



VERIFYING COMPUTATIONS WITH STATE

BENJAMIN BRAUN, ARIEL FELDMAN[†], ZUOCHENG REN, SRINATH SE9Y,
ANDREW BLUMBERG, AND MICHAEL WALFISH

PRESENTED BY: EHSAN HEMMATI

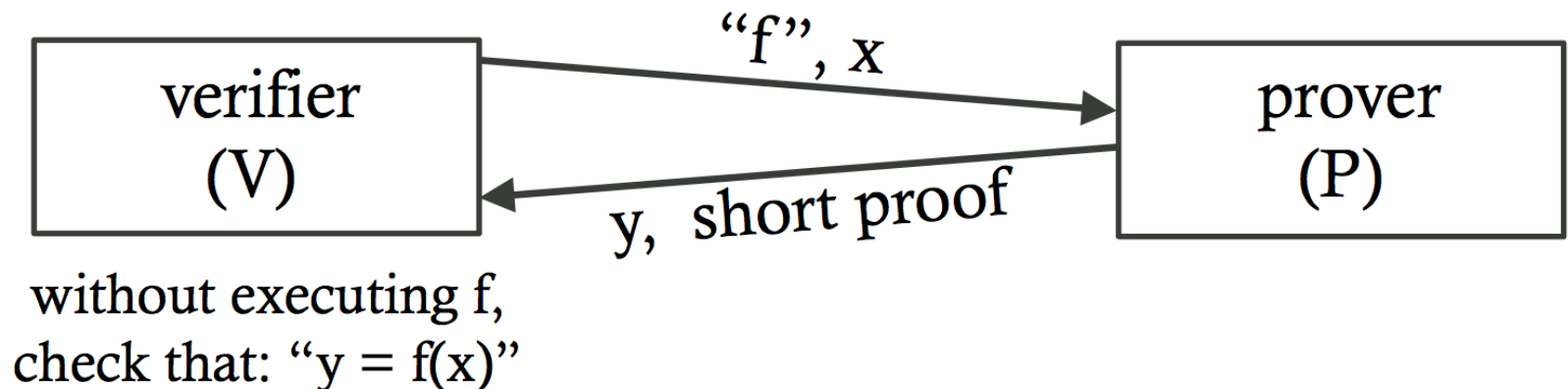
VERIFIABLE COMPUTATION

- VERIFIABLE COMPUTING:
 - \approx DELEGATION OF COMPUTATION
 - \approx SUCCINCT ARGUMENTS
 - \approx EXECUTION INTEGRITY
 - \approx CERTIFIED COMPUTATION
- ENABLING A COMPUTER TO OFFLOAD THE COMPUTATION OF SOME FUNCTION, TO OTHER PERHAPS UNTRUSTED CLIENTS
- THE OTHER CLIENTS EVALUATE THE FUNCTION AND RETURN THE RESULT WITH A PROOF THAT THE COMPUTATION OF THE FUNCTION WAS CARRIED OUT CORRECTLY.

VERIFIABLE COMPUTATION - INTRO

CLASSIC AND FUNDAMENTAL PROBLEM

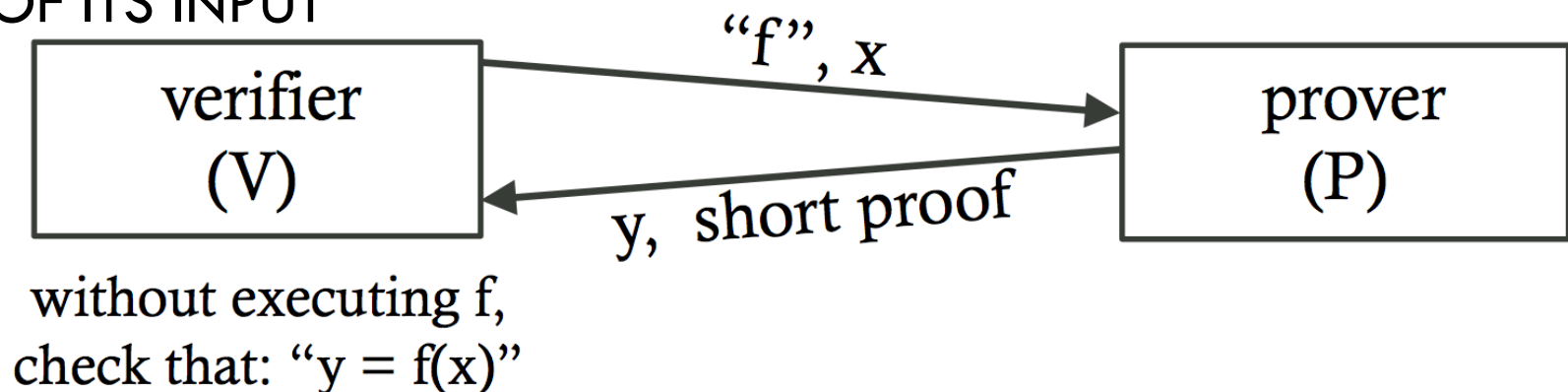
- CLOUD COMPUTING (CONSIDER LARGE DISTRIBUTED JOBS)
- INFORMATION RETRIEVAL (CONSIDER A QUERY AGAINST A REMOTE DATABASE)
- HARDWARE SUPPLY CHAIN (CONSIDER POTENTIALLY ADVERSARIAL CHIPS)
- GENERALIZES TO VERIFYING ASSERTIONS
- MANY OTHER APPLICATIONS



VERIFIABLE COMPUTATION- INTRO

MANY VARIANTS OF THE SETUP

- PROOF DELIVERED OVER ROUNDS OF INTERACTION
- MORE GENERAL CLAIM: “THERE EXISTS A W SUCH THAT $Y = F(X,W)$ ”
 - ... AND FURTHERMORE P “KNOWS” W
 - ... AND FURTHERMORE P CAN KEEP W PRIVATE
- DIFFERENT ASSUMPTIONS (UNCONDITIONAL VS. STANDARD VS. FUNKY)
- V CANNOT ACCESS ALL OF ITS INPUT



- CLASSIC AND FUNDAMENTAL PROBLEM

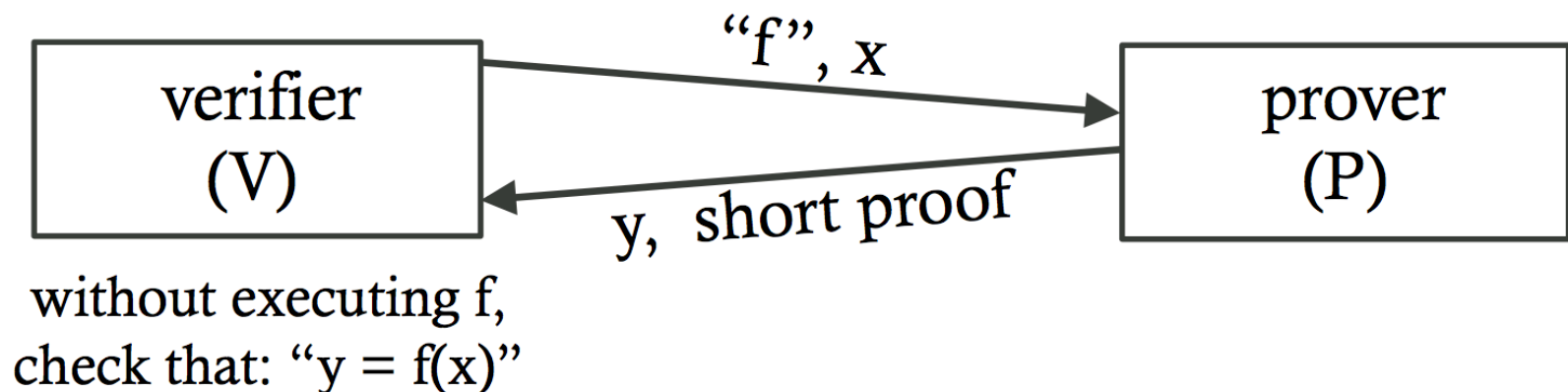
- MANY APPLICATIONS (CLOUD COMPUTING, INFORMATION RETRIEVAL, UNTRUSTED HARDWARE SUPPLY CHAIN, ETC.). GENERALIZES TO VERIFYING ASSERTIONS.

- MANY VARIANTS OF THE SETUP

- COMMONALITY: V GETS ASSURANCE THAT P PERFORMED A TASK AS DIRECTED, WITHOUT REDOING P'S WORK AND WITHOUT ACCESS TO P'S RESOURCES OR INPUTS.

- NOTE: PROGRAM CORRECTNESS IS COMPLEMENTARY

- PROGRAM CORRECTNESS ESTABLISHES THAT F IS CONSISTENT WITH A SPECIFICATION. IN OUR CONTEXT, F IS A (POSSIBLY BUGGY) GIVEN AND IS THE DIRECTIVE FOR P.



IMPLEMENTATION

- PROOF-BASED VERIFIABLE COMPUTATION
- MANY OF THE VARIANTS ARE ADDRESSABLE IN THEORY, WITH PROBABILISTIC PROOF PROTOCOLS (PROBABILISTICALLY CHECKABLE PROOFS – PCPS)
- STRONG GUARANTEES AND IS USUALLY PHRASED AS DEFENDING AGAINST AN ARBITRARILY MALICIOUS PROVER
- MALICIOUSNESS IS NOT AN ACCUSATION BUT RATHER A COMPREHENSIVE MODEL THAT INCLUDES BENIGN MALFUNCTIONS WITH UNPREDICTABLE EFFECTS.

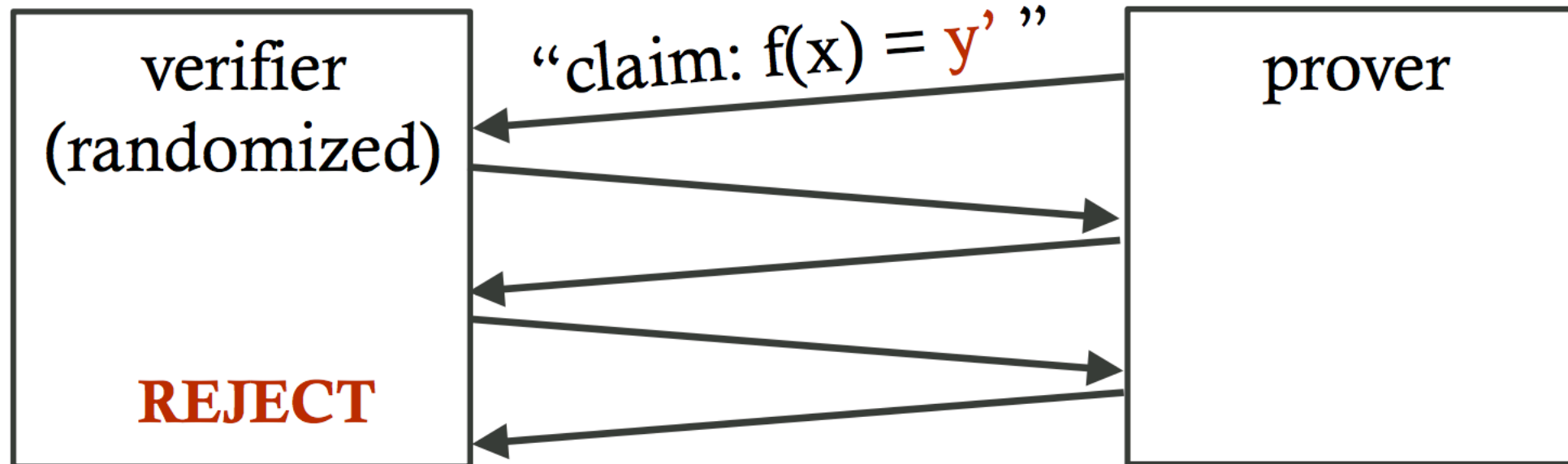
IMPLEMENTATION

- COMPLETENESS:

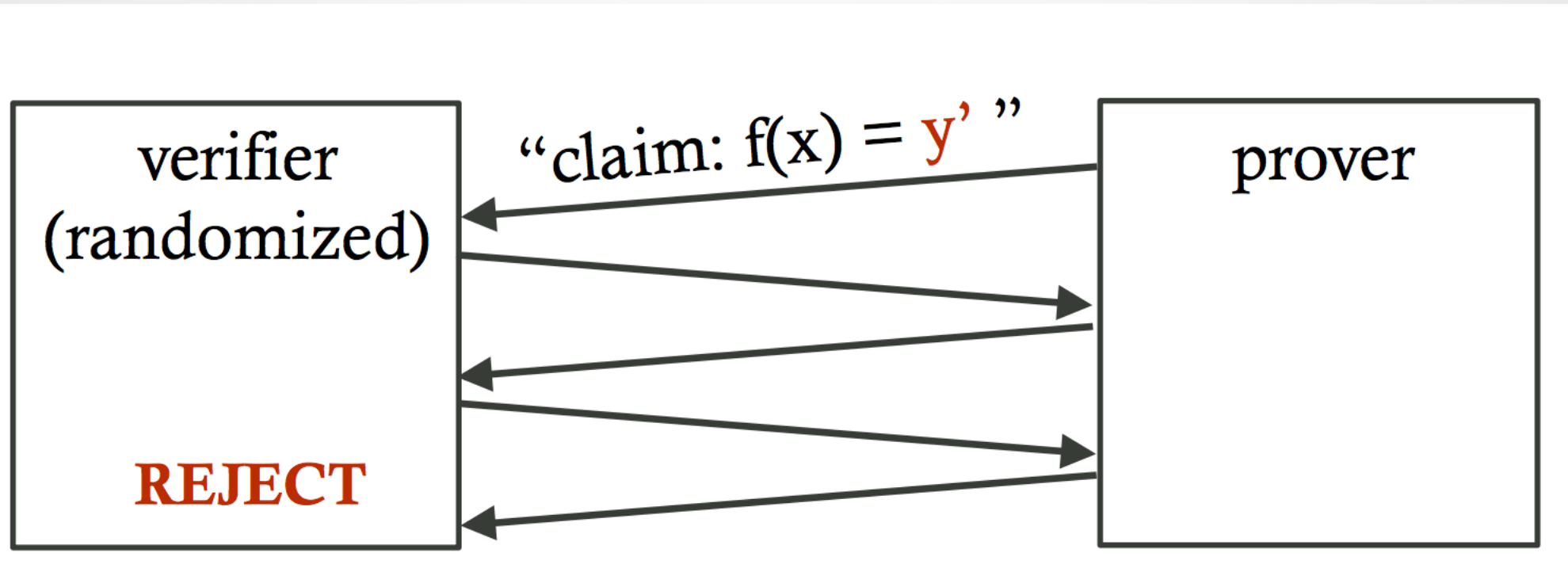
$y = \Psi(x)$, then if \mathcal{P} follows the protocol, $\Pr\{\mathcal{V} \text{ accepts}\} = 1$.

- SOUNDNESS:

$y \neq \Psi(x)$, then $\Pr\{V \text{ rejects}\} > 1 - \epsilon$, where ϵ can be made small.



IMPLEMENTATION



IMPLEMENTATION

- **GOOD NEWS:**
 - RUNNING CODE; COST REDUCTIONS OF 1020 VS. THEORY
 - COMPILERS FROM C TO VERIFIABLE COMPUTATIONS
 - CONCRETELY EFFICIENT VERIFIERS
- **EQUIVOCAL NEWS:**
 - SMALL COMPUTATIONS, EXTREME EXPENSE, ETC.
 - USEFUL ONLY FOR SPECIAL-PURPOSE APPLICATIONS
- **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

BROAD APPROACHES TO VERIFIABLE COMPUTATION

A. MAKE USAGE ASSUMPTIONS

- REPLICATION [MR97, CL99, CRR11]
- ATTESTATION [PMP11, SLSPDK05]
- TRUSTED HARDWARE [CT10, SSW10]
- AUDITING [MWR99, HKD07]

• B. RESTRICT THE CLASS OF COMPUTATIONS

- [FREIVALDS77, GM01, SION05, MSG07, KSC09, BGV11, BF11, BZF11, FG12, ...]

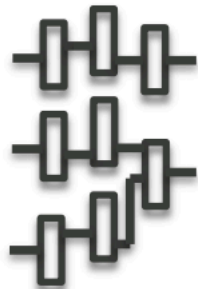
• C. STRIVE FOR GENERALITY

SYNOPSIS OF THE RESEARCH AREA

front-end
(program translator)

```
main(){  
  ...  
}
```

C program



arithmetic circuit



back-end
(probabilistic proof protocol)

verifier

x

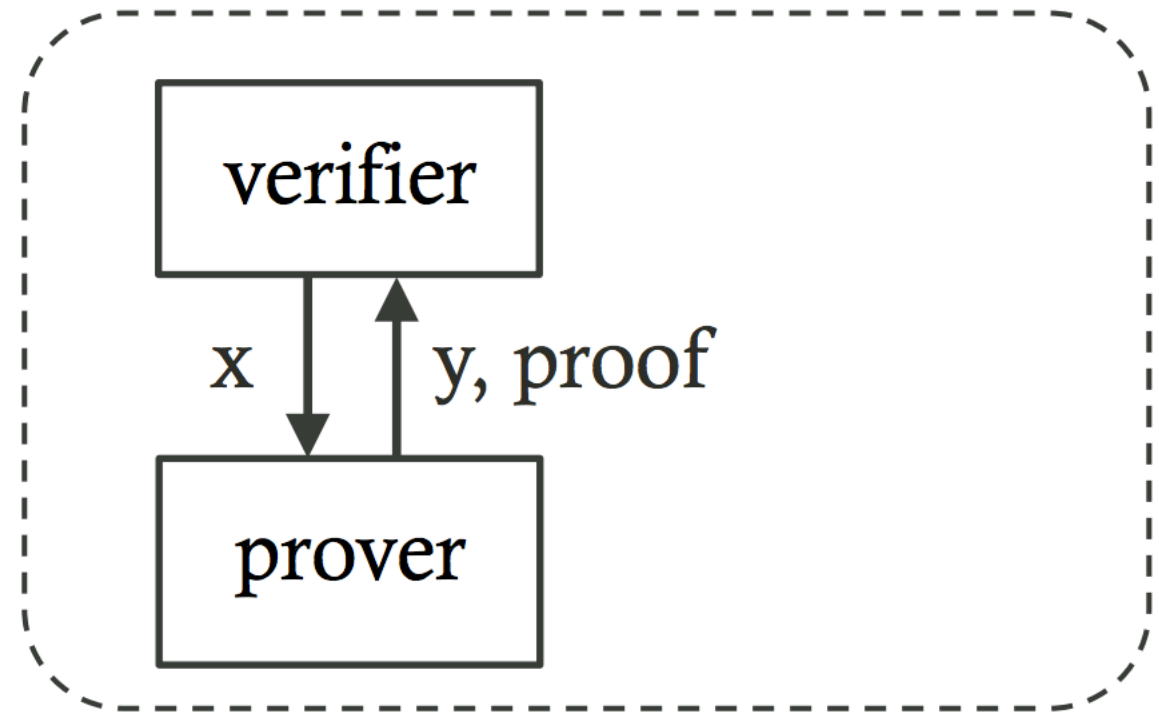
y, proof

prover



- what circuits does it handle?
- what assumptions are needed?
- what are its properties?
- what is the number of messages?
- what are the costs?
- what costs can be amortized?
- what are the mechanics?

back-end
(probabilistic proof protocol)

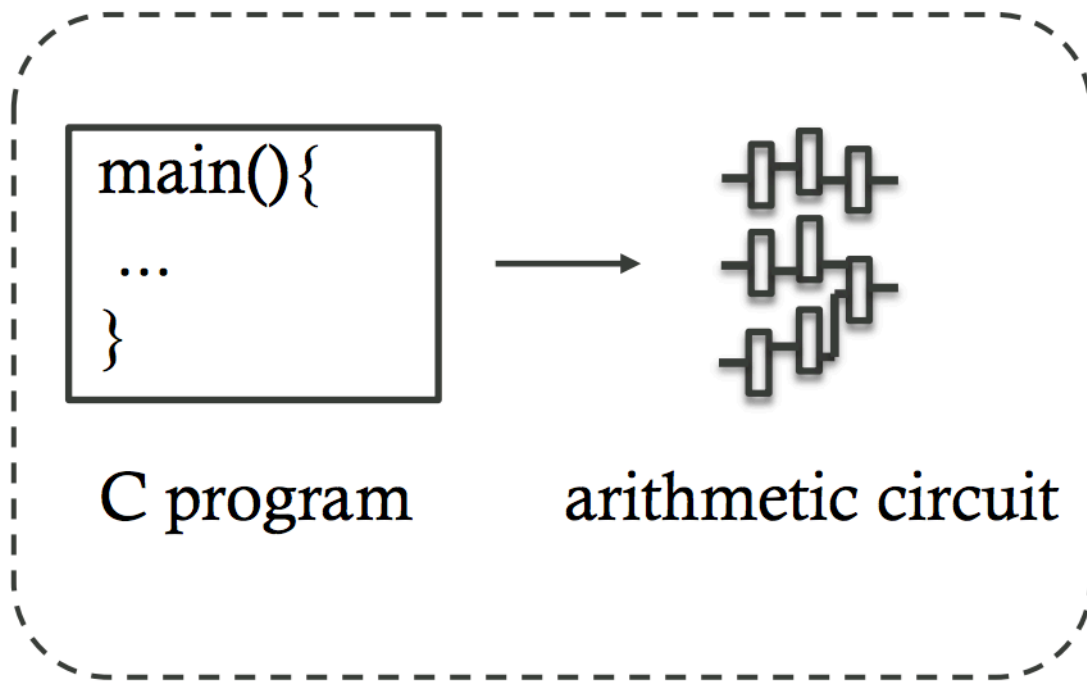


interactive proof

interactive argument

non-interactive argument

front-end (program translator)



- how expressive is it?
- what is programming like?
- how does translation work?
- what are the costs of different program structures?
- how can programs refer to external state?

circuits with repeated structure
circuits without repeated structure
circuits w/ non-deterministic input
“universal” circuits

} “ASIC”
} “CPU”

HOW ARE PROBABILISTIC PROOFS DEFINED?

THERE ARE MANY DEFINITIONS AND VARIANTS:

A PROBABILISTIC PROOF FOR A LANGUAGE L IS AN INTERACTING VERIFIER V_L (WHICH IS PPT) AND PROVER P_L (WHOSE POWER VARIES DEPENDING ON THE DEFINITION). LET $(V, P)(A)$ DENOTE THE INTERACTION BETWEEN V AND P ON INSTANCE A . IF $(V, P)(A) = 1$, V IS SAID TO “ACCEPT” THE INTERACTION. THE INTERACTION MUST MEET:

- COMPLETENESS:

$$\text{If } a \in L, \Pr\{(V_L, P_L)(a) = 1\} = 1.$$

- SOUNDNESS:

$$\text{If } a \notin L, \text{ then } \forall P', \Pr\{(V_L, P')(a) = 1\} < \varepsilon, \text{ for some fixed, constant } \varepsilon.$$

- EFFICIENCY: THE HONEST P (THAT IS, P_L) SHOULD HAVE RUNNING TIME THAT IS POLYNOMIAL (AND IDEALLY LINEAR OR QUASILINEAR) IN THE TIME TO COMPUTE OR DECIDE L (AS NOTED EARLIER, THE ASSUMED POWER OF A DISHONEST P DEPENDS ON THE KIND OF PROBABILISTIC PROOF). V 'S RUNNING TIME IS IDEALLY CONSTANT OR LOGARITHMIC IN THE TIME TO COMPUTE OR DECIDE L ; SAME WITH THE COMMUNICATION COMPLEXITY.

VARIANTS: COMPUTATIONAL SOUNDNESS, NON-DETERMINISTIC LANGUAGES, PROOF OF KNOWLEDGE, ZERO KNOWLEDGE.

LANGUAGE

WHAT LANGUAGE SHOULD WE USE FOR “CORRECT PROGRAM EXECUTION”?

- **BOOLEAN CIRCUIT SATISFIABILITY**

WE USE THIS TERM TO REFER TO THE LANGUAGE OF TRIPLES (C, X, Y) WHERE A BOOLEAN CIRCUIT C , IF GIVEN INPUT X , PRODUCES OUTPUT Y . THIS IS SLIGHTLY NON-STANDARD, BUT IT MATCHES THE PROBLEM SETUP IN DELEGATION.

