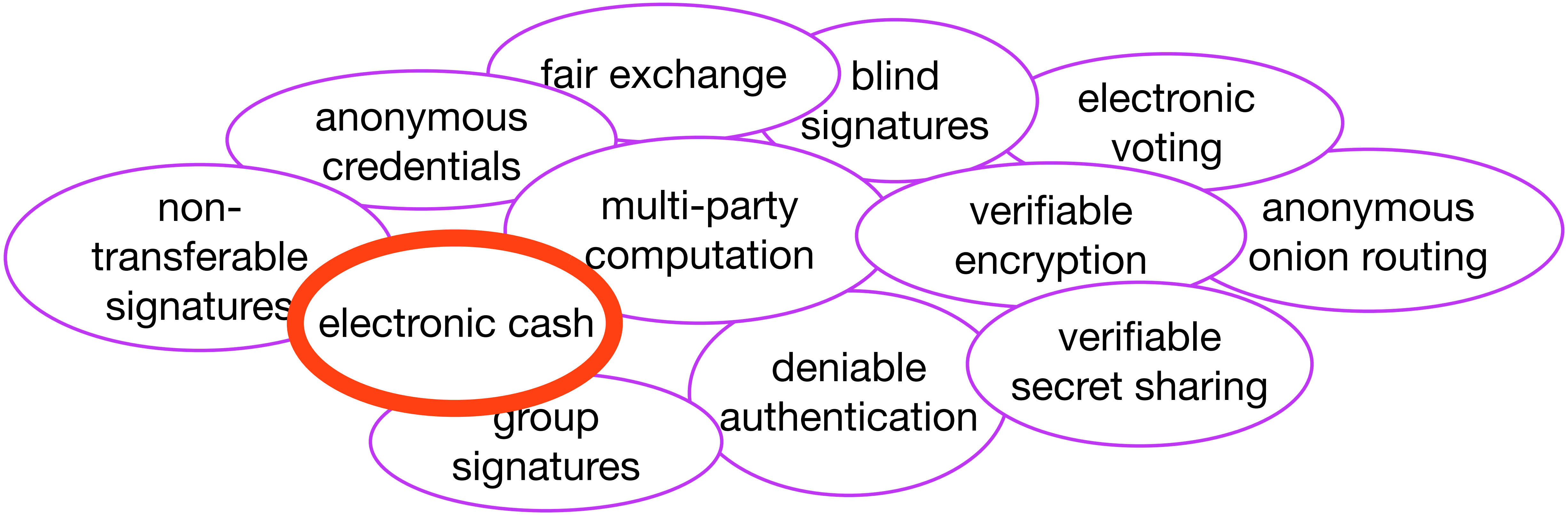
group signatures

Zero-knowledge proofs [GMR89,BdSMP91]



Soundness: system won’t accept incorrect proof

Zero-knowledge: system won’t learn anything it didn’t already know



fair exchange

blind signatures

anonymous credentials

electronic voting

non-transferable signatures

multi-party computation

verifiable encryption

anonymous onion routing

electronic cash

deniable authentication

verifiable secret sharing

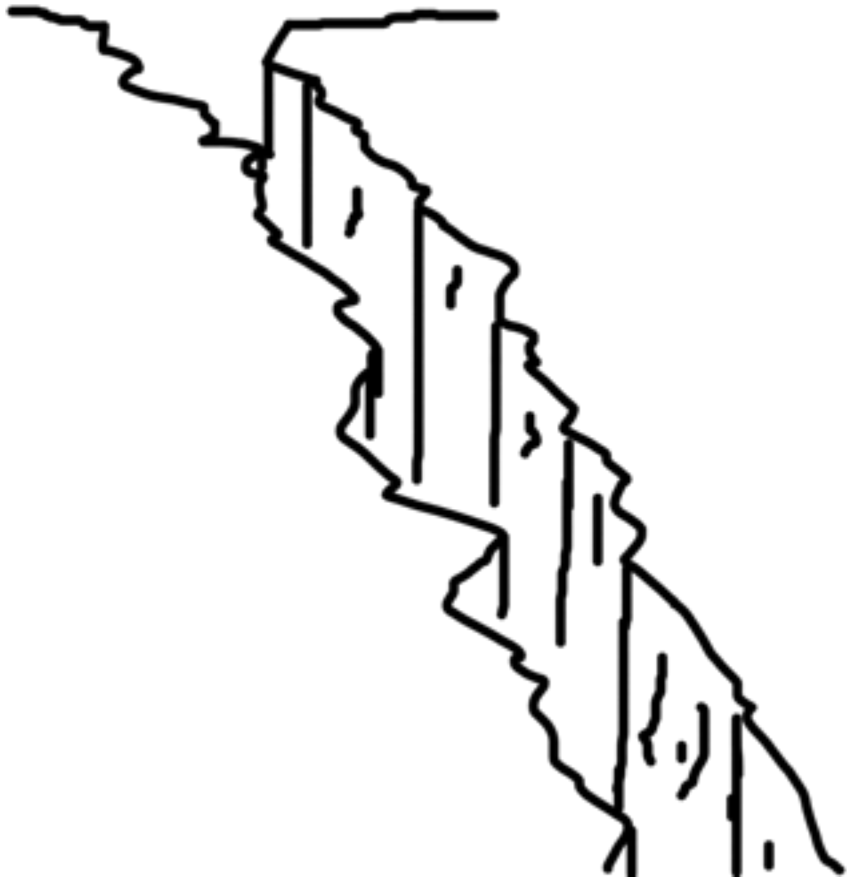
group signatures

Zero-knowledge proofs have applications, but can be complex

Implementing zero knowledge (take 1)

Crypto

Zero knowledge



Systems

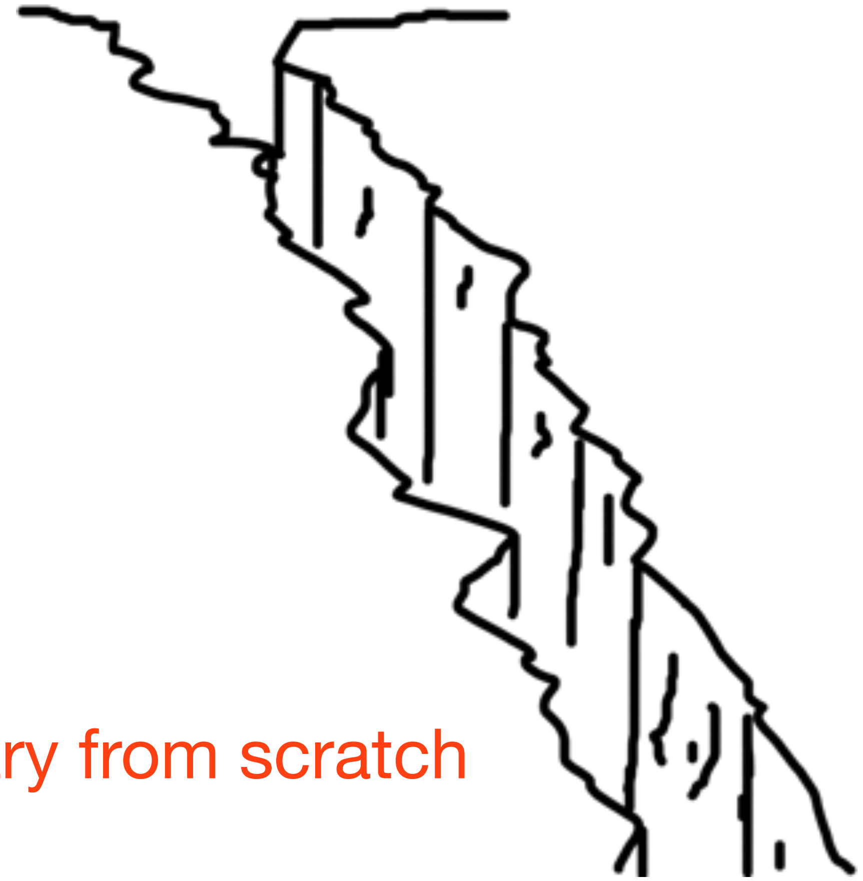
P2P file sharing

Implementing zero knowledge (take 1)

Crypto

Zero knowledge

Our first attempt: write library from scratch



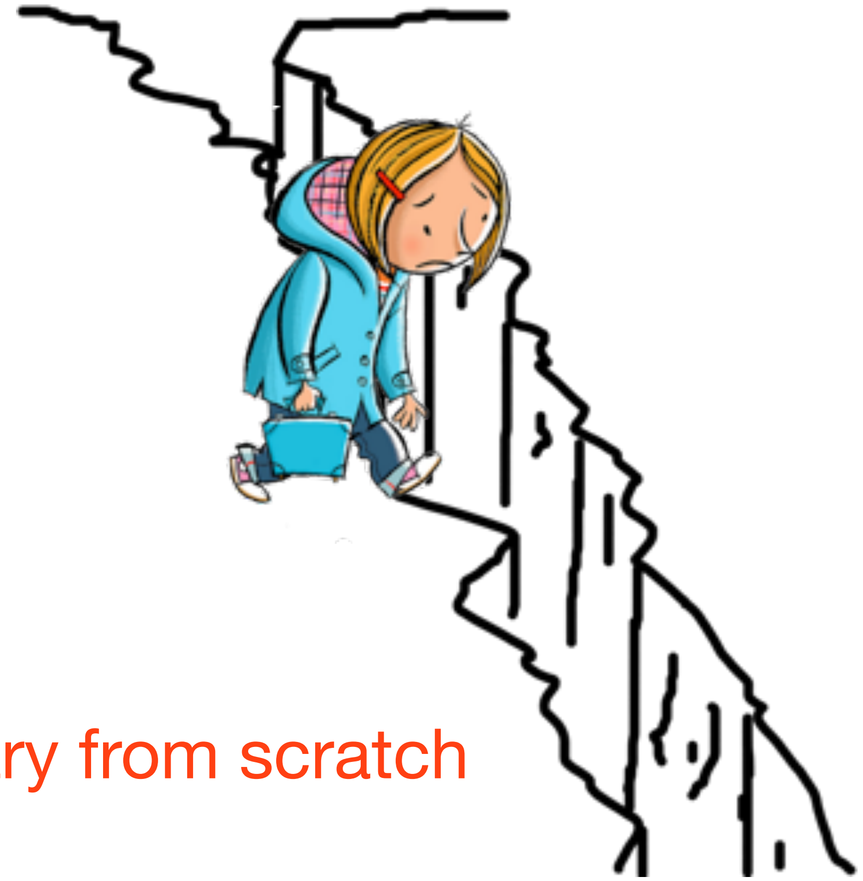
Systems

P2P file sharing

Implementing zero knowledge (take 1)

Crypto

Zero knowledge



Our first attempt: write library from scratch

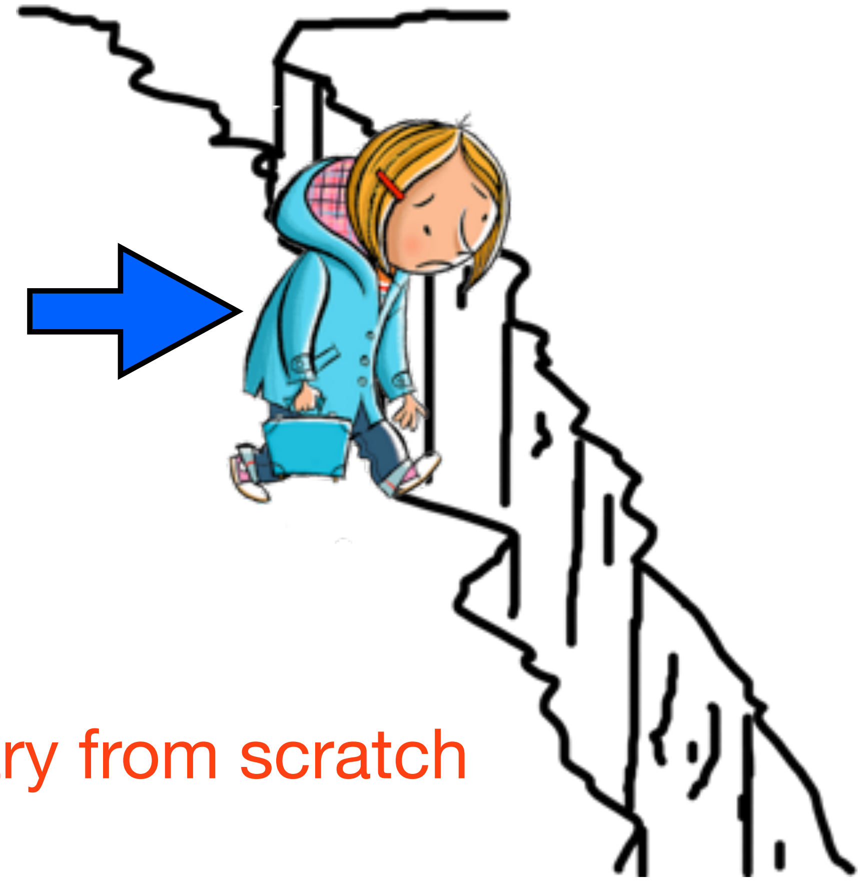
Systems

P2P file sharing

Implementing zero knowledge (take 1)

Crypto

Zero knowledge

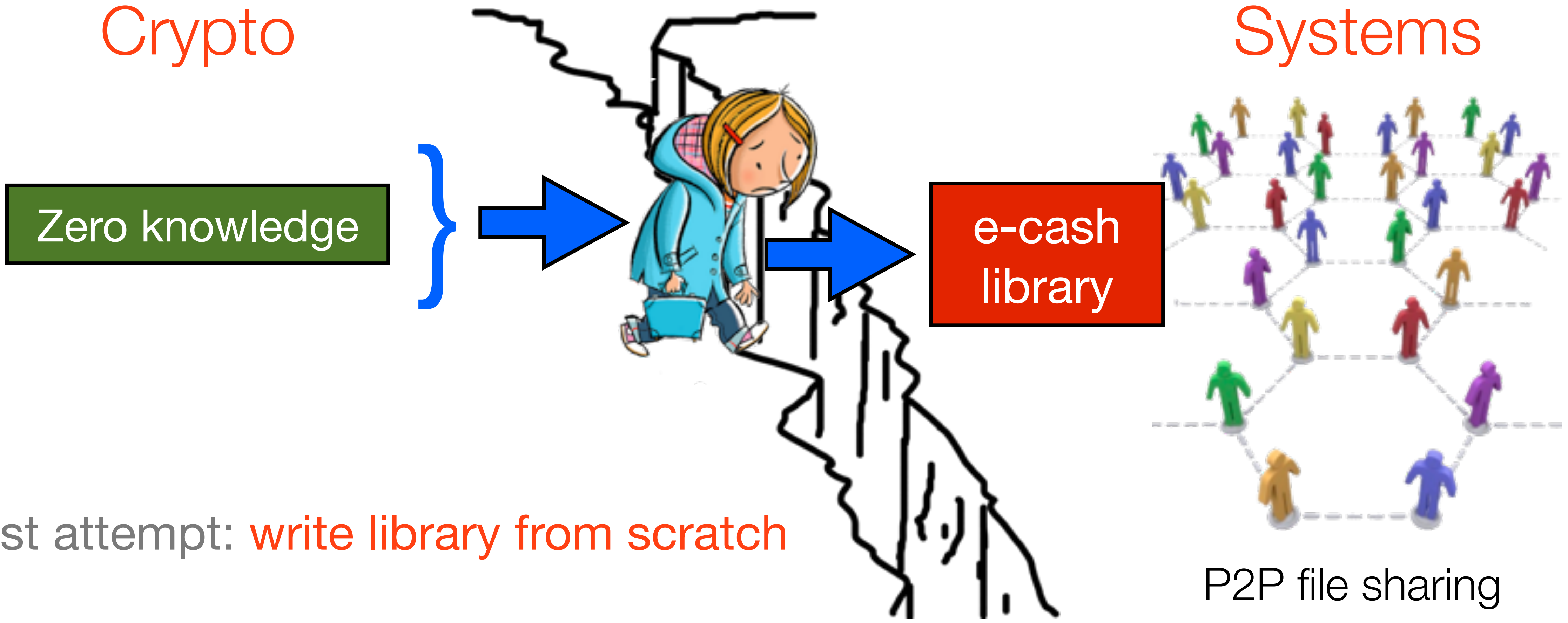


Systems

P2P file sharing

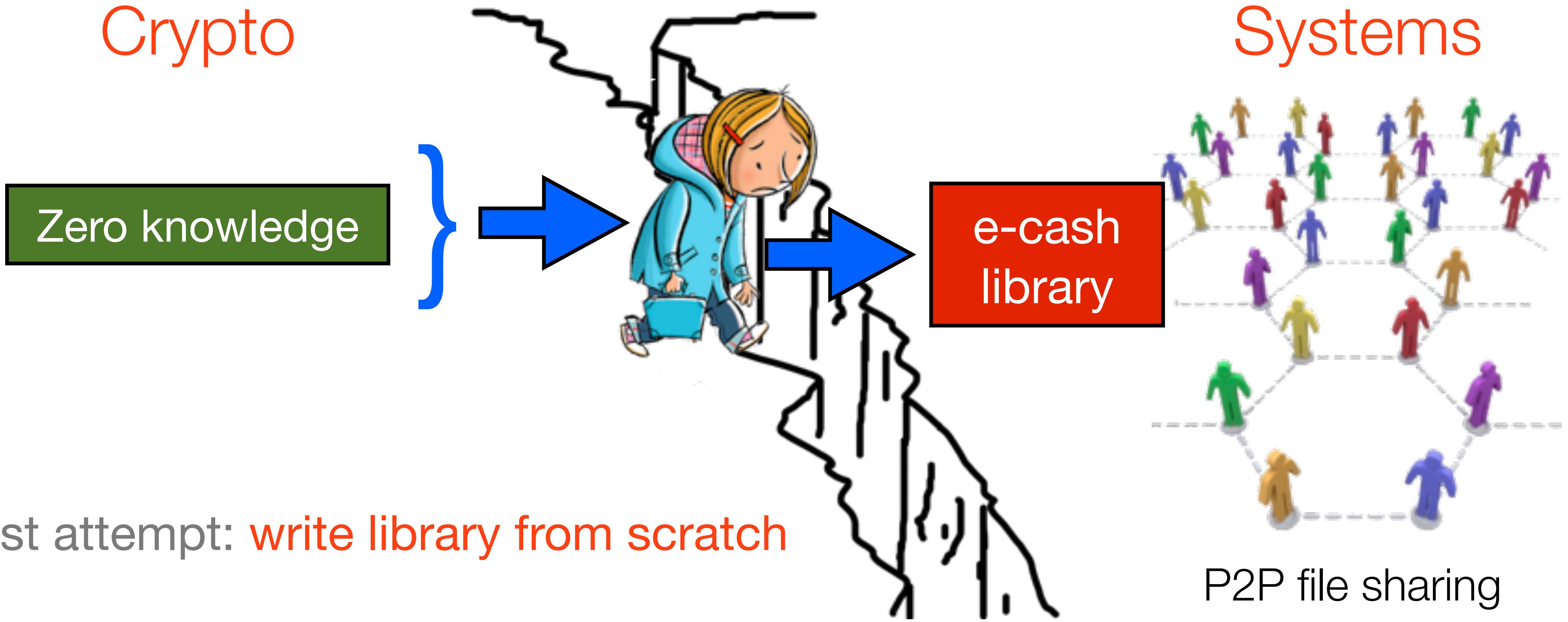
Our first attempt: write library from scratch

Implementing zero knowledge (take 1)



Our first attempt: write library from scratch

Implementing zero knowledge (take 1)



Our first attempt: write library from scratch

- Not reusable
- Time-consuming
- Error prone

Implementing zero knowledge (take 1)

Implementing zero knowledge (take 1)

- Lesson learned: even though you know the math, coding can get messy

Implementing zero knowledge (take 1)

- Lesson learned: even though you know the math, coding can get messy

```
Coin::Coin(const BankParameters *params, int stat, int lx,
  hashalg_t hashAlg, const ZZ &coinIndex, const ZZ
  &walletSize, int coinDenom, const ZZ &sk_u, const ZZ &s,
  const ZZ &t, const vec_ZZ &clSig, const vector<SecretValue>
  &clPrivateSecrets, const vector<SecretValue>
  &clPrivateRandoms, const ZZ &r) { ...
```

Implementing zero knowledge (take 1)

