Proof-Carrying Data:
secure computation on untrusted execution platforms

Eran Tromer

Joint work with
Alessandro Chiesa
Eli Ben-Sasson
Daniel Genkin

Motivation

Motivation

	INTEGRITY	CONFIDENTIALITY
SOFTWARE	• Bugs • Trojans	

- Software engineering (review, tests)
- Formal verification, static analysis
- Language type safety
- Dynamic analysis
- Reference monitors

Motivation

	INTEGRITY	CONFIDENTIALITY
SOFTWARE	• Bugs • Trojans	
NETWORK	• Lack of trust	

- Software engineering (review, tests)
- Formal verification, static analysis
- Language type safety
- Dynamic analysis
- Reference monitors

?

Motivation

	INTEGRITY	CONFIDENTIALITY
SOFTWARE	• Bugs • Trojans	
NETWORK	• Lack of trust	
ENVIRONMENT	• Tampering	• Physical side-channels (EM, power, acoustic)

Motivation

	INTEGRITY	CONFIDENTIALITY
SOFTWARE	• Bugs • Trojans	
NETWORK	• Lack of trust	
ENVIRONMENT	• Tampering	• Physical side-channels

Motivation

	INTEGRITY	CONFIDENTIALITY
SOFTWARE	• Bugs • Trojans	
NETWORK	• Lack of trust	
ENVIRONMENT	• Tampering	• Physical side-channels
PLATFORM	• Cosmic rays • Hardware bugs • Hardware trojans • IT supply chain	

Information technology supply chain: headlines

The New York Times (May 9, 2008)

"F.B.I. Says the Military Had Bogus Computer Gear"

ars technica (October 6, 2008)

"Chinese counterfeit chips causing military hardware crashes"

The New York Times (May 6, 2010)

"A Saudi man was sentenced [...] to four years in prison for selling counterfeit computer parts to the Marine Corps for use in Iraq and Afghanistan."

Assurance? Validation? Certification?

Motivation

	INTEGRITY	CONFIDENTIALITY
SOFTWARE	• Bugs • Trojans	
NETWORK	• Lack of trust	
ENVIRONMENT	• Tampering	• Physical side-channels
PLATFORM	• Cosmic rays • Hardware bugs • Hardware trojans • IT supply chain	

Motivation

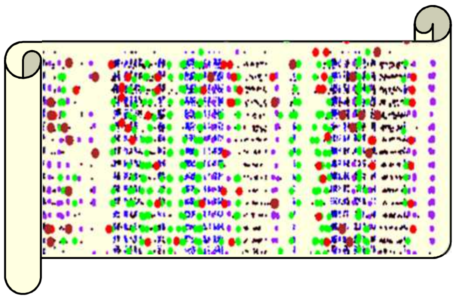
	INTEGRITY	CONFIDENTIALITY
SOFTWARE	• Bugs • Trojans	
NETWORK	• Lack of trust	
ENVIRONMENT	• Tampering	• Physical side-channels
PLATFORM	• Cosmic rays • Hardware bugs • Hardware trojans • IT supply chain	→ Fault analysis • Architectural side-channels (e.g., cache attacks)

Information Leakage in Third-Party Compute Clouds

[Ristenpart Tromer Shacham Savage ‘09]

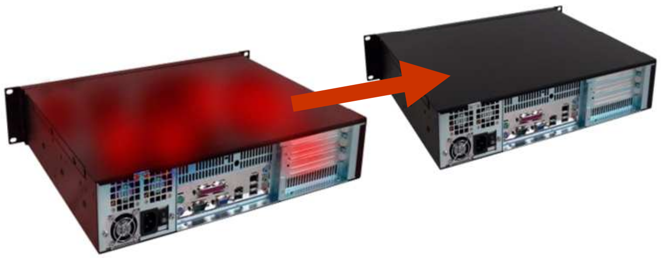
Demonstrated, using Amazon EC2 as a study case:

- Cloud cartography
  Mapping the structure of the “cloud” and locating a target on the map.



- Placement vulnerabilities
  An attacker can place his VM on the same physical machine as a target VM (40% success for a few dollars).

- Cross-VM exfiltration
  Once VMs are co-resident, information can be exfiltrated across VM boundary:
  - Covert channels
  - Load traffic analysis
  - Keystrokes



Motivation

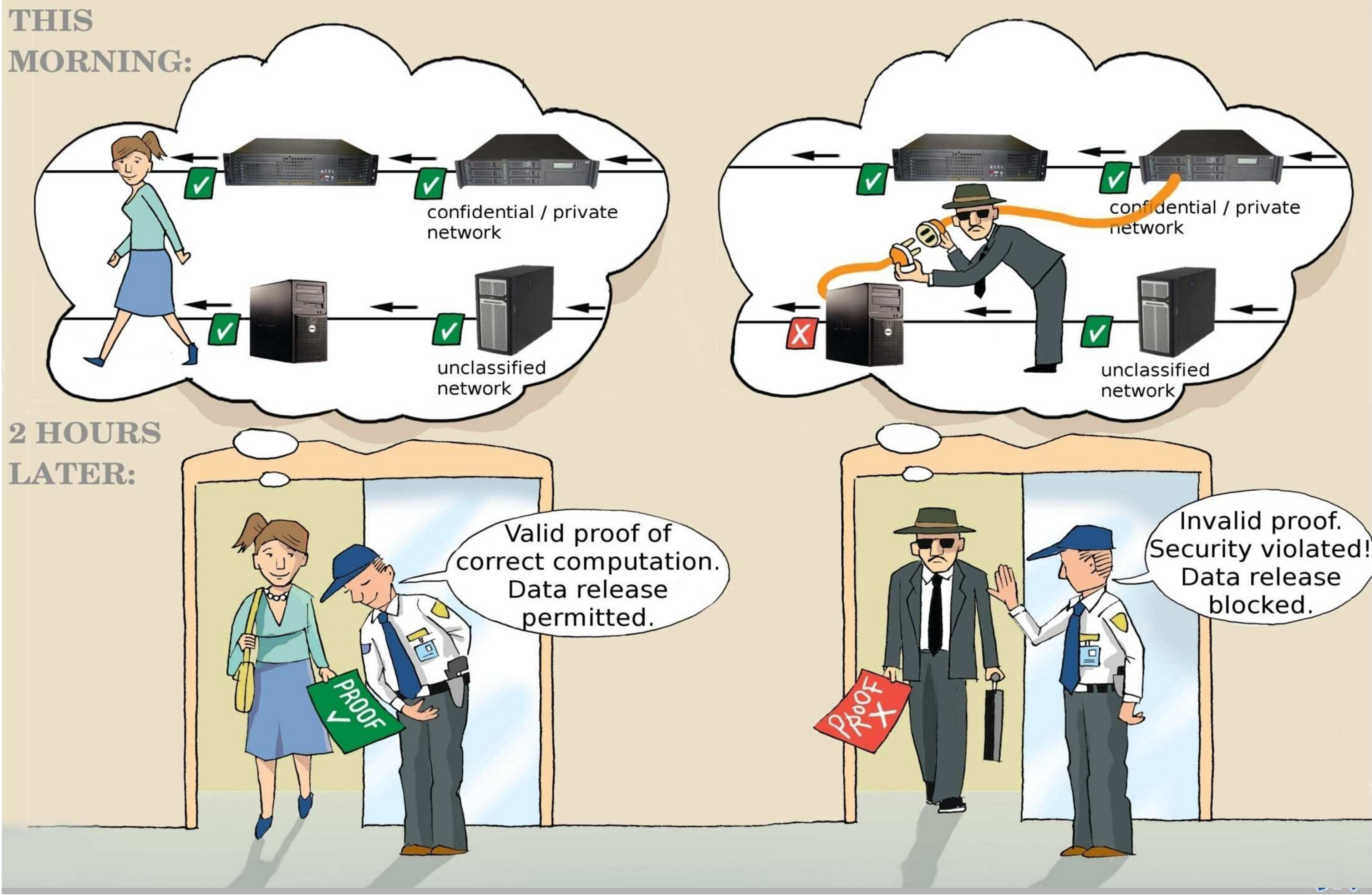
	CORRECTNESS	SECRECY
SOFTWARE	• Bugs • Trojans	
NETWORK	• Lack of trust	
ENVIRONMENT	• Tampering	• Physical side-channels
PLATFORM	• Cosmic rays • Hardware bugs • Hardware trojans • IT supply chain	• Fault analysis • Architectural side-channels (e.g.,cache attacks)