- **ARITHMETIC CIRCUIT SATISFIABILITY**

SIMILAR TO PRIOR ONE, BUT NOW: THE CIRCUIT IS OVER A LARGE FINITE FIELD, THE WIRES ARE INTERPRETED AS FIELD ELEMENTS, AND THE GATES ARE INTERPRETED AS FIELD OPERATIONS (ADD, MULTIPLY).

- **NON-DETERMINISTIC (BOOLEAN, ARITH.) CIRCUIT SATISFIABILITY**

NOW WE IMAGINE THAT THE CIRCUIT TAKES SOME UNCONSTRAINED INPUT (LABEL IT w), AND THIS LANGUAGE IS ALL TRIPLES (C, X, Y) FOR WHICH THERE EXISTS SOME $w = w$ SUCH THAT $C(X, w) = Y$.

“SATISFIABILITY” ENTERS BECAUSE THERE ARE IMPLICIT CONSTRAINTS. SOMETIMES IT IS EASIER TO WORK WITH CONSTRAINTS EXPLICITLY

A CONVENIENT LANGUAGE

ARITHMETIC CONSTRAINT SATISFIABILITY

SYSTEM OF EQUATIONS IN FINITE FIELD F

A COMPUTATION F IS EQUIVALENT TO CONSTRAINTS C IF:

C IS CONSTRAINTS OVER VARIABLES (X, Y, Z) AND FIELD F S.T. (DET CASE) $\forall X, Y: Y=F(X) \Leftrightarrow C(X=X, Y=Y)$
IS SATISFIABLE (NON-DET. CASE) $\forall X, Y: (\exists W \text{ S.T. } Y=F(X, W)) \Leftrightarrow C(X=X, Y=Y)$ IS SATISFIABLE

TERMINOLOGY: CONSTRAINTS C ARE SAID TO BE AN ARITHMETIZATION OF THE COMPUTATION F .

increment-by-one

```
f(X) {  
  Y = X + 1;  
  return Y;  
}
```

[equivalent]

$$\left\{ \begin{array}{l} 0 = Z - X, \\ 0 = Z - Y + 1 \end{array} \right\}$$

A CONVENIENT LANGUAGE

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


$$\begin{aligned}0 &= X - i \\0 &= Y - (X + 1) \\0 &= Y - r\end{aligned}$$

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

$$\begin{aligned}0 &= X - 6 \\0 &= Y - (X + 1) \\0 &= Y - 7\end{aligned}$$

There is a solution

If the output is 8

$$\begin{aligned}0 &= X - 6 \\0 &= Y - (X + 1) \\0 &= Y - 8\end{aligned}$$

There is no solution

Example: “ $Y = (X1 \neq X2)$ ”

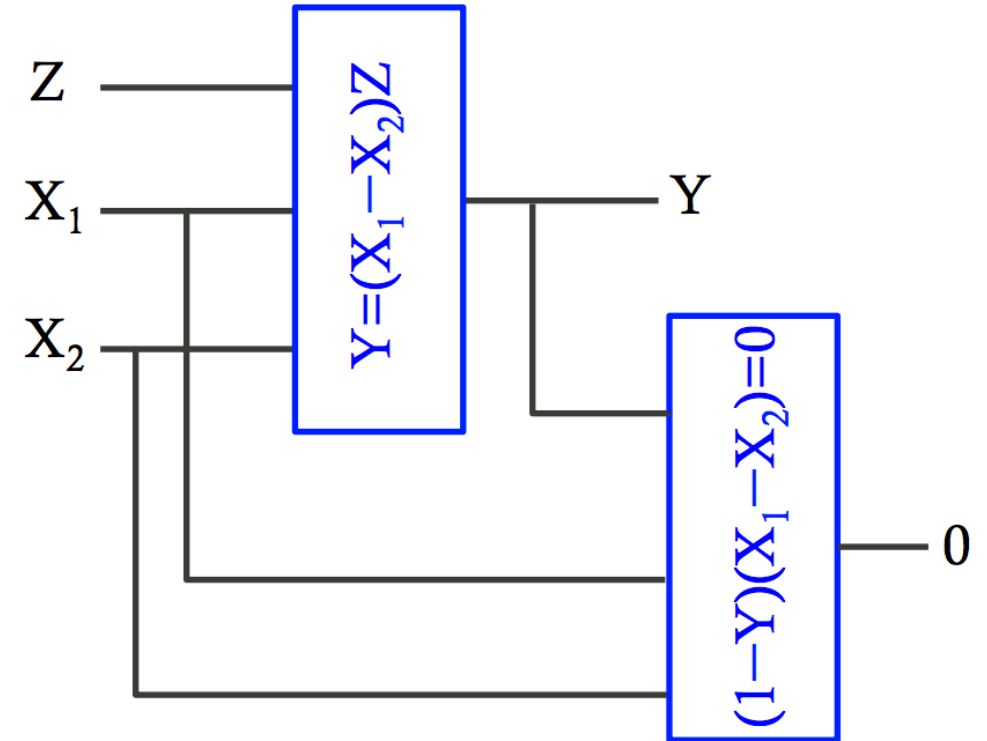
$$0 = (X1 - X2) \cdot M - Y$$

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

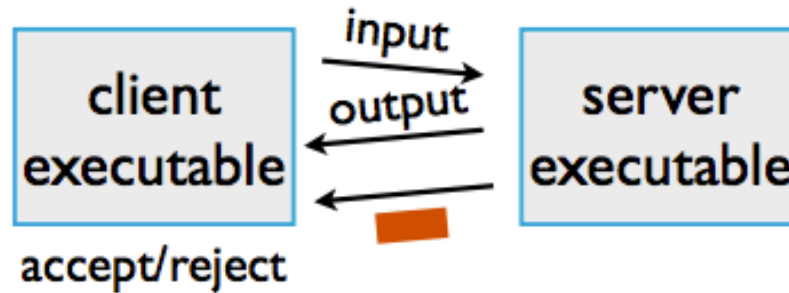
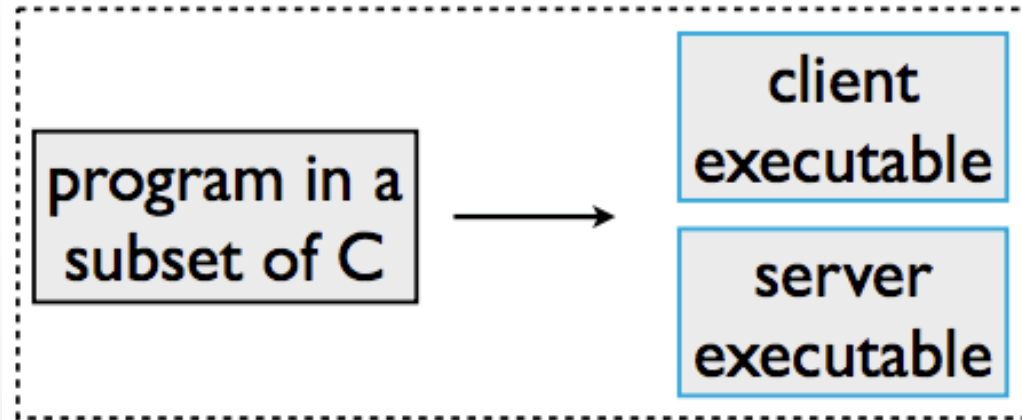
Observe:


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

if $X1 \neq X2$, then Y must be 1, to satisfy the second.



PANTRY



 = short proof

PANTRY - INTRO

$$y = \Psi(x)$$

1. Ψ IS REPRESENTED AS CONSTRAINTS

- A SET OF CONSTRAINTS C IS A SYSTEM OF EQUATIONS IN VARIABLES (X, Y, Z) , OVER A LARGE FINITE FIELD, F
- VARIABLES X AND Y REPRESENT THE INPUT AND OUTPUT VARIABLES
- IF FOR SOME Z , SETTING $Z=z$ MAKES ALL CONSTRAINTS IN $C(X=x, Y=y)$ HOLD SIMULTANEOUSLY, THEN $C(X=x, Y=y)$ IS SAID TO BE SATISFIABLE, AND z IS A SATISFYING ASSIGNMENT.

2. P COMPUTES AND IDENTIFIES A SATISFYING ASSIGNMENT

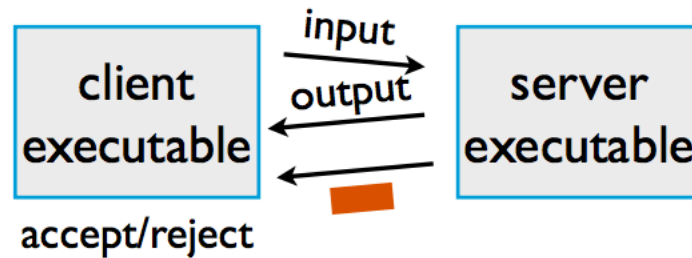
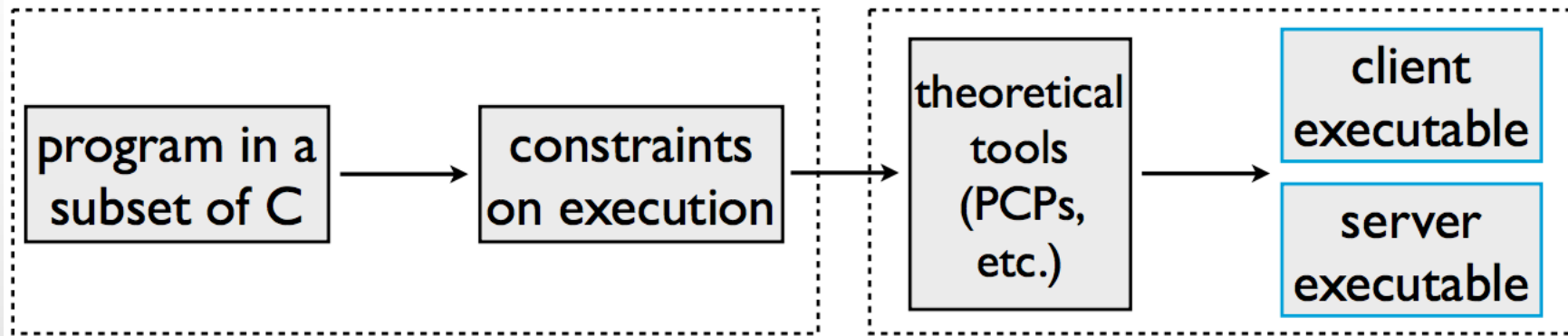
- P "EXECUTES" $\Psi(x)$ BY IDENTIFYING A SATISFYING ASSIGNMENT TO THE EQUIVALENT CONSTRAINTS $C(X=x)$

3. P ARGUES THAT IT HAS A SATISFYING ASSIGNMENT

- P WANTS TO PROVE TO V THAT IT KNOWS A SATISFYING ASSIGNMENT TO $C(X=x, Y=y)$
- THIS WOULD CONVINC V THAT THE OUTPUT Y IS CORRECT