- Lesson learned: even though you know the math, coding can get messy

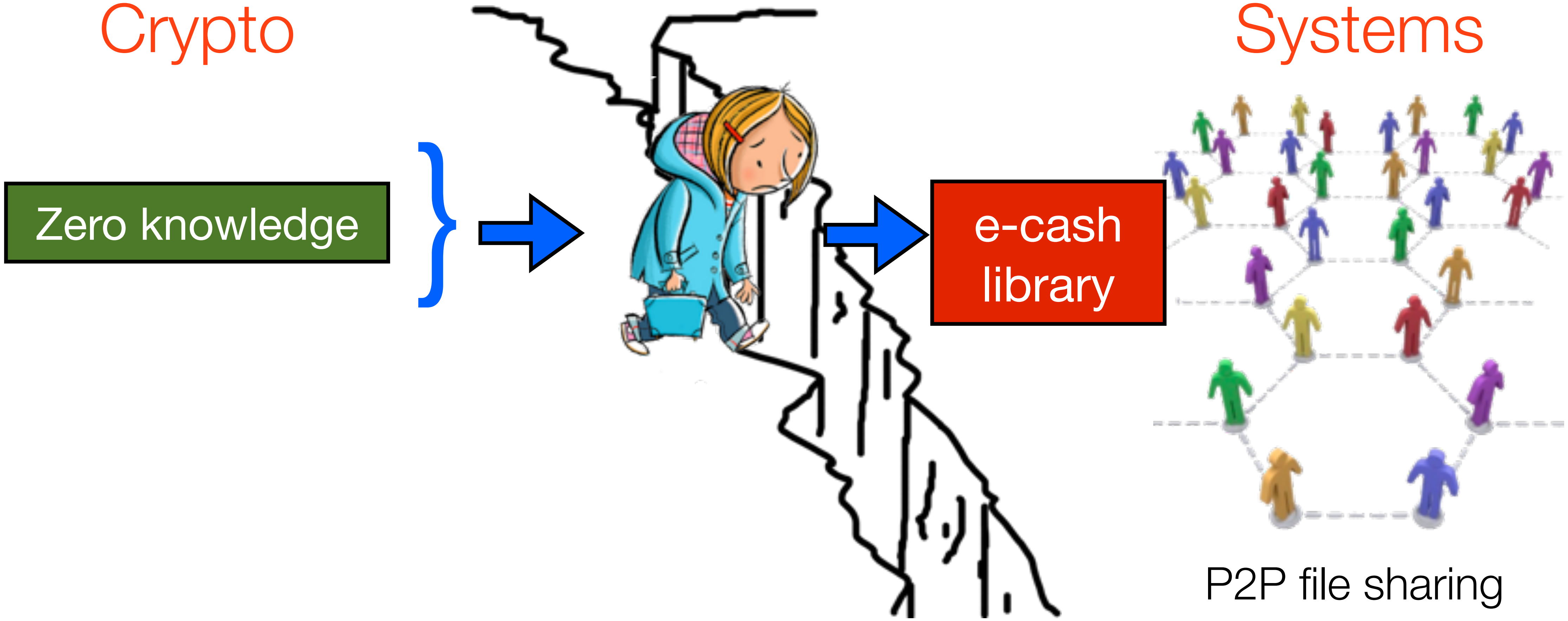
```
Coin::Coin(const BankParameters *params, int stat, int lx,
  hashalg_t hashAlg, const ZZ &coinIndex, const ZZ
  &walletSize, int coinDenom, const ZZ &sk_u, const ZZ &s,
  const ZZ &t, const vec_ZZ &clSig, const vector<SecretValue>
  &clPrivateSecrets, const vector<SecretValue>
  &clPrivateRandoms, const ZZ &r) { ...
```

- Functionality is there, but not easy to use

Implementing zero knowledge (take 2)

Crypto

Systems



Zero knowledge

e-cash library

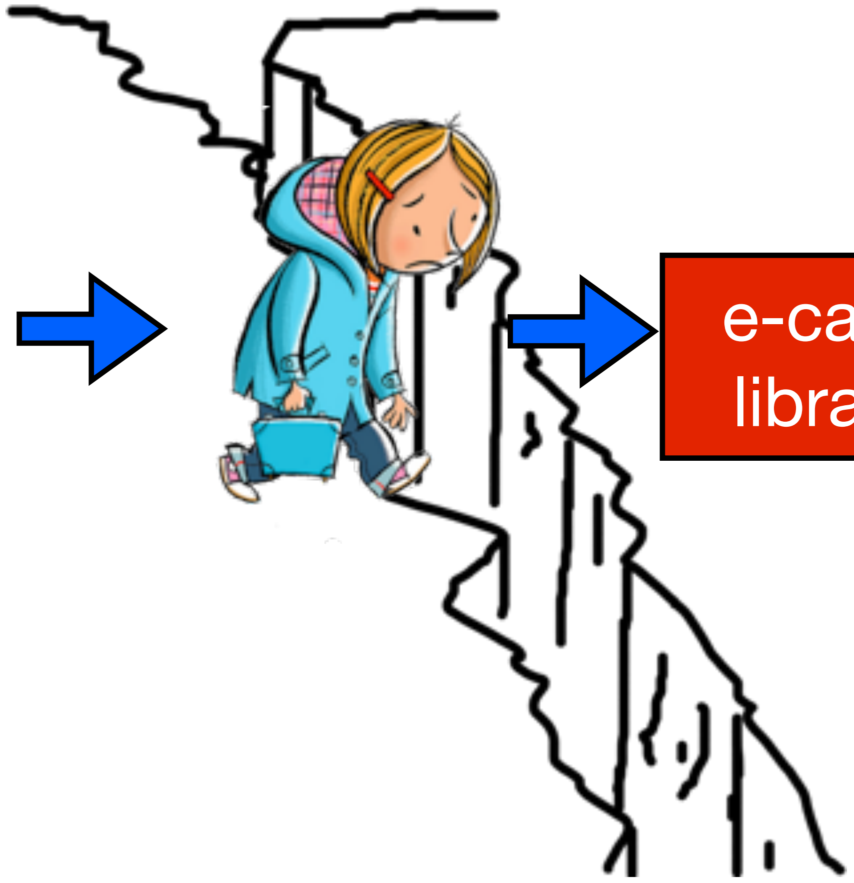
P2P file sharing

Implementing zero knowledge (take 2)

Crypto

Systems

Zero knowledge

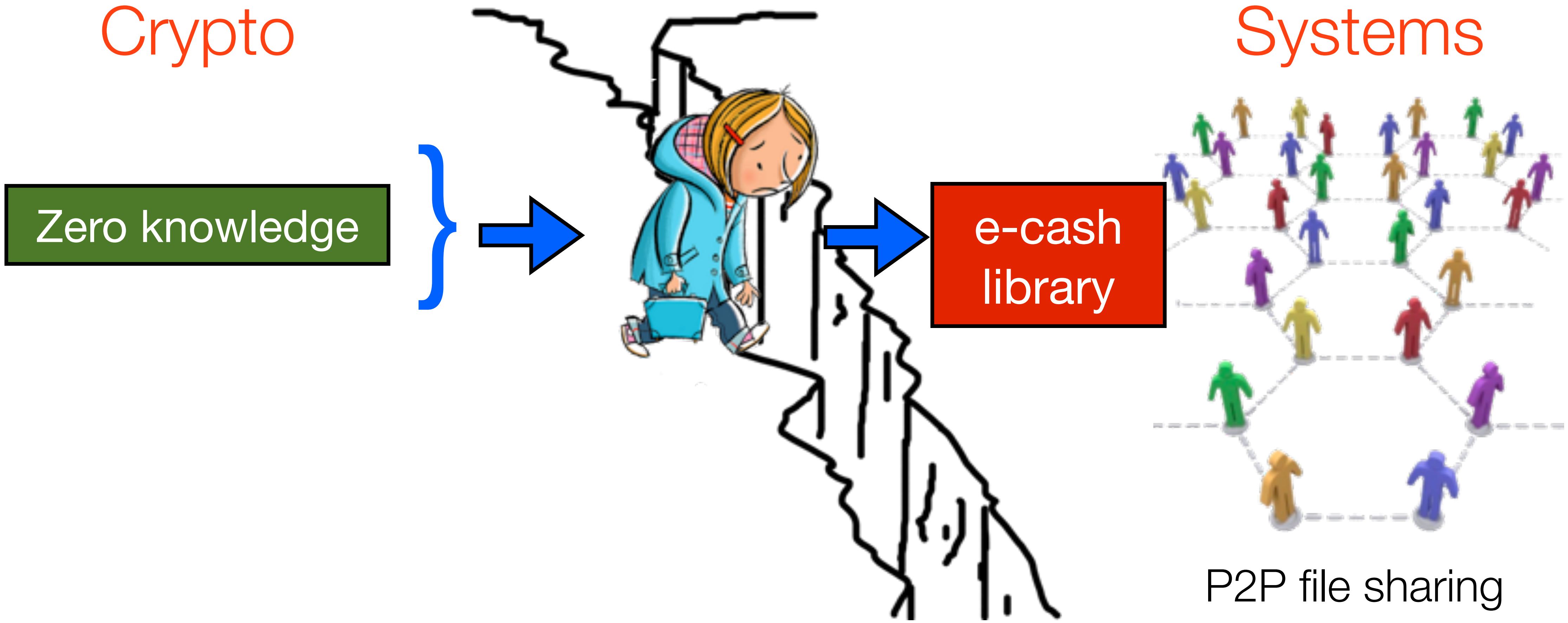


e-cash library

P2P file sharing

How can we lighten the implementation load?

Implementing zero knowledge (take 2)



Crypto

Zero knowledge

e-cash library

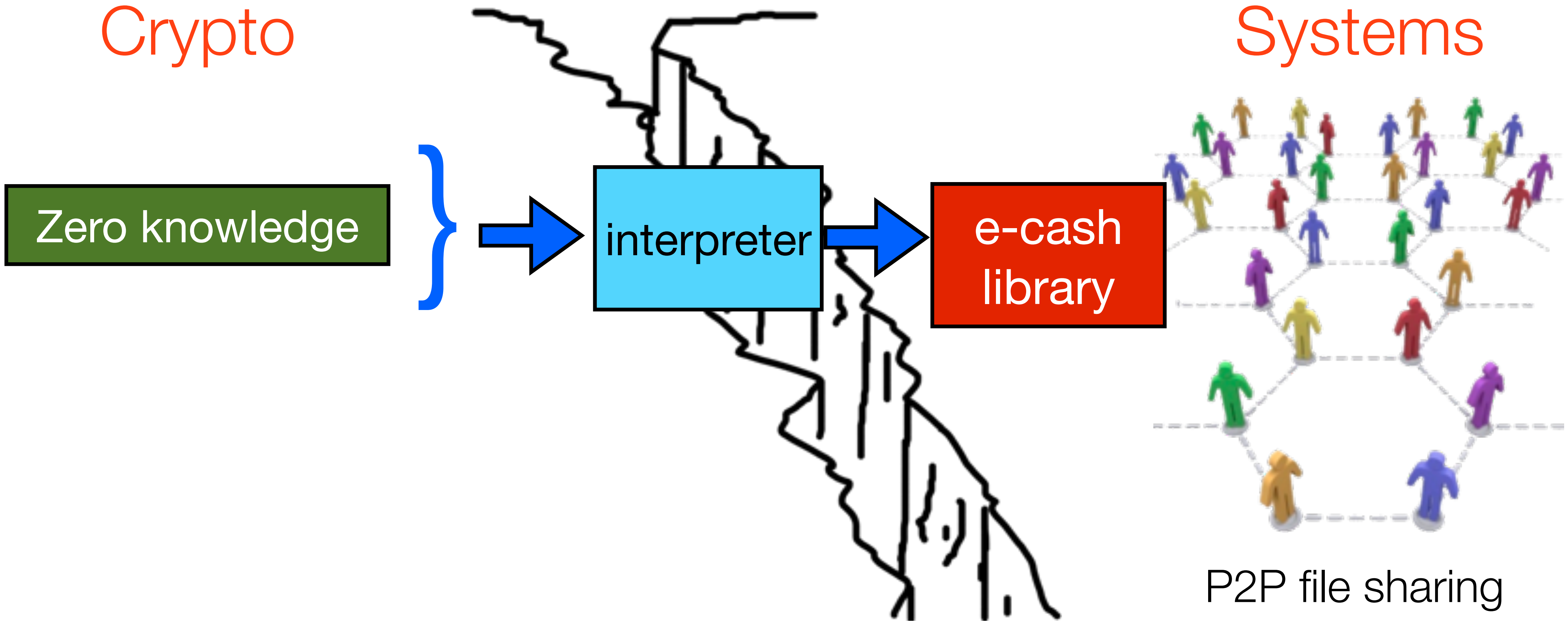
Systems

P2P file sharing

How can we lighten the implementation load?

- Design a language: ZKPDL (Zero Knowledge Proof Description Language)
- Build an interpreter to automatically translate from ZKPDL to proofs

Implementing zero knowledge (take 2)



Crypto

Systems

Zero knowledge

interpreter

e-cash library

P2P file sharing

How can we lighten the implementation load?

- Design a language: ZKPDL (Zero Knowledge Proof Description Language)
- Build an interpreter to automatically translate from ZKPDL to proofs

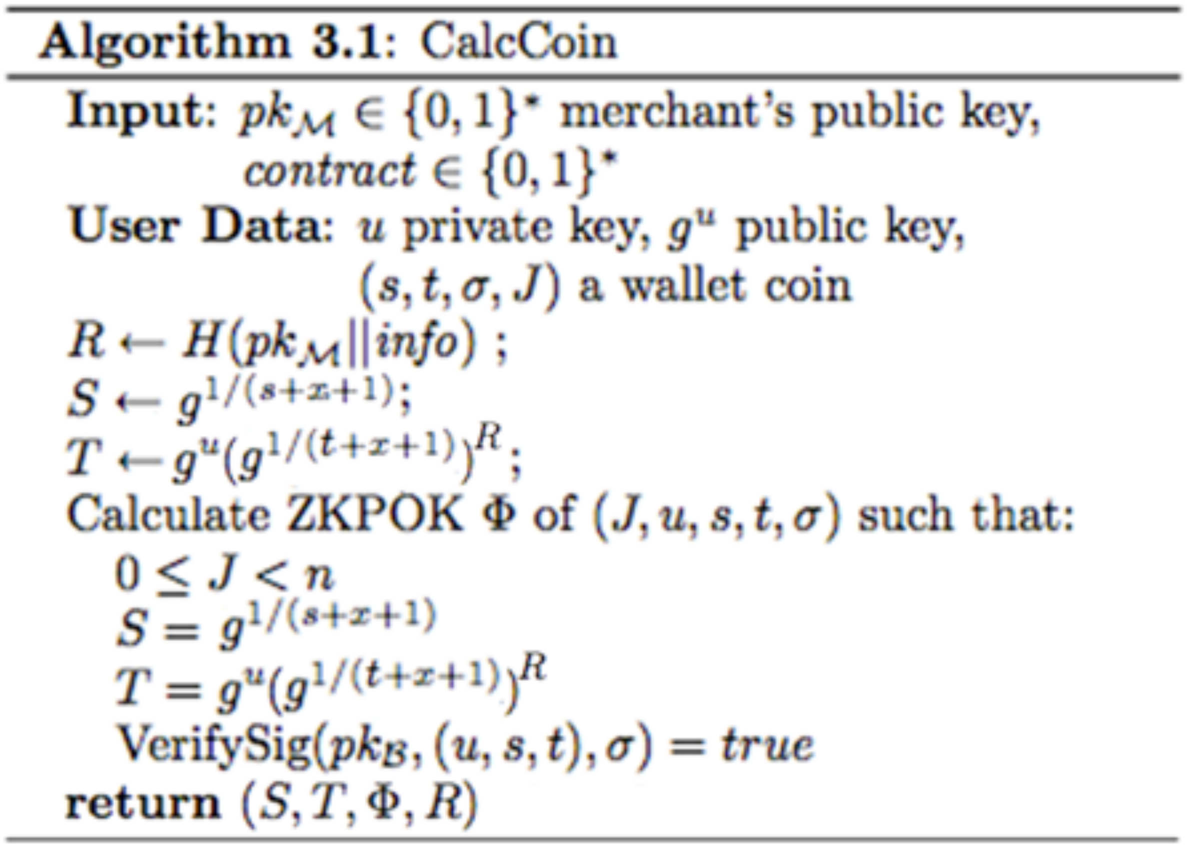
Step 1: writing programs in ZKPDL

High-level language, goal was to mirror theoretical descriptions

Step 1: writing programs in ZKPDL

High-level language, goal was to mirror theoretical descriptions

Description in paper

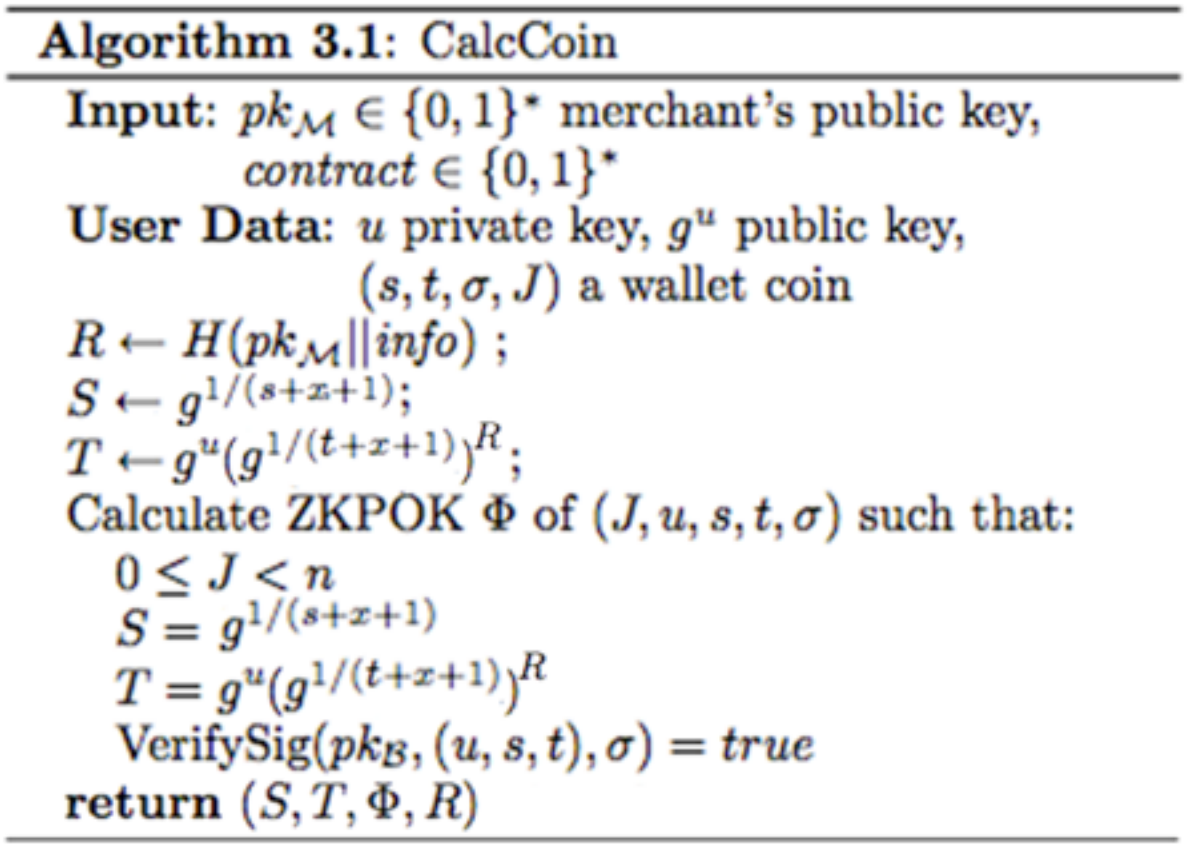


```
Algorithm 3.1: CalcCoin
  Input: $pk_{\mathcal{M}} \in \{0,1\}^*$ merchant's public key,
         $contract \in \{0,1\}^*$
  User Data: $u$ private key, $g^u$ public key,
             $(s, t, \sigma, J)$ a wallet coin
  $R \leftarrow H(pk_{\mathcal{M}} || info)$ ;
  $S \leftarrow g^{1/(s+x+1)}$;
  $T \leftarrow g^u (g^{1/(t+x+1)})^R$;
  Calculate ZKPOK $\Phi$ of $(J, u, s, t, \sigma)$ such that:
    $0 \leq J < n$
    $S = g^{1/(s+x+1)}$
    $T = g^u (g^{1/(t+x+1)})^R$
    $\mathrm{VerifySig}(pk_{\mathcal{B}}, (u, s, t), \sigma) = true$
  return $(S, T, \Phi, R)$
```

Step 1: writing programs in ZKPDL

High-level language, goal was to mirror theoretical descriptions

Description in paper



```
Algorithm 3.1: CalcCoin
  Input: $pk_{\mathcal{M}} \in \{0,1\}^*$ merchant's public key,
         $contract \in \{0,1\}^*$
  User Data: $u$ private key, $g^u$ public key,
             $(s, t, \sigma, J)$ a wallet coin
  $R \leftarrow H(pk_{\mathcal{M}} || info)$ ;
  $S \leftarrow g^{1/(s+x+1)}$;
  $T \leftarrow g^u(g^{1/(t+x+1)})^R$;
  Calculate ZKPOK $\Phi$ of $(J, u, s, t, \sigma)$ such that:
    $0 \leq J < n$
    $S = g^{1/(s+x+1)}$
    $T = g^u(g^{1/(t+x+1)})^R$
    $\mathrm{VerifySig}(pk_{\mathcal{B}}, (u, s, t), \sigma) = true$
  return $(S, T, \Phi, R)$
```