High-level goal

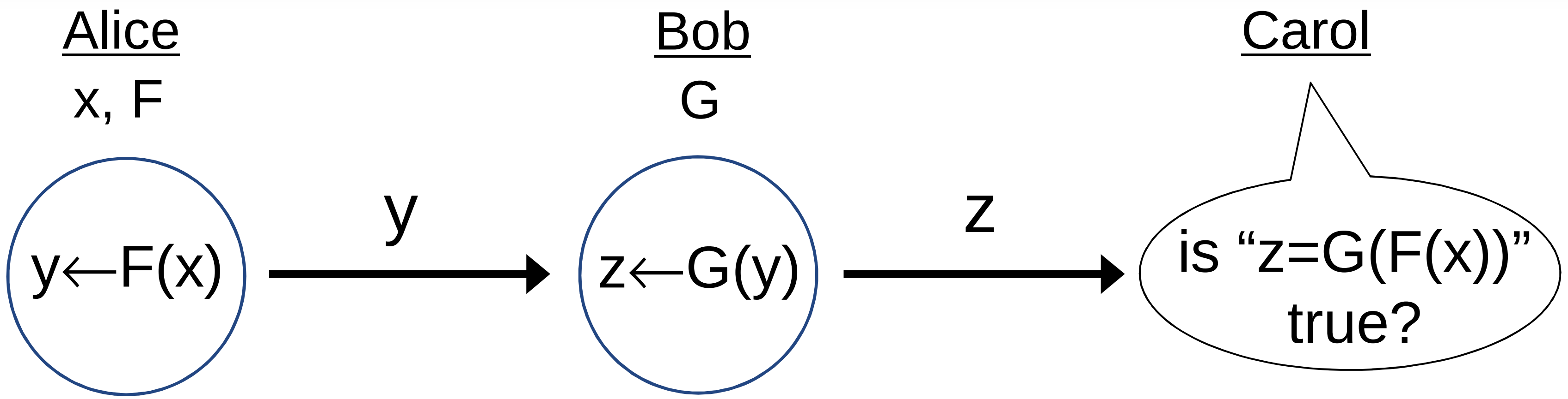
Ensure properties of a distributed computation when parties are mutually untrusting, faulty, leaky & malicious.

Proof-Carrying Data
overview

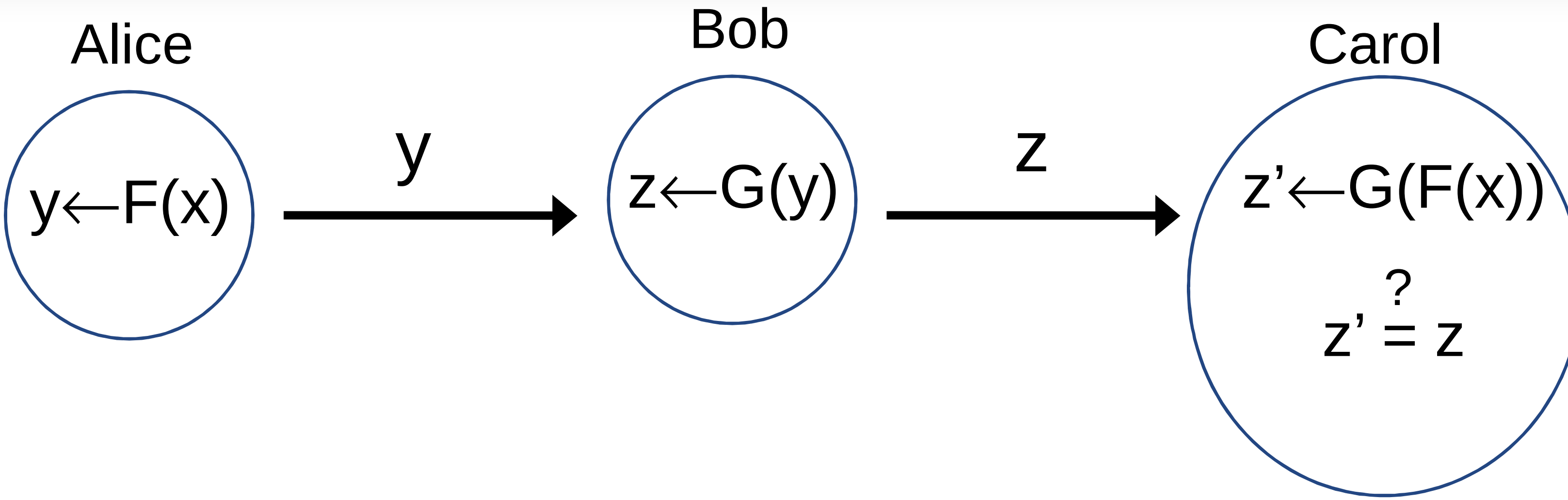
Proof-Carrying Data: an example



Toy example (3-party correctness)



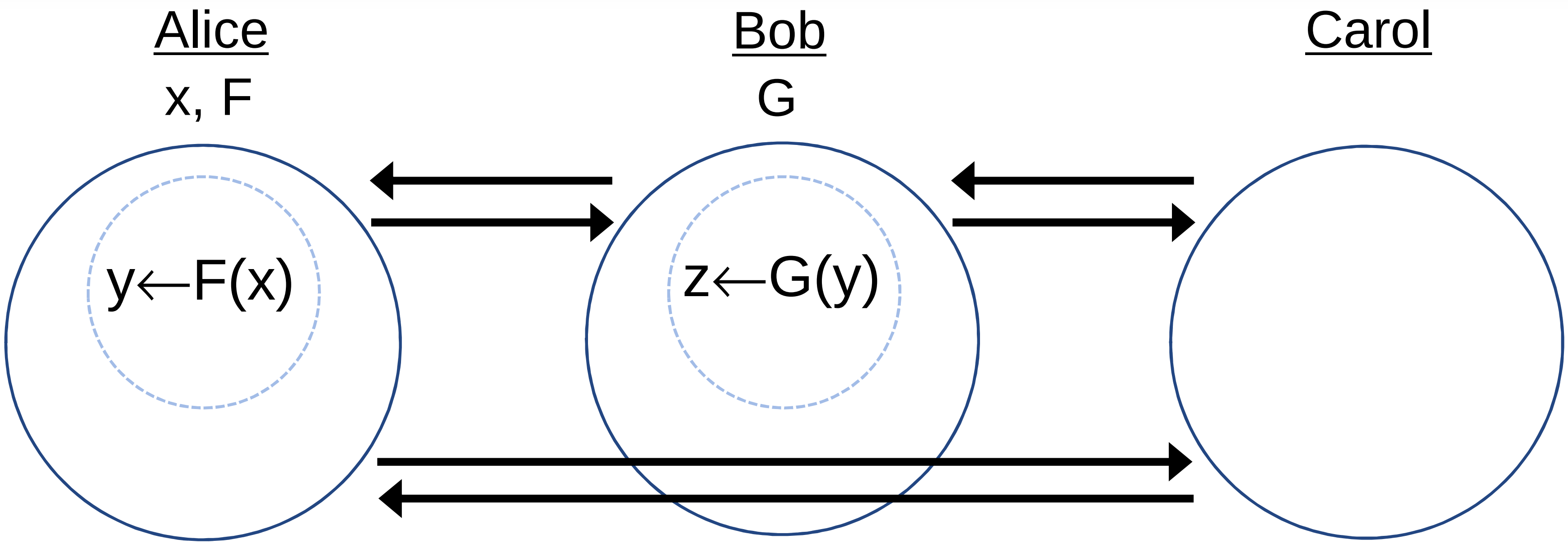
Toy example: trivial solution



Carol can recompute everything, but:

- Uselessly expensive
- Requires Carol to fully know x,F,G
  - We will want to represent these via short hashes/signatures

