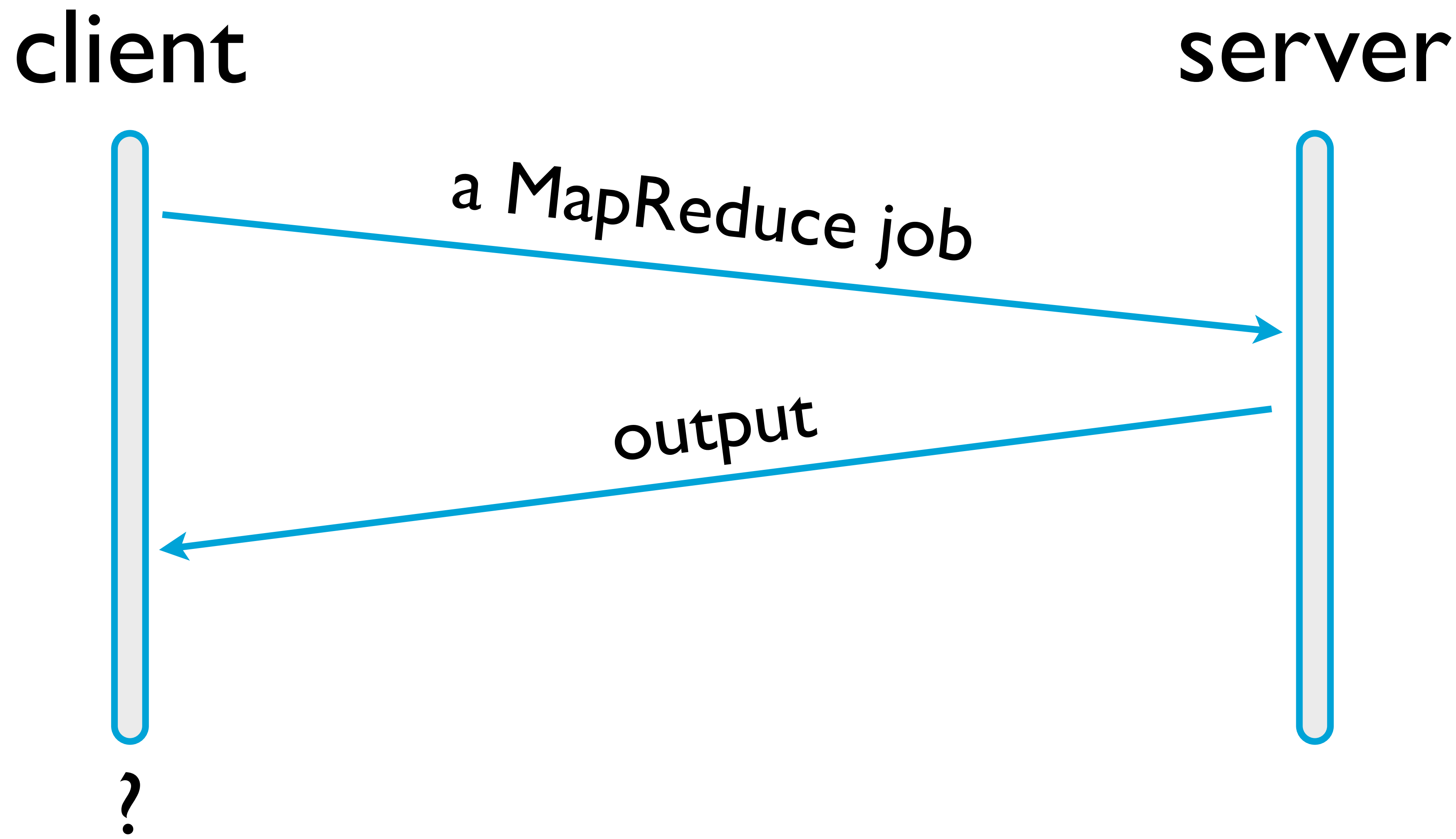
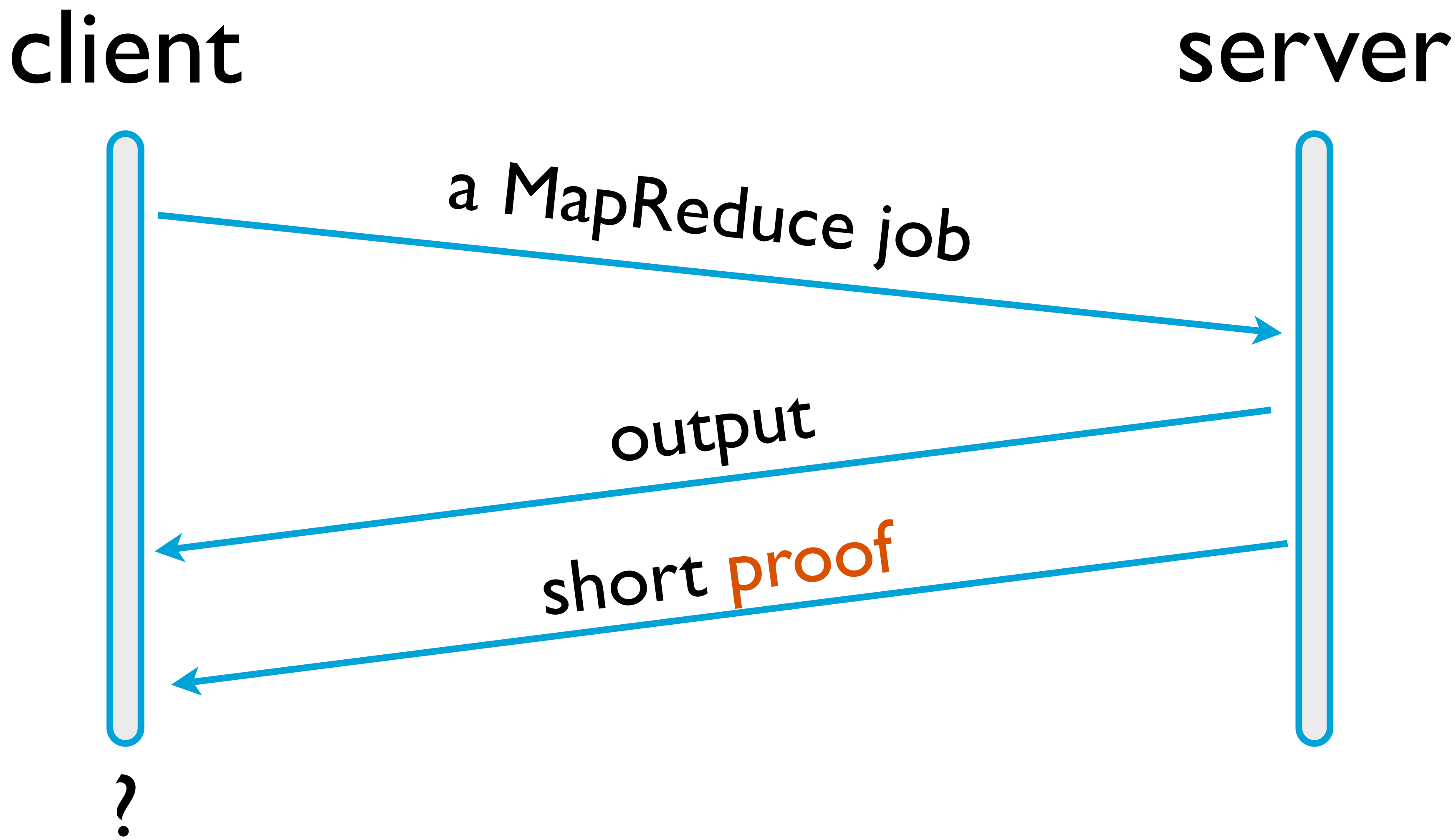
Verifying computations with state

Benjamin Braun, Ariel Feldman[†], Zuocheng Ren,
Srinath Setty, Andrew Blumberg, and Michael Walfish

The University of Texas at Austin
[†]University of Pennsylvania



the server could
compute incorrectly



check the proof quickly

Theory (PCPs, arguments, etc.) offers a solution ...
but only in theory
[ALMSS92, Micali00, BCC88, Kilian92, IKO07]

Recent projects refine and implement the theory

- CMT, TRMP, and Thaler [Cormode et al. ITCS12, Thaler et al. HotCloud12, Thaler CRYPTO13]
- Pepper, Ginger, Zaatar, and Allspice [HotOS11, NDSS12, USENIX SECURITY12, EuroSys13, IEEE S&P13]
- Pinocchio [Gennaro et al. EUROCRYPT13, Parno et al. IEEE S&P13]
- BCGTV [Ben-Sasson et al. CRYPTO13]

Highlights

- Compile C programs into verifiable computations
- Reduce costs by over a factor of 10^{20}

Remaining roadblocks in bringing the theory to practice

- The computations have to be stateless

- The client incurs a large setup cost

- The server's overheads are large

Remaining roadblocks in bringing the theory to practice	Pantry (this talk)
• The computations have to be stateless	[Eliminate]
• The client incurs a large setup cost	[Mitigate]
• The server's overheads are large	[Retain]

Aren’t there more pragmatic alternatives?

Yes and no.

Aren’t there more pragmatic alternatives?

Yes and no. Consider replication, trusted hardware, etc.:

Aren't there more pragmatic alternatives?

Yes and no. Consider replication, trusted hardware, etc.:

(1) Far less expensive than Pantry ... but impose assumptions

- Long-term, we want unconditional, cost-effective guarantees
- Pantry is a step toward this goal

(2) Pantry enables new applications for which there are not pragmatic alternatives

- Computations over private server state, etc.

Rest of this talk:

- The computations have to be stateless

- The client incurs a large setup cost

- The server's overheads are large

1

2

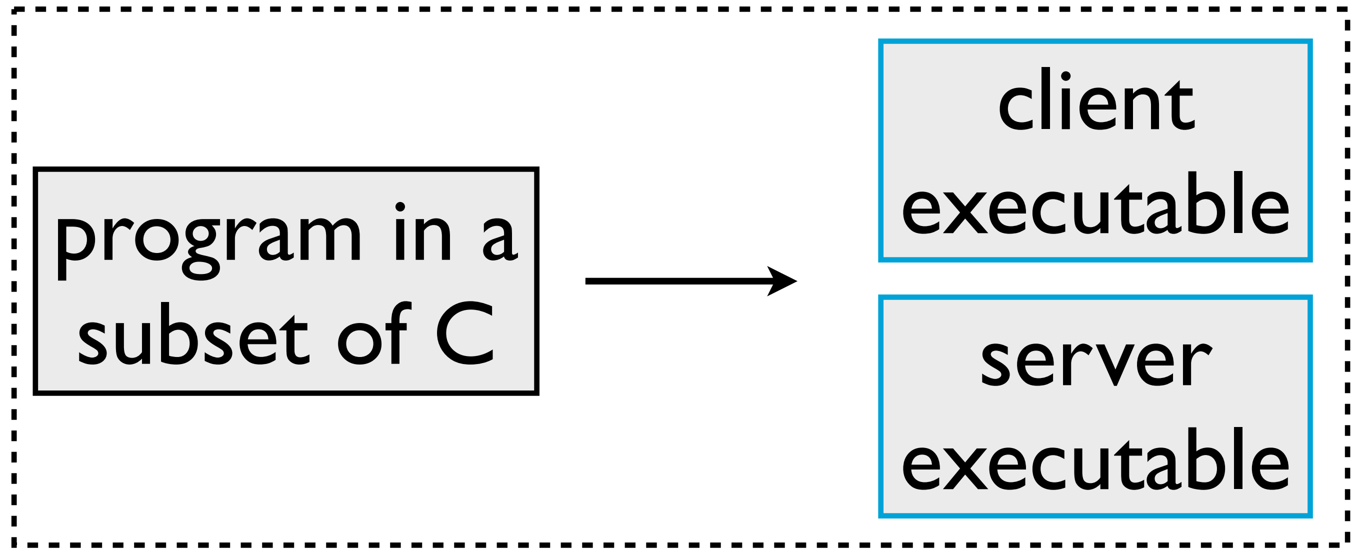
[Eliminate]

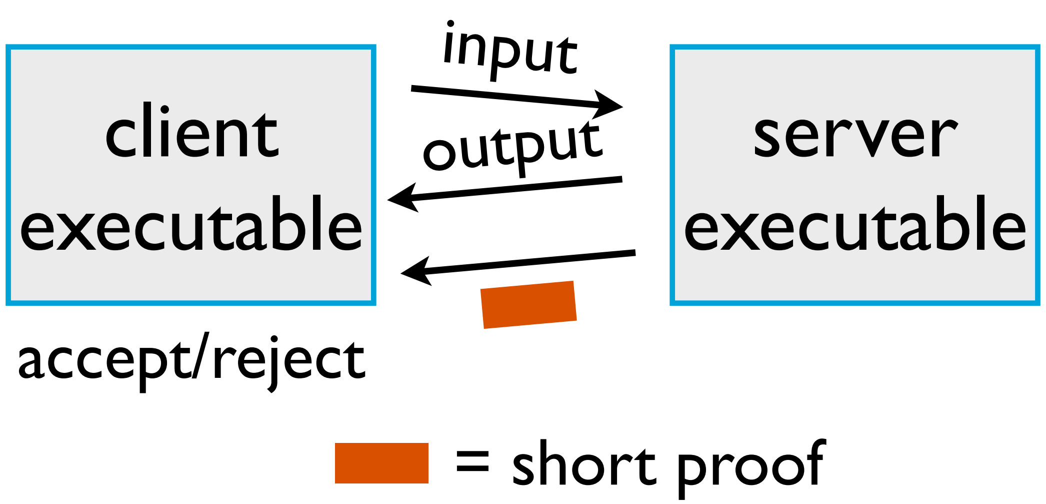
3

[Mitigate]