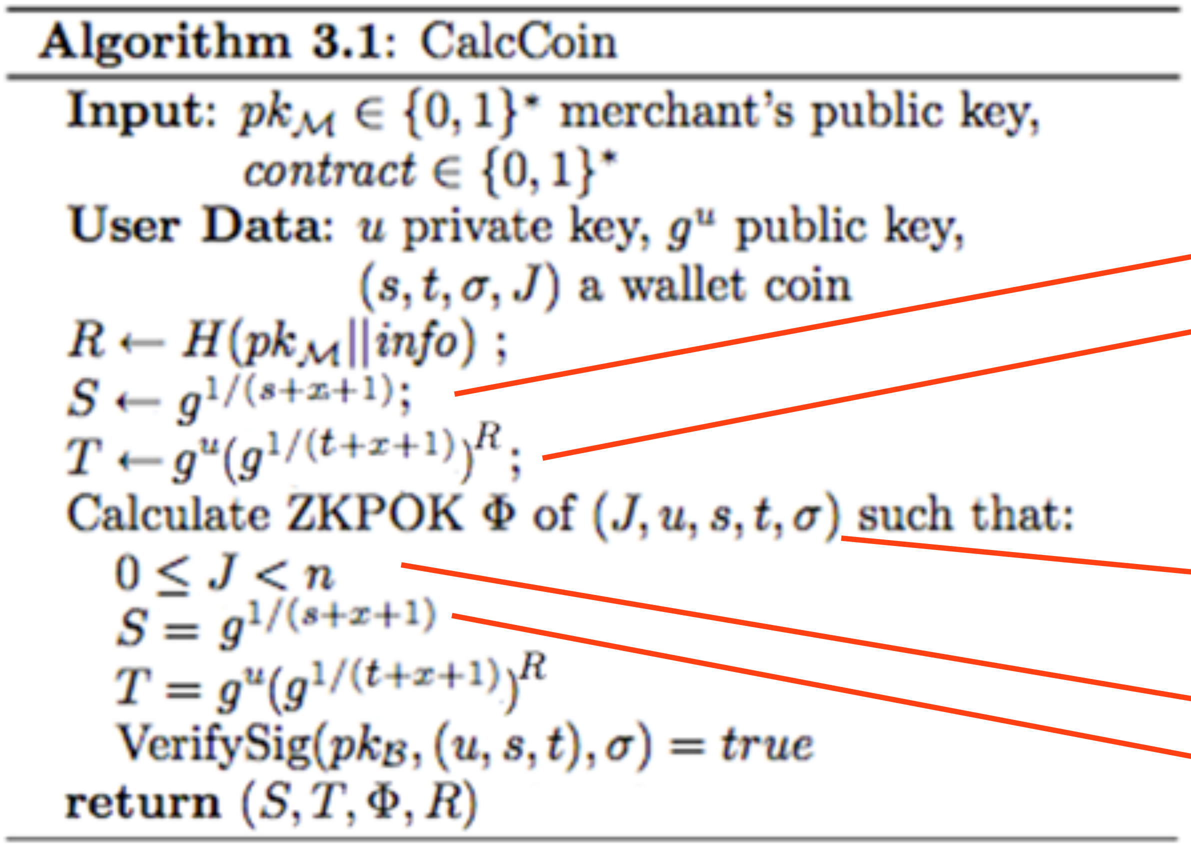
Description in ZKPDL

```
computation:
  ...
  compute:
    S := g^(1/(s+x+1))
    T := g^u * (g^(1/(t+x+1)))^R
proof:
  given:
    group: G = <g,h>
    elements in G: S, T
  prove knowledge of:
    exponents in G: u,s,t,x
    integer: J
  such that:
    range: 0 <= J < n
    S = g^(1/(s+x+1))
```

Step 1: writing programs in ZKPDL

High-level language, goal was to mirror theoretical descriptions

Description in paper



```
Algorithm 3.1: CalcCoin
  Input: $pk_{\mathcal{M}} \in \{0,1\}^*$ merchant's public key,
         $contract \in \{0,1\}^*$
  User Data: $u$ private key, $g^u$ public key,
             $(s,t,\sigma,J)$ a wallet coin
  $R \leftarrow H(pk_{\mathcal{M}}||info)$ ;
  $S \leftarrow g^{1/(s+x+1)}$;
  $T \leftarrow g^u(g^{1/(t+x+1)})^R$;
  Calculate ZKPOK $\Phi$ of $(J,u,s,t,\sigma)$ such that:
    $0 \leq J < n$
    $S = g^{1/(s+x+1)}$
    $T = g^u(g^{1/(t+x+1)})^R$
    VerifySig$(pk_{\mathcal{B}},(u,s,t),\sigma) = true$
  return $(S,T,\Phi,R)$
```

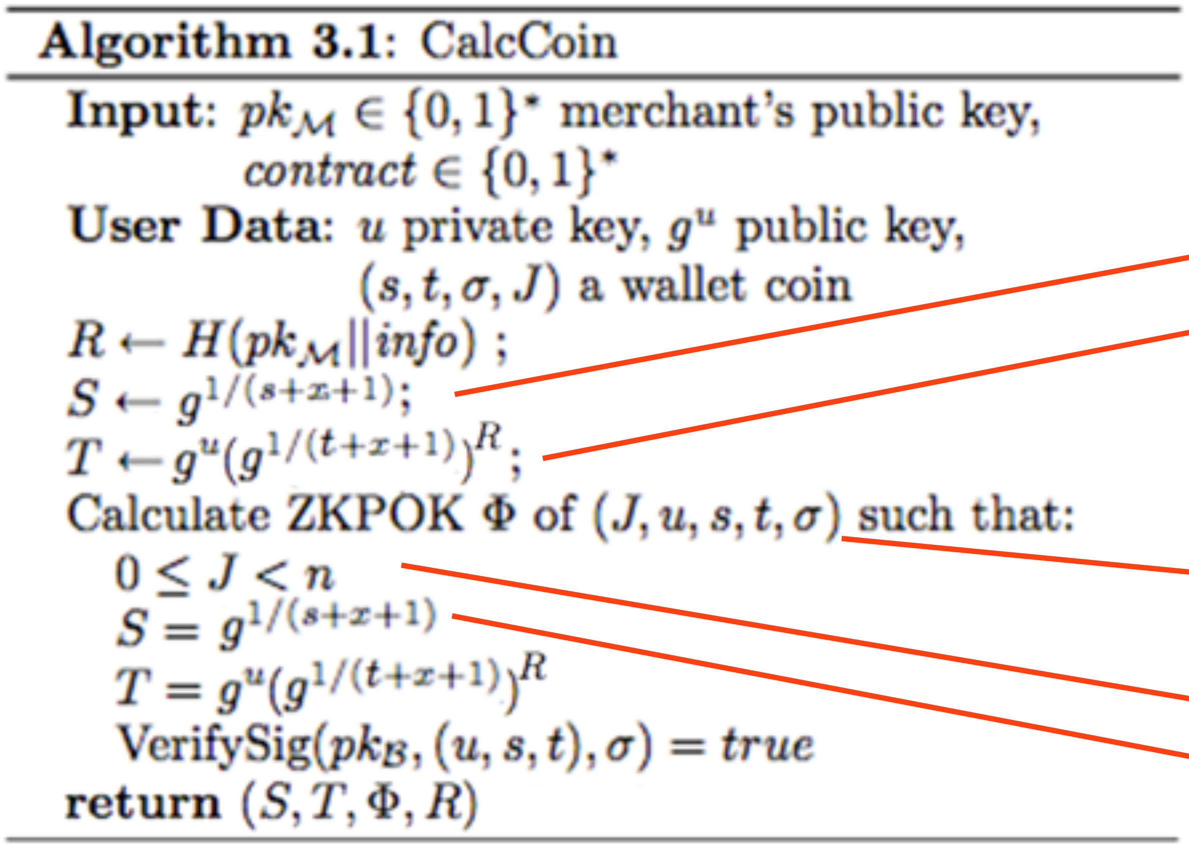
Description in ZKPDL

```
computation:
  ...
  compute:
    S := g^(1/(s+x+1))
    T := g^u * (g^(1/(t+x+1)))^R
proof:
  given:
    group: G = <g,h>
    elements in G: S, T
  prove knowledge of:
    exponents in G: u,s,t,x
    integer: J
  such that:
    range: 0 <= J < n
    S = g^(1/(s+x+1))
```

Step 1: writing programs in ZKPDL

High-level language, goal was to mirror theoretical descriptions

Description in paper



```
Algorithm 3.1: CalcCoin
  Input: $pk_{\mathcal{M}} \in \{0,1\}^*$ merchant's public key,
         $contract \in \{0,1\}^*$
  User Data: $u$ private key, $g^u$ public key,
             $(s, t, \sigma, J)$ a wallet coin
  $R \leftarrow H(pk_{\mathcal{M}} || info)$ ;
  $S \leftarrow g^{1/(s+x+1)}$;
  $T \leftarrow g^u (g^{1/(t+x+1)})^R$;
  Calculate ZKPOK $\Phi$ of $(J, u, s, t, \sigma)$ such that:
    $0 \leq J < n$
    $S = g^{1/(s+x+1)}$
    $T = g^u (g^{1/(t+x+1)})^R$
    $\mathrm{VerifySig}(pk_{\mathcal{B}}, (u, s, t), \sigma) = true$
  return $(S, T, \Phi, R)$
```

Description in ZKPDL

```
computation:
  ...
  compute:
    S := g^(1/(s+x+1))
    T := g^u * (g^(1/(t+x+1)))^R
proof:
  given:
    group: G = <g,h>
    elements in G: S, T
  prove knowledge of:
    exponents in G: u,s,t,x
    integer: J
  such that:
    range: 0 <= J < n
    S = g^(1/(s+x+1))
```

Currently support four ZKP types, enough for vast majority of applications

Should also be easy to add new types if they're needed

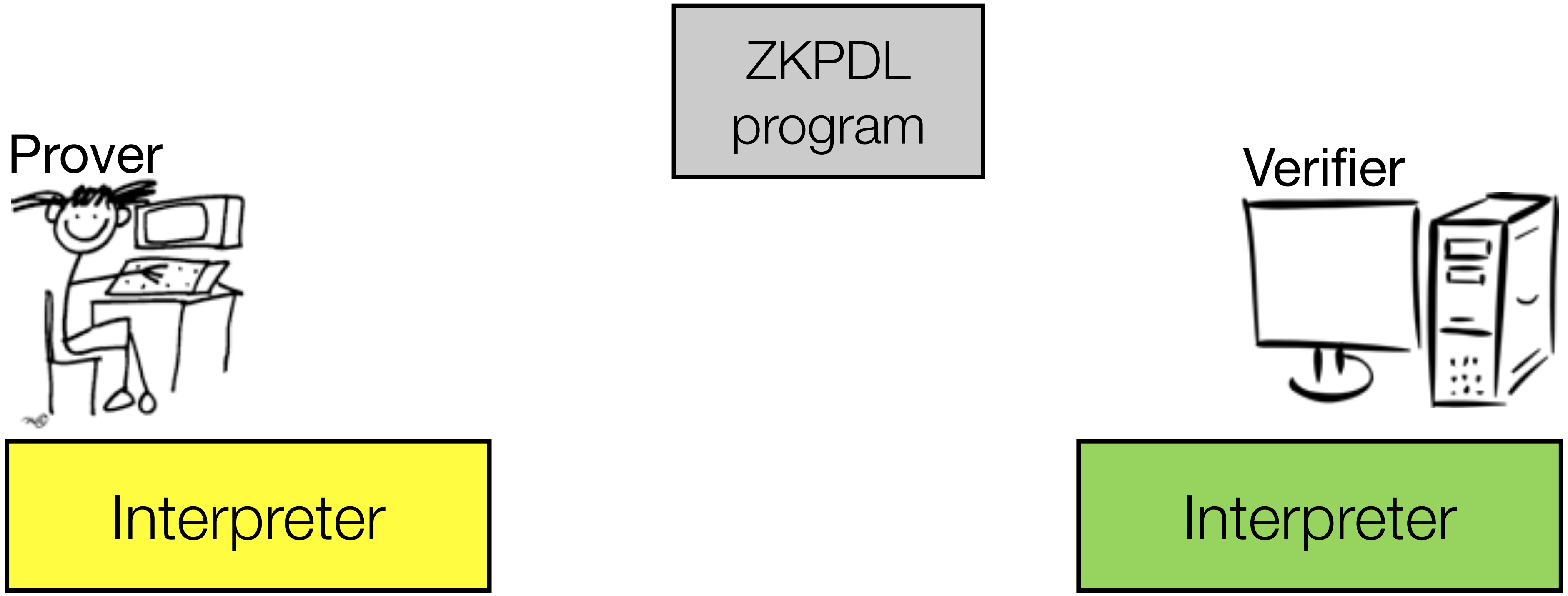
Sample usage of the interpreter

Prover

I have access credentials

Verifier

Sample usage of the interpreter



ZKPDL program

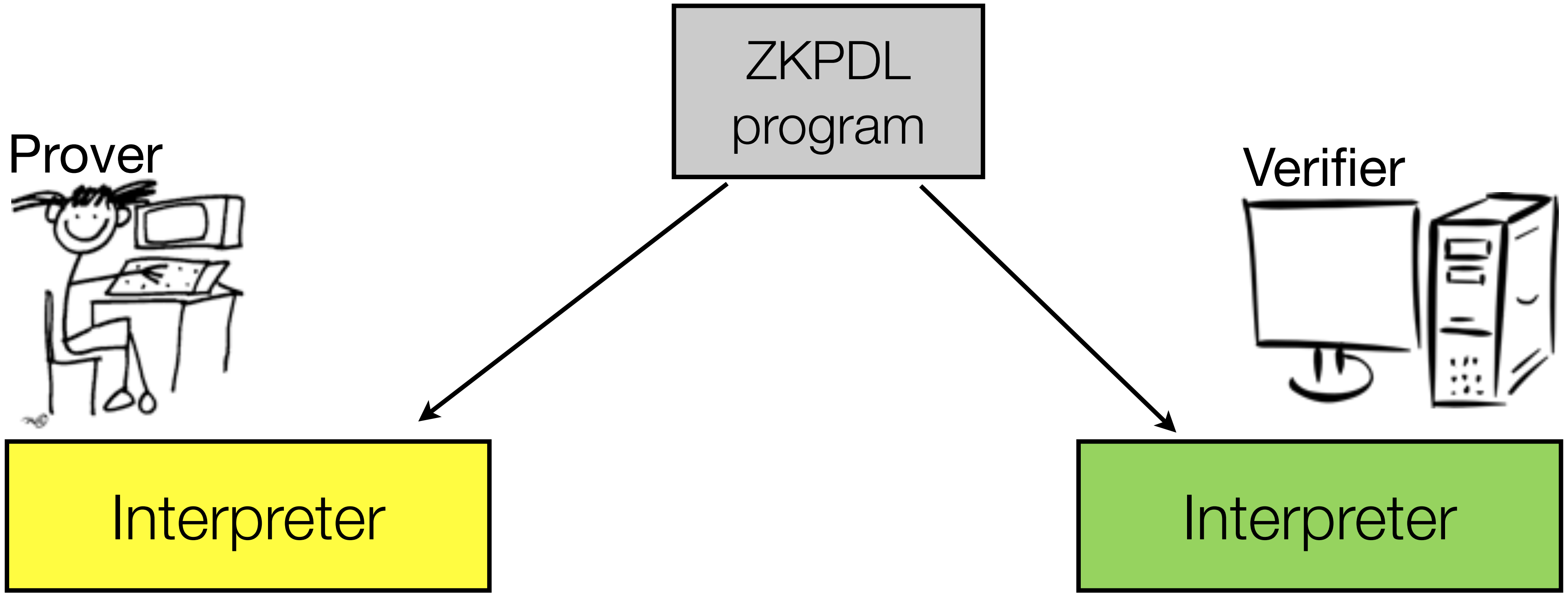
Prover

Verifier

Interpreter

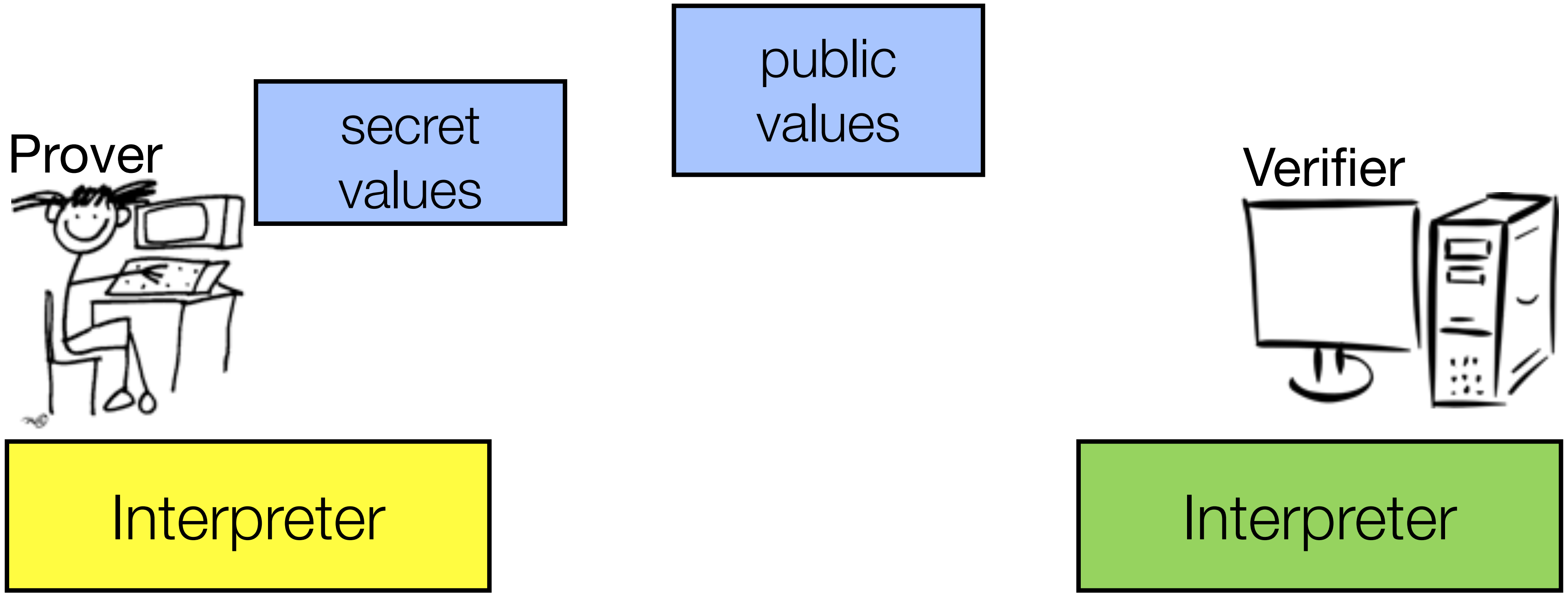
Interpreter

Sample usage of the interpreter



- At compile time, check program syntax, types, etc.

Sample usage of the interpreter



Prover

secret values

public values

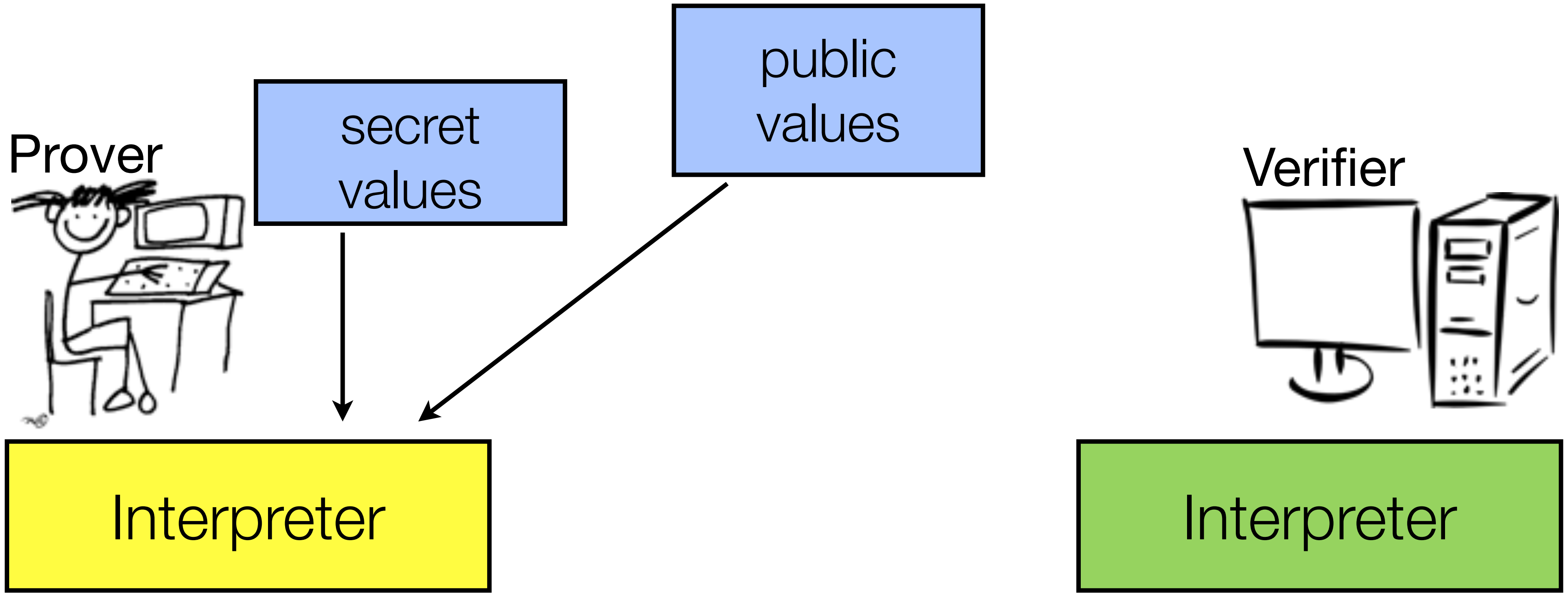
Verifier

Interpreter

Interpreter

- At compile time, check program syntax, types, etc.
- At run time, need all values to be proved

Sample usage of the interpreter



Prover

secret values

public values

Verifier

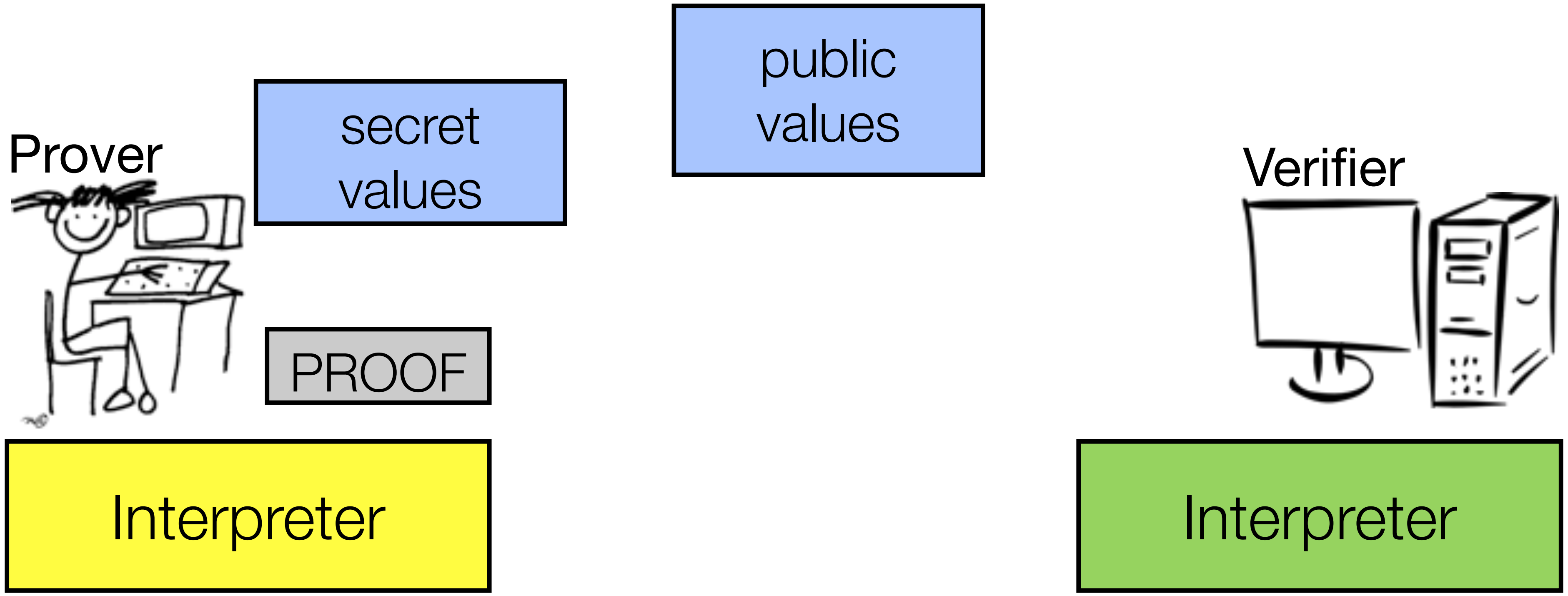
Interpreter

Interpreter

- At compile time, check program syntax, types, etc.