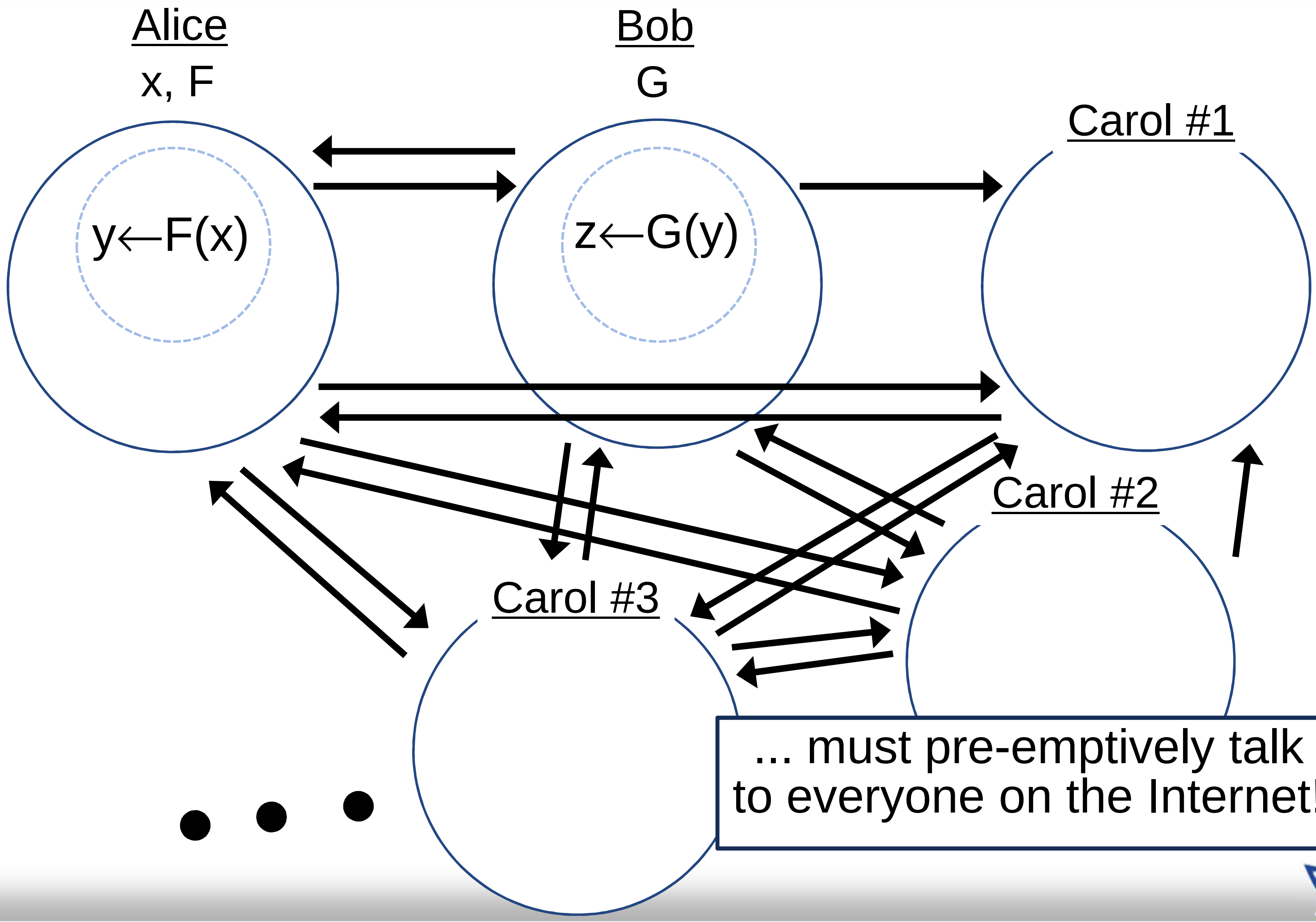
Toy example: secure multiparty computation [GMW87][BGW88][CCD88]



But:

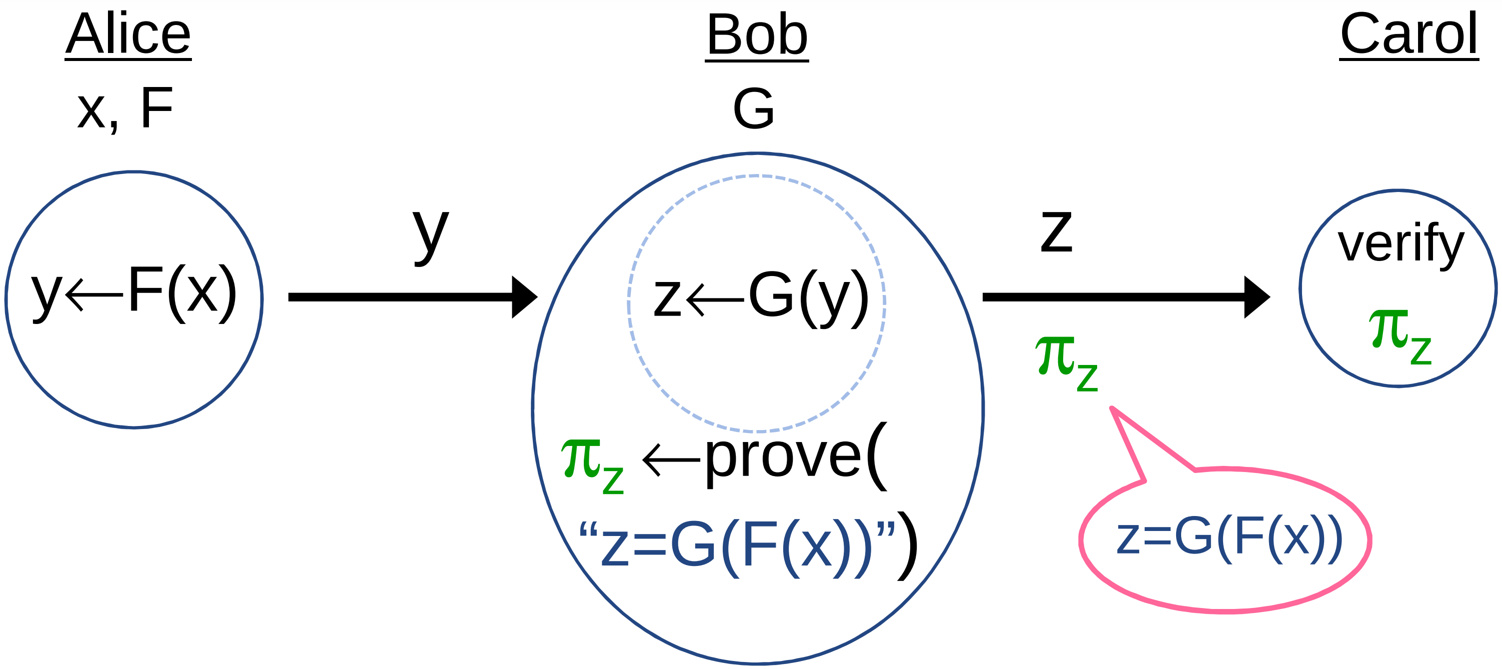
- computational blowup is polynomial in the whole computation, and not in the local computation
- computation (F and G) must be chosen in advance
- does not preserve the communication graph: parties must be fixed in advance, otherwise…

Toy example:
secure multiparty computation [GMW87][BGW88][CCD88]



... must pre-emptively talk to everyone on the Internet!

Toy example:
computationally-sound (CS) proofs [Micali 94]

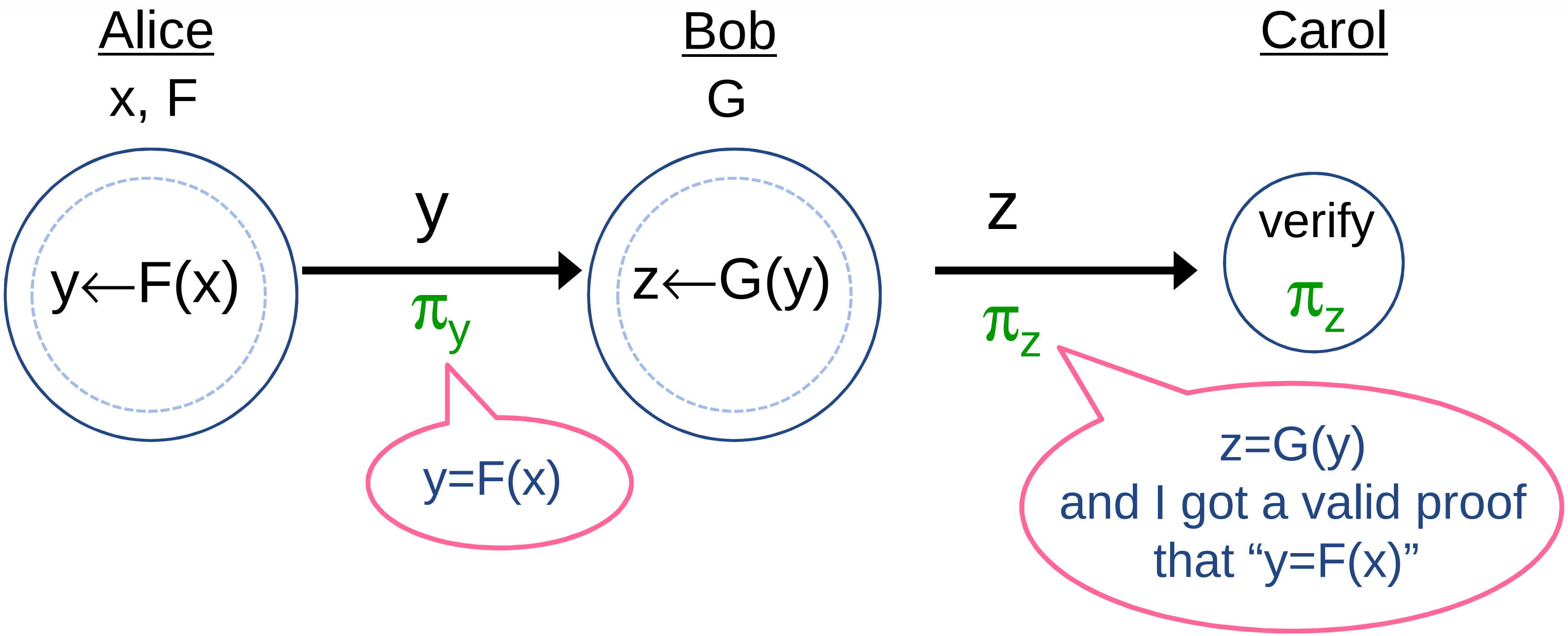


Bob can generate a proof string that is:

- Tiny (polylogarithmic in his own computation)
- Efficiently verifiable by Carol

However, now Bob recomputes everything...