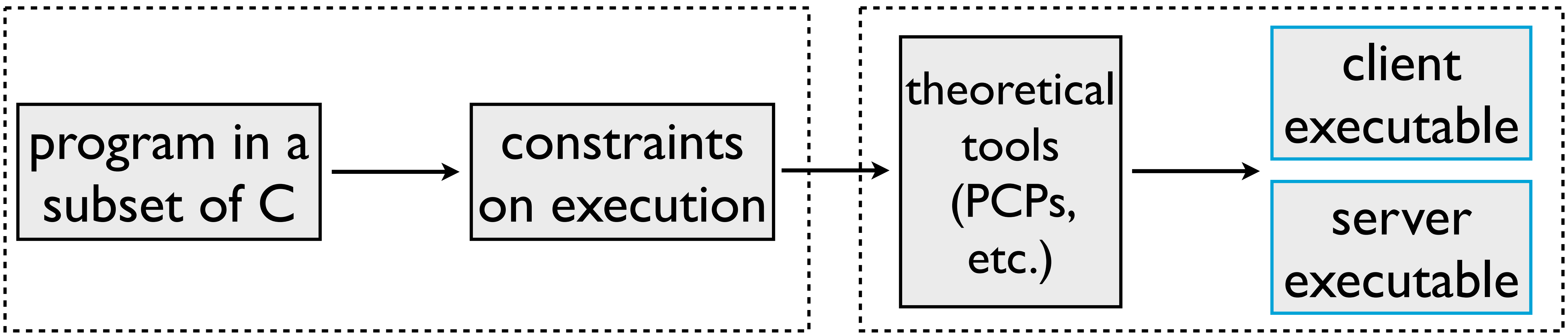
[Retain]

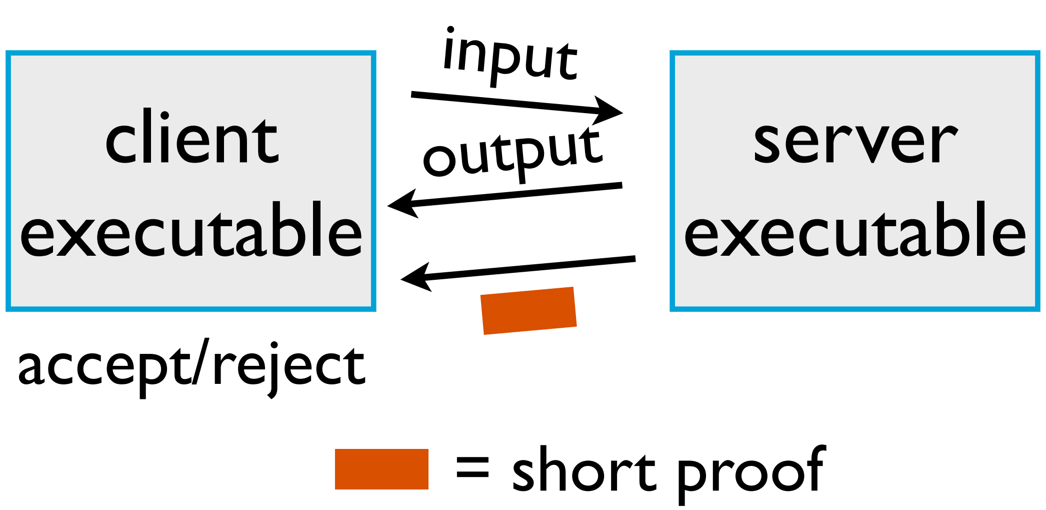
Pantry's base: Zaatar [EuroSys13] and Pinocchio [IEEE S&P13]



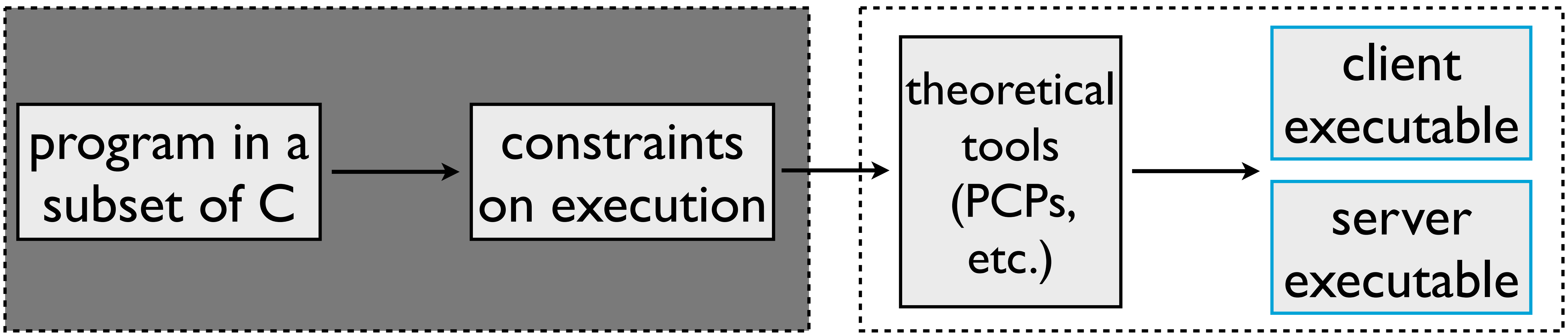


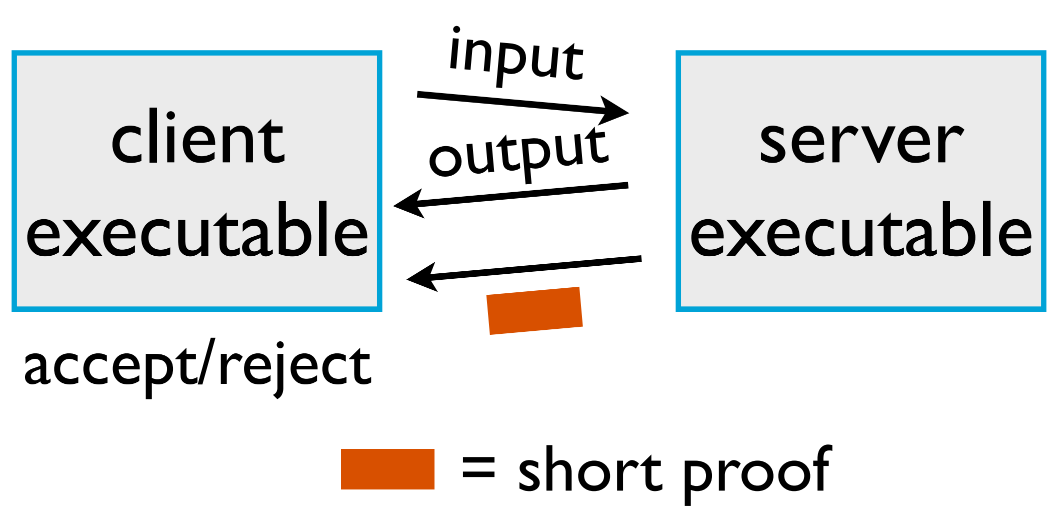
Pantry's base: Zaatar [EuroSys13] and Pinocchio [IEEE S&P13]





Pantry's base: Zaatar [EuroSys13] and Pinocchio [IEEE S&P13]





Programs compile into a set of constraints

```
int increment(int i) {
  r = i + 1;
  return r;
}
```

$\Longleftrightarrow$

$0 = X - i$

$0 = Y - (X + 1)$

$0 = Y - r$

Programs compile into a set of constraints

```
int increment(int i) {
  r = i + 1;
  return r;
}
```

$\longleftrightarrow$

$$0 = X - i$$
$$0 = Y - (X + 1)$$
$$0 = Y - r$$

Correct input/output pair means that the equations have a solution (i.e., constraints are satisfiable)

Suppose the input is 6

If the output is 7

$$0 = X - 6$$
$$0 = Y - (X + 1)$$
$$0 = Y - 7$$

There is a solution

If the output is 8

$$0 = X - 6$$
$$0 = Y - (X + 1)$$
$$0 = Y - 8$$

There is no solution

Constraints can represent various program structures

Example: “Y = (X1 != X2)”

$$0 = (X1 - X2) \cdot M - Y$$

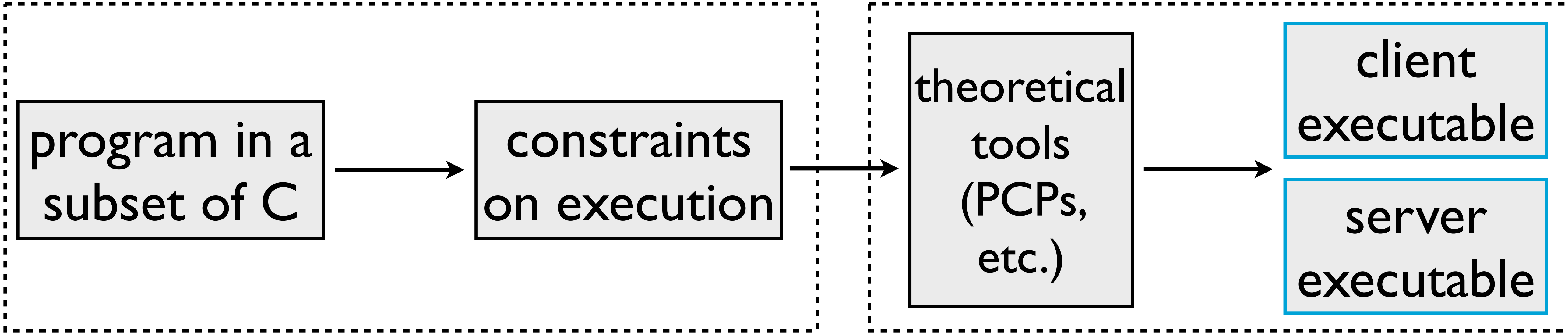
$$0 = (1 - Y) \cdot (X1 - X2)$$

Observe:

if X1 == X2, then Y must be 0, to satisfy the first.

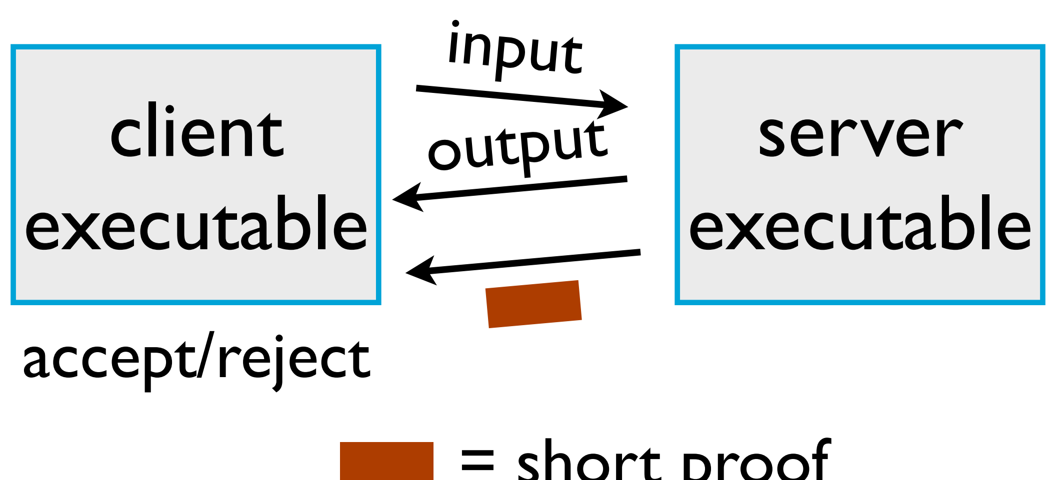
if X1 != X2, then Y must be 1, to satisfy the second.