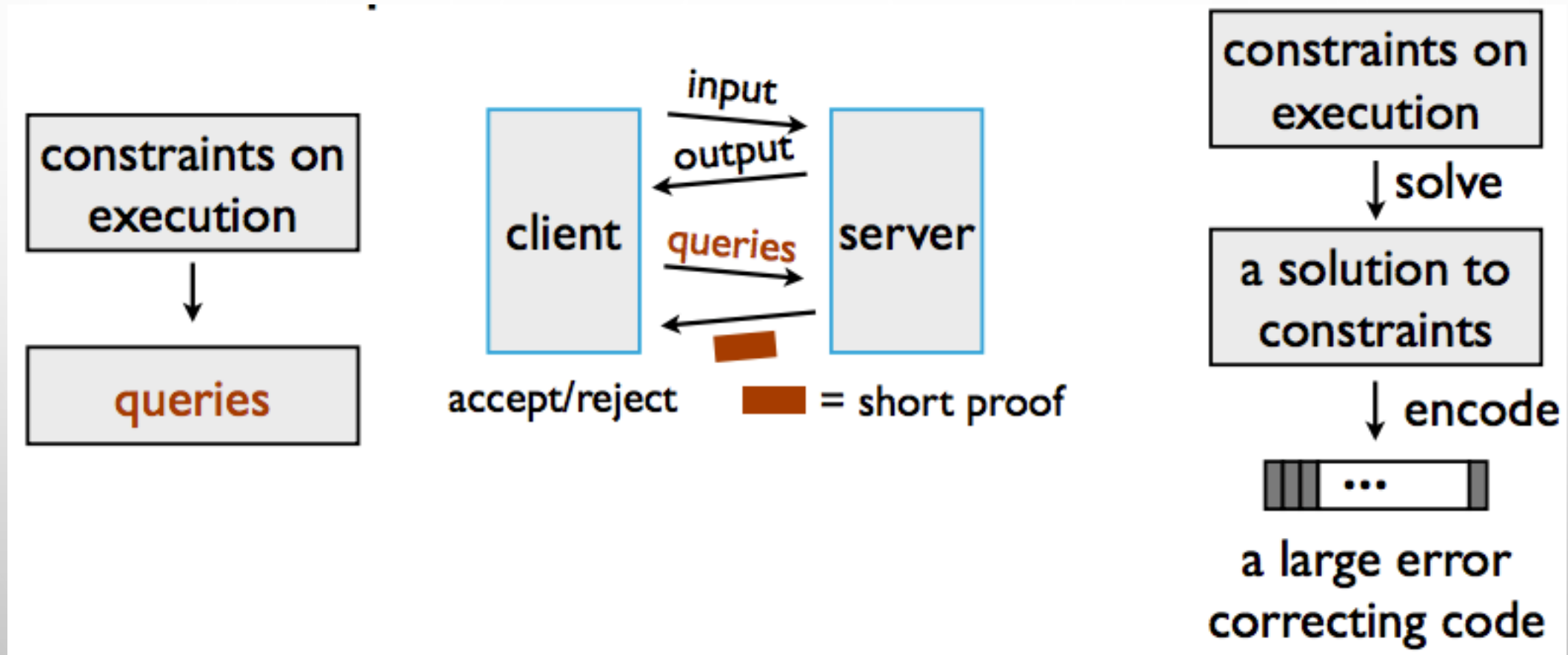
PANTRY

- PANTRY'S BASE: ZAATAR [EUROSYS13] AND PINOCCHIO [IEEE S&P13]



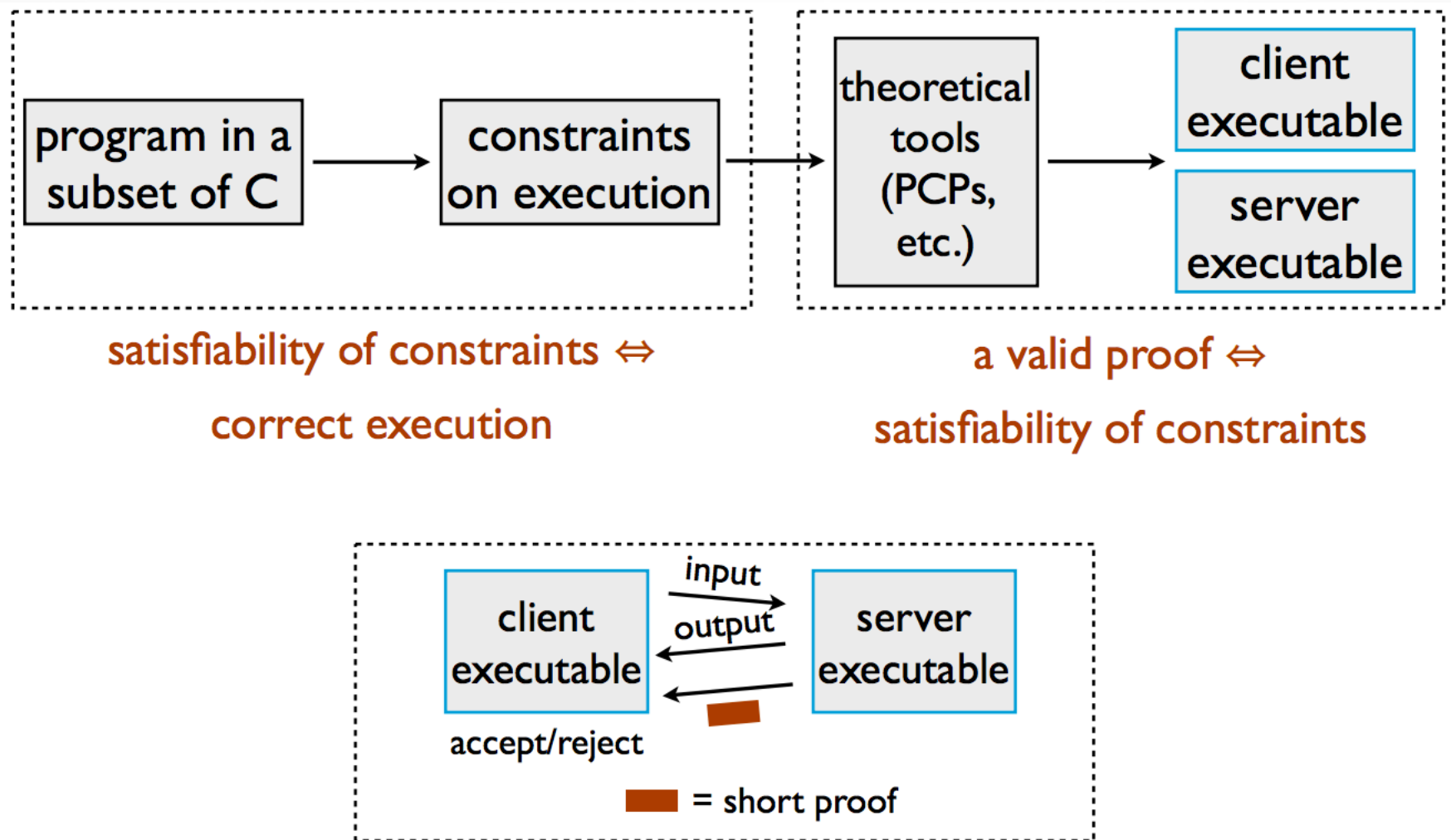
■ = short proof

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 - INTRO



PANTRY - INTRO

THERE IS A SIMPLE PROOF THAT A SATISFYING ASSIGNMENT EXISTS:

- *THE SATISFYING ASSIGNMENT ITSELF*

- HOWEVER, V COULD CHECK THIS PROOF ONLY BY EXAMINING ALL OF IT, WHICH WOULD BE AS MUCH WORK AS EXECUTING THE COMPUTATION

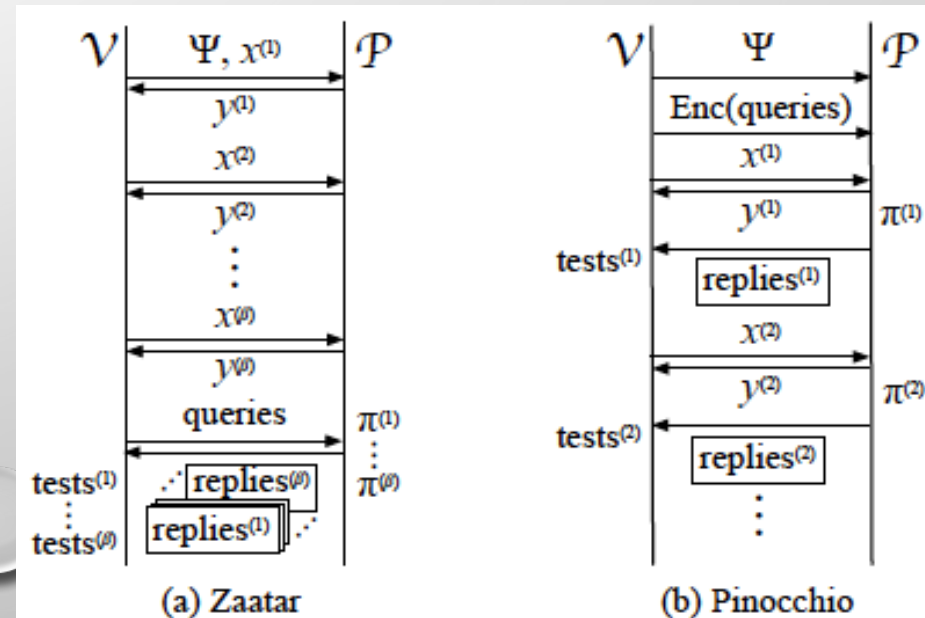
SOLUTION:

- APPLY THE THEORY OF PCP_s

- IMPLIES THAT A CLASSICAL PROOF—A SATISFYING ASSIGNMENT Z , IN THIS CASE—CAN BE ENCODED INTO A LONG STRING $()$ IN A WAY THAT ALLOWS V TO DETECT THE PROOF'S VALIDITY BY:
 - A. INSPECTING A SMALL NUMBER OF RANDOMLY-CHOSEN LOCATIONS IN
 - B. APPLYING EFFICIENT TESTS TO THE CONTENTS FOUND AT THOSE LOCATIONS

ZAATAR VS PINOCCHIO

- ZAATAR INSTANTIATES AN EFFICIENT ARGUMENT:
 - EXTRACTS FROM P A CRYPTOGRAPHIC COMMITMENT TO π , AND THEN V QUERIES P, MEANING THAT V ASKS P WHAT VALUES (p_i) CONTAINS AT PARTICULAR LOCATIONS.
 - V USES PCPS TO CHOOSE THE LOCATIONS AND TEST THE REPLIES, AND CRYPTOGRAPHY TO ENSURE THAT P'S REPLIES PASS V'S TESTS ONLY IF P'S REPLIES ARE CONSISTENT WITH A PROOF (π) THAT A SATISFYING ASSIGNMENT EXISTS.
- PINOCCHIO [65] AND KNOWN AS A NON-INTERACTIVE ARGUMENT:
 - V PREENCRYPTS QUERIES AND SENDS THEM TO P
 - P REPLIES TO THE QUERIES WITHOUT KNOWING WHICH LOCATIONS V IS QUERYING. THIS PROCESS (HIDING THE QUERIES, REPLYING TO THEM, TESTING THE ANSWERS) RELIES ON SOPHISTICATED CRYPTOGRAPHY LAYERED ATOP THE PCP MACHINERY



PANTRY - INTRO

- SOLVES THE STATELESS LIMITATIONS:

HOW CAN WE DESIGN CONSTRAINTS SUCH THAT THEIR
SATISFIABILITY IS TANTAMOUNT TO CORRECT STORAGE
INTERACTION?

A NAIVE APPROACH

...

$B = \text{read}(A)$



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

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

....

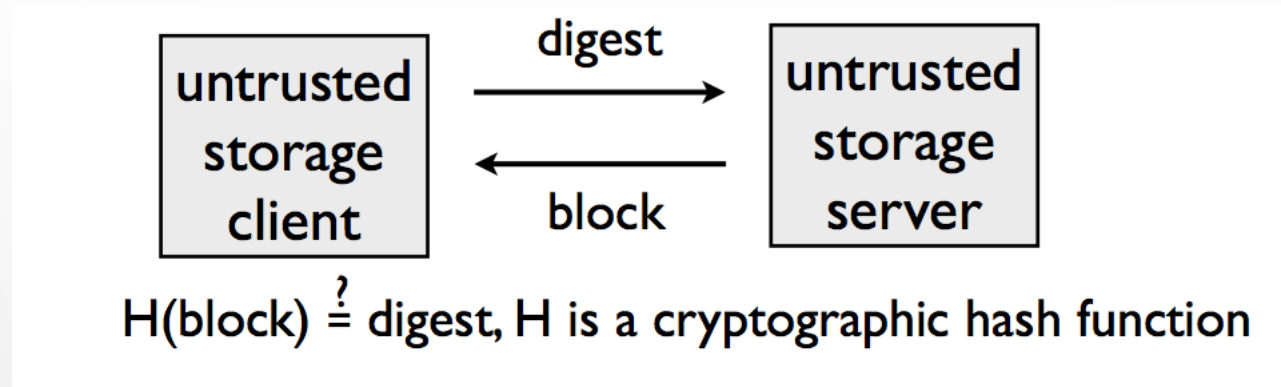
$$B = S_{\text{size}} - (A-\text{size}) \cdot C_{\text{size}}$$

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

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

- EXPENSIVE: REQUIRES ONE CONSTRAINT FOR EACH ADDRESS
- INCOMPLETE: PROVIDES ONLY VOLATILE STATE

CONSIDER AN UNTRUSTED BLOCK STORE:

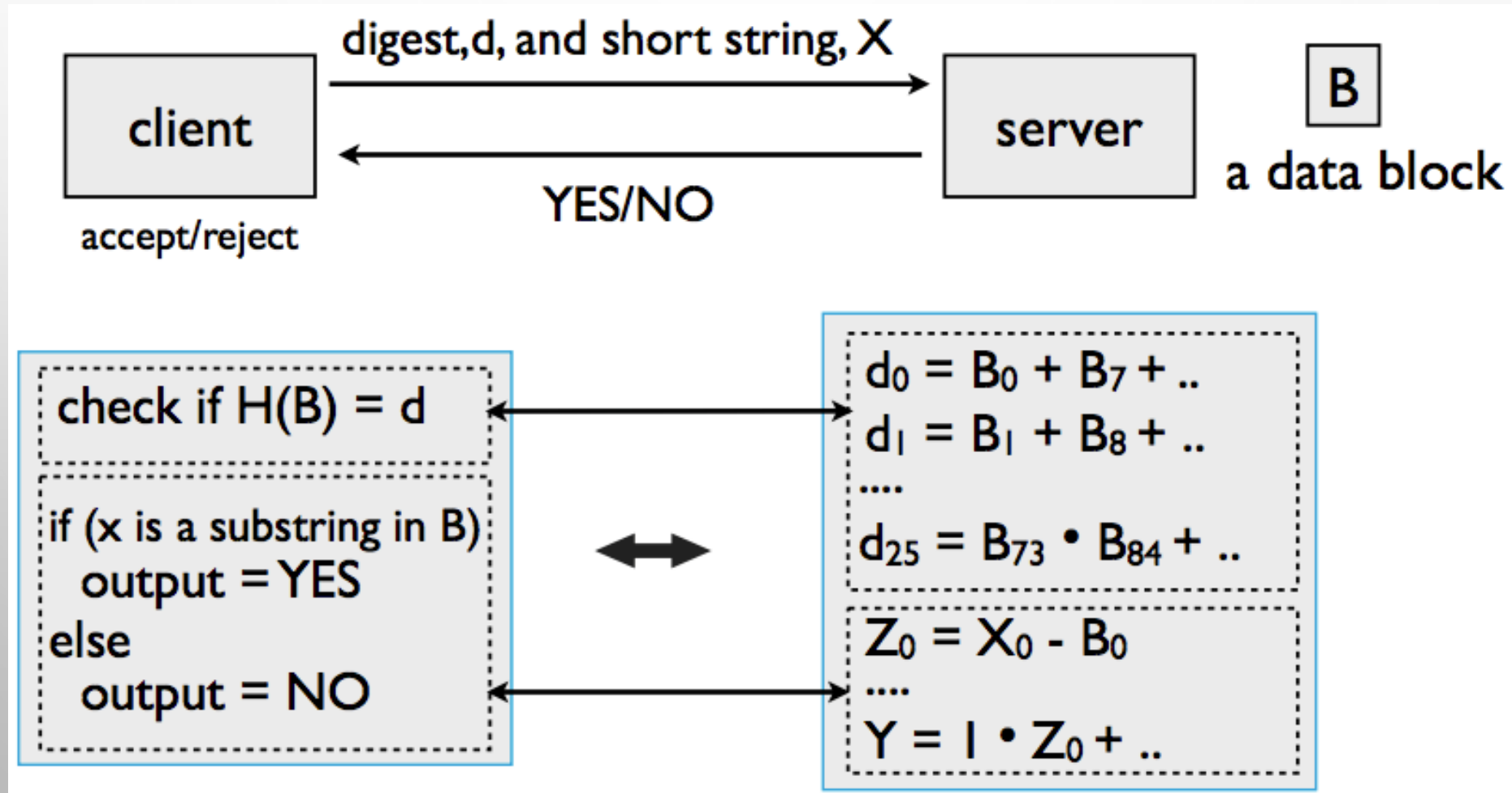


- TO RUN A COMPUTATION WITH REMOTE INPUTS, THE ABOVE CLIENT WILL NEED TO:
 1. FETCH BLOCKS FROM THE STORAGE SERVER
 2. CHECK THE INTEGRITY OF THE BLOCKS USING DIGESTS
 3. RUN THE COMPUTATION

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

PANTRY'S APPROACH TO STATE

- CONSIDER A SUBSTRING SEARCH WITH A REMOTE DATA BLOCK



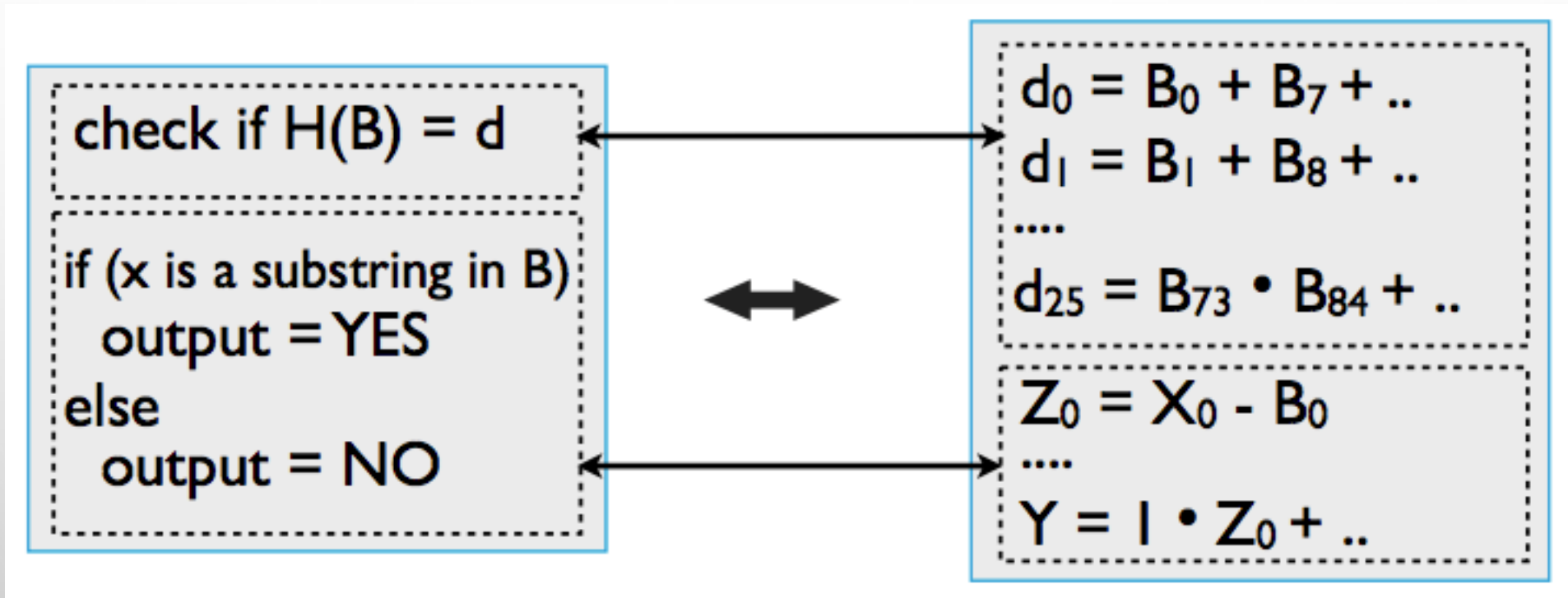
PANTRY'S APPROACH TO STATE

- VERIFIABLE BLOCKS

$$S: \textit{name} \rightarrow \textit{block} \cup \perp$$

- BLOCK = S(NAME), THEN H(BLOCK) = NAME
- THIS NAMING SCHEME ALLOWS CLIENTS TO USE UNTRUSTED STORAGE SERVERS
- GIVEN A NAME FOR DATA, THE CLIENT CAN CHECK THAT THE RETURNED BLOCK IS CORRECT, IN THE SENSE OF BEING CONSISTENT WITH ITS NAME.
- A CLIENT THAT CREATES NEW BLOCKS CAN COMPUTE THEIR NAMES AND USE THOSE NAMES AS REFERENCES LATER IN THE COMPUTATION.
- V CANNOT ACTUALLY CHECK THE CONTENTS OF THE BLOCKS THAT IT "RETRIEVES" OR IMPOSE THE CORRECT NAMES OF THE BLOCKS THAT IT "STORES"
- V USES THE VERIFICATION MACHINERY TO OUTSOURCE THE STORAGE CHECKS TO P; IN FACT, P ITSELF COULD BE USING AN UNTRUSTED BLOCK STORE!

PANTRY'S APPROACH TO STATE



- SATISFIABILITY OF THE ABOVE CONSTRAINTS \Leftrightarrow PASSING HASH CHECKS
- PASSING HASH CHECKS IS COMPUTATIONALLY INFEASIBLE WITHOUT THE RIGHT DATA BLOCKS

PANTRY'S C TO EXPOSE STATE

- **GETBLOCK(DIGEST / NAME):**
RETURNS A BLOCK SUCH THAT $H(\text{BLOCK}) = \text{DIGEST /NAME}$
- **PUTBLOCK(BLOCK):**
STORES "BLOCK" AT LOCATION $H(\text{BLOCK})$

GetBlock(name n):

$block \leftarrow$ read block with name n in block store S
assert $n == H(block)$
return $block$

PutBlock(block):

$n \leftarrow H(block)$
store $(n, block)$ in block store S
return n

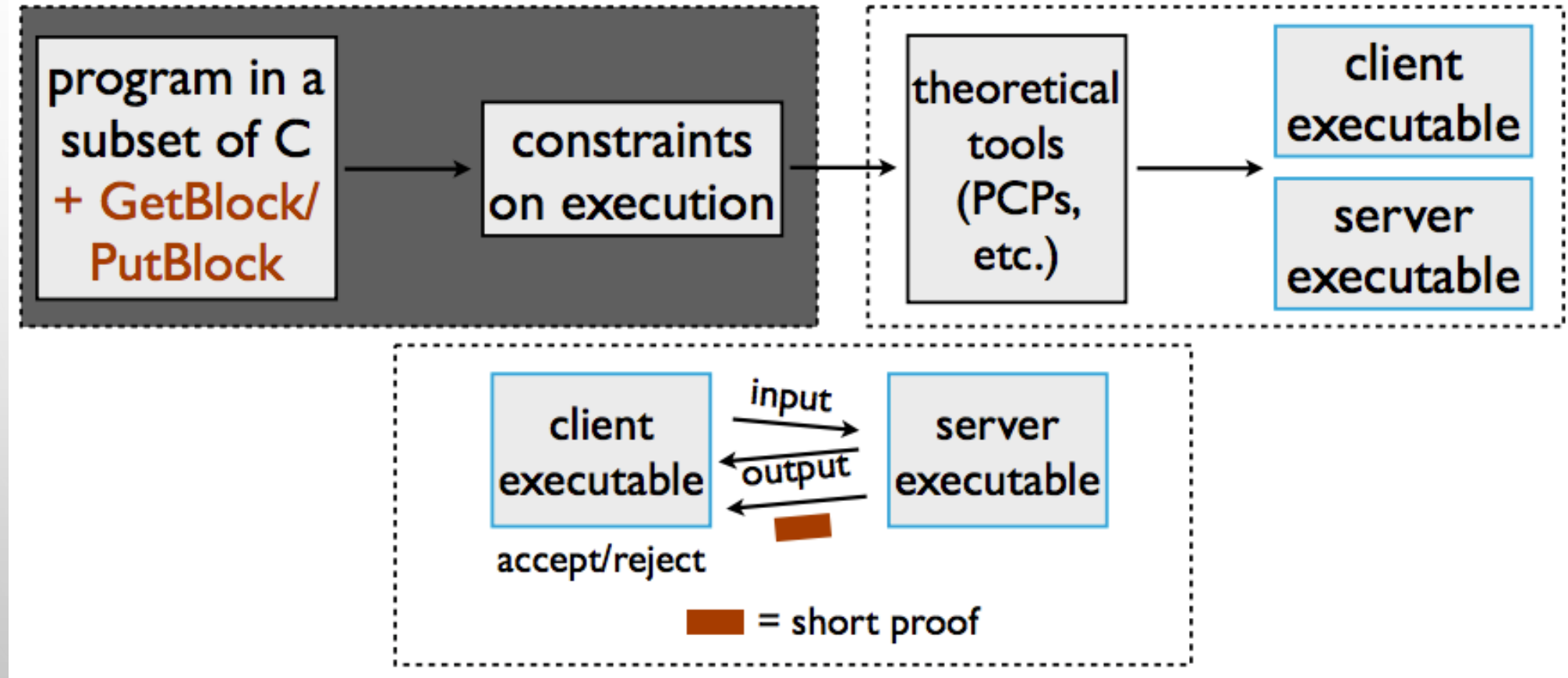
PANTRY'S C TO EXPOSE STATE

- CODE $b = \text{GETBLOCK}(n)$ COMPILES TO CONSTRAINTS $C_{H-1}(X=n, Y=b)$
 - X , REPRESENTS THE NAME;
 - Y , REPRESENTS THE BLOCK CONTENTS;
- CODE $N = \text{PUTBLOCK}(b)$ COMPILES TO THE CONSTRAINTS $C_H(X=b, Y=n)$
 - SAME CONSTRAINTS, EXCEPT THAT THE INPUTS AND OUTPUTS ARE SWITCHED.
- EXAMPLE:

```
add(int x1, name x2) {  
    block b = GetBlock(x2);  
    /* assume that b is a field element */  
    return b + x1;  
}
```

$$C = \{Y - B - X_1 = 0\} \cup C_{H-1}(X=X_2, Y=B),$$

PANTRY: AN EXTENSION TO ZAATAR & PINOCCHIO



- A VALID PROOF \Leftrightarrow "I KNOW A SATISFYING ASSIGNMENT TO CONSTRAINTS"
- SATISFIABILITY OF CONSTRAINTS \Leftrightarrow HASH CHECKS PASS
- HASH CHECKS PASS \Leftrightarrow CORRECT STORAGE INTERACTION