Toy example: Proof-Carrying Data [Chiesa Tromer 09] following Incrementally-Verifiable Computation [Valiant 08]



Each party prepares a proof string for the next one.

Each proof is:

- Tiny (polylogarithmic in party's own computation).
- Efficiently verifiable by the next party.

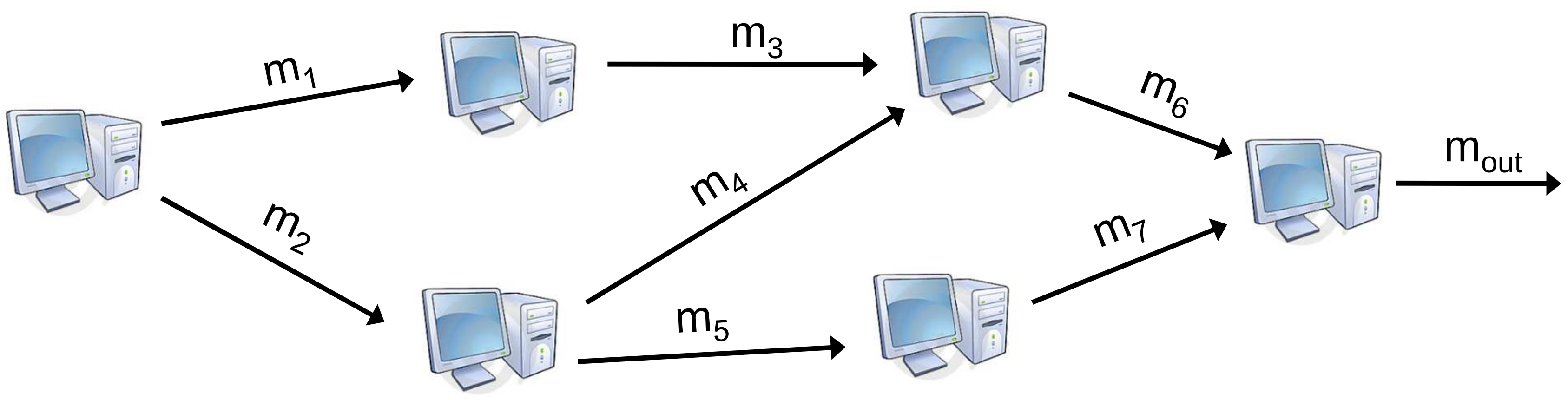
Generalizing:

The Proof-Carrying Data framework

Generalizing: distributed computations

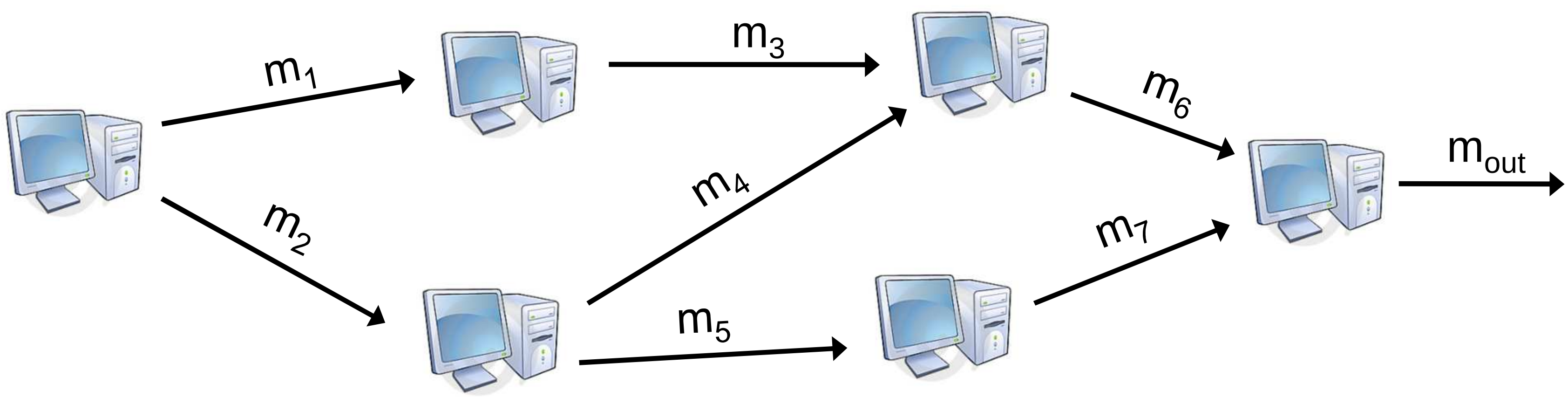
Distributed computation:

Parties exchange messages and perform computation.



Generalizing: arbitrary interactions

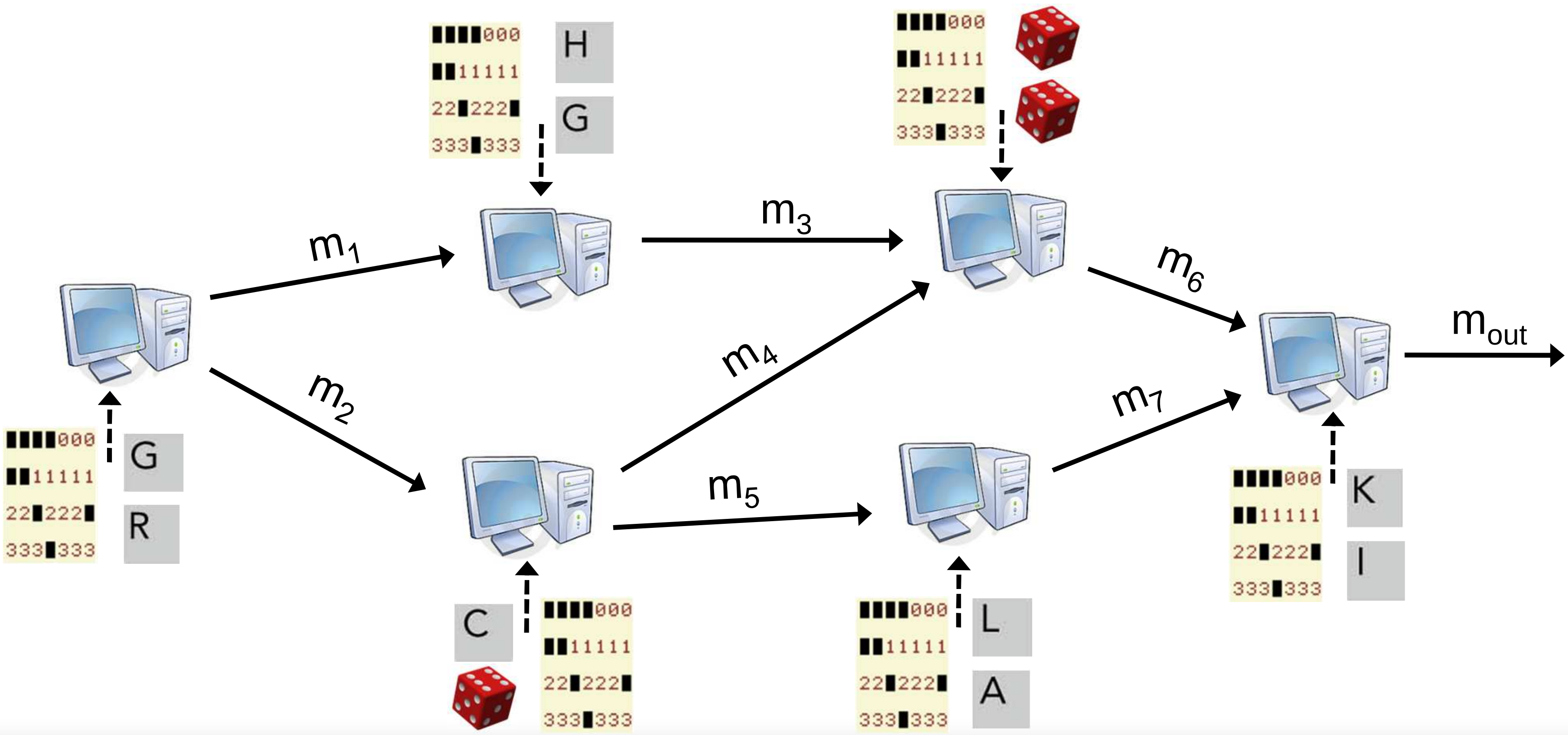
- Arbitrary interactions
  - communication graph over time is any direct acyclic graph



Generalizing: arbitrary interactions

- Computation and graph are determined on the fly
  - by each party's local inputs:

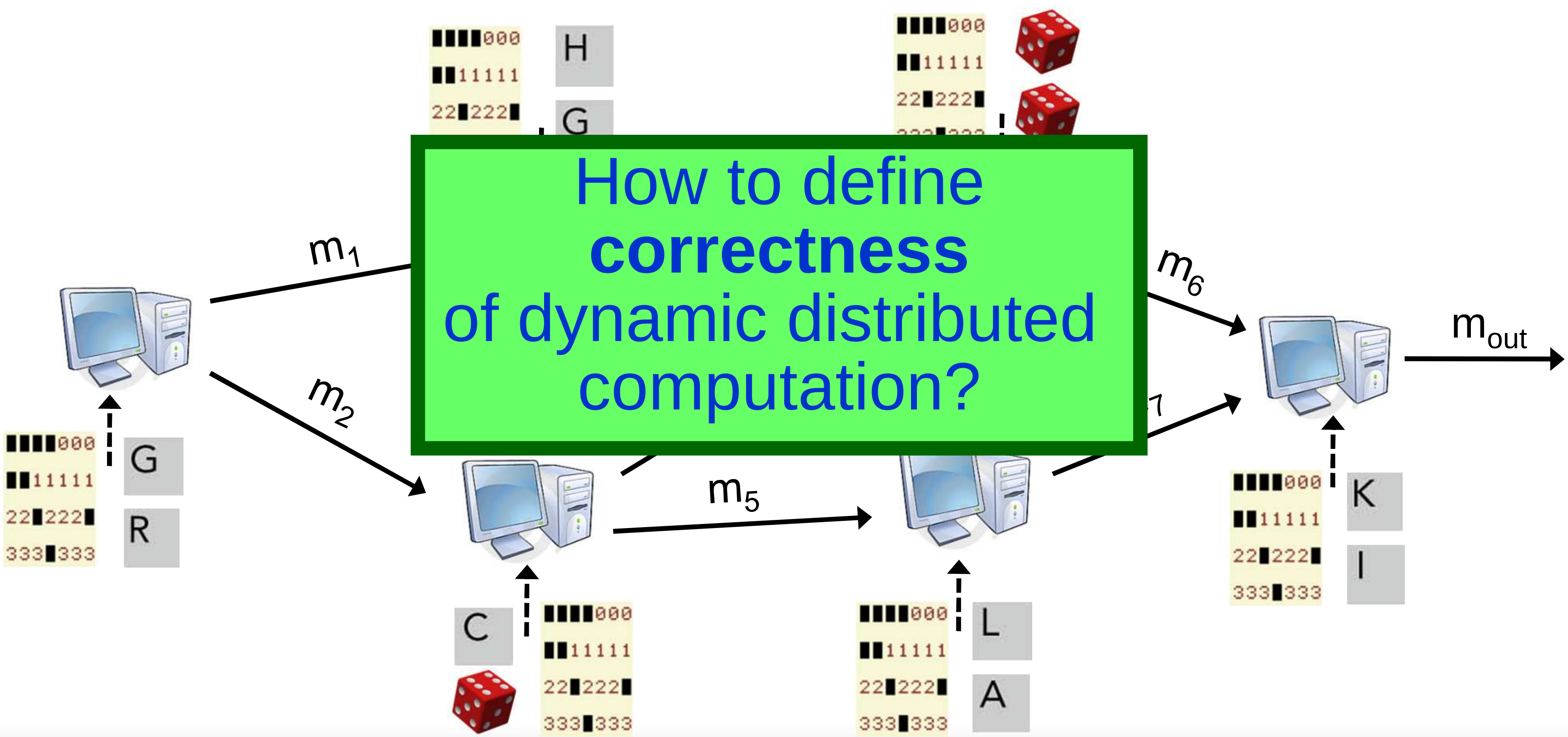
human inputs randomness program



Generalizing: arbitrary interactions

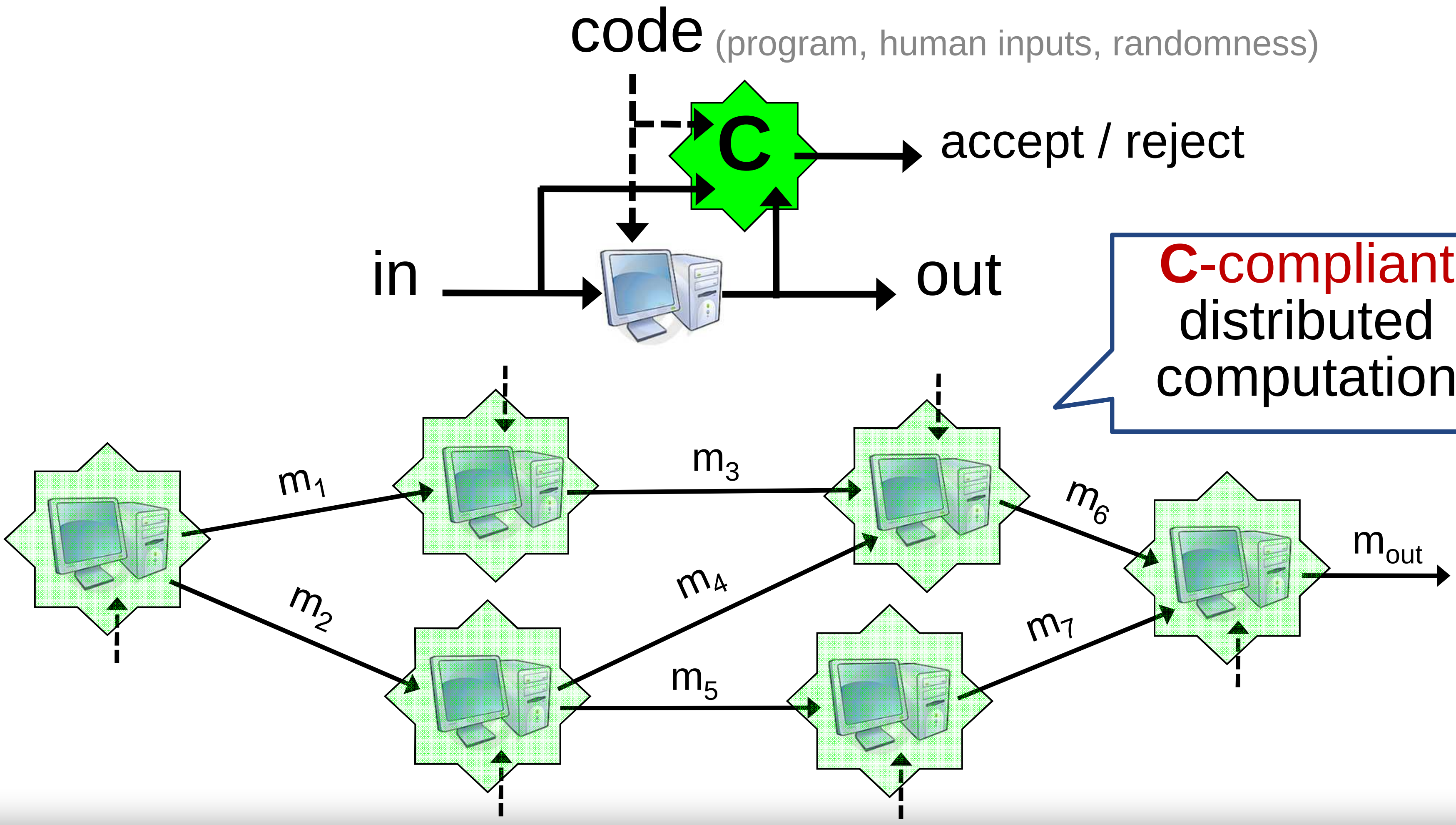
- Computation and graph are determined on the fly
  - by each party's local inputs:

human inputs randomness program



C-compliance

System designer specifies his notion of **correctness** via a **compliance predicate C**(in,code,out) that must be locally fulfilled at every node.

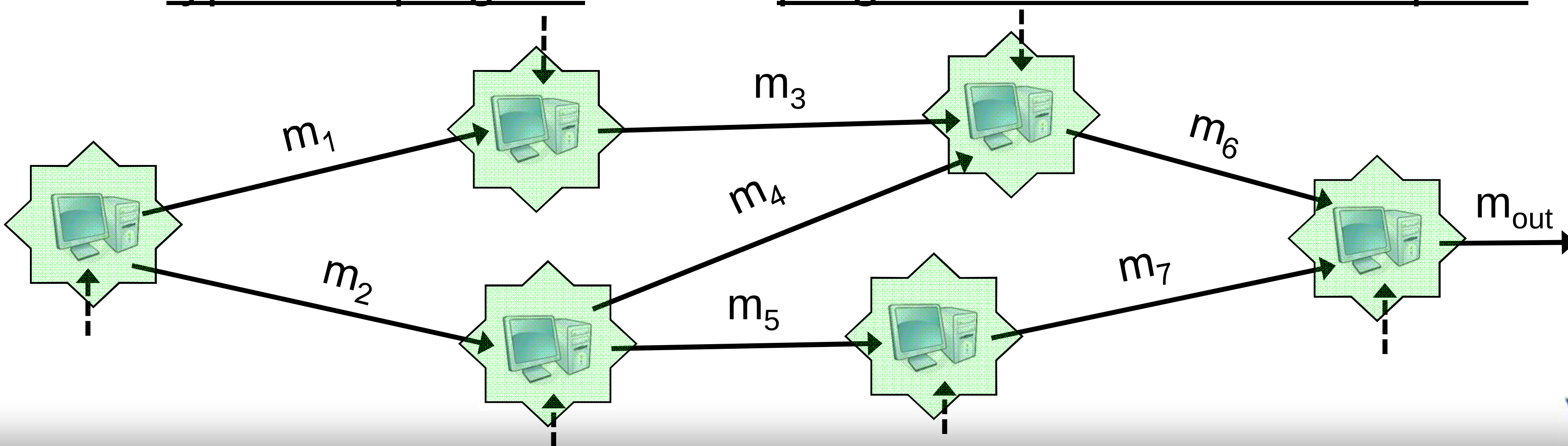


Examples of **C**-compliance

correctness is a **compliance predicate** **C**(in,code,out)
that must be locally fulfilled at every node