Pantry's base: Zaatar [EuroSys13] and Pinocchio [IEEE S&P13]

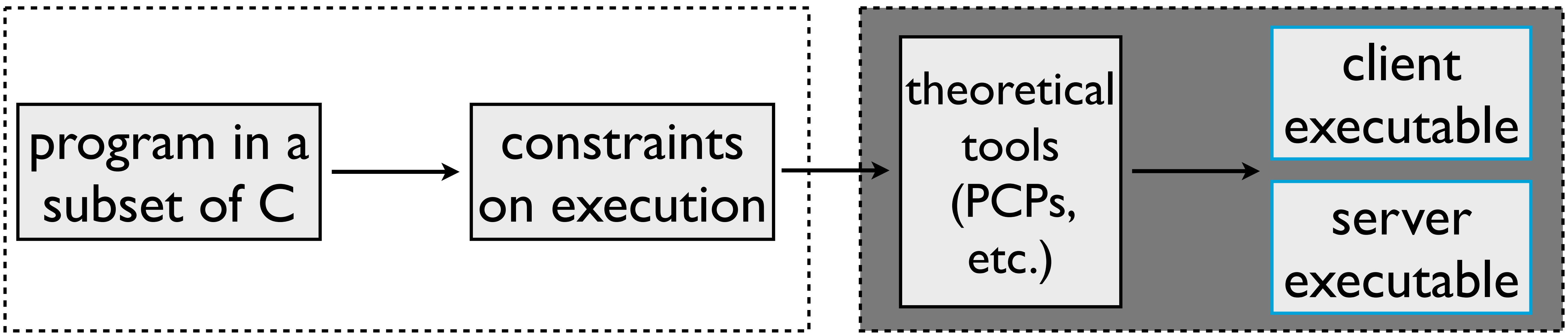


satisfiability of constraints ⇔

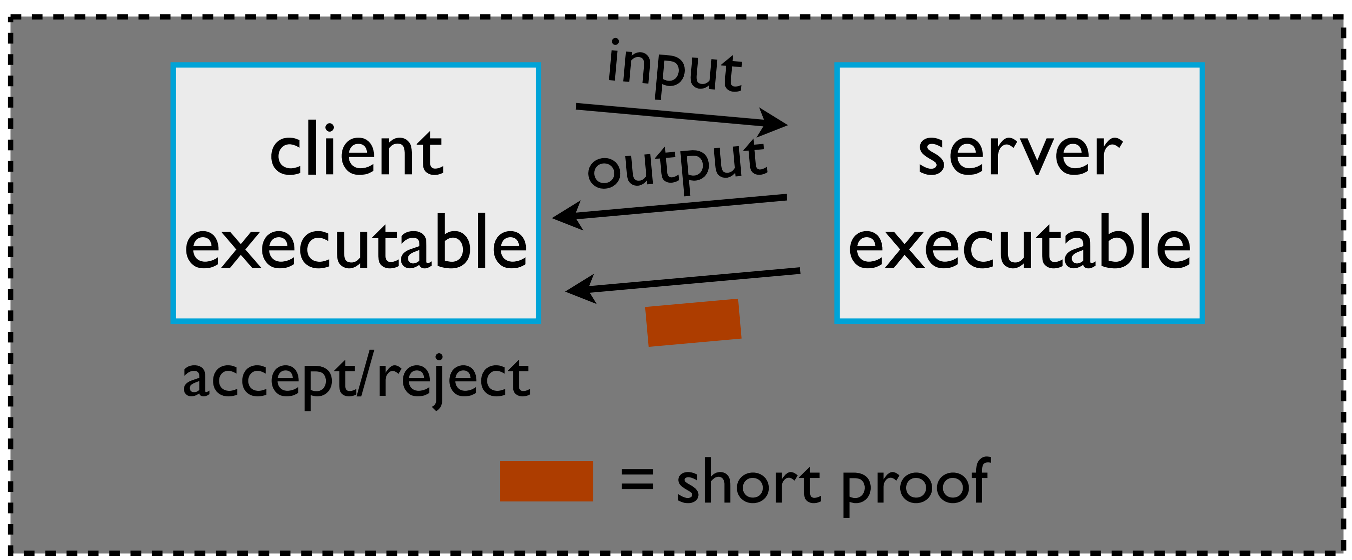
correct execution



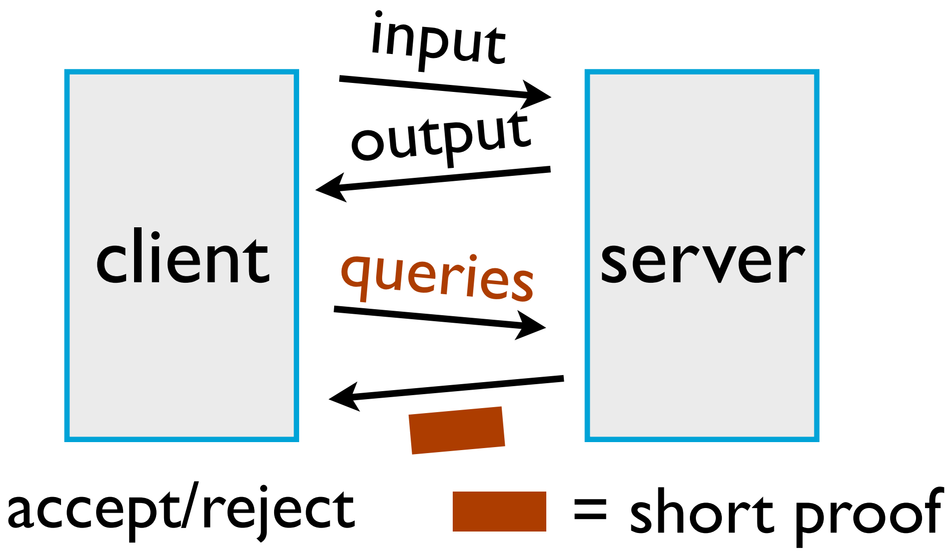
Pantry's base: Zaatar [EuroSys13] and Pinocchio [IEEE S&P13]

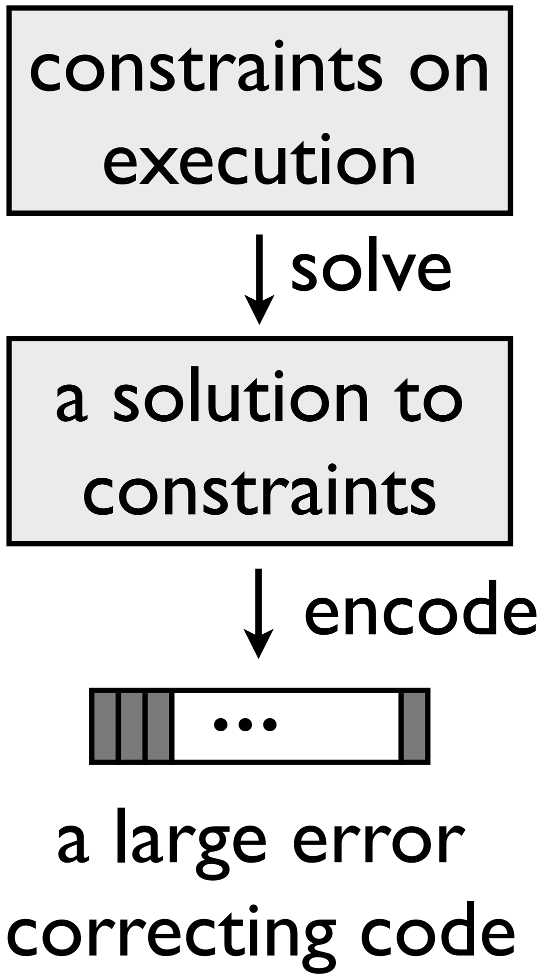


satisfiability of constraints ⇔ correct execution

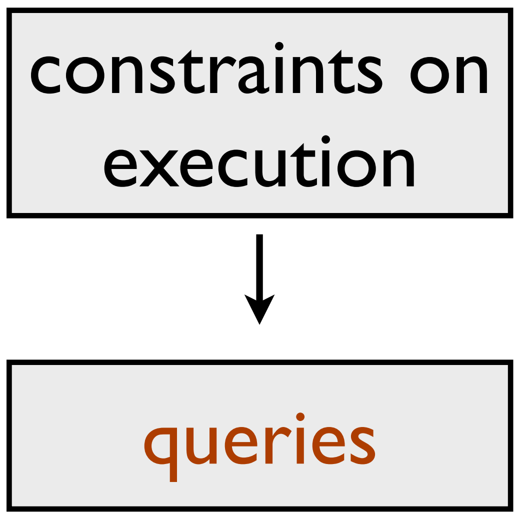


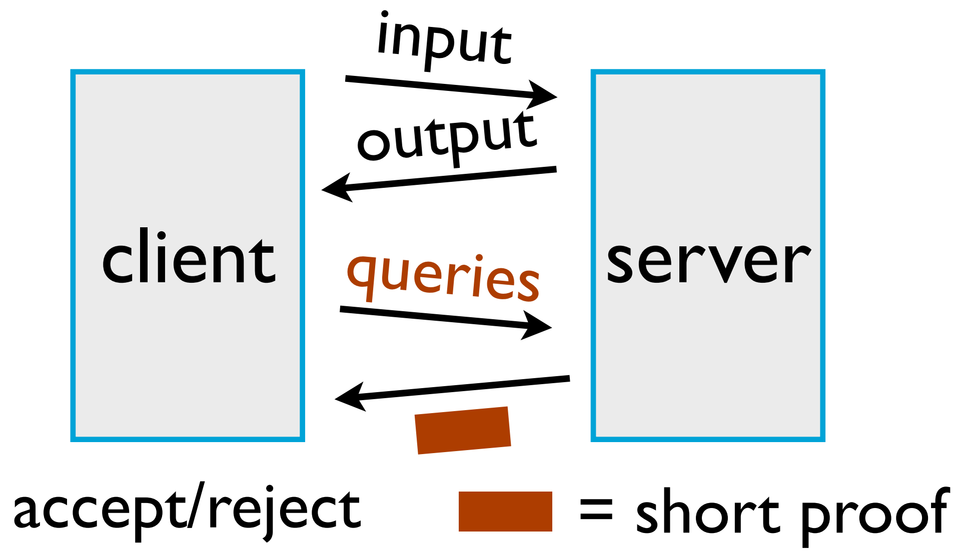
Verification protocol:

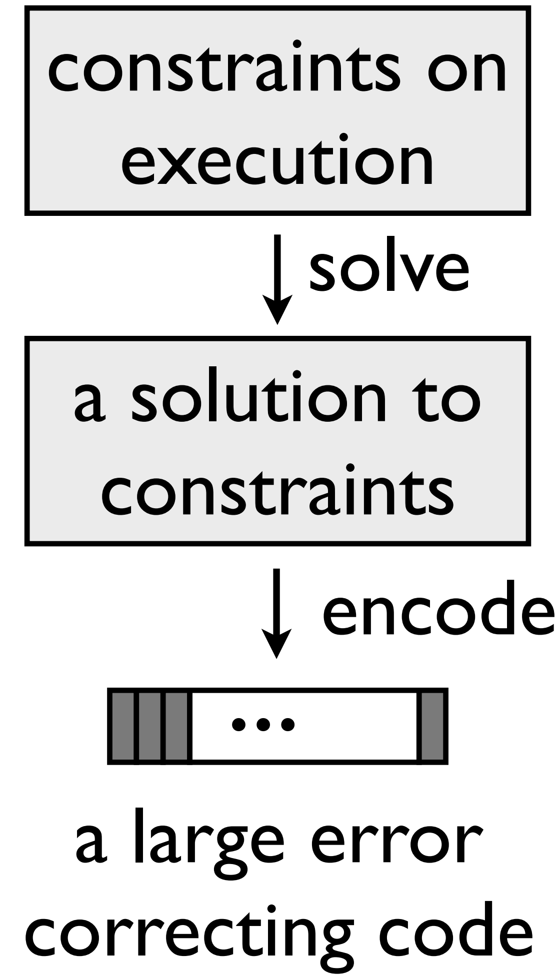




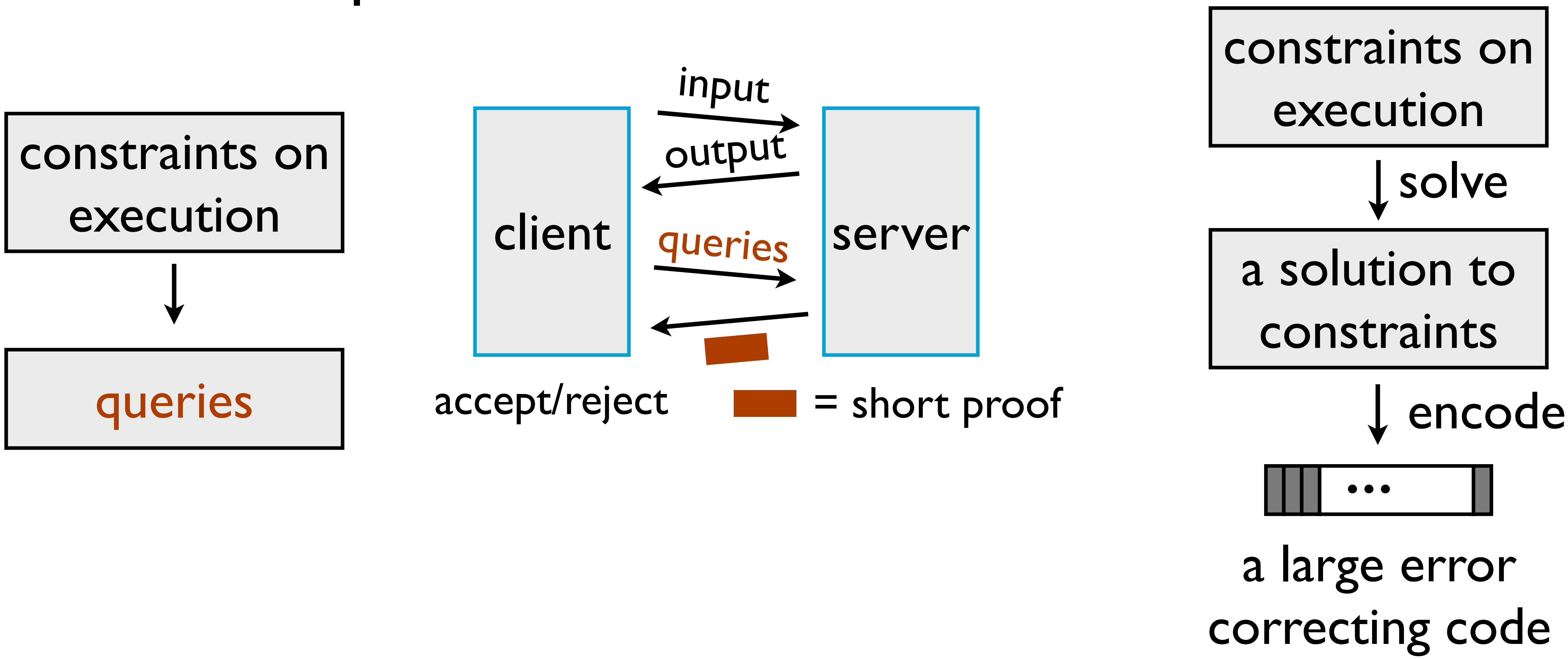
Verification protocol:







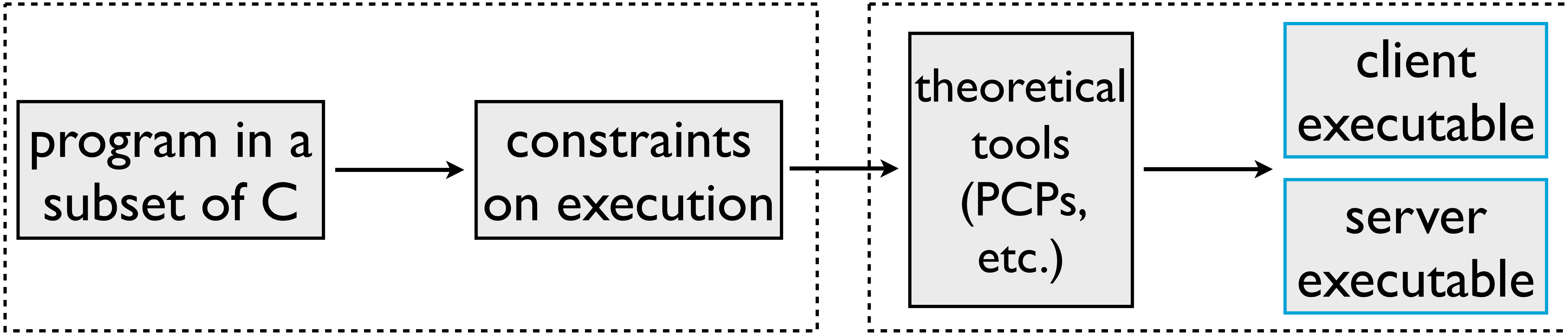
Verification protocol:



The client has to amortize its query generation costs to save resources relative to local execution

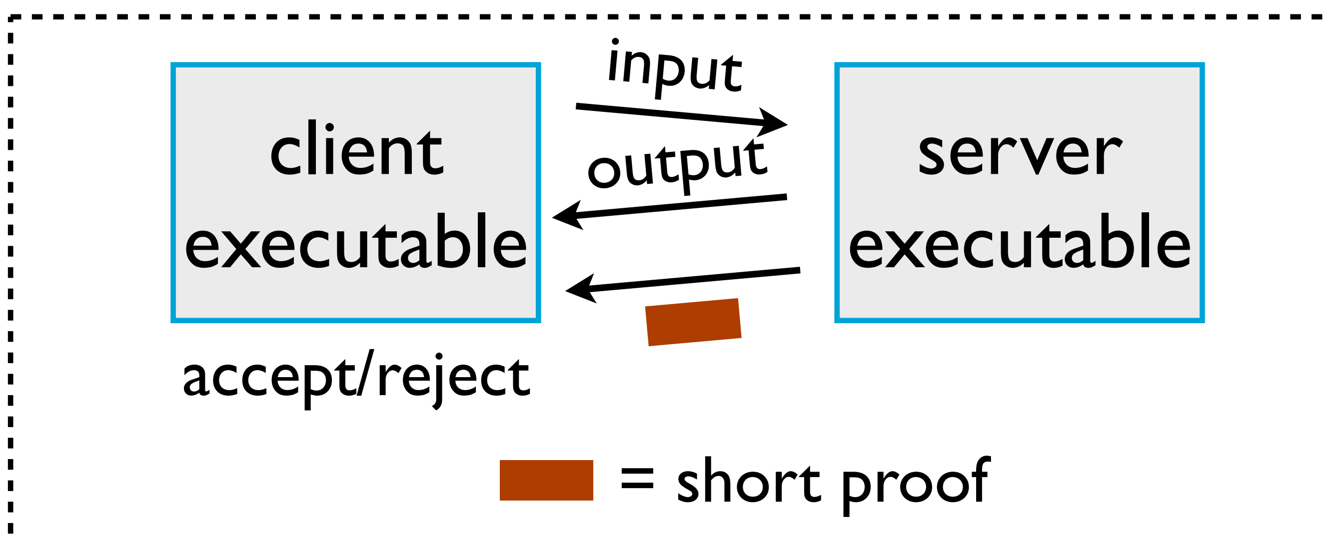
The short proof is the queried values from the large encoding

Pantry's base: Zaatar [EuroSys13] and Pinocchio [IEEE S&P13]

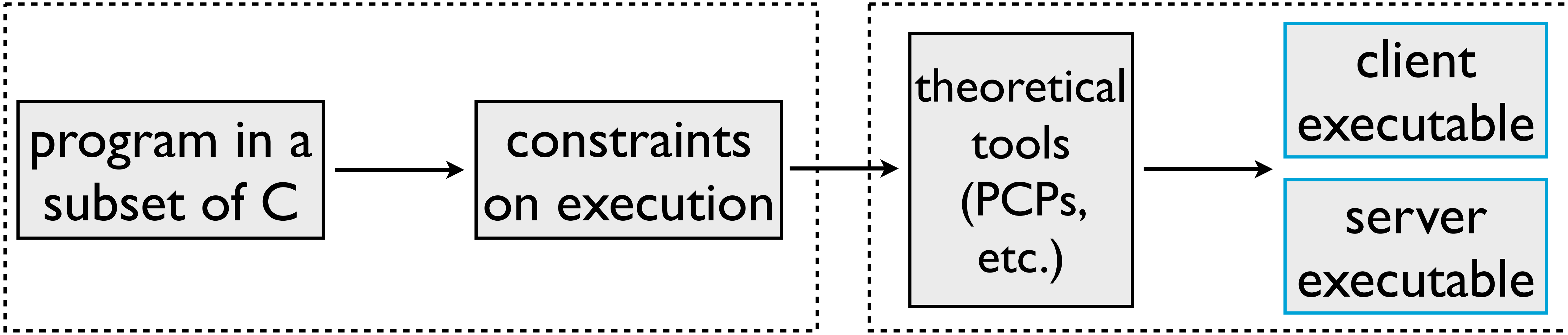


satisfiability of constraints ⇔ correct execution

a valid proof ⇔ satisfiability of constraints



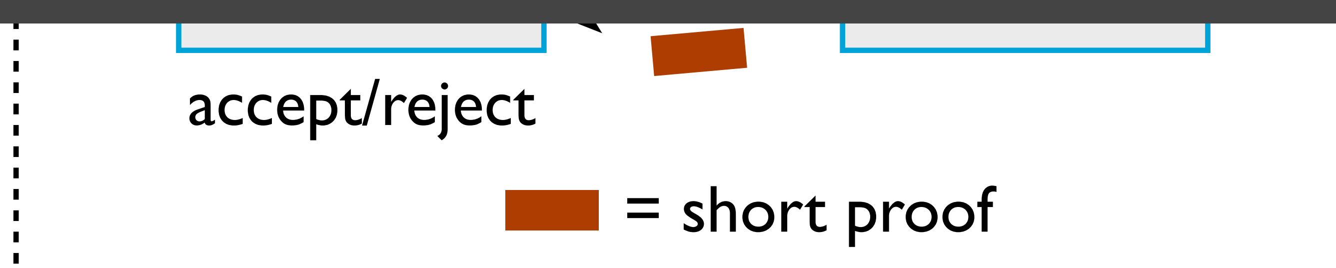
Pantry's base: Zaatar [EuroSys13] and Pinocchio [IEEE S&P13]



satisfiability of constraints ⇔ correct execution

a valid proof ⇔ satisfiability of constraints

How can we design constraints such that their satisfiability is tantamount to correct storage interaction?



A naive approach

$B = \text{read}(A)$ ⟷

$B = S_0 - (A-0) \cdot C_0$

$B = S_1 - (A-1) \cdot C_1$

....

$B = S_{size} - (A-size) \cdot C_{size}$

$S_0, S_1, \ldots, S_{size}$ correspond to cells of storage

A naive approach

$B = read(A)$ $\longleftrightarrow$

$B = S_0 - (A-0) \cdot C_0$

$B = S_1 - (A-1) \cdot C_1$

....

$B = S_{size} - (A-size) \cdot C_{size}$

switch (A) {
  case 0: B = S_0; break;
  case 1: B = S_1; break;
  ...
  case size: B = S_{size}; break;
}

S_0, S_1, ..., S_{size} correspond to cells of storage

A naive approach

B = read(A) ⟷

$B = S_0 - (A-0) \cdot C_0$

$B = S_1 - (A-1) \cdot C_1$

....

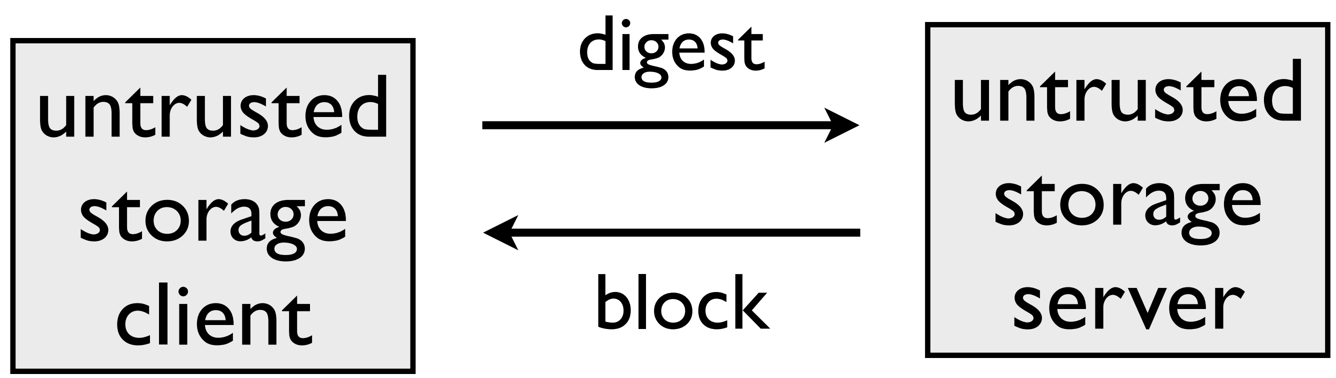
$B = S_{size} - (A\text{-}size) \cdot C_{size}$

```
switch (A) {
  case 0: B = S0; break;
  case 1: B = S1; break;
  ...
  case size: B = Ssize; break;
}
```

$S_0, S_1, \ldots, S_{size}$ correspond to cells of storage

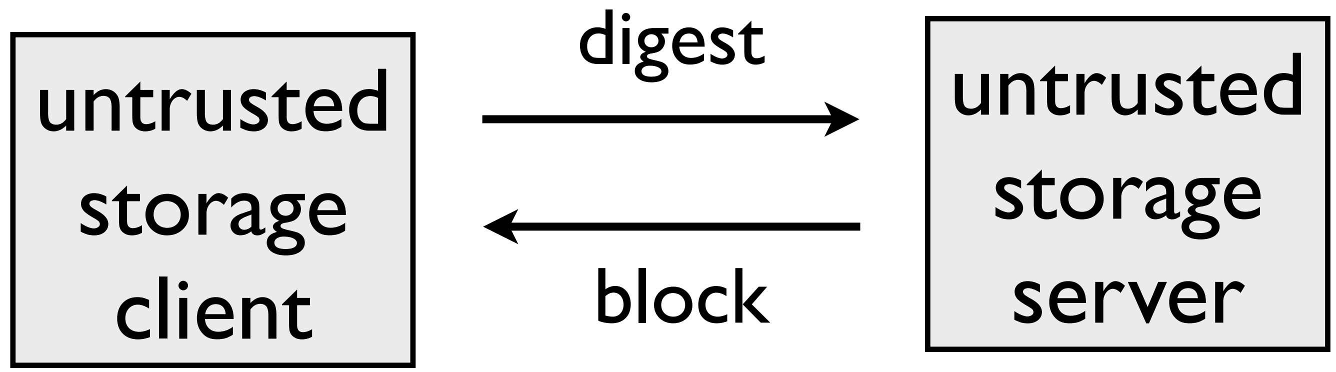
- Expensive: requires one constraint for each address
- Incomplete: provides only volatile state