LIMITATIONS

- PANTRY DOES NOT FORMALLY ENFORCE DURABILITY:
 - MALICIOUS P COULD DISCARD BLOCKS INSIDE PUTBLOCK YET STILL EXHIBIT A SATISFYING ASSIGNMENT
- PANTRY (LIKE ITS PREDECESSORS) DOES NOT ENFORCE AVAILABILITY:
 - P COULD REFUSE TO ENGAGE, OR FAIL TO SUPPLY A SATISFYING ASSIGNMENT, EVEN IF IT KNOWS ONE.

PANTRY ENFORCES **INTEGRITY**, MEANING THAT PURPORTED MEMORY VALUES (THE BLOCKS THAT ARE USED IN THE COMPUTATION) ARE CONSISTENT WITH THEIR NAMES, OR ELSE THE COMPUTATION DOES NOT VERIFY.

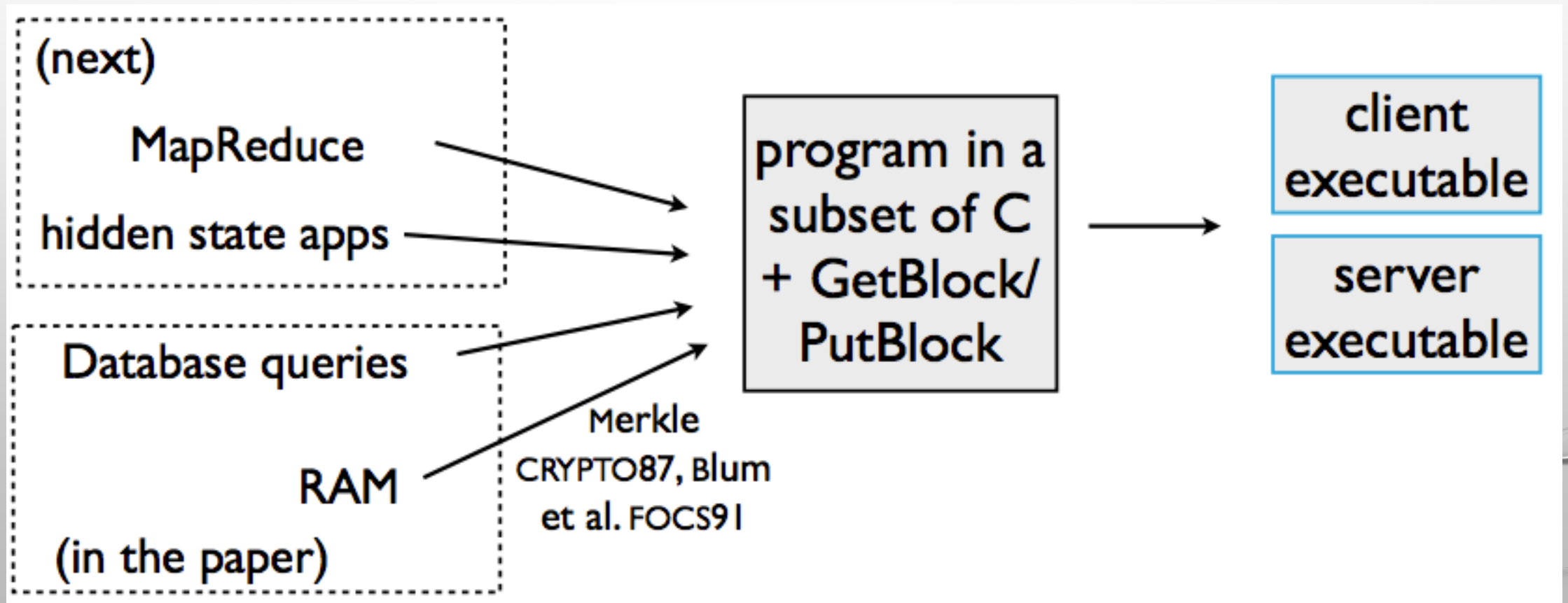
APPLICATION

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

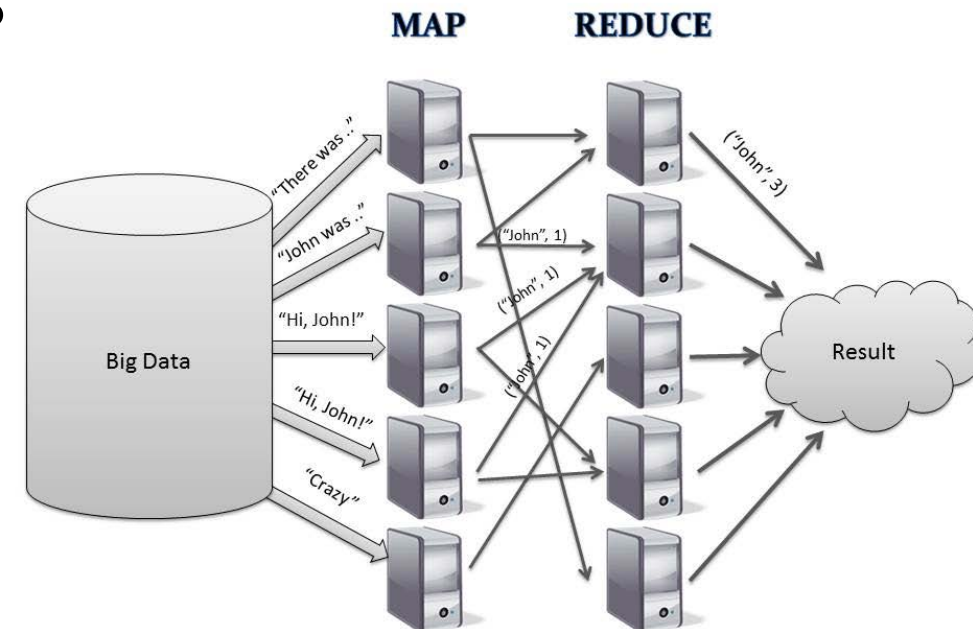
VERIFIABLE STATEFUL APPLICATIONS FROM C CODE WITH PANTRY

- THE APPLICATIONS



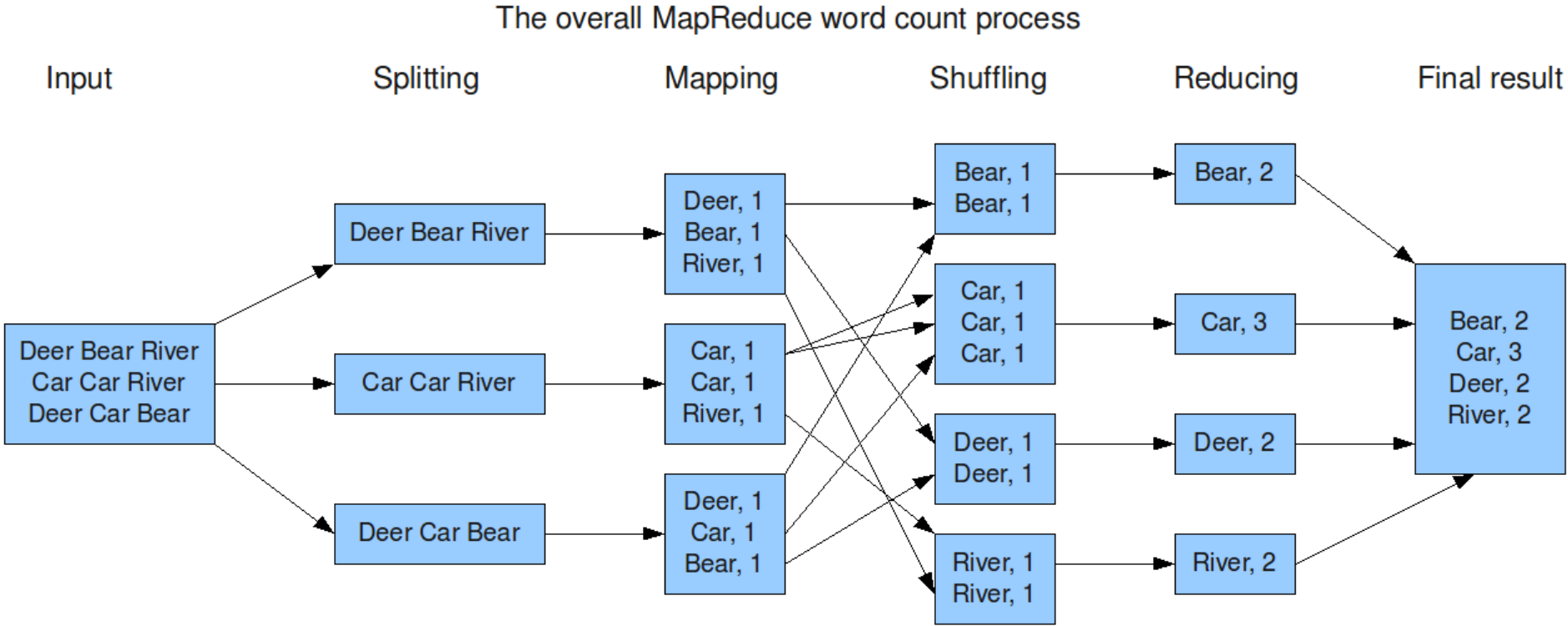
MAPREDUCE

- **MAPREDUCE** IS A [PROGRAMMING MODEL](#) AND AN ASSOCIATED IMPLEMENTATION FOR PROCESSING AND GENERATING [BIG DATA](#) SETS WITH A [PARALLEL, DISTRIBUTED](#) ALGORITHM ON A [CLUSTER](#)
- A MAPREDUCE JOB CONSISTS OF MAP AND REDUCE FUNCTIONS, AND INPUT DATA STRUCTURED AS A LIST OF KEY-VALUE PAIRS