- At run time, need all values to be proved

Sample usage of the interpreter



Prover

secret values

public values

Verifier

PROOF

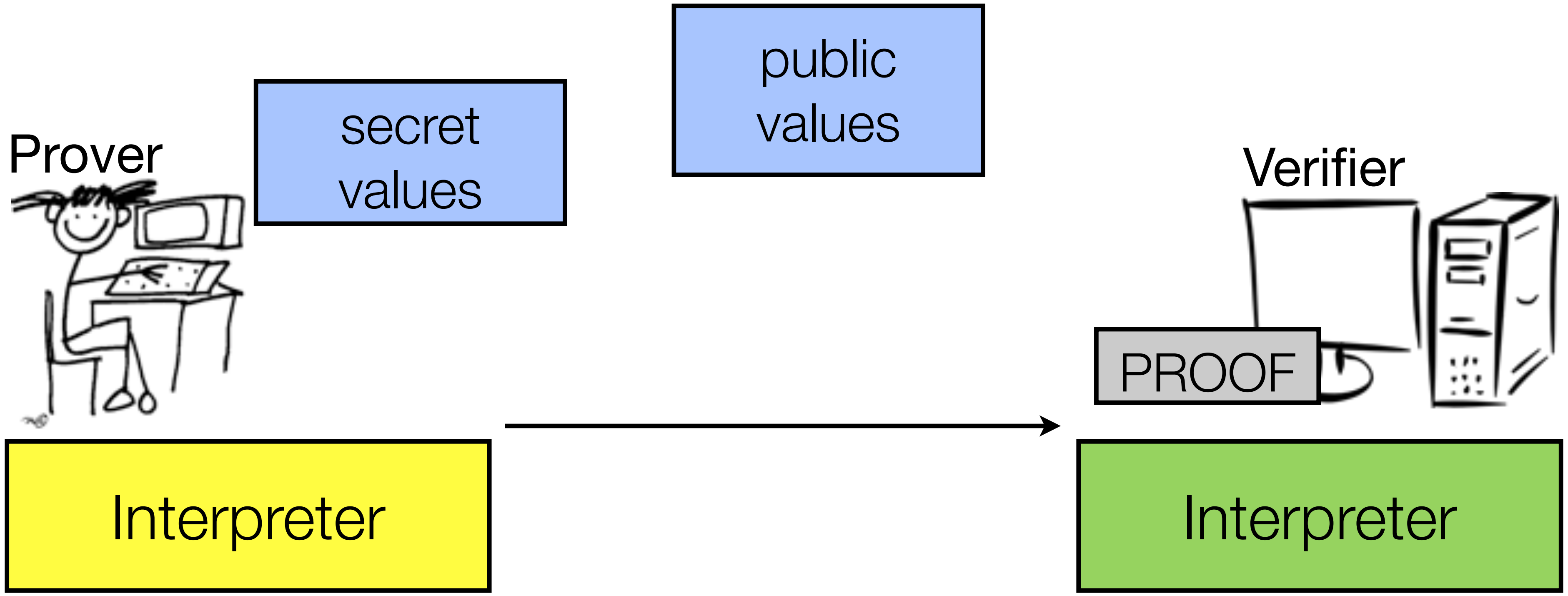
Interpreter

Interpreter

- At compile time, check program syntax, types, etc.

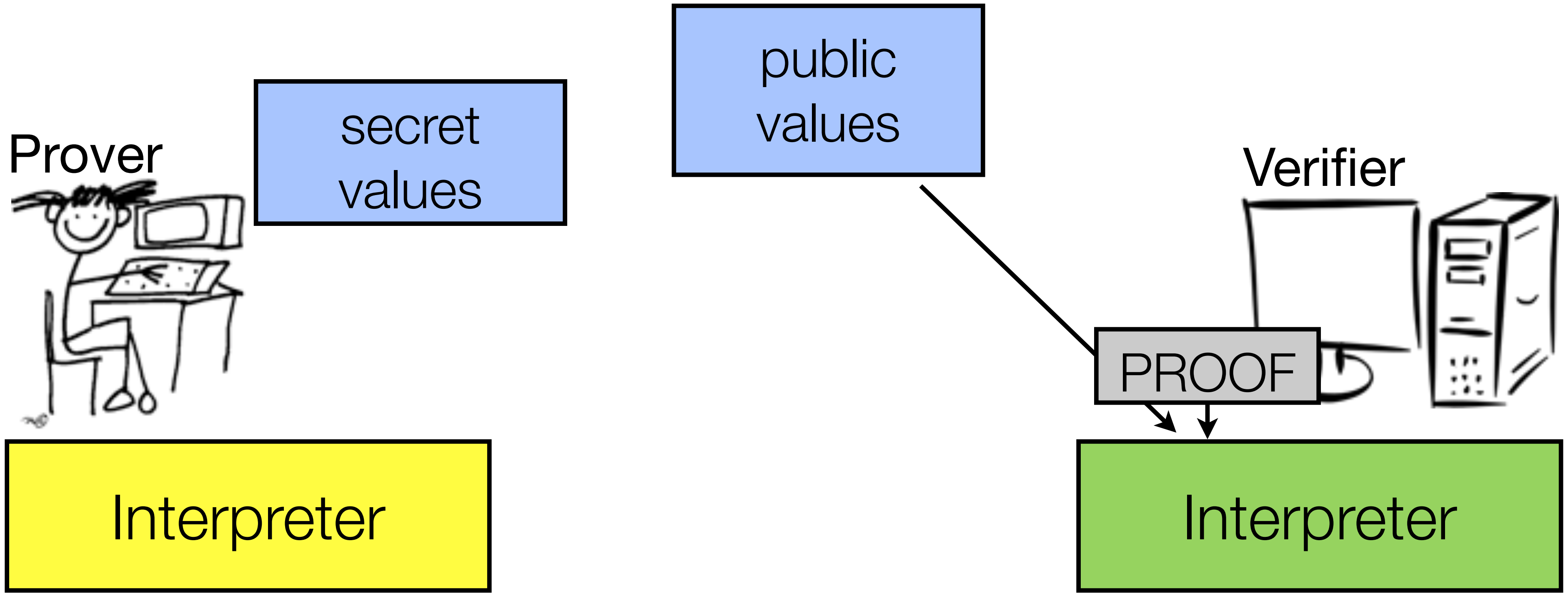
- At run time, need all values to be proved

Sample usage of the interpreter



- At compile time, check program syntax, types, etc.
- At run time, need all values to be proved

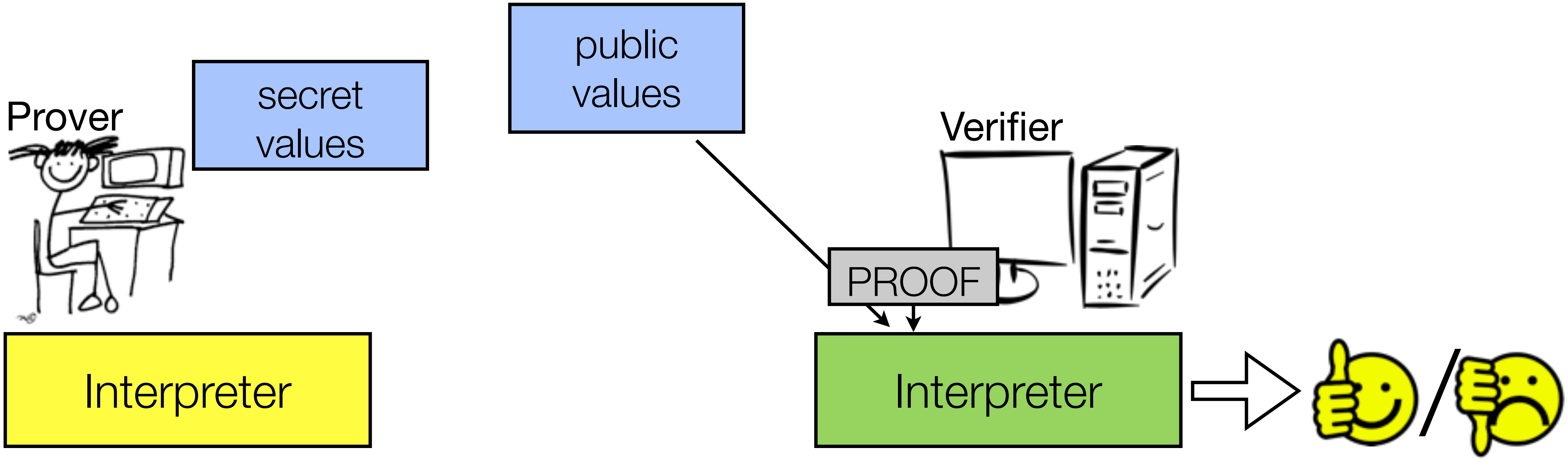
Sample usage of the interpreter



- At compile time, check program syntax, types, etc.

- At run time, need all values to be proved

Sample usage of the interpreter



- At compile time, check program syntax, types, etc.
- At run time, need all values to be proved

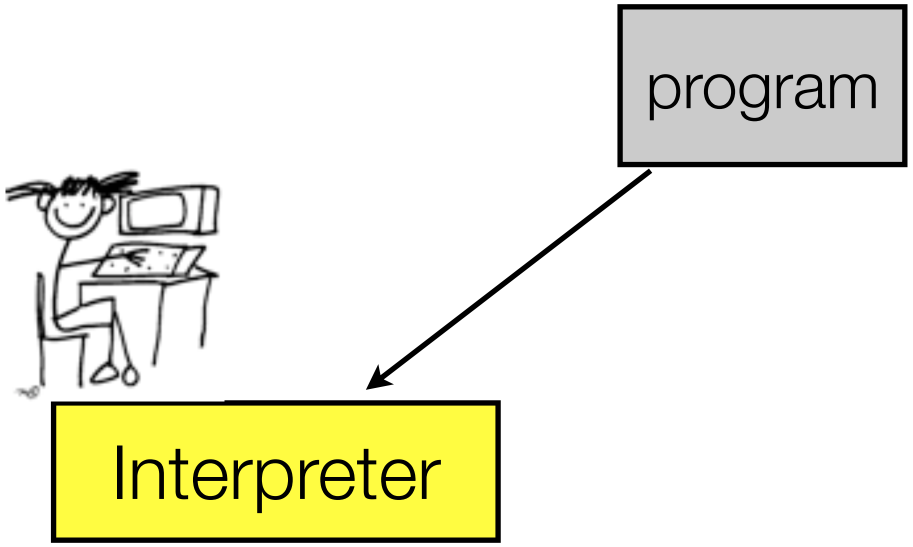
Step 2: using the interpreter to write a library

Step 2: using the interpreter to write a library

Use simple procedure to create wrapper classes for interpreter

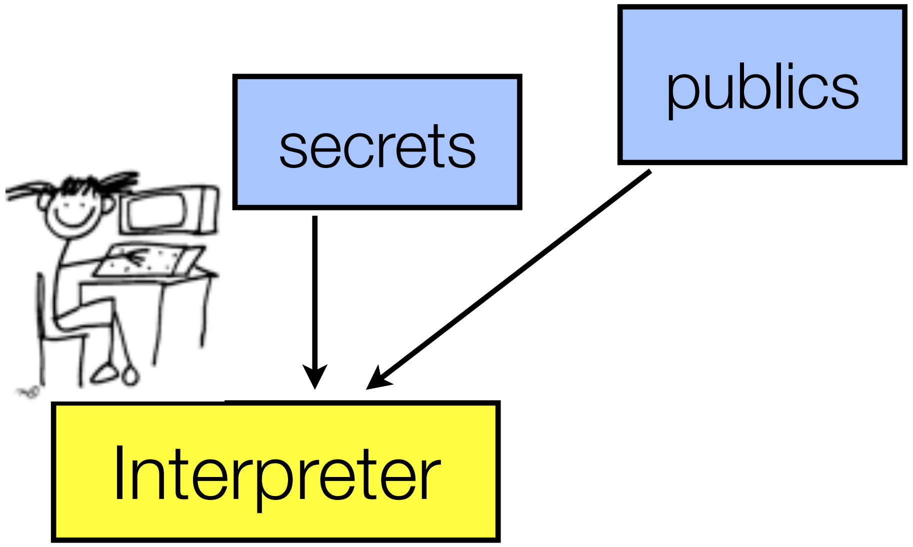
Step 2: using the interpreter to write a library

Use simple procedure to create wrapper classes for interpreter



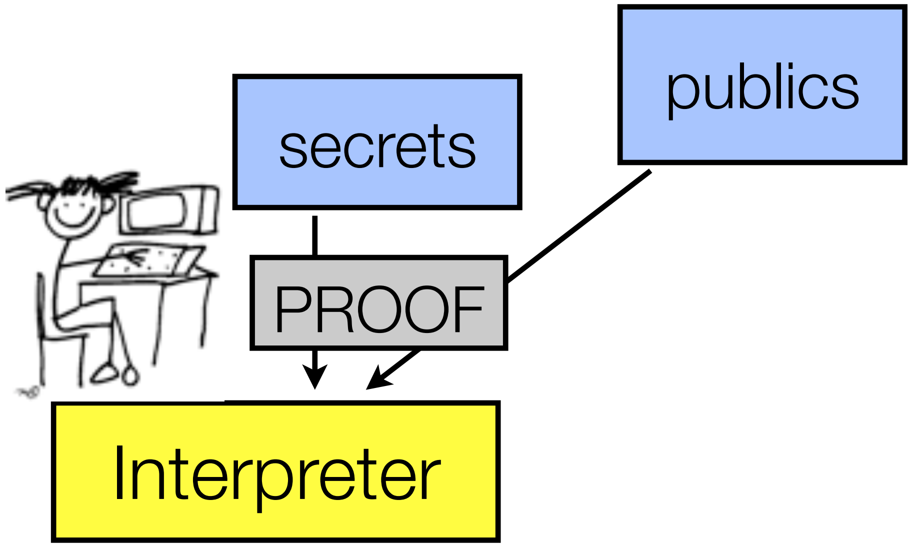
Step 2: using the interpreter to write a library

Use simple procedure to create wrapper classes for interpreter



Step 2: using the interpreter to write a library

Use simple procedure to create wrapper classes for interpreter



Step 2: using the interpreter to write a library

Use simple procedure to create wrapper classes for interpreter

```
Proof MyZKP::prove(group_map g, variable_map v,
                   string program) {
   InterpreterProver p;
   p.check(program);
   p.compute(g,v);
   return p.prove();
}
```

Step 2: using the interpreter to write a library

Use simple procedure to create wrapper classes for interpreter

```
Proof MyZKP::prove(group_map g, variable_map v,
                   string program) {
   InterpreterProver p;
-→ p.check(program);
   p.compute(g,v);
   return p.prove();
}
```

- Specify crypto protocol of choice in the `program` string

Step 2: using the interpreter to write a library

Use simple procedure to create wrapper classes for interpreter

```
Proof MyZKP::prove(group_map g, variable_map v,
                   string program) {
   InterpreterProver p;
→  p.check(program);
→  p.compute(g,v);
→  return p.prove();
}
```

- Specify crypto protocol of choice in the `program` string
- Feed numeric values in and you’re done!

Step 2: using the interpreter to write a library

Use simple procedure to create wrapper classes for interpreter

```
Proof MyZKP::prove(group_map g, variable_map v,
                   string program) {
   InterpreterProver p;
→  p.check(program);
→  p.compute(g,v);
→  return p.prove();
}
```

- Specify crypto protocol of choice in the `program` string

- Feed numeric values in and you’re done!

Solves issues of reusability and of time

Step 2: using the interpreter to write a library

Use simple procedure to create wrapper classes for interpreter

```
Proof MyZKP::prove(group_map g, variable_map v,
                   string program) {
   InterpreterProver p;
 → p.check(program);
 → p.compute(g,v);
 → return p.prove();
}
```

- Specify crypto protocol of choice in the `program` string
- Feed numeric values in and you’re done!

Solves issues of reusability and of time

Took 3-4 months to build interpreter, then one month to reconstruct library

Optimizations: caching

In addition to usability, can achieve improvements in efficiency

Optimizations: caching

In addition to usability, can achieve improvements in efficiency

Have optimizations built into the interpreter

Optimizations: caching

In addition to usability, can achieve improvements in efficiency

Have optimizations built into the interpreter

- Cache powers of bases used for modular exponentiation

  Often have `g^x*h^r mod N`, numbers are 1000 bits long!

  Use common single- and multi-exponentiation techniques

Optimizations: caching

In addition to usability, can achieve improvements in efficiency

Have optimizations built into the interpreter

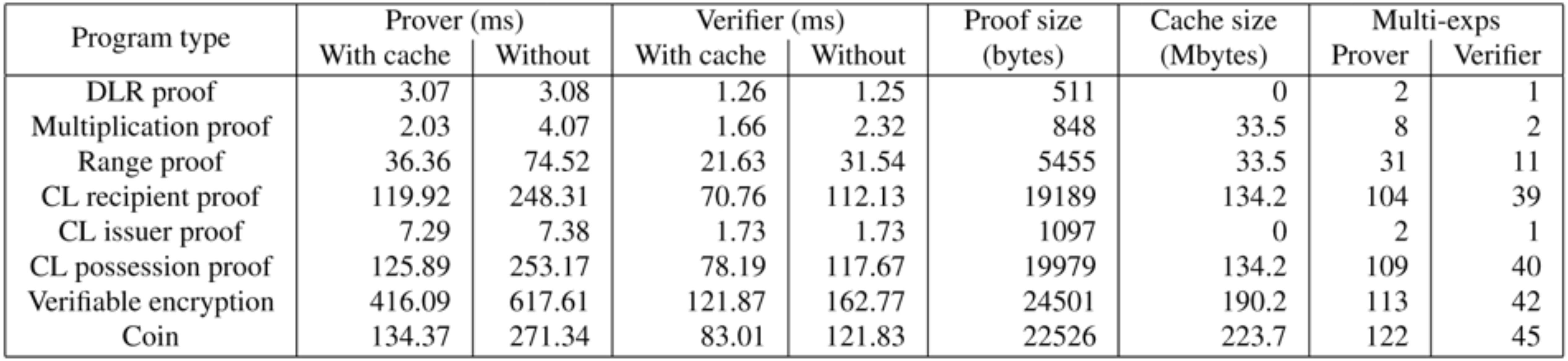
- Cache powers of bases used for modular exponentiation

  Often have `g^x*h^r mod N`, numbers are 1000 bits long!

  Use common single- and multi-exponentiation techniques

- Save copy of interpreter state after compilation

Did caching help?