Consider an untrusted block store:



H(block) $\stackrel{?}{=}$ digest, H is a cryptographic hash function

Consider an untrusted block store:

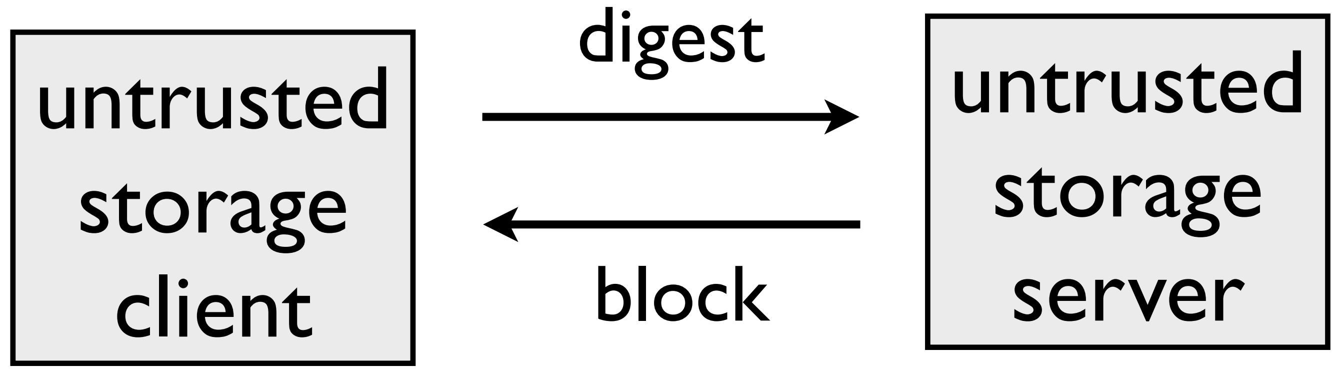


$H(\text{block}) \stackrel{?}{=} \text{digest}$, H is a cryptographic hash function

To run a computation with remote inputs, the above client will need to:

1. Fetch blocks from the storage server
2. Check the integrity of the blocks using digests
3. Run the computation

Consider an untrusted block store:

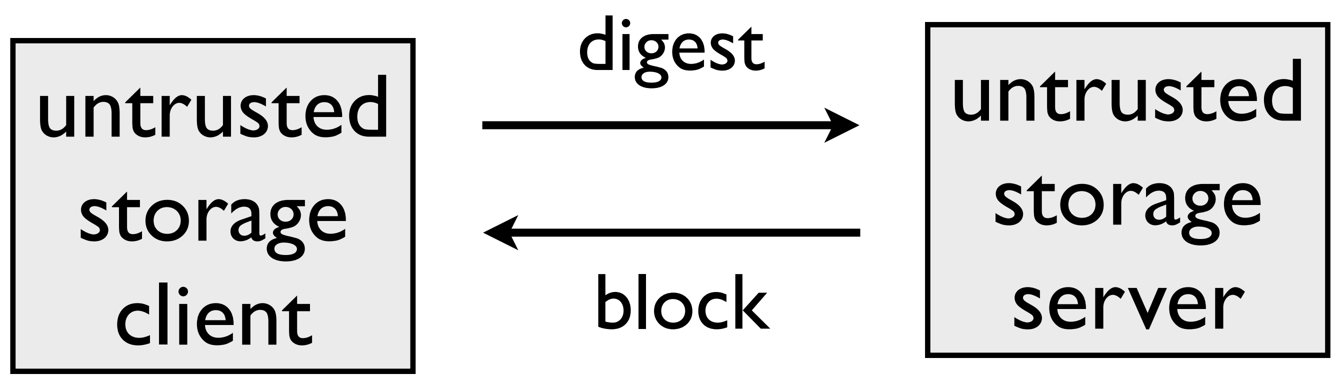


Pantry's approach to state: verifiably run the steps below on the server, by compiling these steps into constraints, without having to handle data blocks

client will need to:

1. Fetch blocks from the storage server
2. Check the integrity of the blocks using digests
3. Run the computation

Consider an untrusted block store:



Pantry's approach to state: verifiably run the steps below on the server, by compiling these steps into constraints, without having to handle data blocks

client will need to:

1. Fetch blocks from the storage server

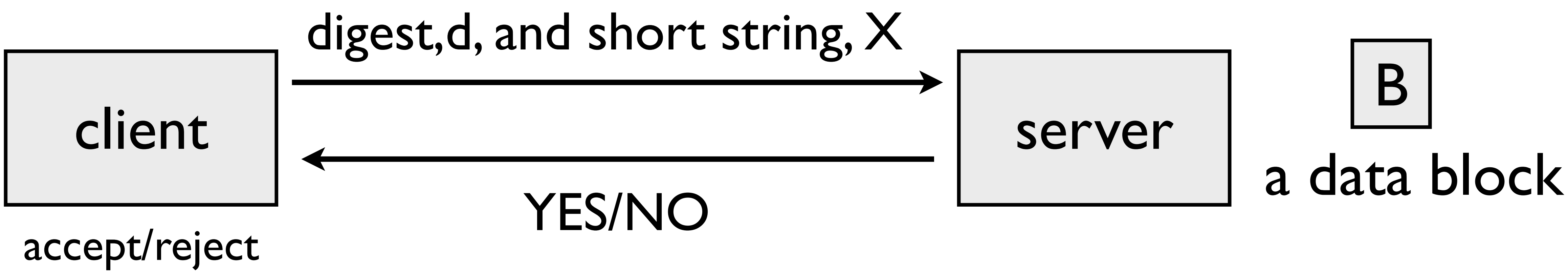
2. Check the integrity of the blocks using digests

3. Run the computation

Existing work designs higher level abstractions using an untrusted block store [Merkle CRYPTO87, Blum et al. FOCS91, Fu et al. OSDI00, Li et al. OSDI04]

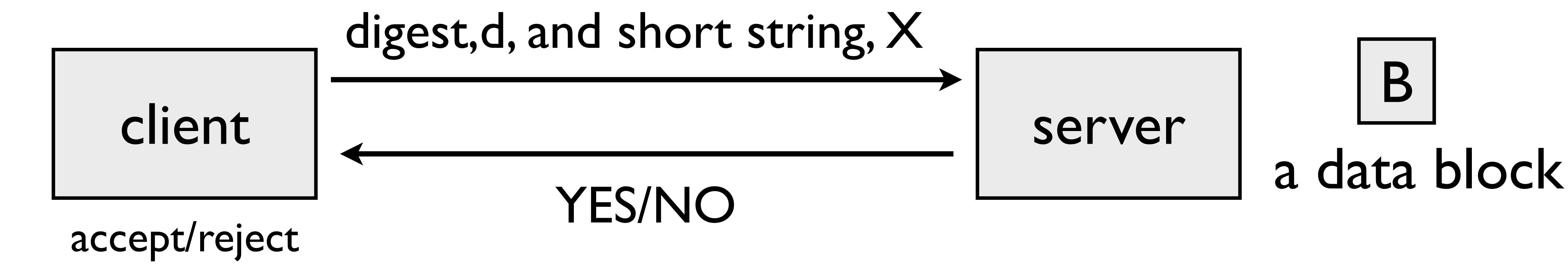
Pantry's approach to state, with an example

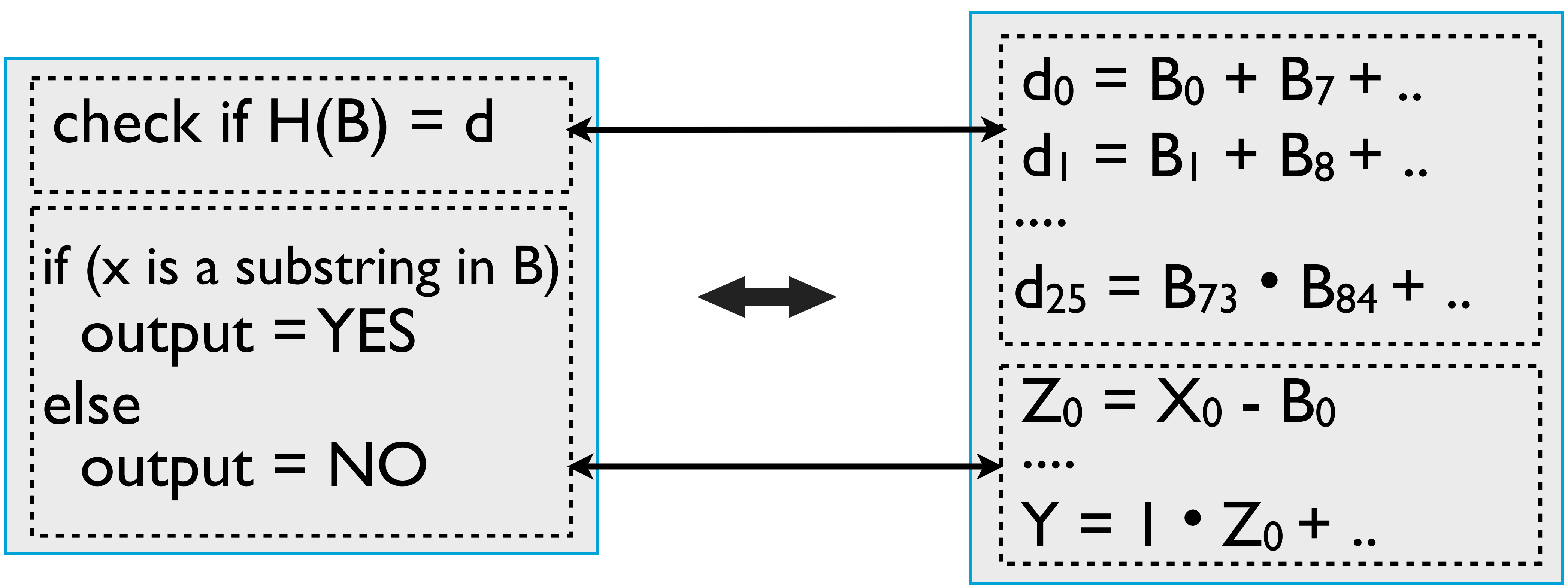
Consider a substring search with a remote data block

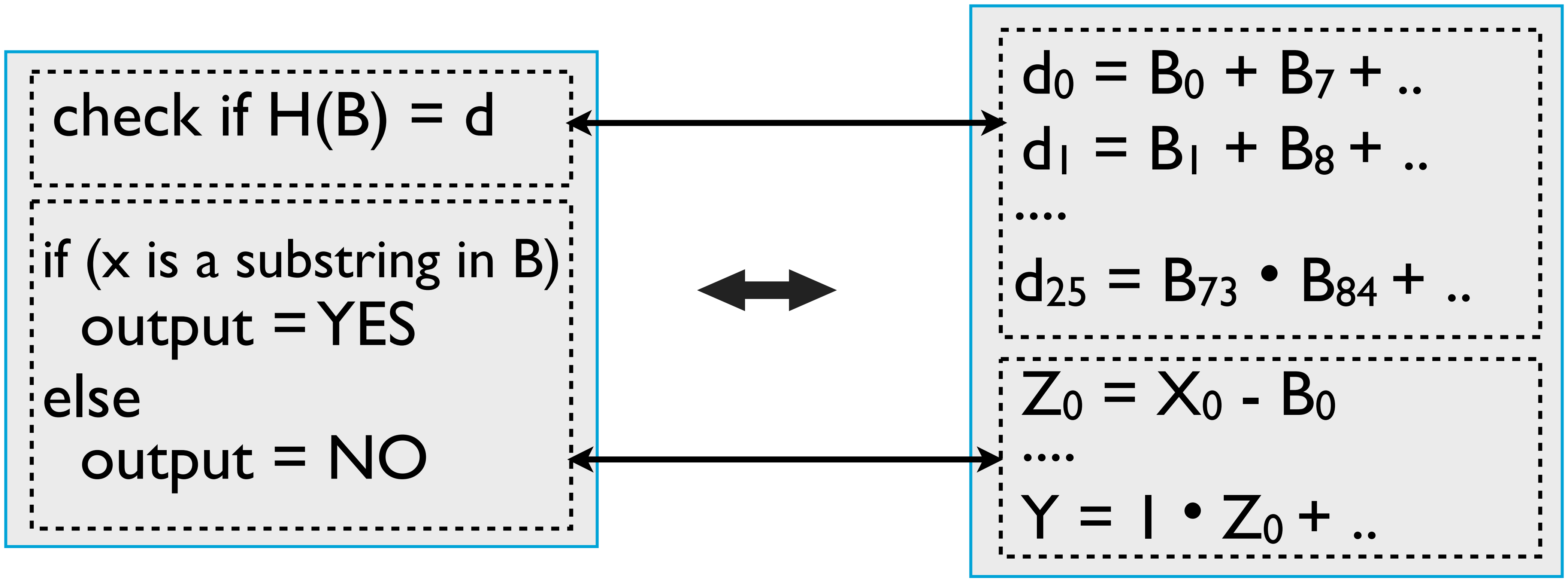


Pantry's approach to state, with an example

Consider a substring search with a remote data block







- Satisfiability of the above constraints ⇔ passing hash checks
- Passing hash checks is computationally infeasible without the right data blocks

We add two primitives to Pantry's C to expose state

- PutBlock(block): stores "block" at location H(block)
- GetBlock(digest): returns a block such that H(block) = digest

We add two primitives to Pantry's C to expose state

- PutBlock(block): stores "block" at location H(block)
- GetBlock(digest): returns a block such that H(block) = digest

```
function (digest d)
  int b = GetBlock(d)
  y = b+1
  return y
```

$d_0 = B_0 + B_7 + ..$

$d_1 = B_1 + B_8 + ..$

....