Some examples:

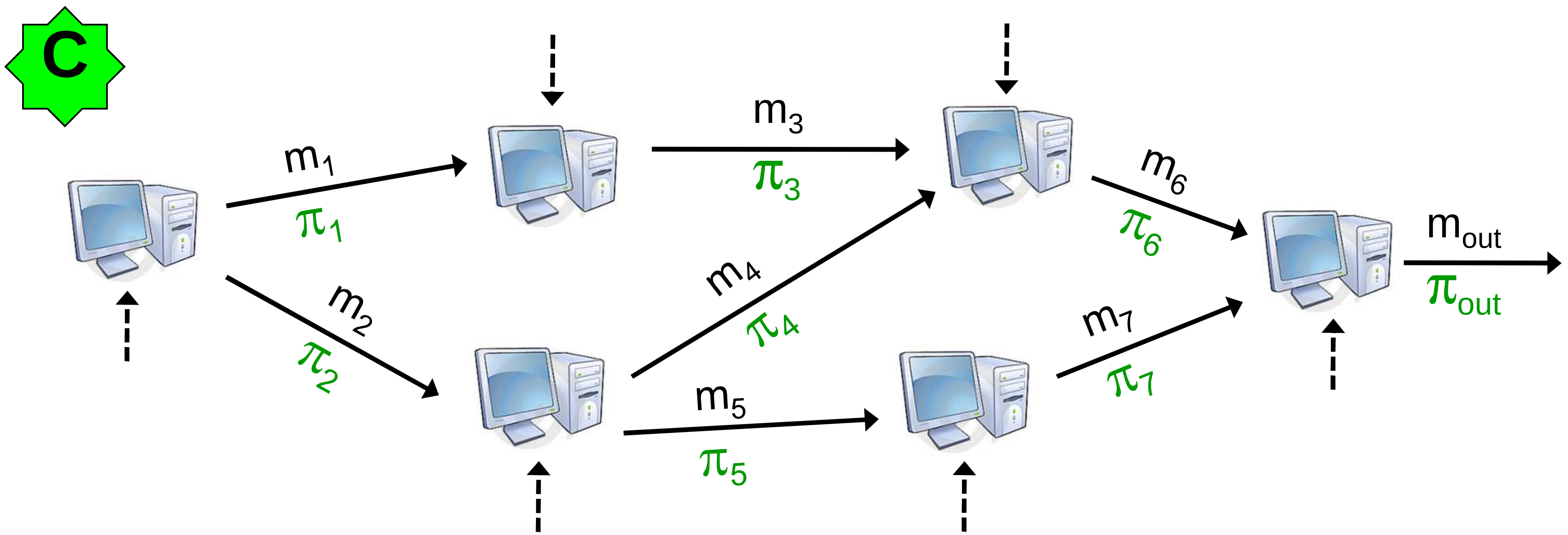
- **C** = “the output is the result of correctly computing a prescribed program”
- **C** = “the output is the result of correctly executing some program signed by the sysadmin”
- **C** = “the output is the result of correctly executing some type-safe program” or “… “program with a valid formal proof”



Dynamically augment computation with proofs strings

In PCD, messages sent between parties are augmented with concise proof strings attesting to their "compliance".

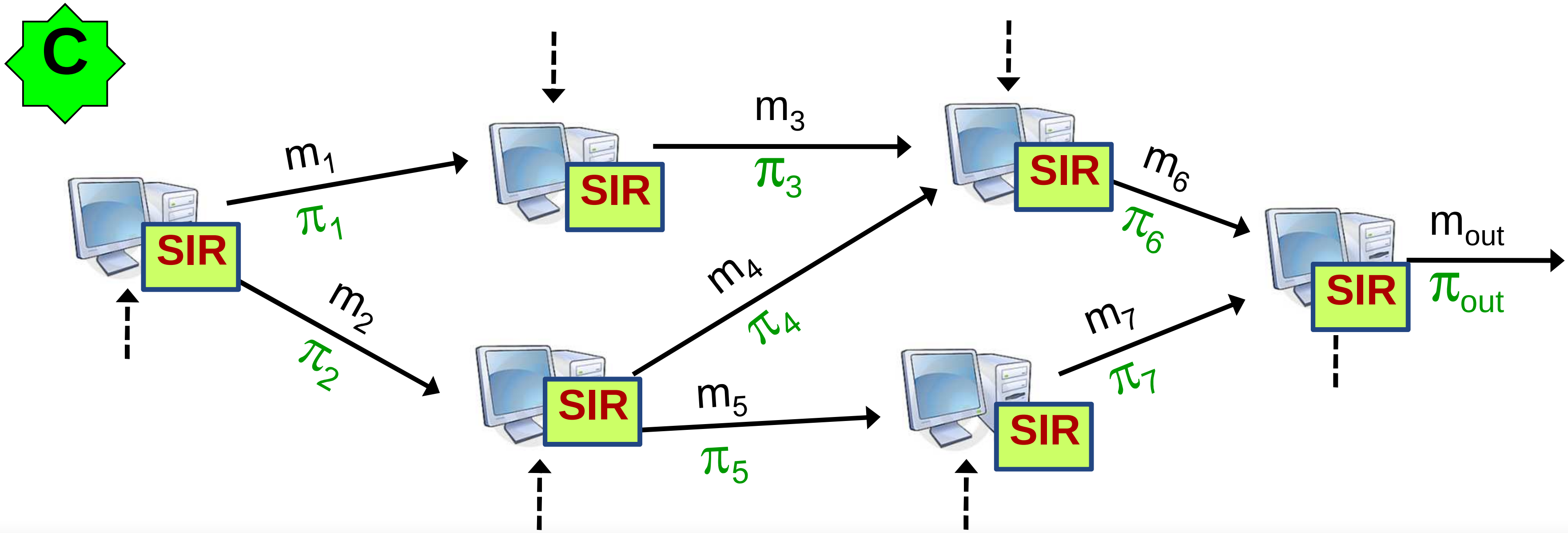
Distributed computation evolves like before, except that each party also generates on the fly a proof string to attach to each output message.



Extra setup ("model")

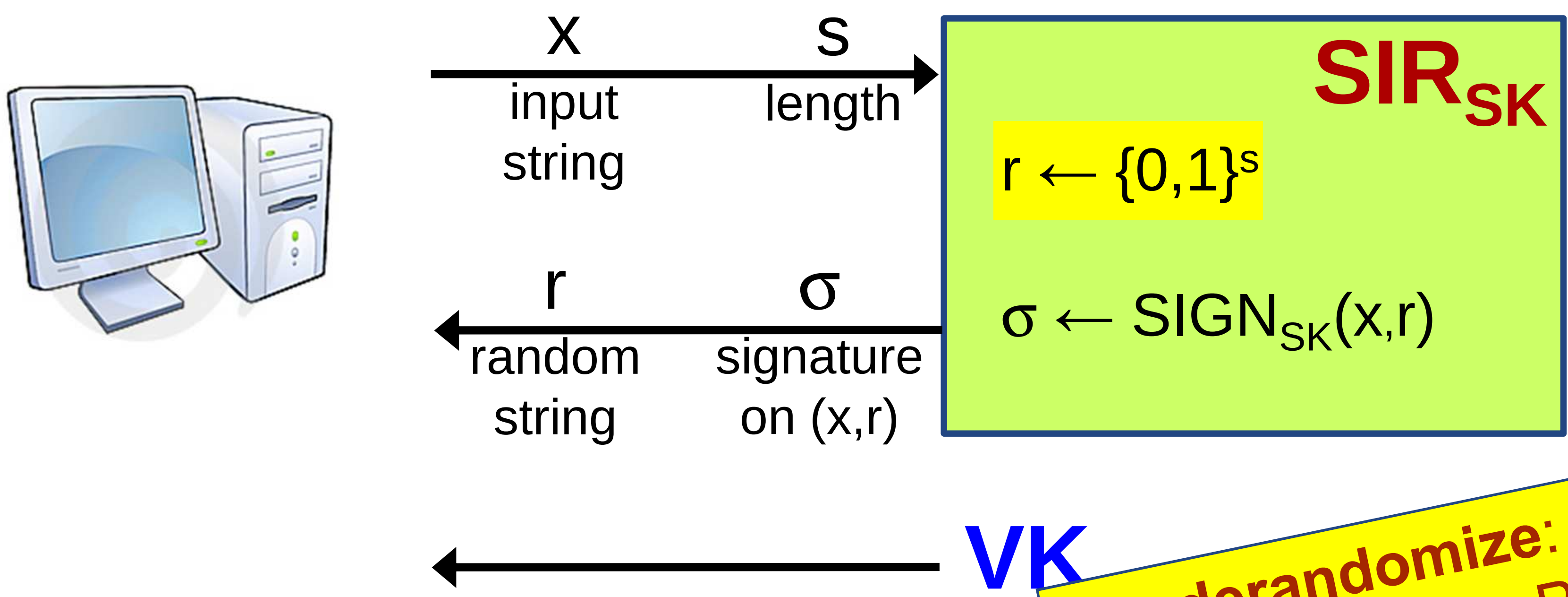
Every node has access to a simple, fixed, stateless trusted functionality

- Signed-Input-and-Randomness (SIR) oracle



Every node has access to a simple, fixed, stateless trusted functionality: essentially, a signature card.