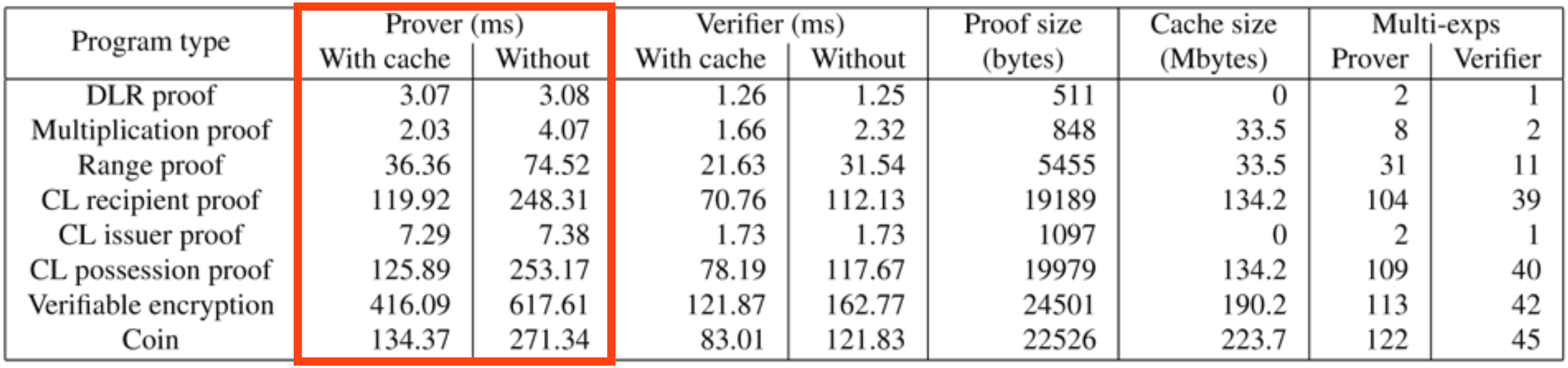


Program type	Prover (ms)		Verifier (ms)		Proof size	Cache size	Multi-exps	
	With cache	Without	With cache	Without	(bytes)	(Mbytes)	Prover	Verifier
DLR proof	3.07	3.08	1.26	1.25	511	0	2	1
Multiplication proof	2.03	4.07	1.66	2.32	848	33.5	8	2
Range proof	36.36	74.52	21.63	31.54	5455	33.5	31	11
CL recipient proof	119.92	248.31	70.76	112.13	19189	134.2	104	39
CL issuer proof	7.29	7.38	1.73	1.73	1097	0	2	1
CL possession proof	125.89	253.17	78.19	117.67	19979	134.2	109	40
Verifiable encryption	416.09	617.61	121.87	162.77	24501	190.2	113	42
Coin	134.37	271.34	83.01	121.83	22526	223.7	122	45

On the prover side, saw about a 50% speed-up using all optimizations

On the verifier side, about 30% (less computation)

Did caching help?

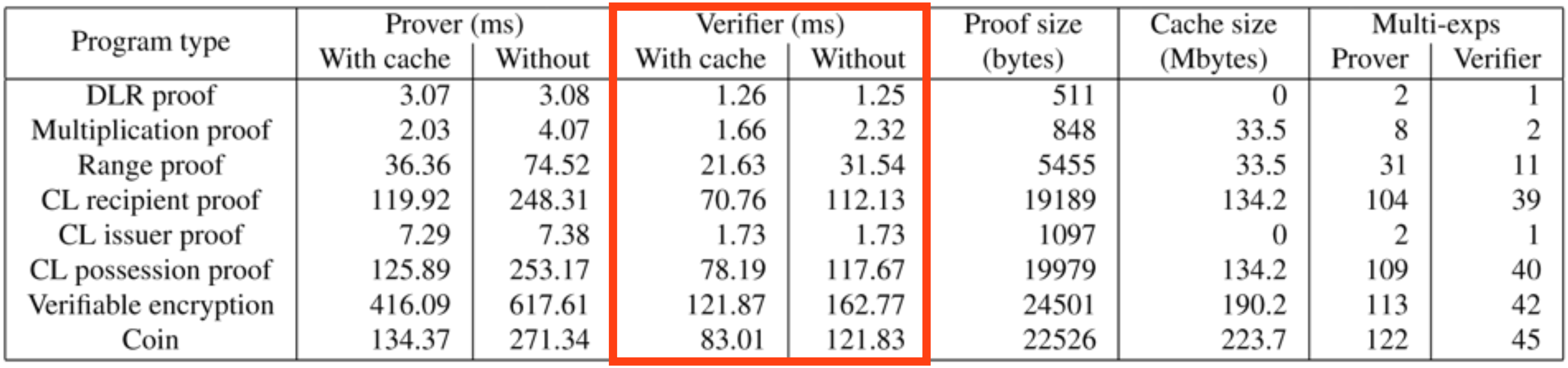


Program type	Prover (ms)		Verifier (ms)		Proof size	Cache size	Multi-exps	
	With cache	Without	With cache	Without	(bytes)	(Mbytes)	Prover	Verifier
DLR proof	3.07	3.08	1.26	1.25	511	0	2	1
Multiplication proof	2.03	4.07	1.66	2.32	848	33.5	8	2
Range proof	36.36	74.52	21.63	31.54	5455	33.5	31	11
CL recipient proof	119.92	248.31	70.76	112.13	19189	134.2	104	39
CL issuer proof	7.29	7.38	1.73	1.73	1097	0	2	1
CL possession proof	125.89	253.17	78.19	117.67	19979	134.2	109	40
Verifiable encryption	416.09	617.61	121.87	162.77	24501	190.2	113	42
Coin	134.37	271.34	83.01	121.83	22526	223.7	122	45

On the prover side, saw about a 50% speed-up using all optimizations

On the verifier side, about 30% (less computation)

Did caching help?

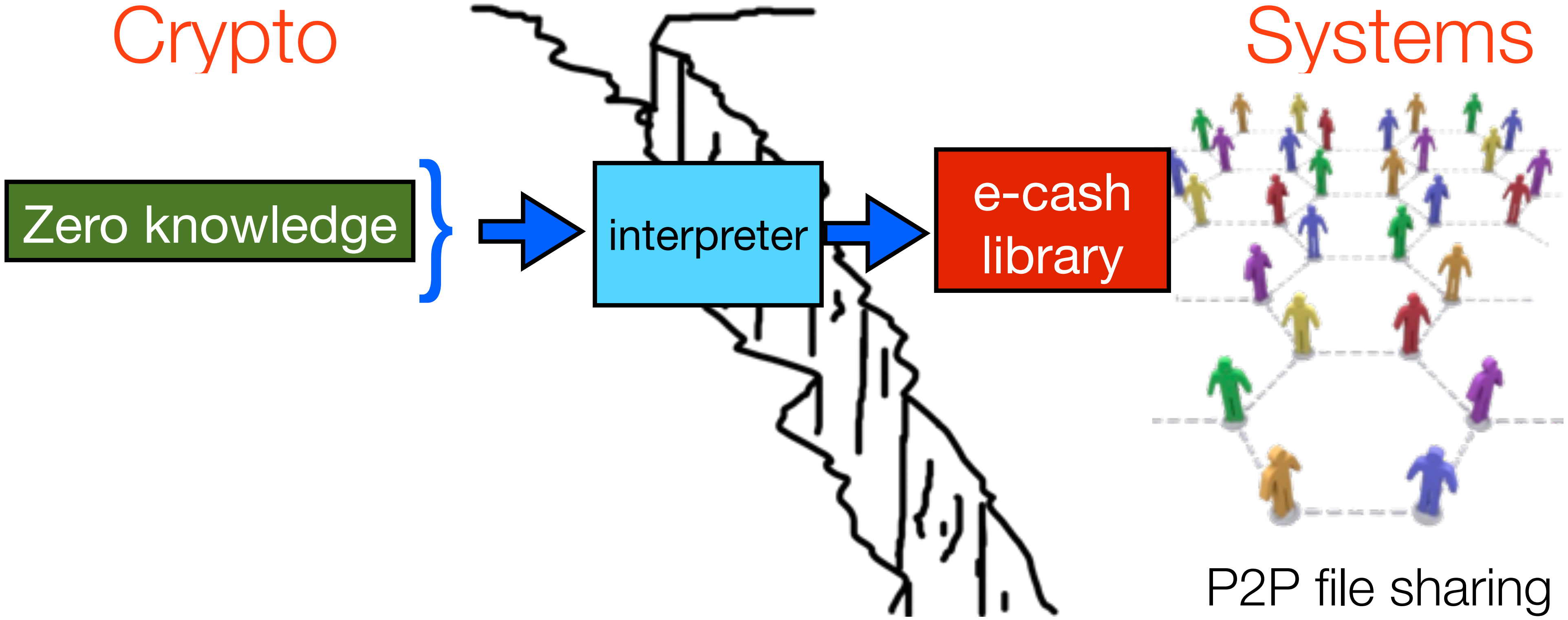


Program type	Prover (ms)		Verifier (ms)		Proof size	Cache size	Multi-exps	
	With cache	Without	With cache	Without	(bytes)	(Mbytes)	Prover	Verifier
DLR proof	3.07	3.08	1.26	1.25	511	0	2	1
Multiplication proof	2.03	4.07	1.66	2.32	848	33.5	8	2
Range proof	36.36	74.52	21.63	31.54	5455	33.5	31	11
CL recipient proof	119.92	248.31	70.76	112.13	19189	134.2	104	39
CL issuer proof	7.29	7.38	1.73	1.73	1097	0	2	1
CL possession proof	125.89	253.17	78.19	117.67	19979	134.2	109	40
Verifiable encryption	416.09	617.61	121.87	162.77	24501	190.2	113	42
Coin	134.37	271.34	83.01	121.83	22526	223.7	122	45

On the prover side, saw about a 50% speed-up using all optimizations

On the verifier side, about 30% (less computation)