$d_{25} = B_{73} \bullet B_{84} + ..$

$Y = B+1$

Representation of H in constraints

We add two primitives to Pantry's C to expose state

- PutBlock(block): stores "block" at location H(block)
- GetBlock(digest): returns a block such that H(block) = digest

```
function (digest d)
  int b = GetBlock(d)
  y = b+1
  return y
```

$d_0 = B_0 + B_7 + ..$

$d_1 = B_1 + B_8 + ..$

....

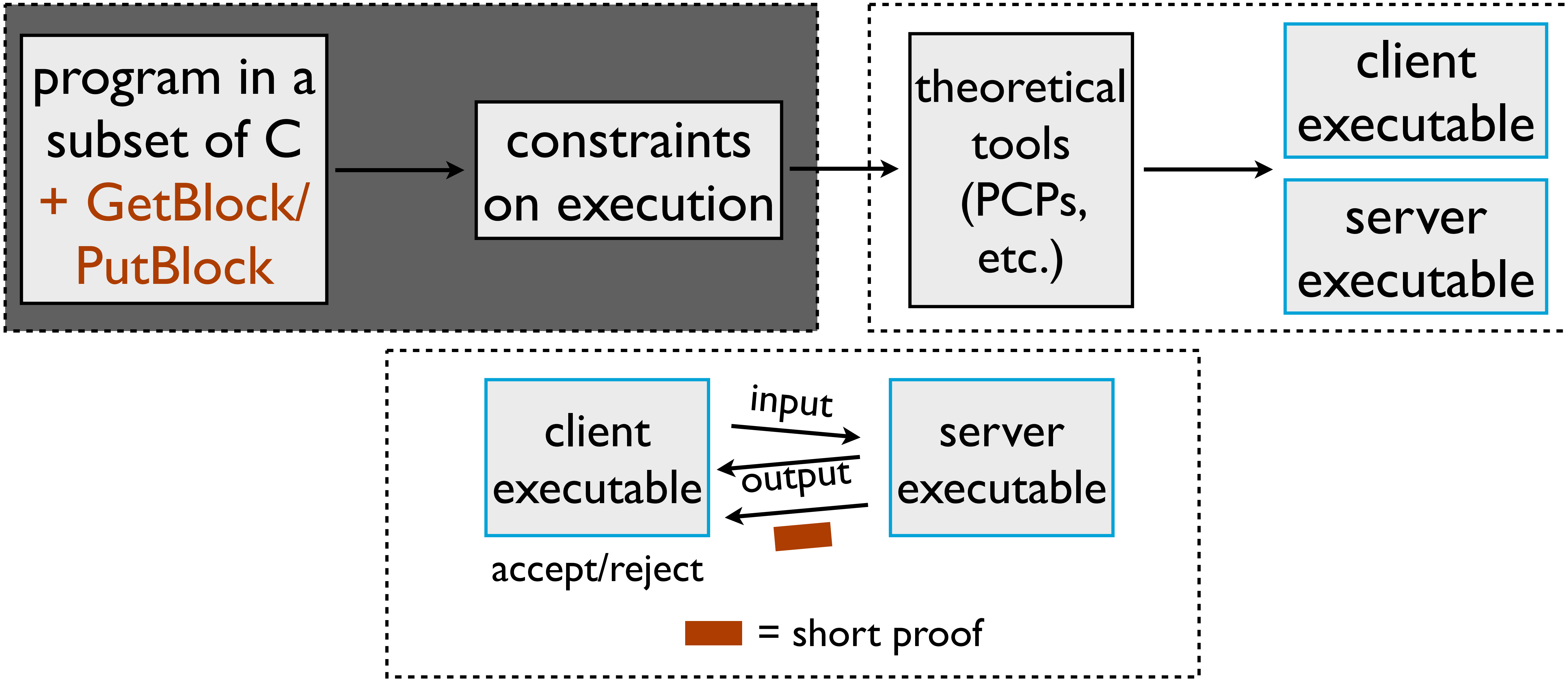
$d_{25} = B_{73} \bullet B_{84} + ..$

$Y = B+1$

Representation of H in constraints

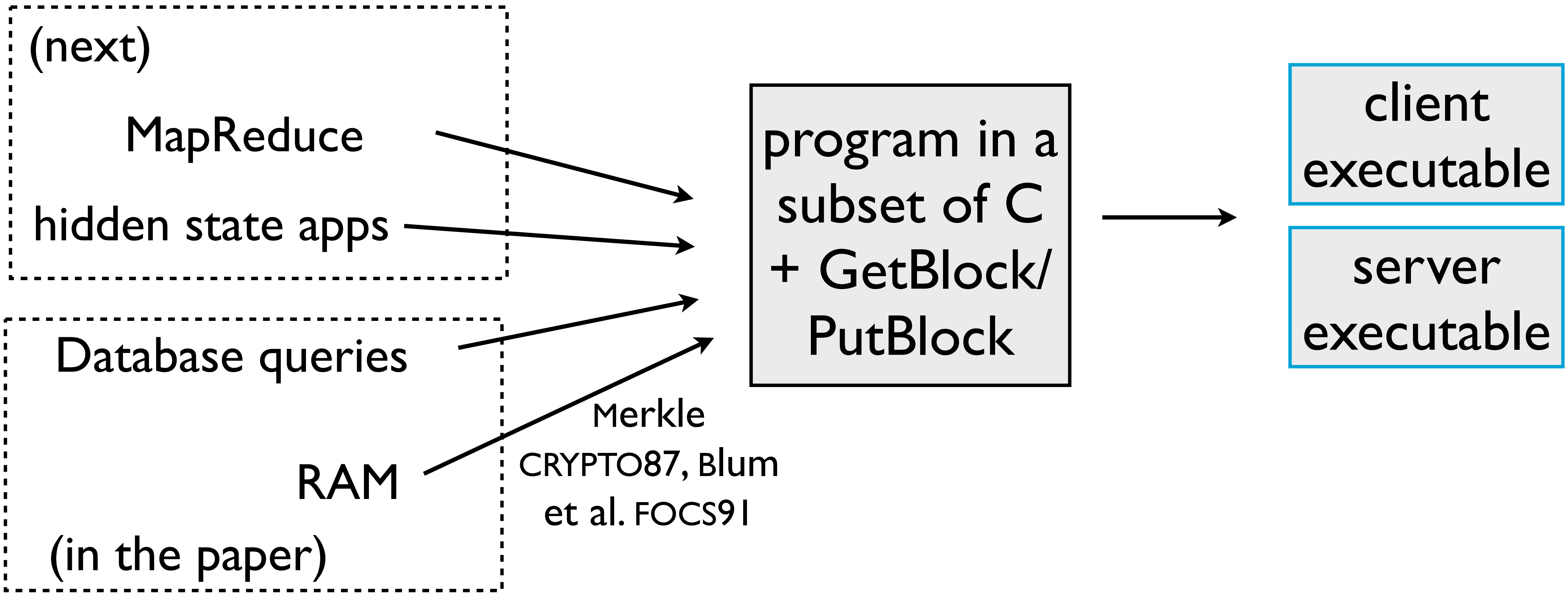
We use a hash function that has an efficient representation as a set of constraints [Ajtai STOC96]

Pantry: an extension to Zaatar and Pinocchio



- a valid proof ⇔ "I know a satisfying assignment to constraints"
- satisfiability of constraints ⇔ hash checks pass
- hash checks pass ⇔ correct storage interaction

Verifiable stateful applications from C code with Pantry:



- The computations have to be stateless

- The client incurs a large setup cost

- The server's overheads are large

1

2

[Eliminate]

3

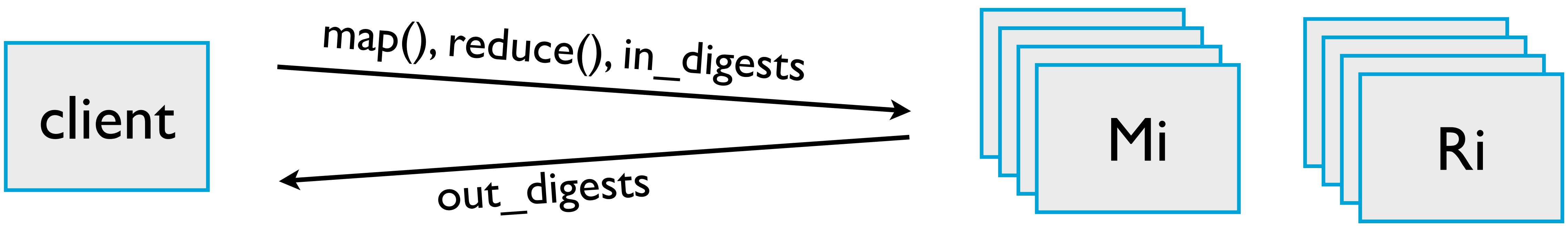
[Mitigate]

[Retain]

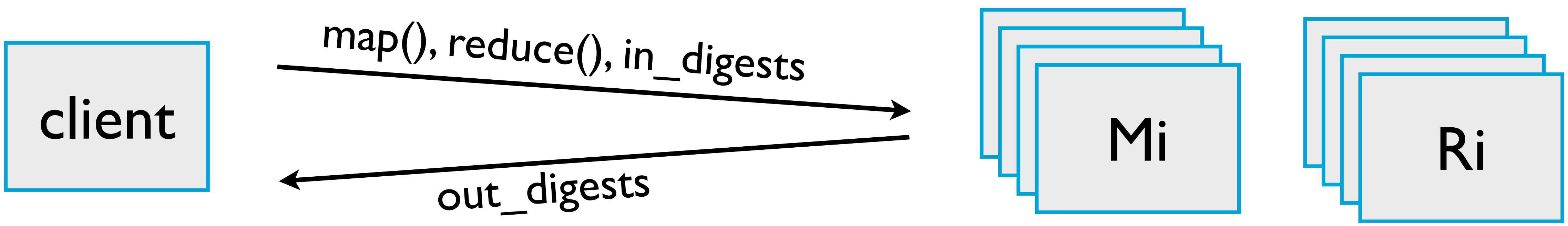
Pantry enables applications where the client's setup costs are tolerable:

- Data parallel computations (MapReduce, etc.) that compute over remote state
  - Have multiple identical computations
- Hidden state applications
  - The client cannot, in principle, execute on its own

The client is assured that a MapReduce job was executed correctly—without ever touching the data



The client is assured that a MapReduce job was executed correctly—without ever touching the data

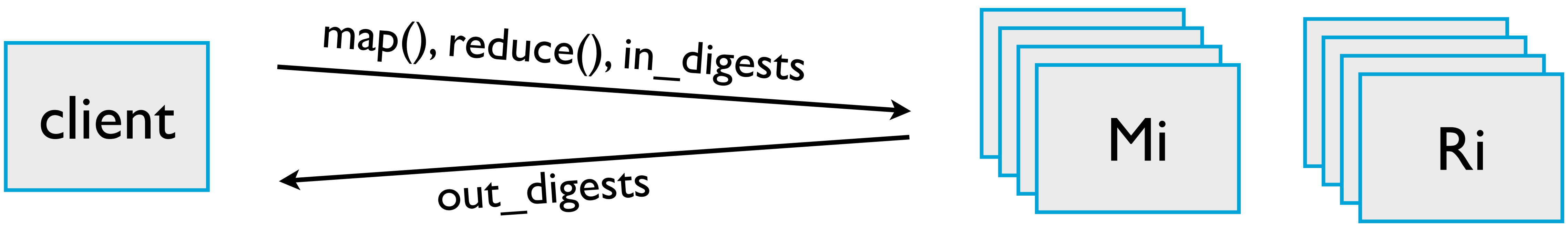


map() and reduce() are expressed in Pantry's subset of C

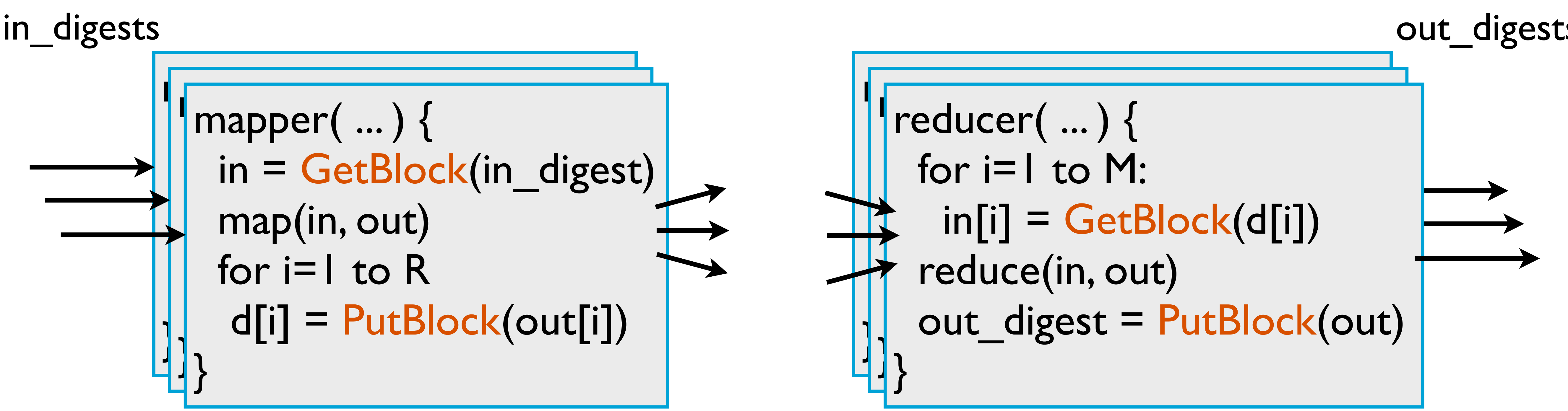
```
mapper(Dig in_digest, Dig *d) {
  in = GetBlock(in_digest)
  map(in, out)
  for i=1 to R
   d[i] = PutBlock(out[i])
}
```

```
reducer(Dig *d, Dig *out_digest) {
  for i=1 to M:
    in[i] = GetBlock(d[i])
  reduce(in, out)
  out_digest = PutBlock(out)
}
```