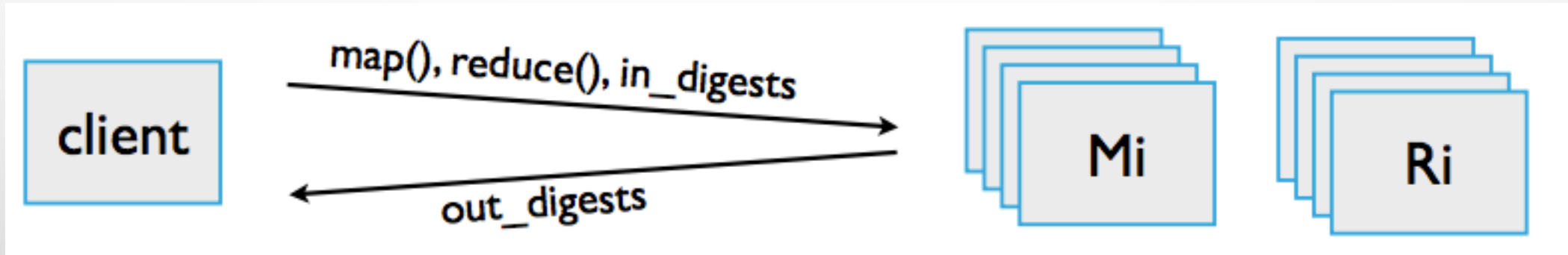
MAPREDUCE

- EXAMPLE:
 - COUNTING WORDS
 - TEMPRETURE AVERAGE
 - TOP N



PANTRY APPLICATION: MAPREDUCE

- 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)  
}
```


PANTRY APPLICATION: MAPREDUCE

- When the mappers execute, each instance $j \in \{1, \dots, M\}$ gets as its input, $x^{(j)}$, the digest of some data. The output of an instance, $\text{map_out}(j)$, is a vector of R digests, one for each reducer that this mapper is “Feeding”;
- the framework receives this output and forwards it to V . Verification Convinces V that each mapper j knows a satisfying assignment to $C_{\text{mapper}}(x=x^{(j)}, y=\text{map_out}(j))$, which establishes for V that the mapper worked over the correct data, applied map correctly, partitioned the transformed data over the reducers correctly, and—in outputting $\text{map_out}(j)$ —named the transformed data correctly. Note that $\{\text{map_out}^{(j)}\}_{j=\{1, \dots, M\}}$ are the M R intermediate digests mentioned above.
- The framework then supplies the inputs to the second phase, by shuffling the digests $\{\text{map_out}(j)\}_{j=\{1, \dots, m\}}$ and regrouping them as $\text{reduce in}(i)_{g_j=\{1, \dots, rg\}}$, where each $\text{reduce in}(i)$ is a vector of M digests, one for each mapper. (V does this regrouping too, in order to know the reducers’ inputs.)

VERIFIABLE RAM

- PANTRY'S VERIFIABLE RAM ABSTRACTION ENABLES RANDOM ACCESS TO CONTIGUOUSLY-ADDRESSABLE, FIXED-SIZE MEMORY CELLS
 - $VALUE = LOAD(ADDRESS, DIGEST);$
 - $NEW\ DIGEST = STORE(ADDRESS, VALUE, DIGEST);$
- THE HIGH-LEVEL IDEA IS THAT THE DIGEST COMMITS TO THE FULL STATE OF MEMORY
- A LOAD GUARANTEES THAT THE CLAIM "ADDRESS CONTAINS VALUE" IS CONSISTENT WITH DIGEST.
- FOR STORE, THE GUARANTEE IS THAT NEW DIGEST CAPTURES THE SAME MEMORY STATE THAT DIGEST DOES WITH THE EXCEPTION THAT ADDRESS NOW HOLDS VALUE

VERIFIABLE RAM

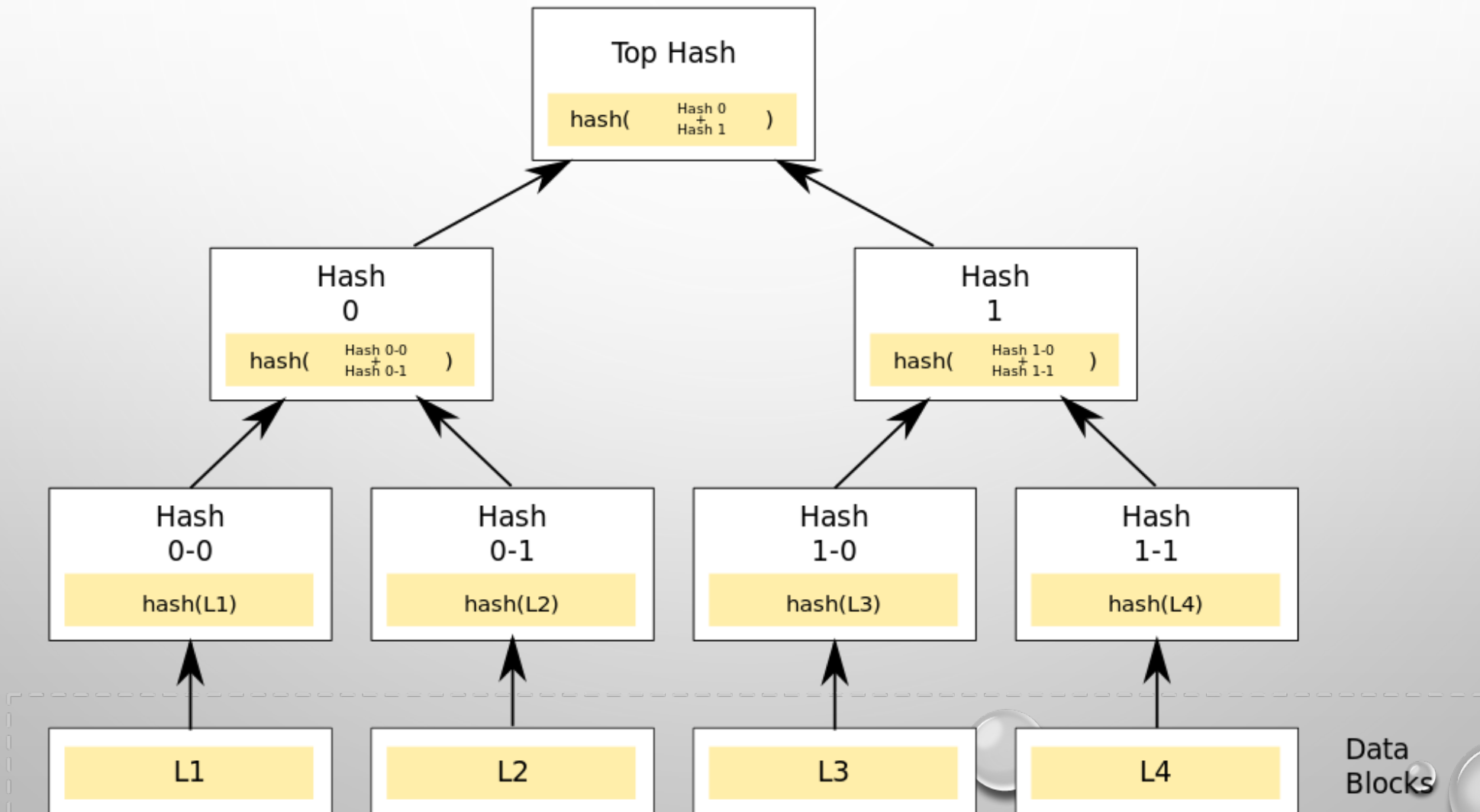
- **MERKLE TREES:**

- EVERY NODE IS NAMED BY A COLLISION-RESISTANT HASH (DENOTED H) OF ITS CONTENTS
- AN INTERIOR NODE'S CONTENTS ARE THE NAMES (OR HASHES) OF THE NODE'S LEFT AND RIGHT CHILDREN.
- EACH LEAF NODE CORRESPONDS TO A MEMORY ADDRESS, AND CONTAINS THE VALUE CURRENTLY HELD AT THE MEMORY ADDRESS.
- THEN, THE DIGEST D IS THE HASH OF THE ROOT NODE'S CONTENTS.

FOR B TO SUCCEED IN A SPURIOUS CLAIM, IT WOULD HAVE TO IDENTIFY A COLLISION IN H .

- **DOES THE LEVEL OF THE COLLISION MATTER?**

MERKLE TREES



VERIFIABLE RAM

Load(address a , digest d):

```
 $\ell \leftarrow \lceil \log N \rceil$   
 $h \leftarrow d$   
for  $i = 1$  to  $\ell$ :  
   $node \leftarrow \text{GetBlock}(h)$   
   $x \leftarrow$   $i$ th bit of  $a$   
  if  $x = 0$ :  
     $h \leftarrow node.left$   
  else:  
     $h \leftarrow node.right$   
 $node \leftarrow \text{GetBlock}(h)$   
return  $node.value$ 
```

Store(address a , value v , digest d):