Case study: using ZKPDL for e-cash



Crypto

Systems

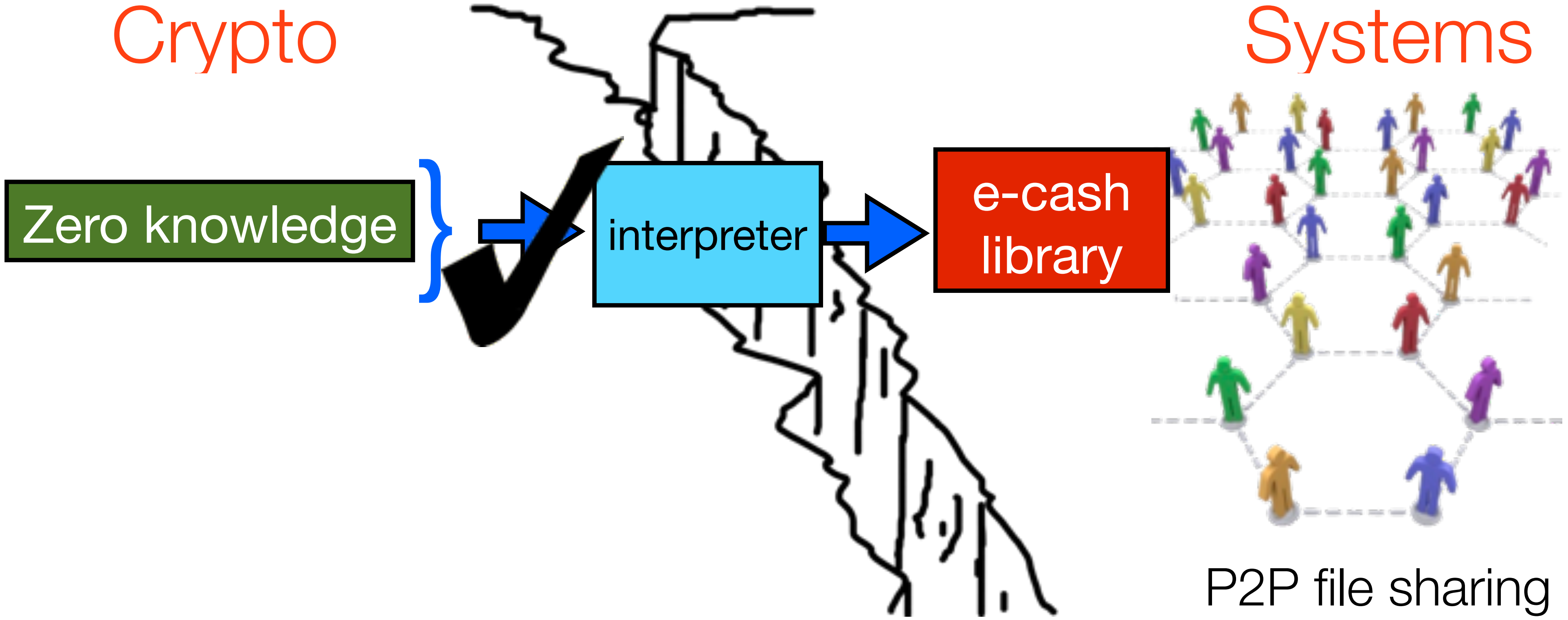
Zero knowledge

interpreter

e-cash library

P2P file sharing

Case study: using ZKPDL for e-cash



Crypto

Zero knowledge

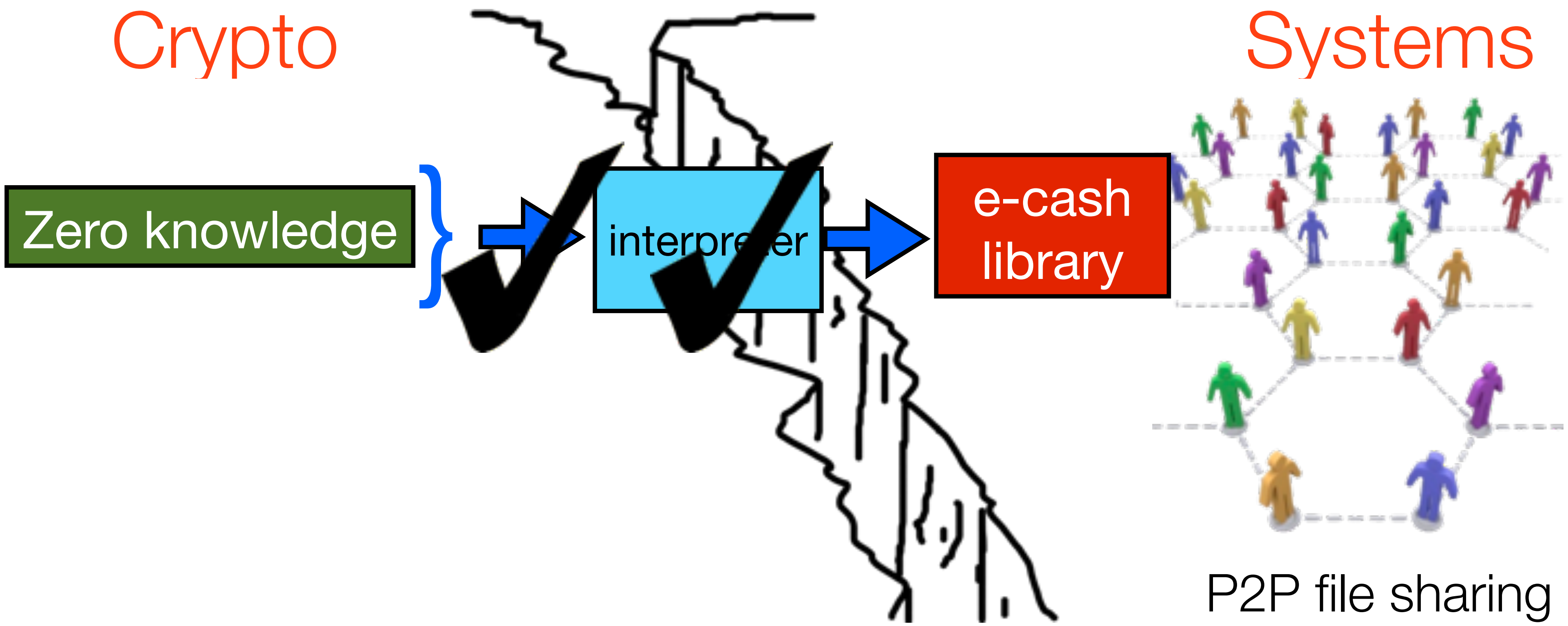
interpreter

e-cash library

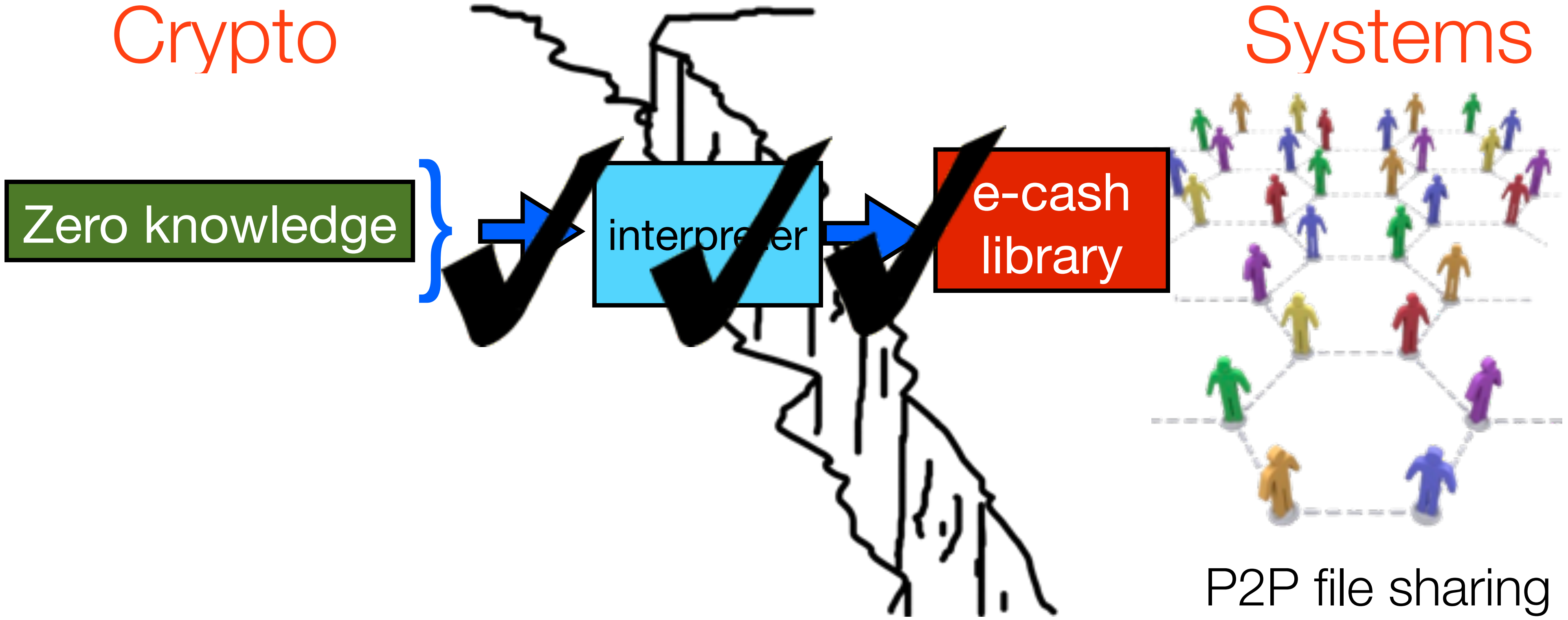
Systems

P2P file sharing

Case study: using ZKPDL for e-cash



Case study: using ZKPDL for e-cash



Crypto

Systems

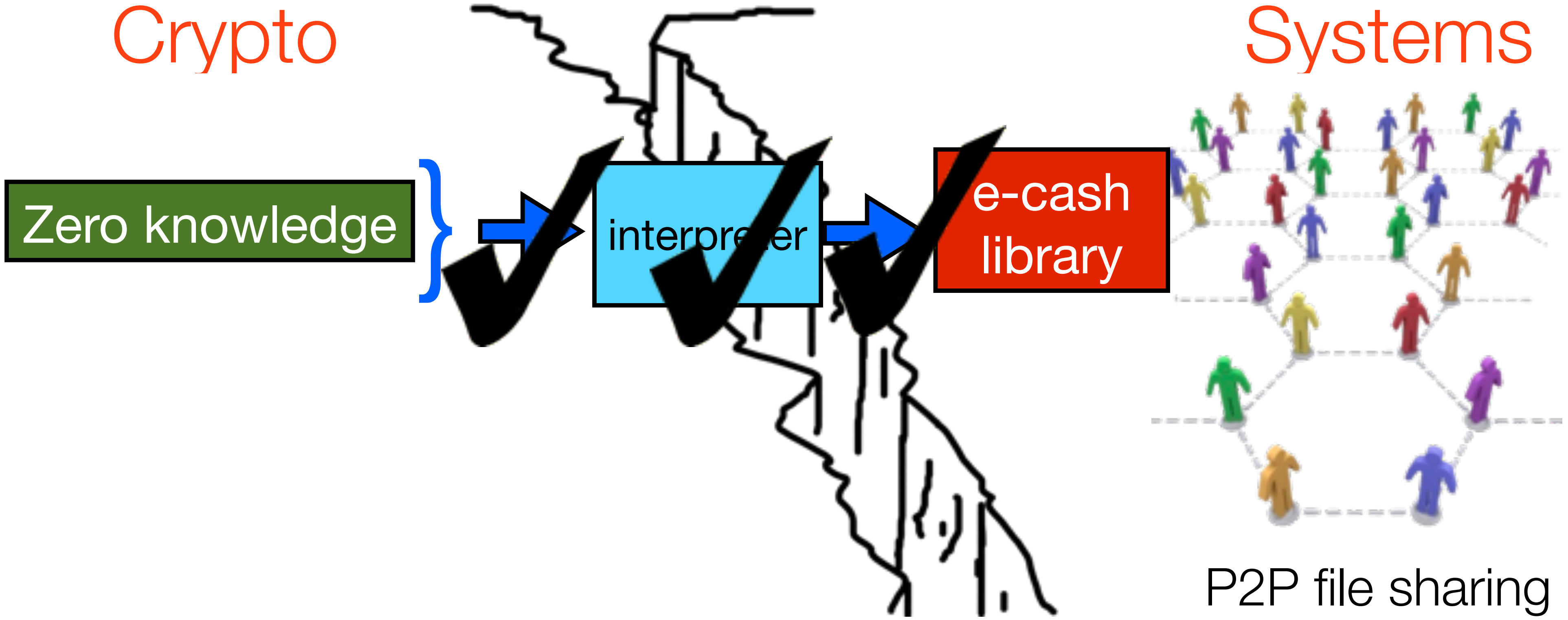
Zero knowledge

interpreter

e-cash library

P2P file sharing

Case study: using ZKPDL for e-cash



P2P file sharing

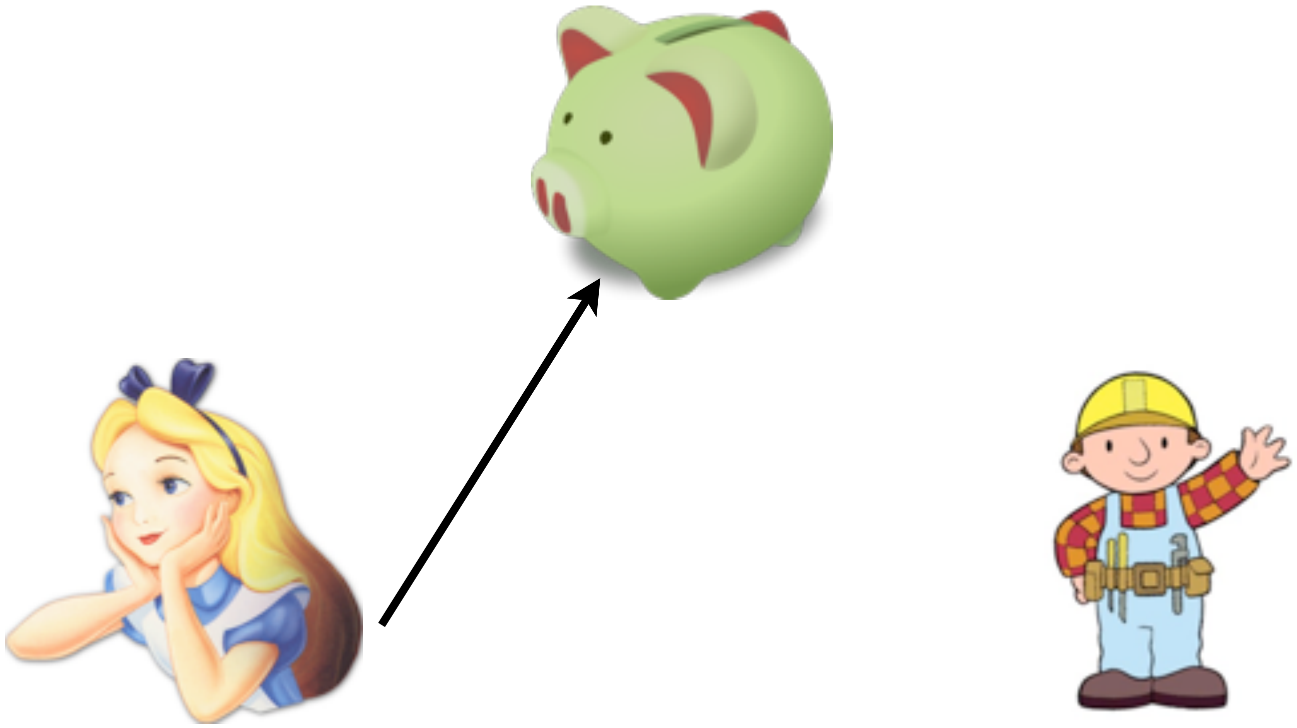
E-cash was originally developed [Ch82] as replacement for currency

Now, view e-cash in context of token systems

- Our usage in P2P file-sharing schemes [BCE+07]
- Provides anonymous transportation ticketing (future work)

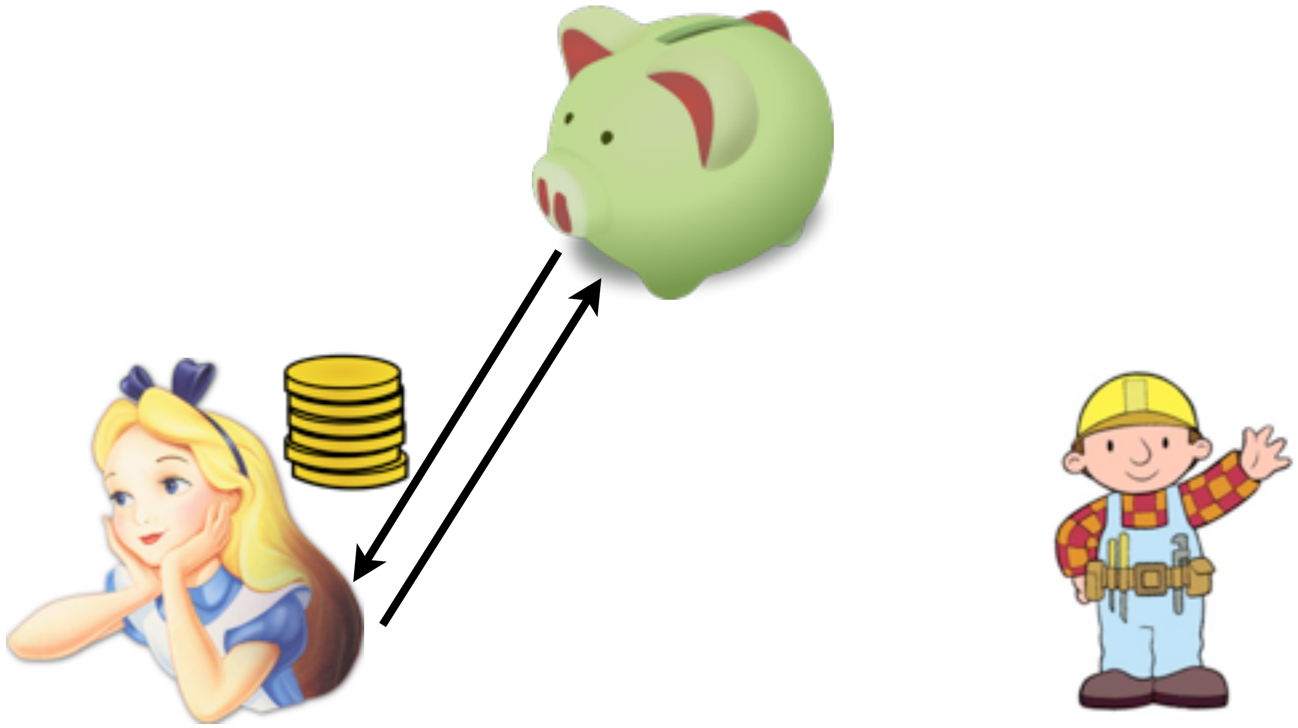
How e-cash works [Ch82, CHL05, CLM07]

How e-cash works [Ch82, CHL05, CLM07]



Withdraw: Alice gets coins from bank

How e-cash works [Ch82, CHL05, CLM07]



Withdraw: Alice gets coins from bank

How e-cash works [Ch82, CHL05, CLM07]

Withdraw: Alice gets coins from bank

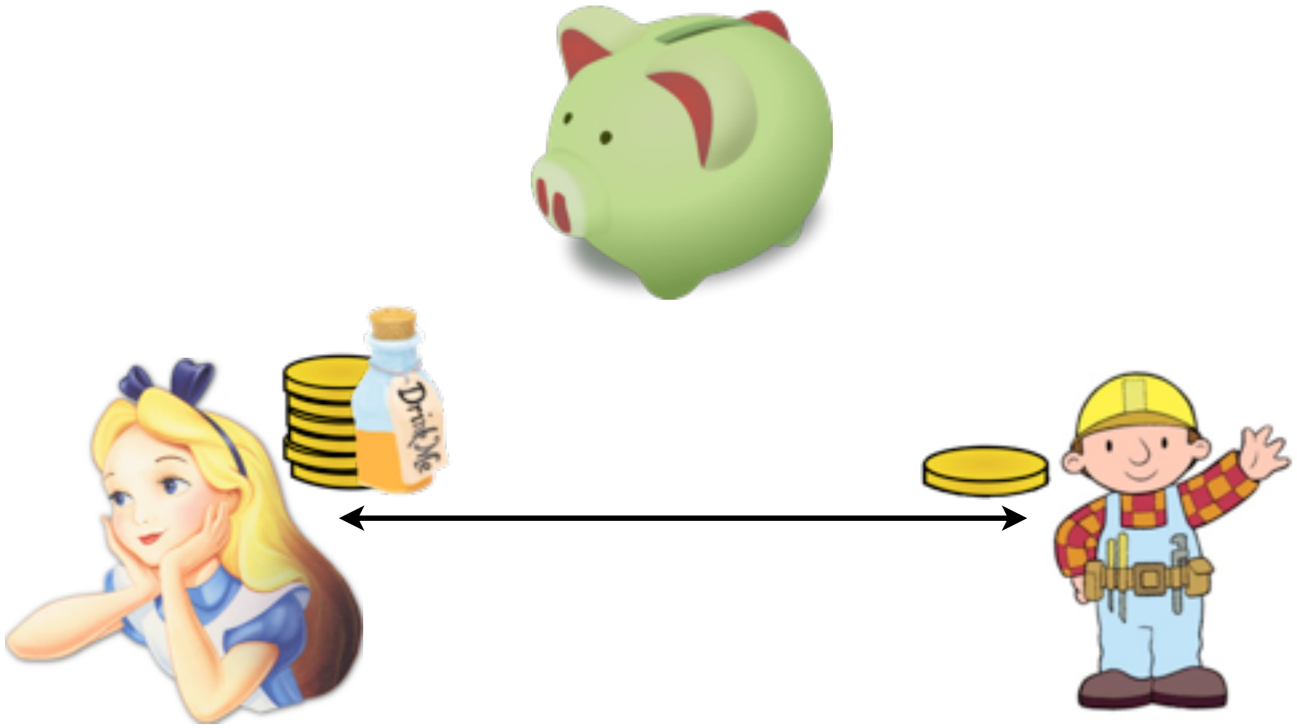
Buy: Alice gives Bob coin in exchange for her purchase

How e-cash works [Ch82, CHL05, CLM07]

Withdraw: Alice gets coins from bank

Buy: Alice gives Bob coin in exchange for her purchase

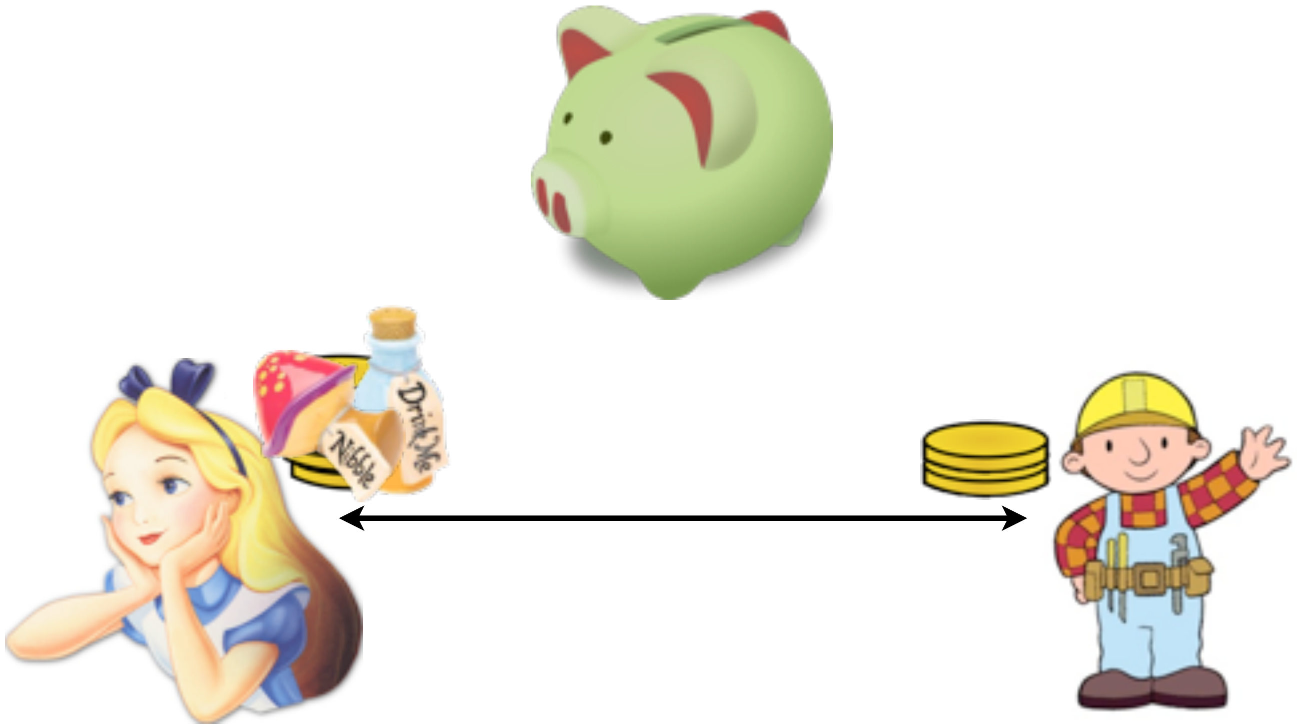
How e-cash works [Ch82, CHL05, CLM07]



Withdraw: Alice gets coins from bank

Buy: Alice gives Bob coin in exchange for her purchase

How e-cash works [Ch82, CHL05, CLM07]



Withdraw: Alice gets coins from bank

Buy: Alice gives Bob coin in exchange for her purchase

Unlinkability: if Alice spends twice, Bob won’t even know it’s the same person

How e-cash works [Ch82, CHL05, CLM07]

Withdraw: Alice gets coins from bank

Buy: Alice gives Bob coin in exchange for her purchase

Unlinkability: if Alice spends twice, Bob won't even know it's the same person

Deposit: Bob deposits these coins with the bank

How e-cash works [Ch82, CHL05, CLM07]

Withdraw: Alice gets coins from bank

Buy: Alice gives Bob coin in exchange for her purchase

Unlinkability: if Alice spends twice, Bob won’t even know it’s the same person

Deposit: Bob deposits these coins with the bank

How e-cash works [Ch82, CHL05, CLM07]

Nibble
Drink Me

Withdraw: Alice gets coins from bank

Buy: Alice gives Bob coin in exchange for her purchase

Unlinkability: if Alice spends twice, Bob won’t even know it’s the same person

Deposit: Bob deposits these coins with the bank

Untraceability: Bank cannot trace the deposited coins back to Alice

CashLib: integrating e-cash into a P2P system

CashLib: integrating e-cash into a P2P system

CashLib: integrating e-cash into a P2P system

Operations:

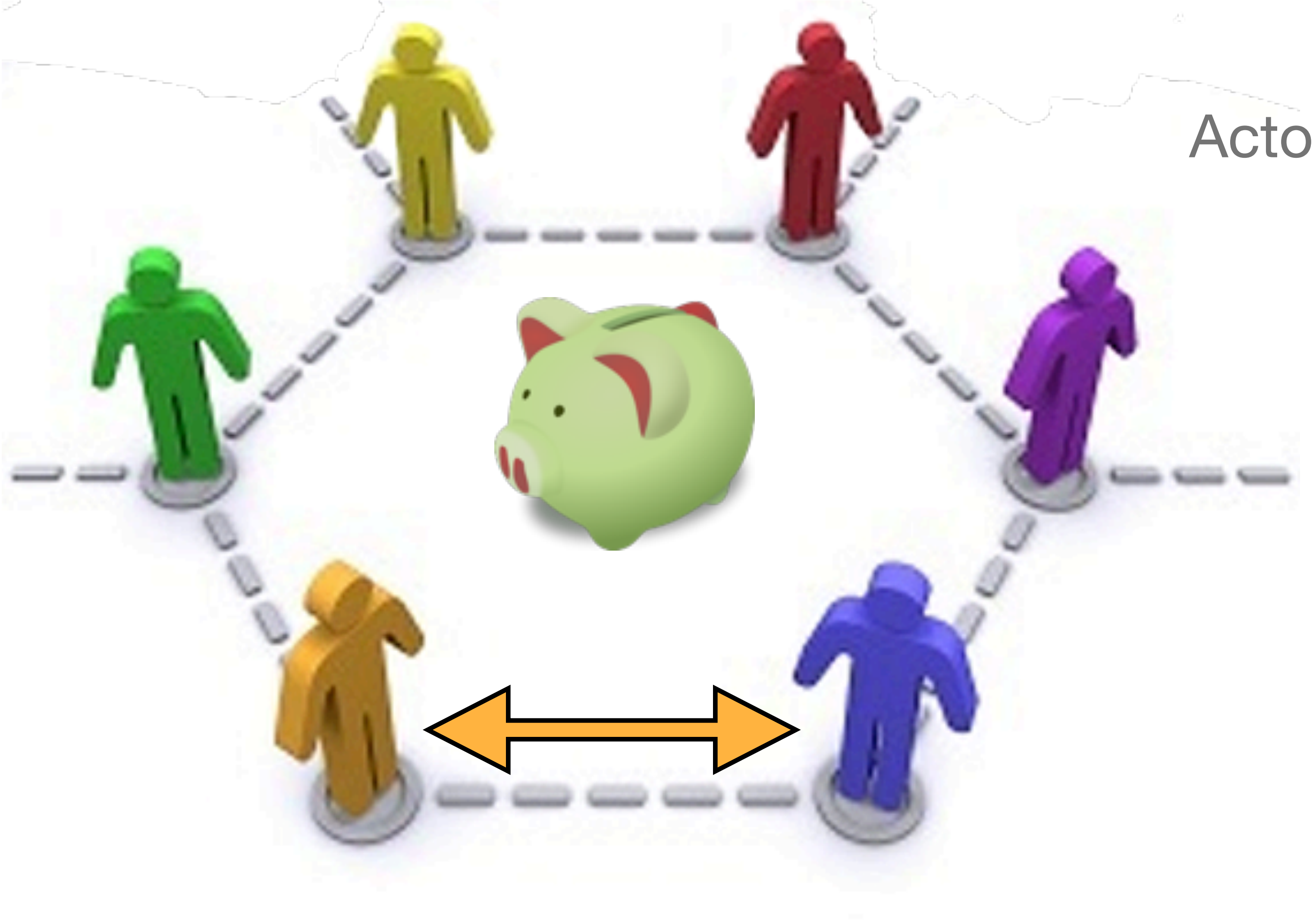
Actors:

How e-cash can improve P2P interactions:

CashLib: integrating e-cash into a P2P system

Operations:

- Buy



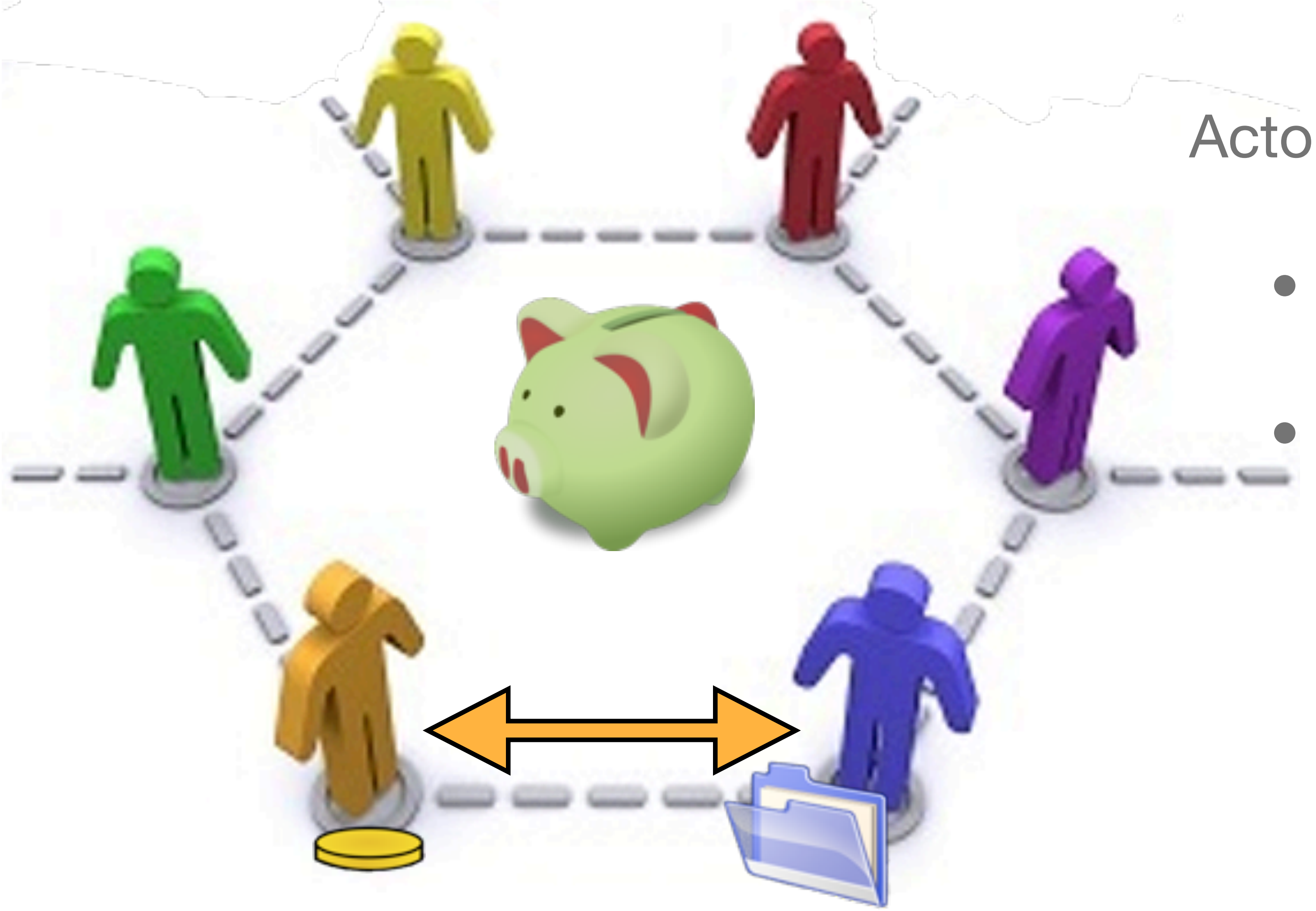
Actors:

How e-cash can improve P2P interactions:

CashLib: integrating e-cash into a P2P system

Operations:

- Buy



Actors:

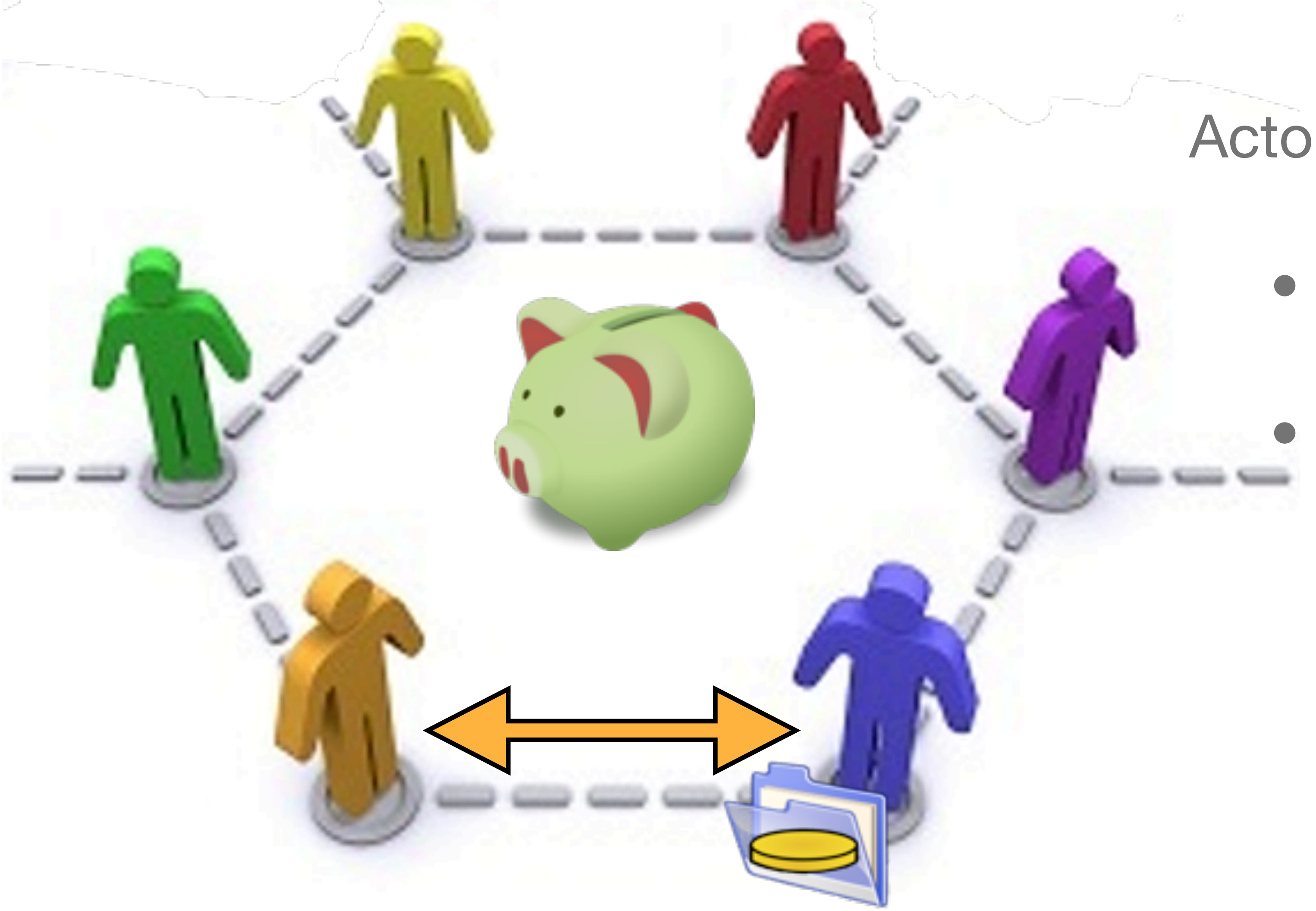
- Buyer
- Seller

How e-cash can improve P2P interactions:

CashLib: integrating e-cash into a P2P system

Operations:

- Buy



Actors:

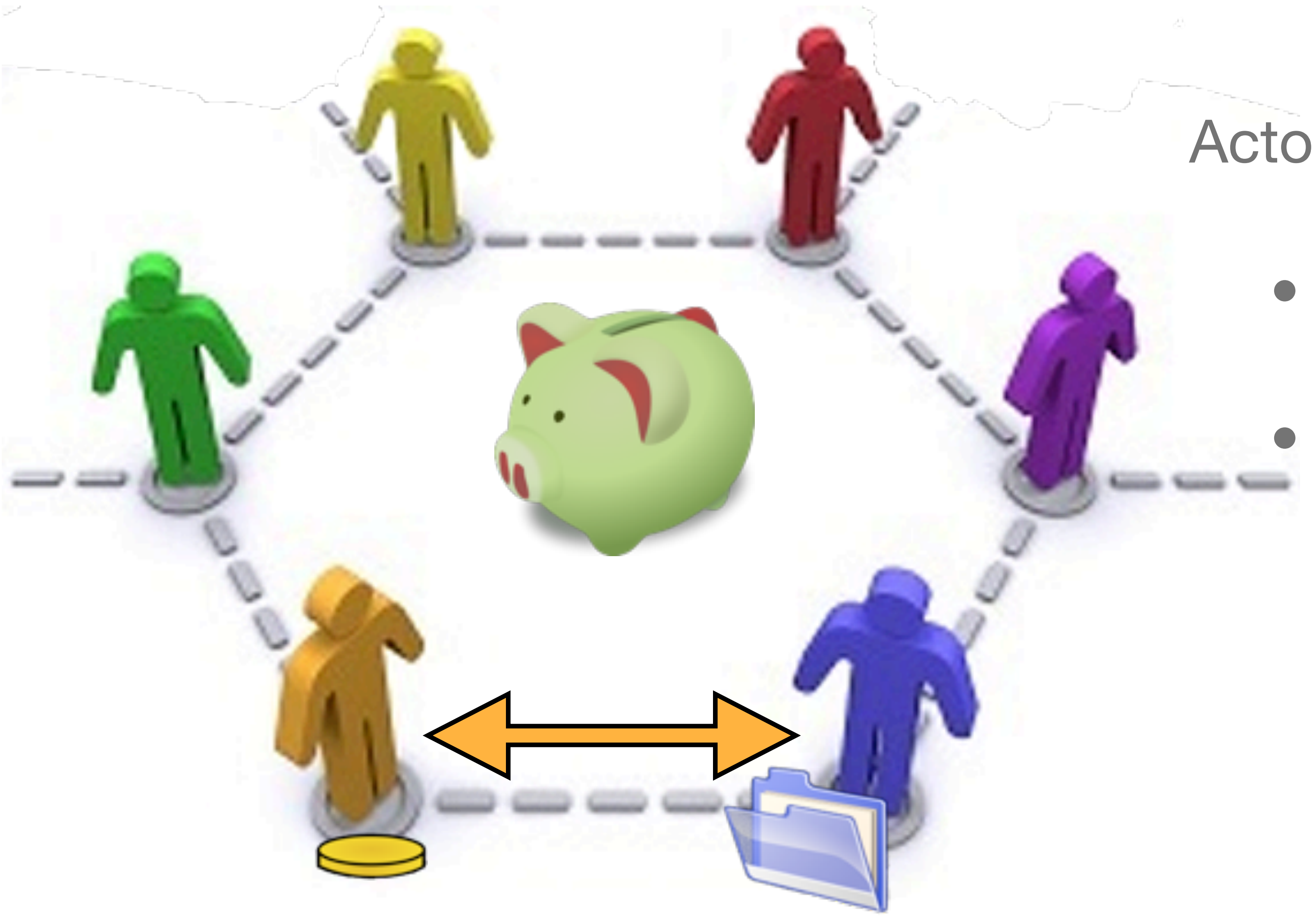
- Buyer
- Seller

How e-cash can improve P2P interactions:

CashLib: integrating e-cash into a P2P system

Operations:

- Buy



Actors:

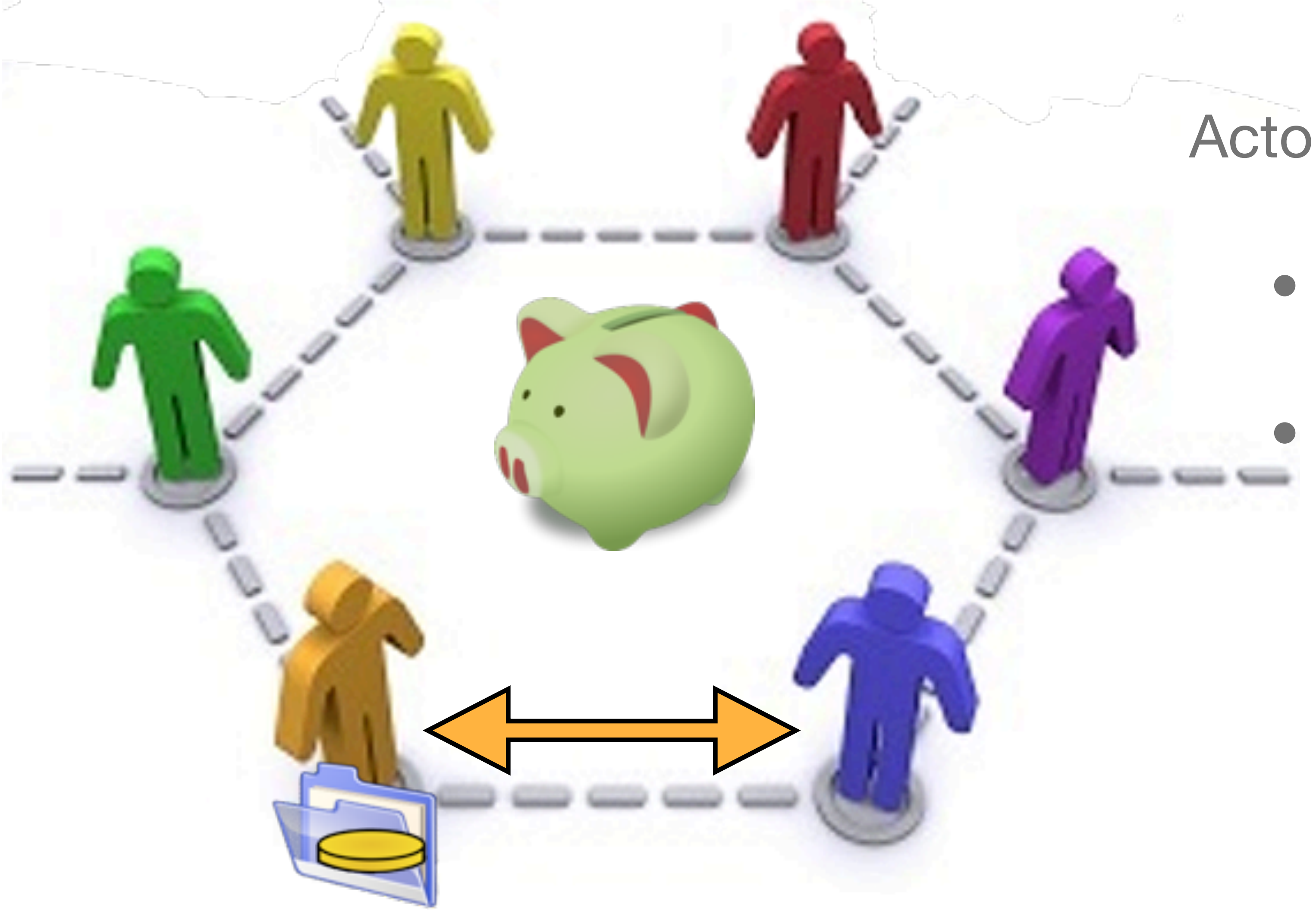
- Buyer
- Seller

How e-cash can improve P2P interactions:

CashLib: integrating e-cash into a P2P system

Operations:

- Buy



Actors:

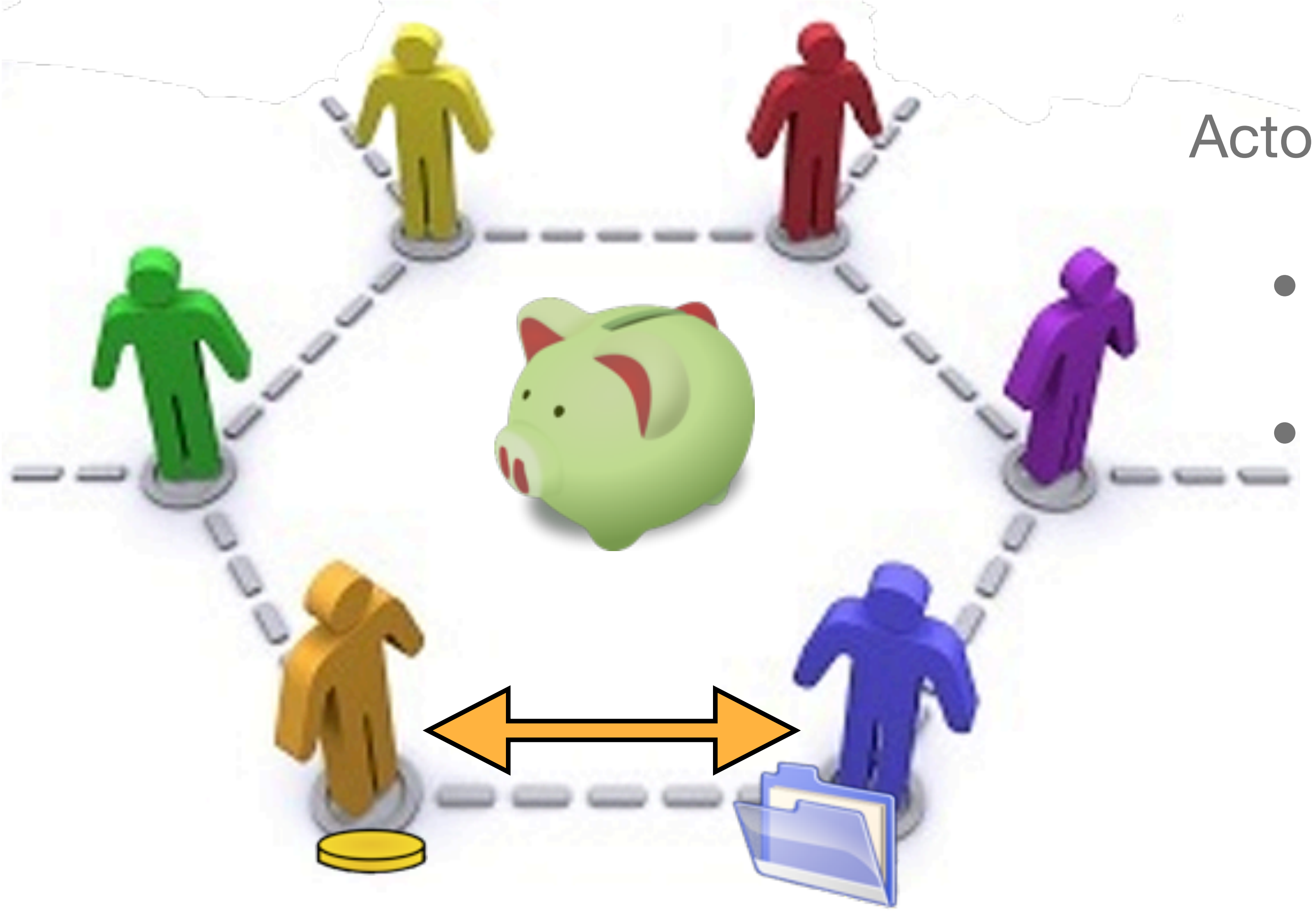
- Buyer
- Seller

How e-cash can improve P2P interactions:

CashLib: integrating e-cash into a P2P system

Operations:

- Buy



Actors:

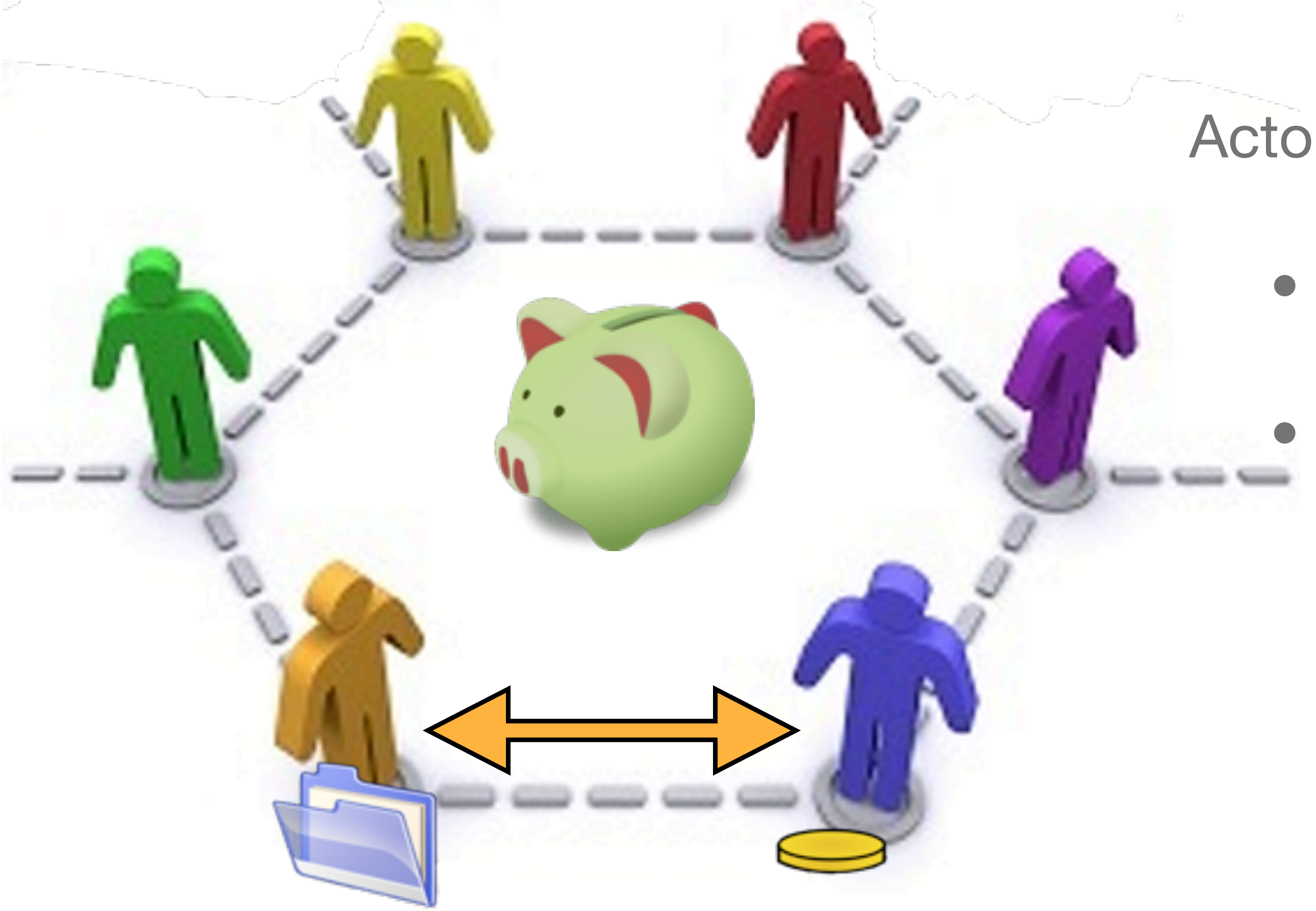
- Buyer
- Seller

How e-cash can improve P2P interactions:

CashLib: integrating e-cash into a P2P system

Operations:

- Buy



Actors:

- Buyer
- Seller

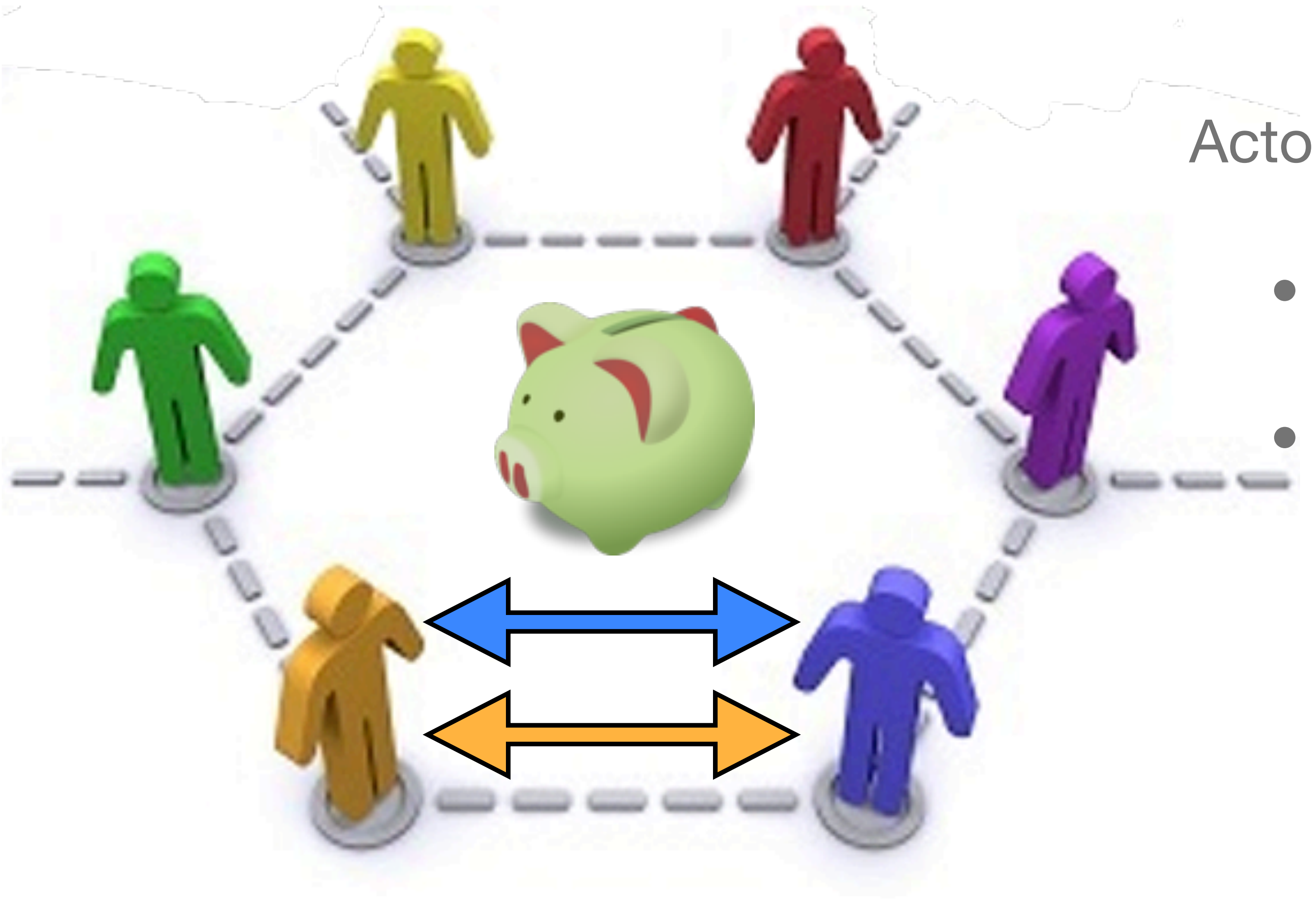
How e-cash can improve P2P interactions:

- Guarantees fair exchange [BCE+07,KL10] between peers

CashLib: integrating e-cash into a P2P system

Operations:

- Buy
- Barter



Actors:

- Buyer
- Seller

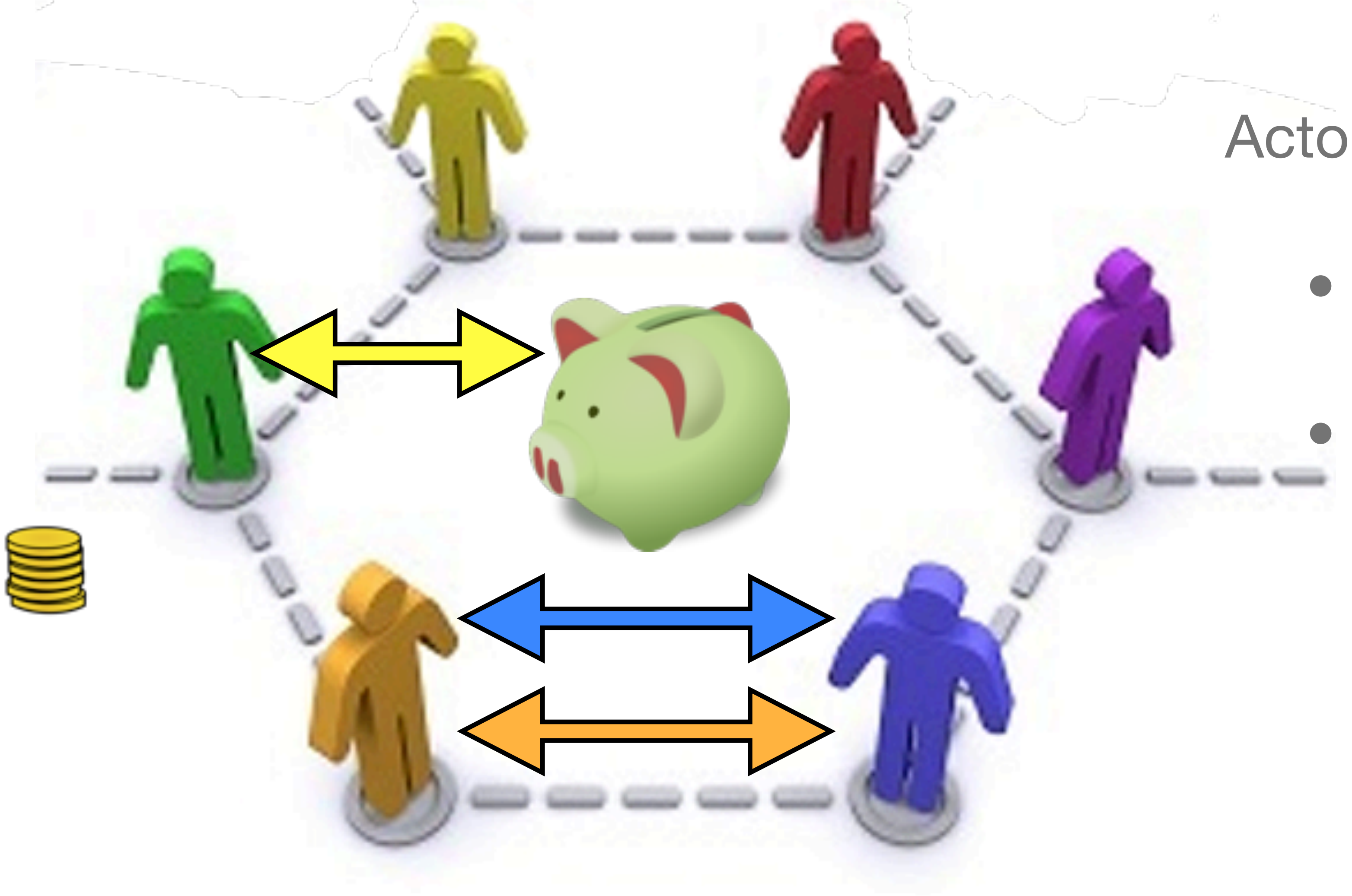
How e-cash can improve P2P interactions:

- Guarantees fair exchange [BCE+07,KL10] between peers

CashLib: integrating e-cash into a P2P system

Operations:

- Buy
- Barter
- Withdraw



Actors:

- Buyer
- Seller

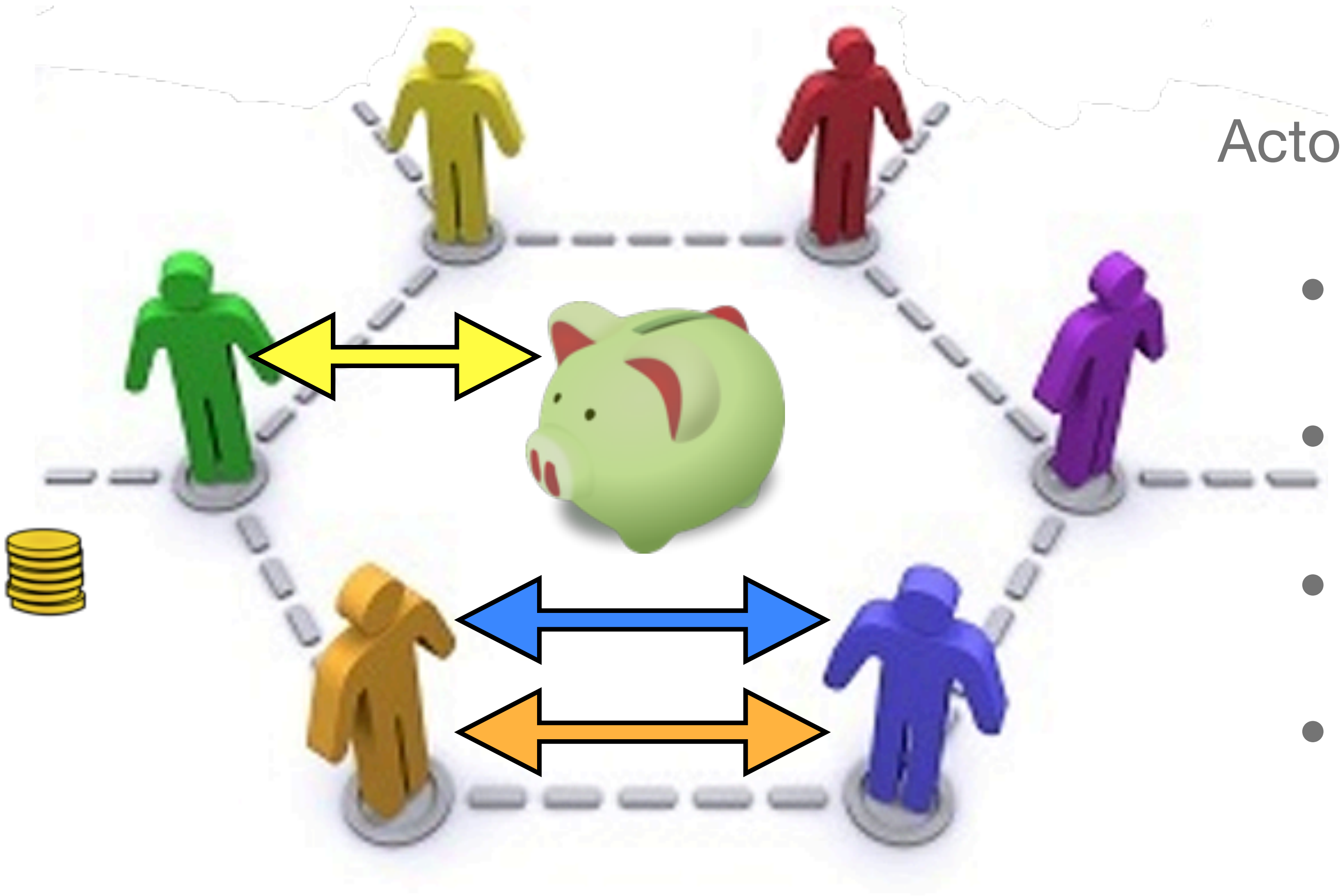
How e-cash can improve P2P interactions:

- Guarantees fair exchange [BCE+07,KL10] between peers

CashLib: integrating e-cash into a P2P system

Operations:

- Buy
- Barter
- Withdraw



Actors:

- Buyer
- Seller
- Bank
- Peer

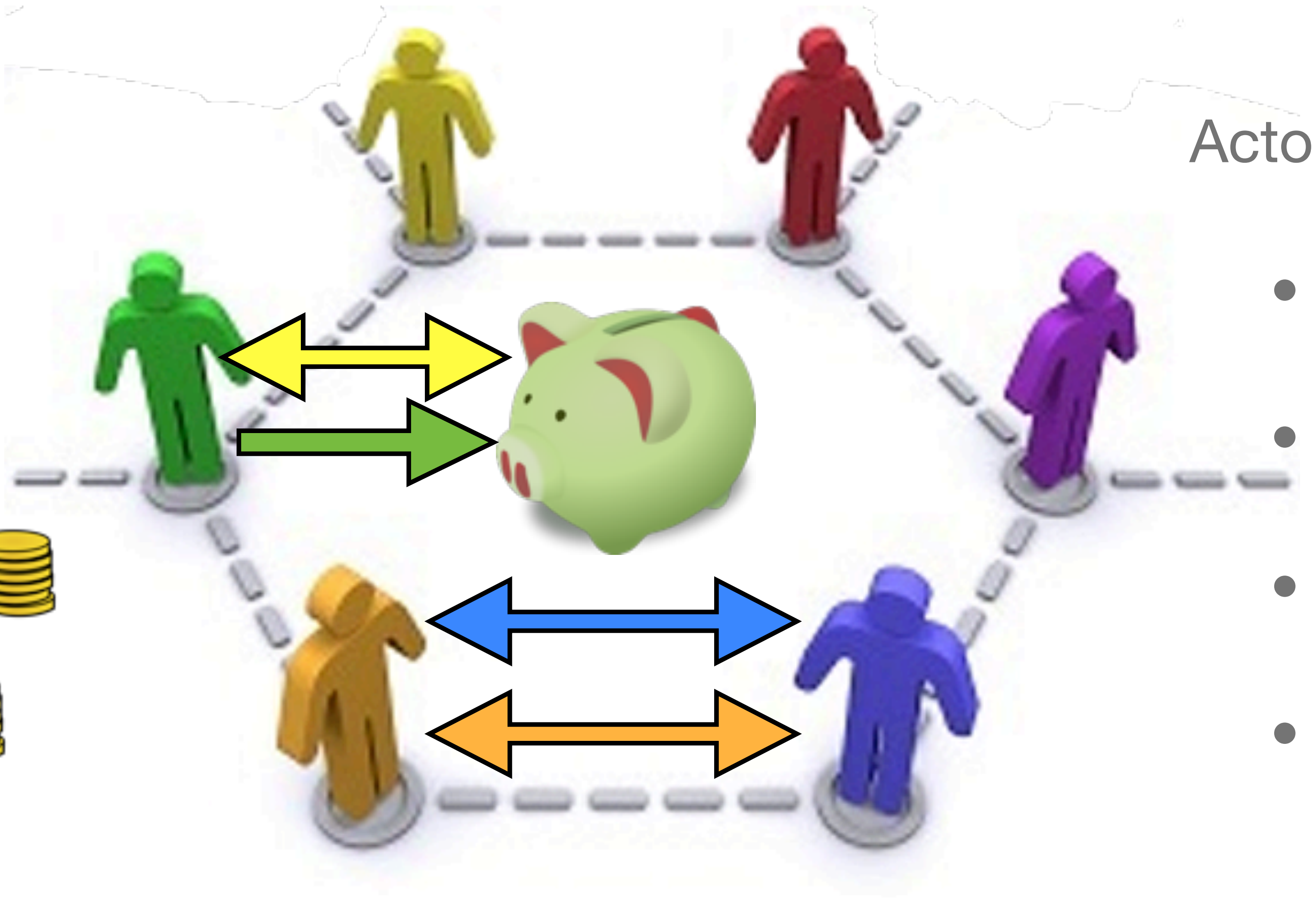
How e-cash can improve P2P interactions:

- Guarantees fair exchange [BCE+07,KL10] between peers

CashLib: integrating e-cash into a P2P system

Operations:

- Buy
- Barter
- Withdraw
- Deposit



Actors:

- Buyer
- Seller
- Bank
- Peer

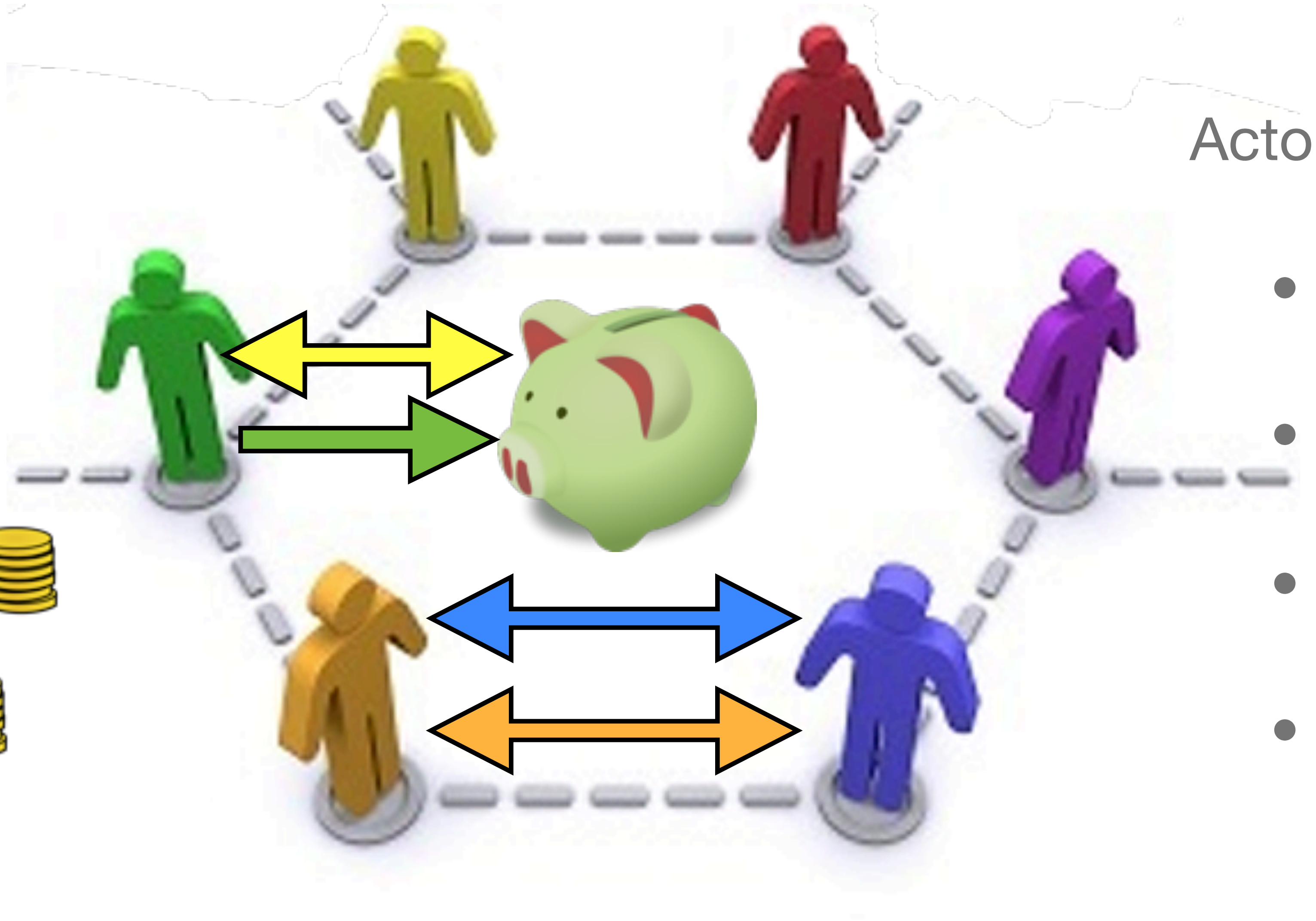
How e-cash can improve P2P interactions:

- Guarantees fair exchange [BCE+07,KL10] between peers

CashLib: integrating e-cash into a P2P system

Operations:

- Buy
- Barter
- Withdraw
- Deposit



Actors:

- Buyer
- Seller
- Bank
- Peer

How e-cash can improve P2P interactions:

- Guarantees fair exchange [BCE+07,KL10] between peers
- Allows bank to monitor upload/download ratio without sacrificing privacy

Related work

Related work

So what aren’t we doing?

Related work

So what aren't we doing?

- Aren't guaranteeing anything about the quality of the proofs

  You give us a bad (e.g., not sound) proof, get a bad proof back

  Checking soundness is well studied by others [CACE]

Related work

So what aren't we doing?

- Aren't guaranteeing anything about the quality of the proofs

  You give us a bad (e.g., not sound) proof, get a bad proof back

  Checking soundness is well studied by others [CACE]

- As application of zero knowledge, provide library only for e-cash

  Idemix project [CH02, BBC+09] provides anonymous credentials

In summary...

- Wrote interpreter to make cryptographer’s job easier
  - Demonstrated efficiency and usability
- Wrote library to make programmer’s job easier
- All source code and documentation available freely online:
  - `http://github.com/brownie/cashlib`

In summary...

- Wrote interpreter to make cryptographer's job easier
  - Demonstrated efficiency and usability
- Wrote library to make programmer's job easier
- All source code and documentation available freely online:
  - http://github.com/brownie/cashlib

Any questions?

Zero knowledge proof types

- What types of proofs do we support?
    - Proof of discrete log representation (DLR): given c, prove `c = g^x*h^r`
    - Equality of DLR: given c and d, prove `c = g^x*h^r` and `d = g^x*h^s`
    - Multiplication: prove `x = y*z` for secret values x, y, z
    - Range: for secret x and public lo, hi, prove `lo <= x < hi`