The client is assured that a MapReduce job was executed correctly—without ever touching the data



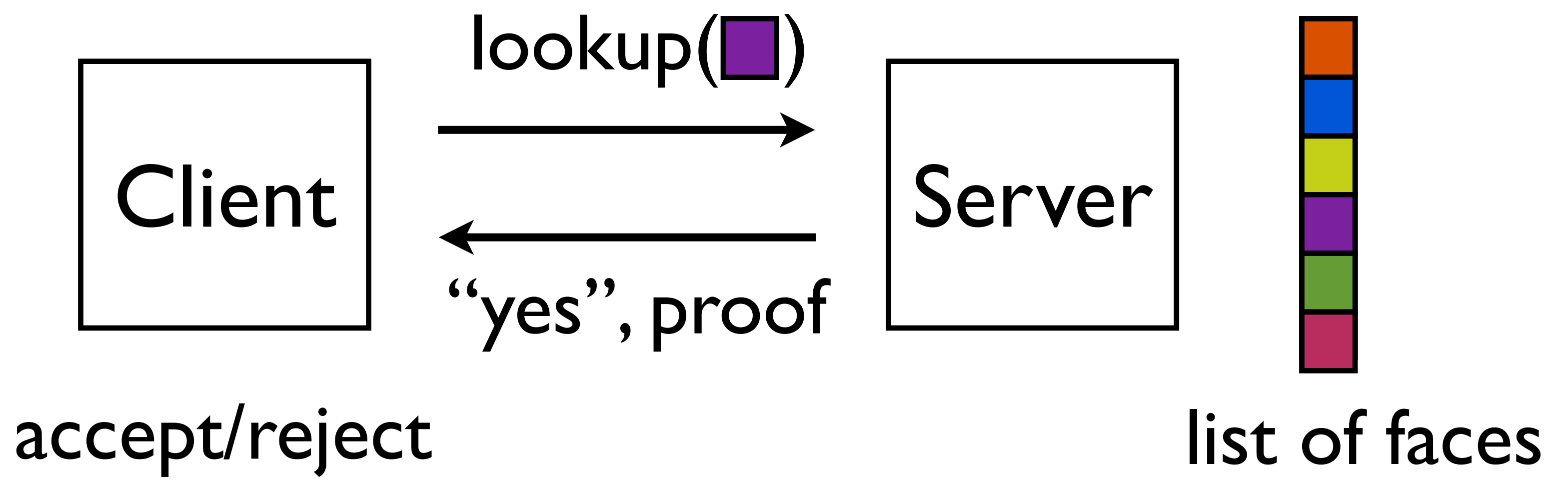
The two phases are handled separately:



Pantry enables applications where the client's setup costs are tolerable:

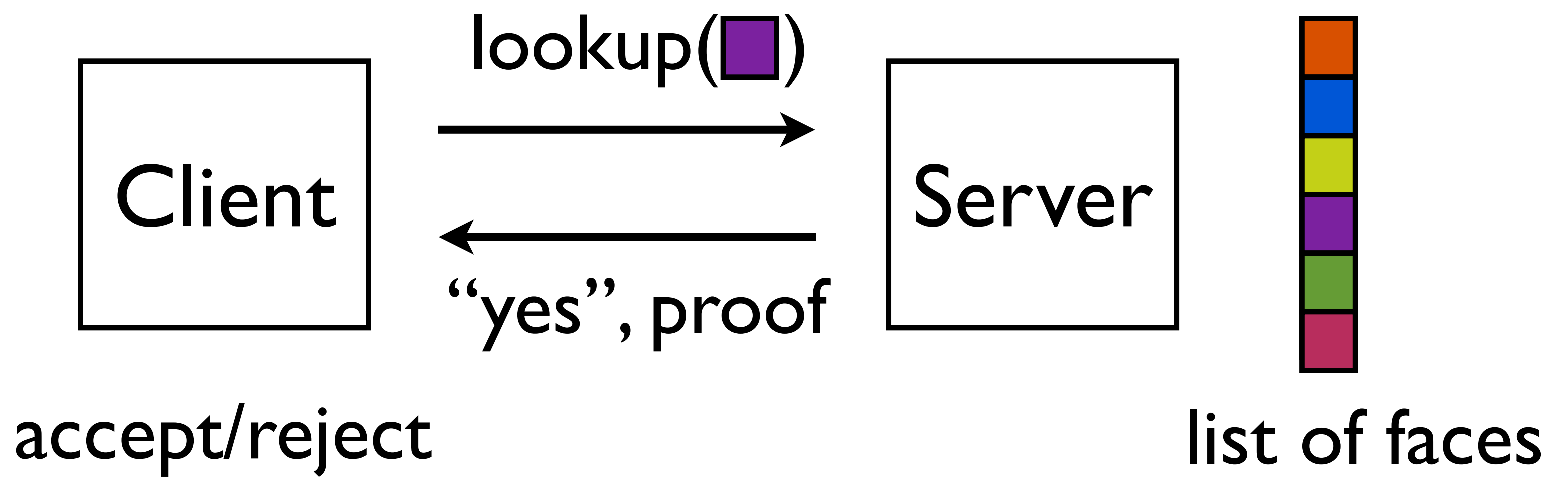
- Data parallel computations (MapReduce, etc.) that compute over remote state
  - Have multiple identical computations
- Hidden state applications
  - The client cannot, in principle, execute on its own

Hidden state applications



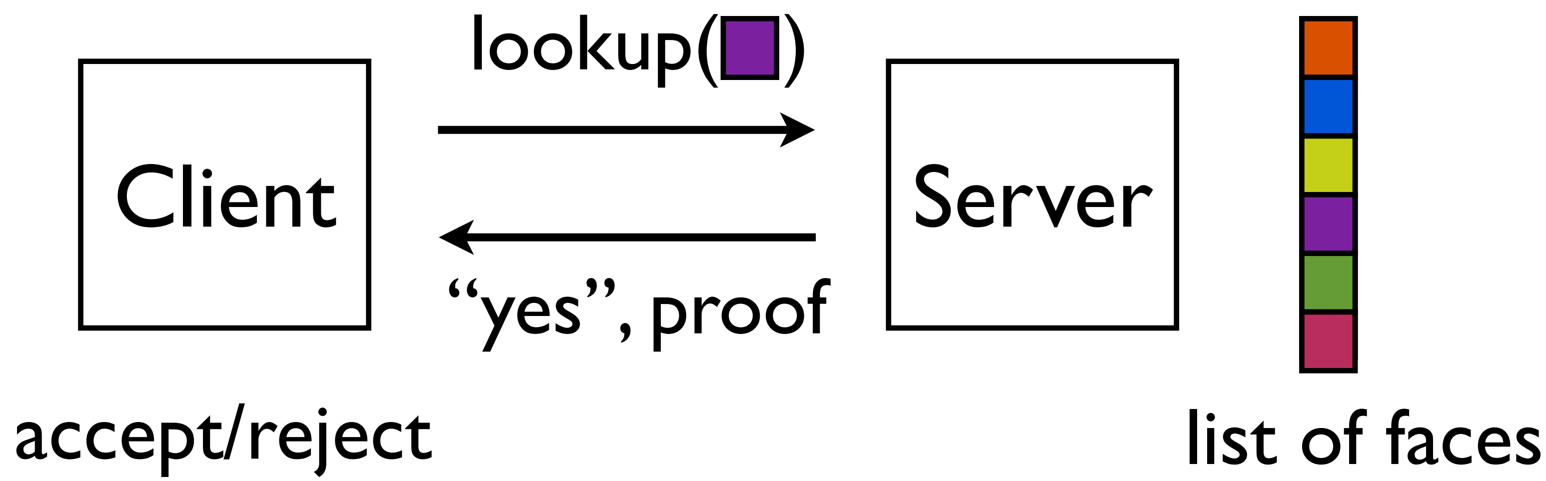
- Key idea: Pantry's storage + Pinocchio's zero-knowledge

Hidden state applications



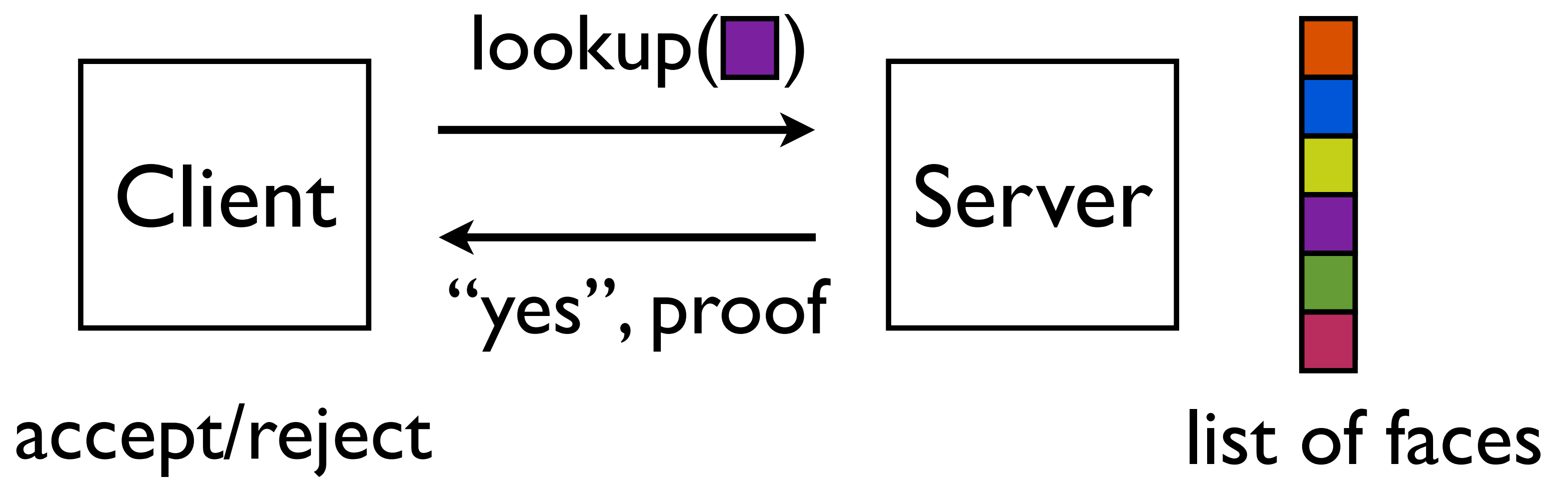
- Key idea: Pantry's storage + Pinocchio's zero-knowledge
- Wrinkles:
  - Pantry's digests aren't information hiding (wrap digests with a cryptographic commitment scheme)
  - Standard commitment schemes are expensive (use an HMAC-based scheme that is 10x cheaper)

Hidden state applications



- Key idea: Pantry's storage + Pinocchio's zero-knowledge
- Wrinkles:
  - Pantry's digests aren't information hiding (wrap digests with a cryptographic commitment scheme)
  - Standard commitment schemes are expensive (use an HMAC-based scheme that is 10X cheaper)
- Other applications: tolling, regression analysis, etc.