```
 $path \leftarrow \text{LoadPath}(a, d)$   
 $\ell \leftarrow \lceil \log N \rceil$   
 $node \leftarrow path[\ell]$   
 $node.value \leftarrow v$   
 $d' \leftarrow \text{PutBlock}(node)$   
for  $i = \ell$  to 1:  
   $node \leftarrow path[i - 1]$   
   $x \leftarrow$   $i$ th bit of  $a$   
  if  $x = 0$ :  
     $node.left \leftarrow d'$   
  else:  
     $node.right \leftarrow d'$   
 $d' \leftarrow \text{PutBlock}(node)$   
return  $d'$ 
```

SEARCH TREE

- SUPPORT EFFICIENT RANGE SEARCHES OVER ANY KEYS:

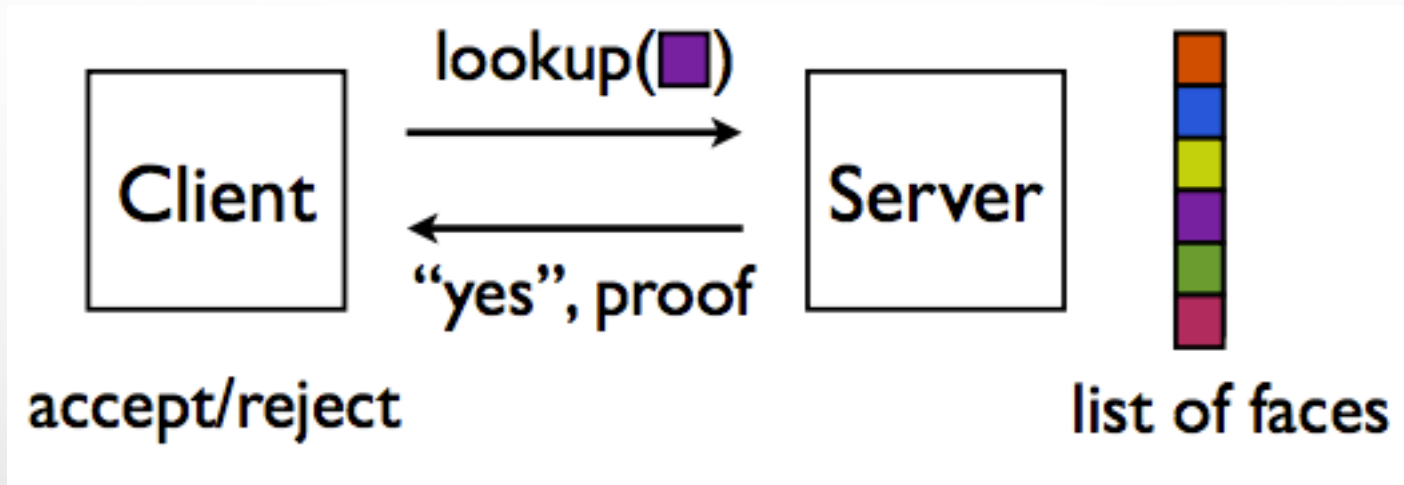
```
values = FindEquals(key, digest)|
values = FindRange(key_start, key_end, digest)
new_digest = Insert(key, value, digest)
new_digest = Remove(key, digest)
```

- BUILD A SEARCHABLE MERKLE TREE:
 - EACH NODE IN THE TREE CONTAINS A KEY, ONE OR MORE VALUES CORRESPONDING TO THAT KEY, AND POINTERS TO (THAT IS, HASHES OF) ITS CHILDREN.
 - THE NODES ARE IN SORTED ORDER, AND THE TREE IS A BALANCED (AVL) TREE, SO OPERATIONS TAKE TIME THAT IS LOGARITHMIC IN THE NUMBER OF KEYS STORED.
- A SEARCH OPERATION (FINDEQUALS, FINDRANGE) DESCENDS THE TREE, VIA A SERIES OF GETBLOCK CALLS.
- AN UPDATE OPERATION (INSERT, REMOVE):
 1. DESCENDS THE TREE TO IDENTIFY THE NODE WHERE THE OPERATION WILL BE PERFORMED;
 2. THEN MODIFIES THAT NODE (VIA PUTBLOCK, THEREBY GIVING IT A NEW NAME
 3. UPDATES THE NODES ALONG THE PATH TO THE ROOT (AGAIN VIA PUTBLOCK), RESULTING IN A NEW DIGEST.

VERIFIABLE DATABASE QUERIES

- V SPECIFIES QUERIES IN A PRIMITIVE SQL-LIKE LANGUAGE
- EACH ROW OF EVERY TABLE IS STORED AS A VERIFIABLE BLOCK, ACCESSED THROUGH GETBLOCK/PUTBLOCK
- THESE BLOCKS ARE POINTED TO BY ONE OR MORE INDEXES
- THERE IS A SEPARATE INDEX FOR EACH COLUMN THAT THE AUTHOR OF THE COMPUTATION WANTS TO BE **SEARCHABLE**
- INDEXES ARE IMPLEMENTED AS VERIFIABLE SEARCH TREES
- DATABASE QUERIES ARE CONVERTED INTO A SERIES OF CALLS TO THE TREES' FINDEQUALS, FINDRANGE, INSERT, AND REMOVE OPERATIONS.

HIDDEN STATE APPLICATIONS



- USING PANTRY, THE CLIENT IS ASSURED THAT THE RESPONSE IS CORRECT, BUT NO INFORMATION ABOUT THE PROVER'S DATABASE LEAKS (BEYOND WHAT THE OUTPUT IMPLIES).
- 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)

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

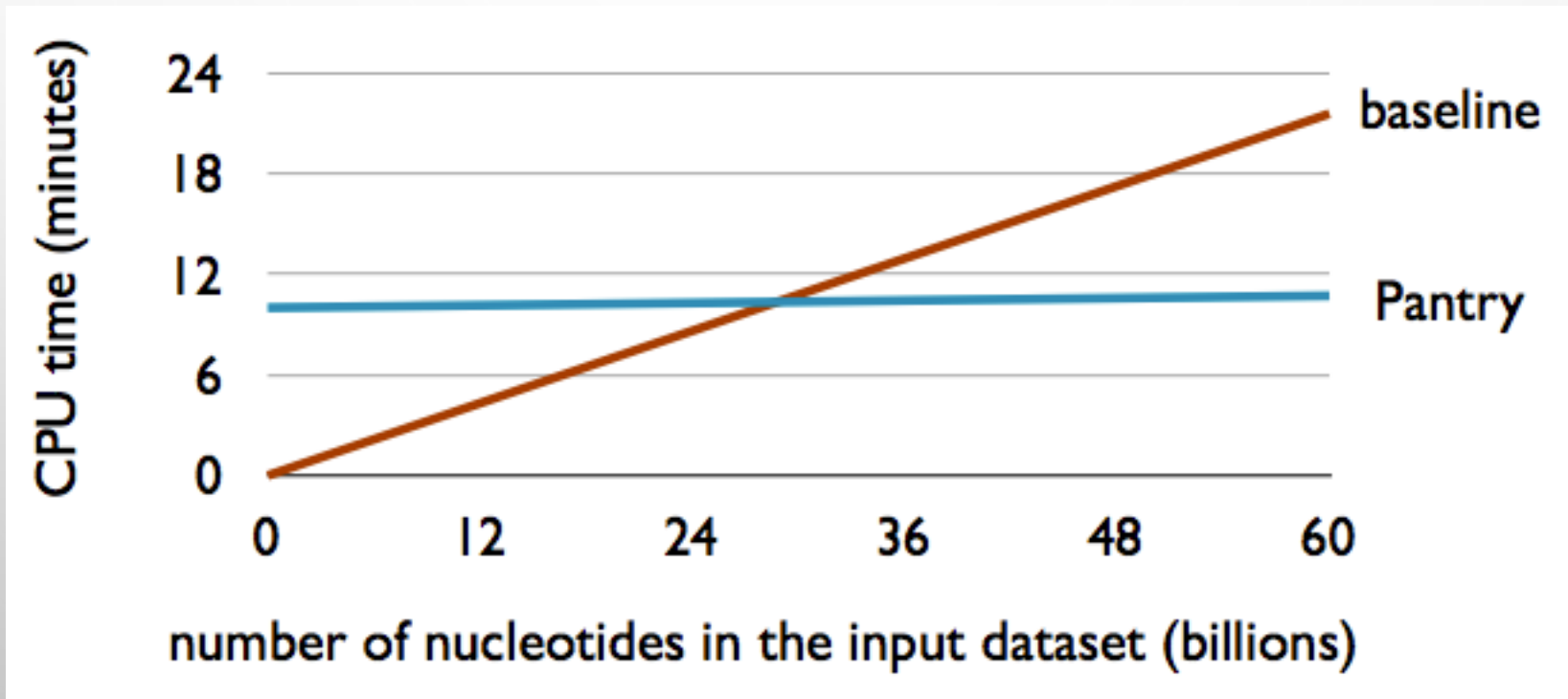
C++, JAVA, GO, AND PYTHON CODE; HTTP/OPEN MPI TO DISTRIBUTE SERVER'S WORK

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?

EVALUATION

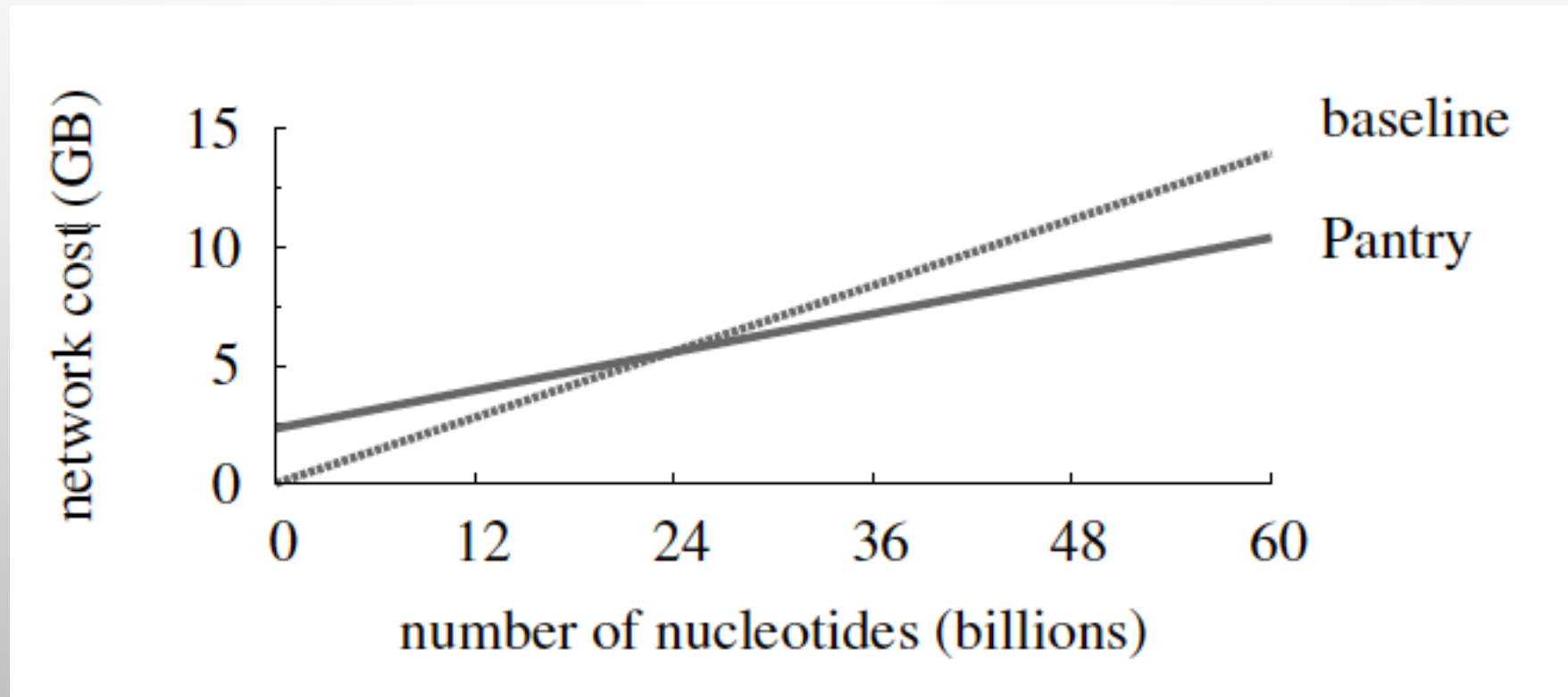
- MAPREDUCE JOB: NUCLEOTIDE SUBSTRING SEARCH IN WHICH A MAPPER GETS 600K NUCLEOTIDES AND OUTPUTS MATCHING LOCATIONS



- GRAPH IS AN EXTRAPOLATION (SLOPES AND Y INTERCEPTS DETERMINED WITH EXPERIMENTS THAT USE UP TO 250 MACHINES AND UP TO 1.2 BILLION NUCLEOTIDES)

EVALUATION

- MAPREDUCE JOB: NUCLEOTIDE SUBSTRING SEARCH IN WHICH A MAPPER GETS 600K NUCLEOTIDES AND OUTPUTS MATCHING LOCATIONS



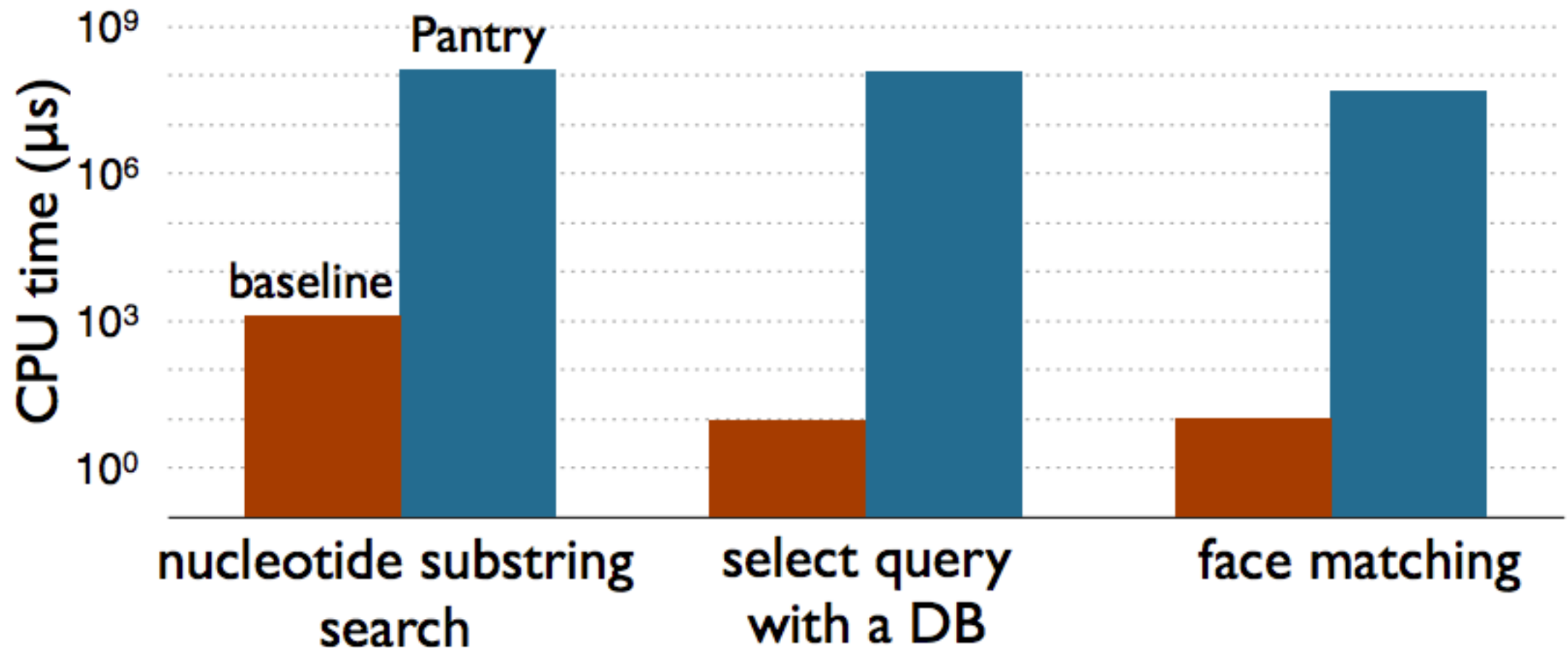
EVALUATION

COST OF SUPPORTING HIDDEN STATE APPLICATIONS

- SERVER HOLDS 128 FACE FINGERPRINTS (HIDDEN STATE: 1.5 KB) PROOF SIZE: 288 BYTES
CLIENT'S CPU TIME: 7 MS GOOD NEWS: NETWORK (SETUP), SERVER'S STORAGE (ONGOING):
170 MB SERVER'S CPU TIME: 7.8 MIN

EVALUATION

- PANTRY'S SERVER'S COST IS MANY ORDERS OF MAGNITUDE SLOWER THAN SIMPLY EXECUTING THE COMPUTATION



EVALUATION

WHY PANTRY?



Q/A