- Signed-Input-and-Randomness (SIR) oracle



can **derandomize**:
Generate r using PRF

(Some) envisioned applications

Application:
Correctness and integrity of IT supply chain

- Consider a system as a collection of components, with specified functionalities
  - Chips on a motherboard
  - Servers in a datacenter
  - Software modules
- **C**(in,code,out) checks if the component's specification holds
- Proofs are attached across component boundaries
- If a proof fails, computation is locally aborted

  $\rightarrow$ integrity, attribution

Application:
Fault and leakage resilient Information Flow Control

Application: Fault and leakage resilient Information Flow Control

- Computation gets "`secret`" / "`non-secret`" inputs
- "`non-secret`" inputs are signed as such
- Any output labeled "`non-secret`" must be independent of `secrets`
- System perimeter is controlled and all output can be checked (but internal computation can be leaky/faulty).
- **C** allows only:
  - Non-secret inputs:
    Initial inputs must be signed as "`non-secret`".
  - IFC-compliant computation:
    Subsequent computation respect
    Information Flow Control rules and follow fixed schedule
- Censor at system's perimeter inspects all outputs:
  - Verifies proof on every outgoing message
  - Releases only non-secret data.

Application:
Fault and leakage resilient Information Flow Control

- Computation gets “`secret`” / “`non-secret`” inputs
- “`non-secret`” inputs are signed as such
- Any output labeled “`non-secret`” must be independent of `secrets`
- System perimeter is controlled and all output can be checked (but internal computation can be leaky/faulty).
- **C** allows only:
  - Non-secret inputs:
    Initial inputs must be signe
  - IFC-compliant computation:
    Subsequent computation re
    Information Flow Control rul
- Censor at system’s perime
  - Verifies proof on every outg
  - Releases only non-secret d

Big assumption, but otherwise no hope for retroactive leakage blocking (by the time you verify, the EM emanations are out of the barn).

Applicable when interface across perimeter is well-understood (e.g., network packets).

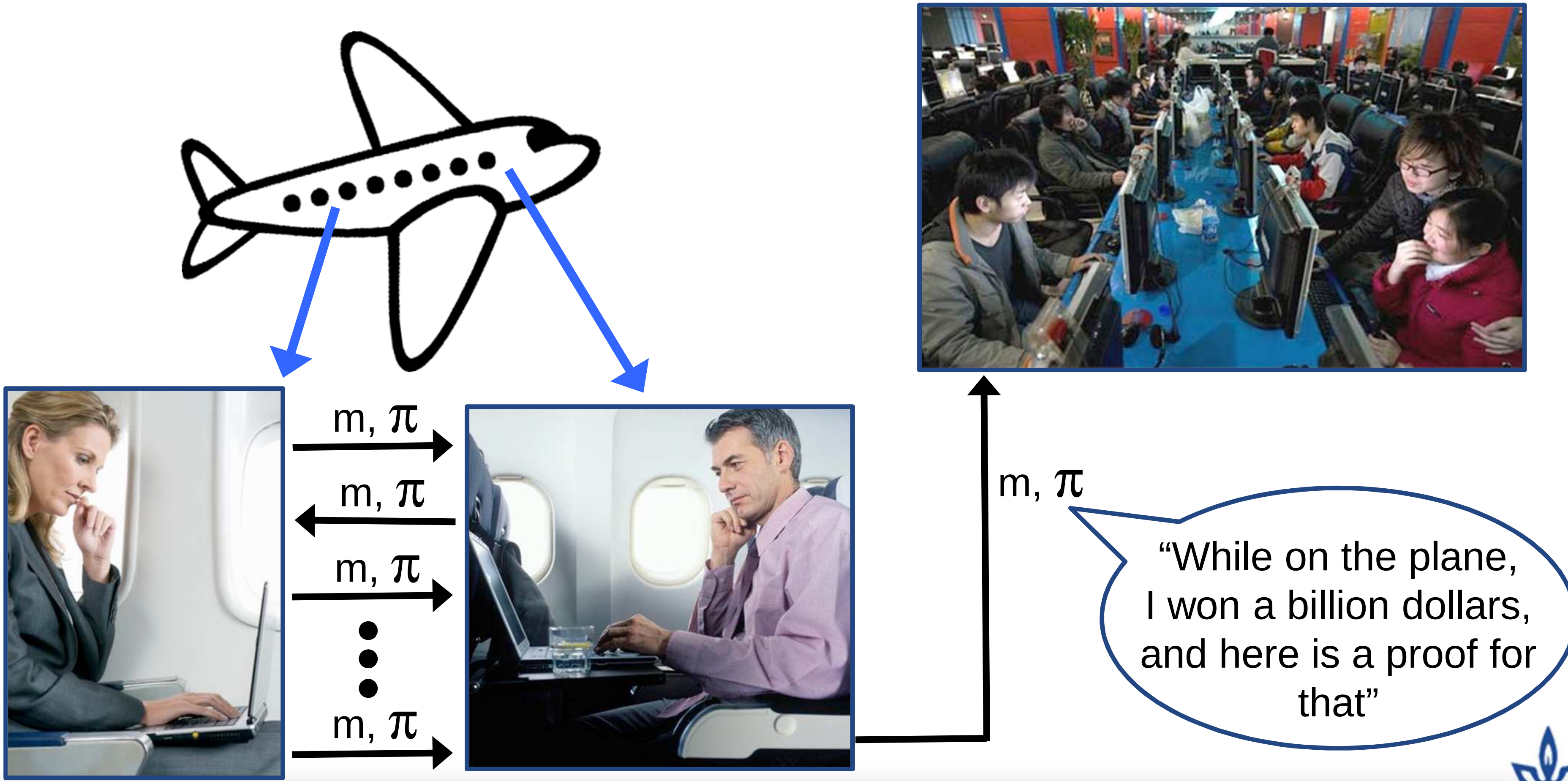
Verify using existing assurance methodology.

Application:
Simulations and MMO

- Distributed simulation:
  - Physical models
  - Virtual worlds (massively multiplayer online virtual reality)
- How can participants prove they have "obeyed the laws of physics"?
  (e.g., cannot reach through wall into bank safe)
- Traditional: centralized.
- P2P architectures strongly motivated but insecure
  [Plummer '04] [GauthierDickey et al. '04]
- Use **C**-compliance to enforce the laws of physics.

Application:
Simulations and MMO – example

- Alice and Bob playing on an airplane, can later rejoin a larger group of players, and prove they did not cheat while offline.



Application: type safety

C(in,code,out) verifies that
code is type-safe & out=code(in)

- Using PCD, type safety can be maintained
  - even if underlying execution platform is untrusted
  - even across mutually untrusting platforms

- Type safety is very expressive:
  - Using dependent types (e.g., Coq) or refinement types: Can express **any computable property**. Extensive literature on what that can be verified efficiently (at east with heuristic completeness – good enough!)
  - Using object-oriented model (subclassing as constraint specification): **leverage OO programming methodology**.

More applications

Mentioned:

- Fault isolation and accountability, type safety, multilevel security, simulations.

Many others:

- Enforcing rules in financial systems
- Proof-carrying code
- Distributed dynamic program analysis
- Antispam email policies

Security design reduces to "compliance engineering": write down a suitable compliance predicate **C**.

- Recurring patterns: signatures, censors, verify-code-then-verify-result…
- Introduce design patterns (a la software engineering) [GHJV95]

A few words about realization

Probabilistically Checkable Proofs (partial history)

early 80's

[Goldwasser, Micali, Rackoff][Babai and Moran]

generalization of NP proofs to include probabilistic verification and interaction

late 80's

[Ben-Or Goldwasser Kilian Wigderson][Shamir][Babai Fortnow Lund]

probabilistic verification and interaction buys a lot of "expressive" power

early 90's

[Babai Fortnow Levin Szegedy]

NP proofs can be written in a format that can be checked with only logarithmic queries and in polylogarithmic time!