Hidden state applications



- Key idea: Pantry's storage + Pinocchio's zero-knowledge
- Wrinkles:
  - Pantry's digests aren't information hiding (wrap digests with a cryptographic commitment scheme)
  - Standard commitment schemes are expensive (use an HMAC-based scheme that is 10X cheaper)
- Other applications: tolling, regression analysis, etc.
- Upshot: with only C programs, one can get powerful guarantees

Benchmark applications and implementation

Benchmark applications:

- MapReduce: nucleotide substring search, dot product, nearest neighbor search, and covariance computation
- Hidden state: face matching, tolling, and regression analysis

Distributed implementation of the server

C++, Java, Go, and Python code; HTTP/Open MPI to distribute server's work

Evaluation questions

1. When does Pantry's client save resources relative to locally executing the computation?

2. What are the costs of supporting hidden state?

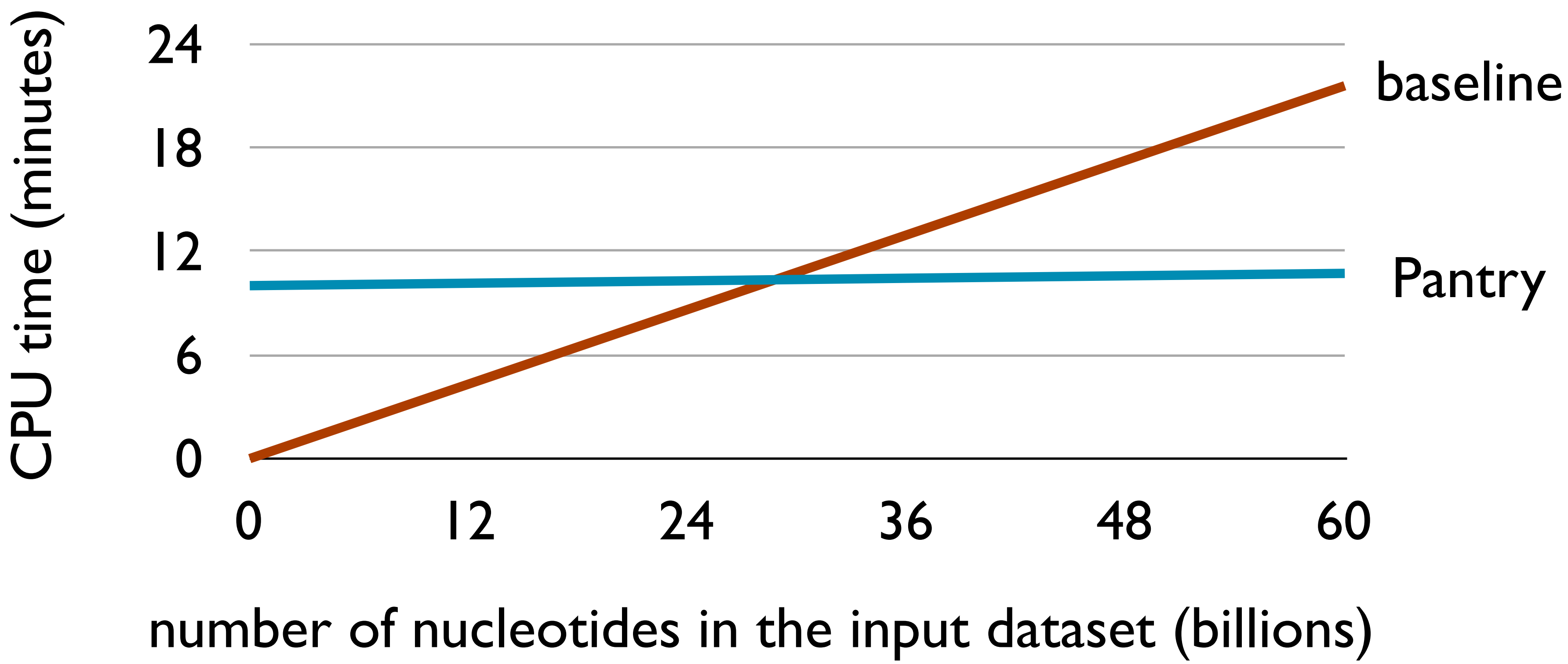
3. What are the costs of Pantry's server, relative to simply executing the computation?

Pantry's client saves resources at sufficiently large input sizes

MapReduce job: nucleotide substring search in which a mapper gets 600K nucleotides and outputs matching locations

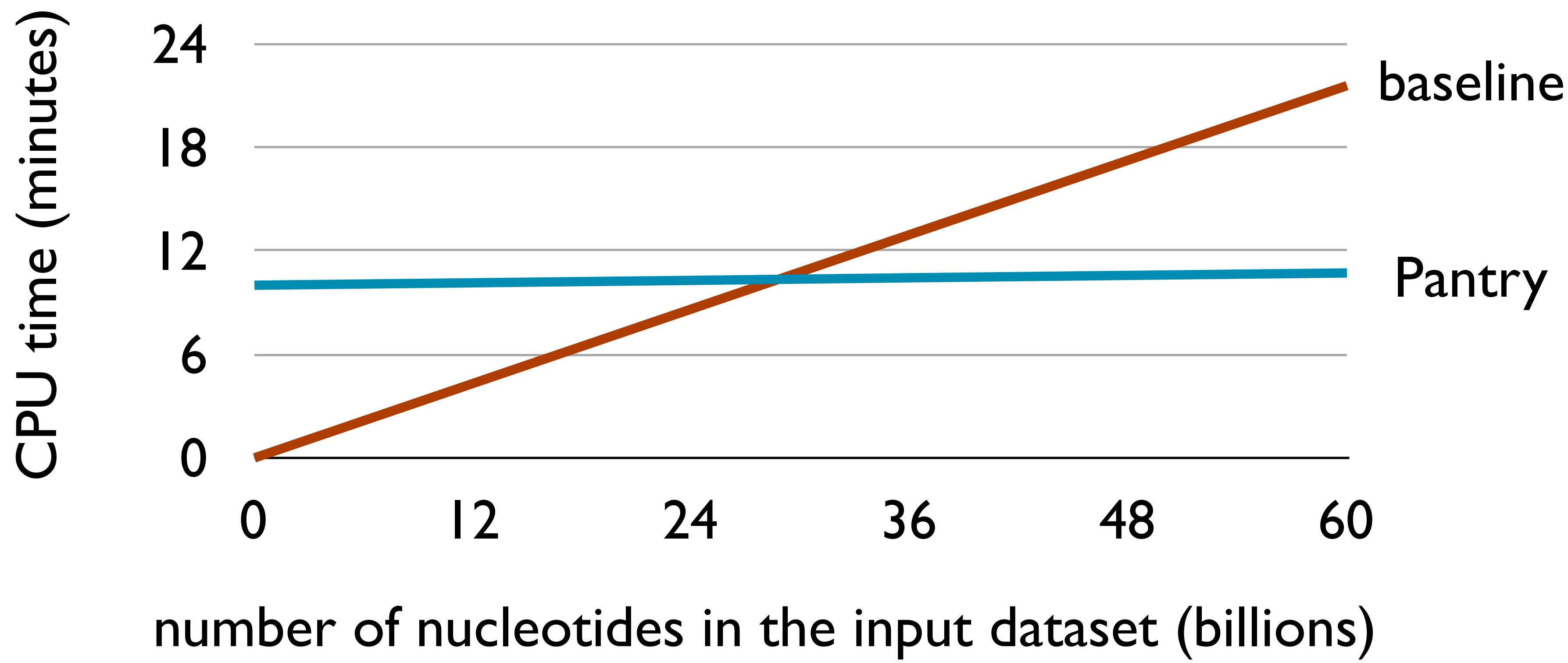
Pantry's client saves resources at sufficiently large input sizes

MapReduce job: nucleotide substring search in which a mapper gets 600K nucleotides and outputs matching locations



Pantry's client saves resources at sufficiently large input sizes

MapReduce job: nucleotide substring search in which a mapper gets 600K nucleotides and outputs matching locations



Graph is an extrapolation (slopes and y intercepts determined with experiments that use up to 250 machines and up to 1.2 billion nucleotides)

Cost of supporting hidden state applications

Server holds 128 face fingerprints (hidden state: 15 KB)

good news:

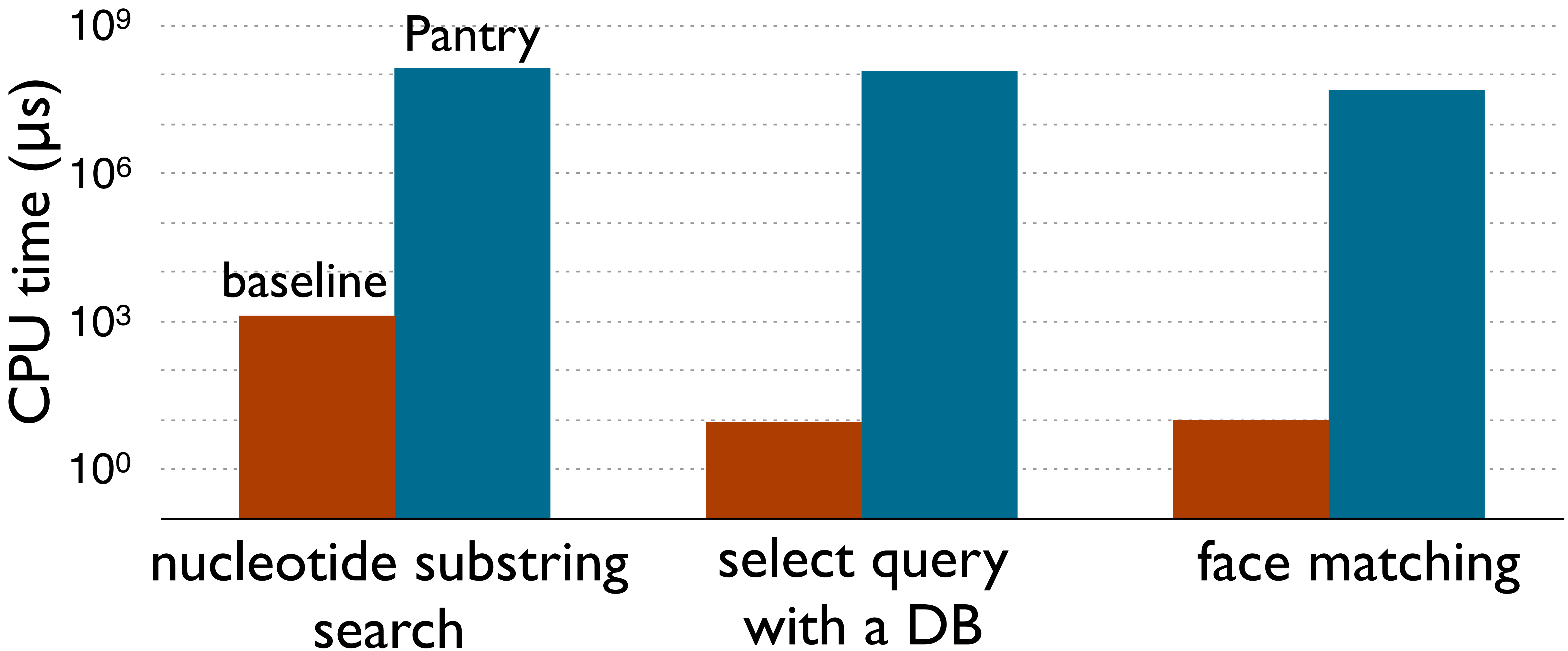
- proof size: 288 bytes
- client’s CPU time: 7 ms

bad news:

- network (setup), server’s storage (ongoing): 170 MB
- server’s CPU time: 7.8 min

Pantry's server's cost is many orders of magnitude slower than simply executing the computation

sources of overhead: constraints + crypto ops. proportional to #constraints



Recap

- The computations have to be stateless [Eliminate]

- The client incurs a large setup cost [Mitigate]

- The server's overheads are large [Retain]

Prior work on verifiable computation

Make assumptions about the server's failure modes or give up generality:

Replication [Castro & Liskov TOCS02], trusted hardware [Chiesa & Tromer ICS10, Sadeghi et al. TRUST10], and auditing [Monrose et al. NDSS99, Haeberlen et al. SOSP07]

Special-purpose [Freivalds MFCS79, Golle & Mironov RSA01, Sion VLDB05, Benabbas et al. CRYPTO11, Boneh & Freeman EUROCRYPT11]

Unconditional guarantees and general but not geared to practice:

Use fully homomorphic encryption [GGP, Chung et al.CRYPTO10]

Theory of PCPs, IPs, arguments [GMR85, Ben-Or et al. STOC88, Babai et al. STOC91, Kilian STOC92, ALMSS92, AS92, Goldwasser et al. STOC 2008, Bitansky et al. ITCS12]

Four projects have produced implementations

Pepper, Ginger, Zaatar, Allspice	HotOS11 NDSS12 USENIX SECURITY12 EuroSys13 IEEE S&P13
CMT, Thaler	Cormode et al. ITCS12 Thaler et al. HotCloud12 Thaler CRYPTO13
Pinocchio, GGPR	Gennaro et al. EUROCRYPT13 Parno et al. IEEE S&P13
BCGTV	Ben-Sasson et al. CRYPTO13 Ben-Sasson et al. ITCS13 Bitansky et al. TCC13

Next steps for the area of verifiable computing

- Reducing the server’s overhead (currently 3-6 orders of magnitude more than native execution)

- Avoiding the client’s setup costs efficiently

- Enhancing the computational model (currently loops are unrolled, storage operations need a lot of constraints, etc.)

Takeaways

- Pantry takes another step in bringing powerful theory behind verifiable computation into practice
  - Pantry enables realistic, stateful computations: MapReduce, database queries, hidden state applications, etc.

- We think: the machinery underlying Pantry or its variant will be a key tool in building future secure systems