[Arora Safra][Arora Lund Motwani Sudan Szegedy]

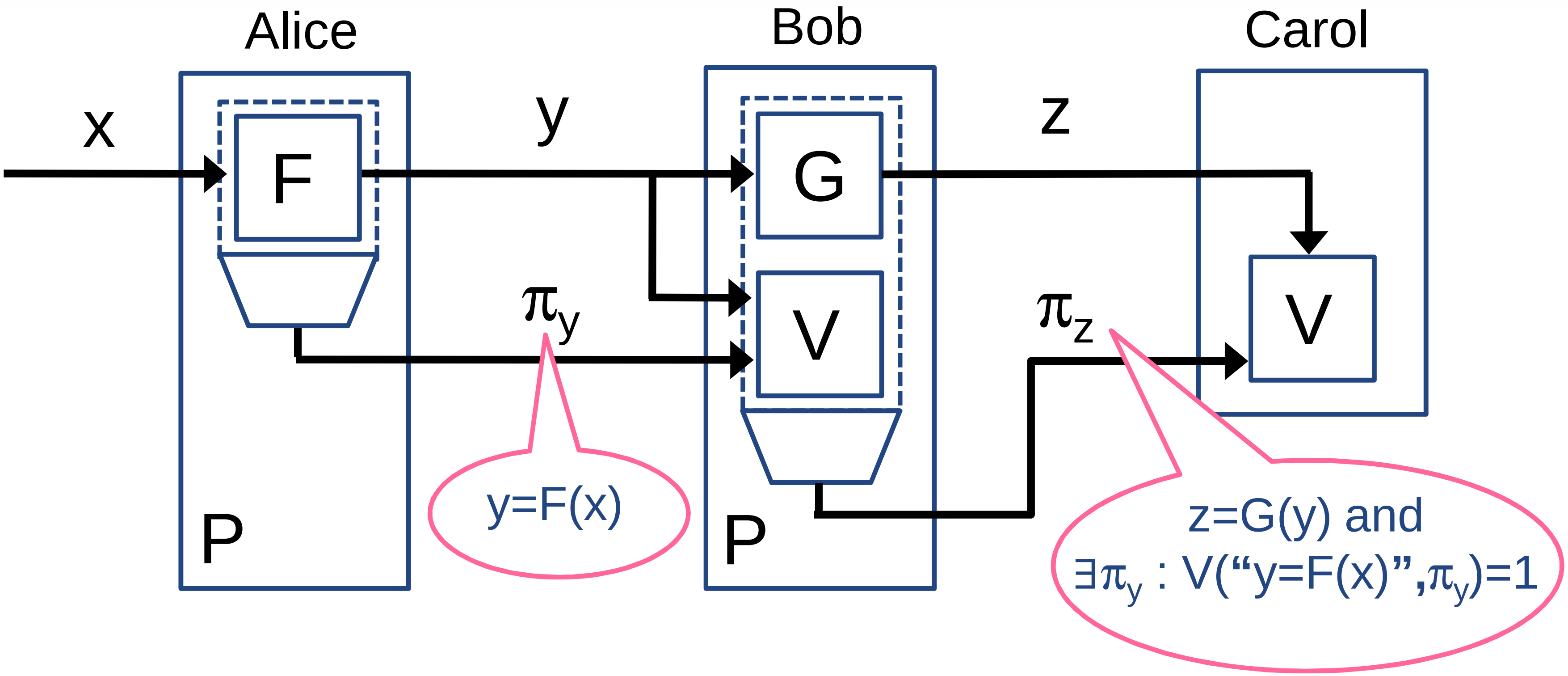
reduce number of queries to a constant

by now: many improved parameters, e.g., [Håstad] [Dinur]

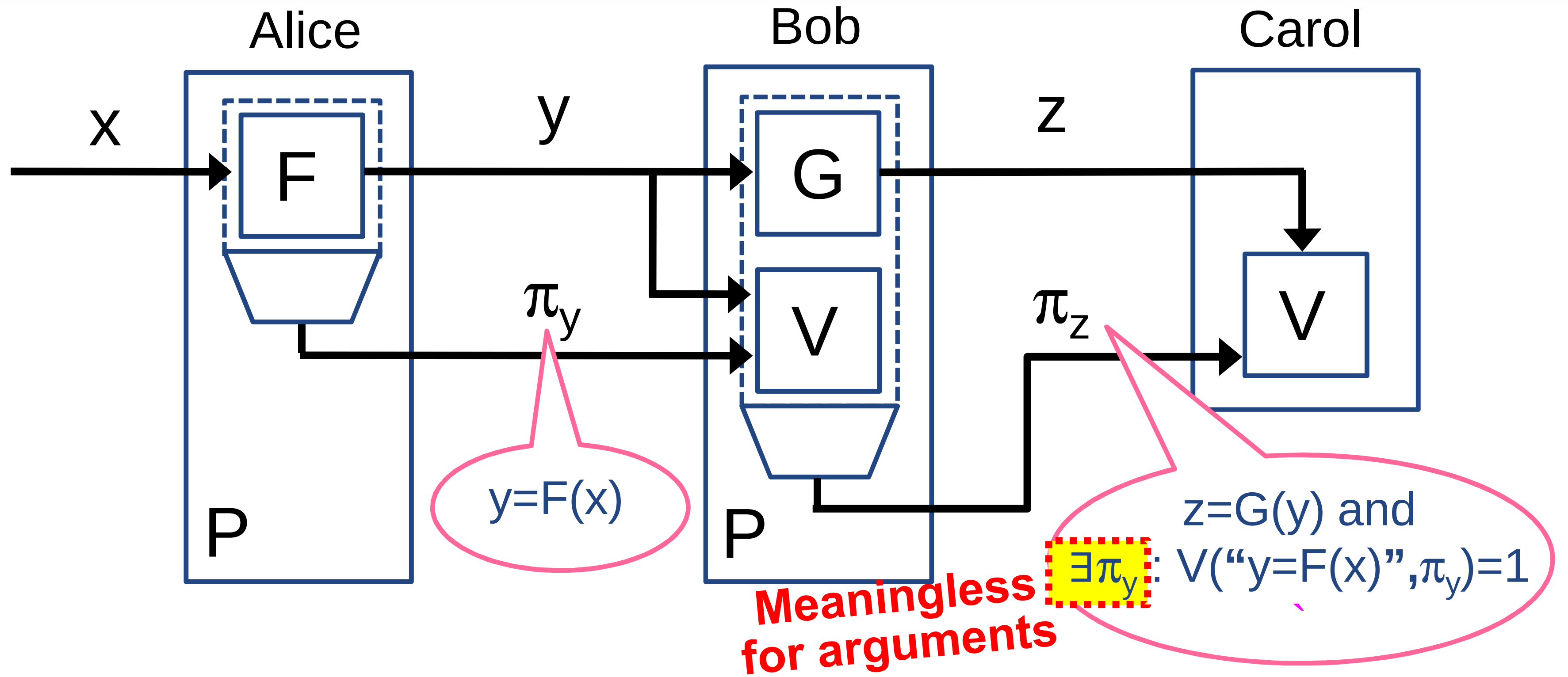
h[Ben-Sasson Sudan] [Ben-Sasson Goldreich Harsha Sudan Vadhan]

now

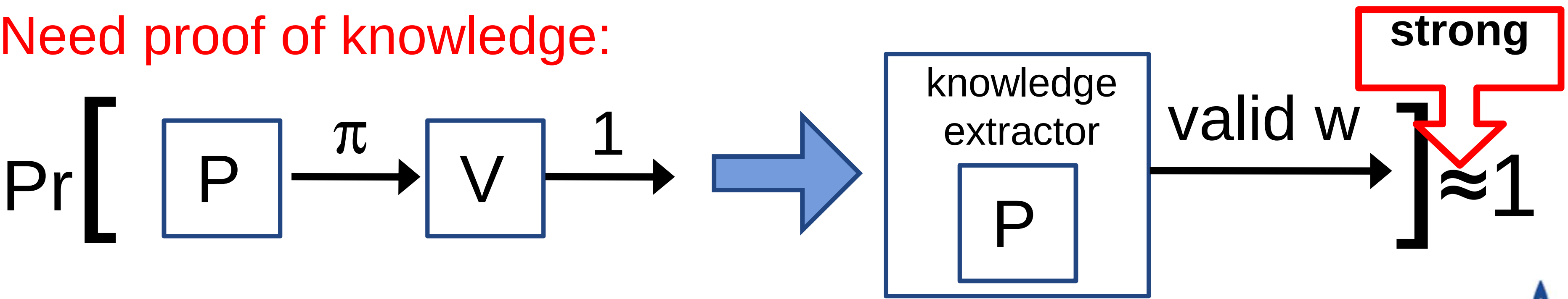
Proof aggregation



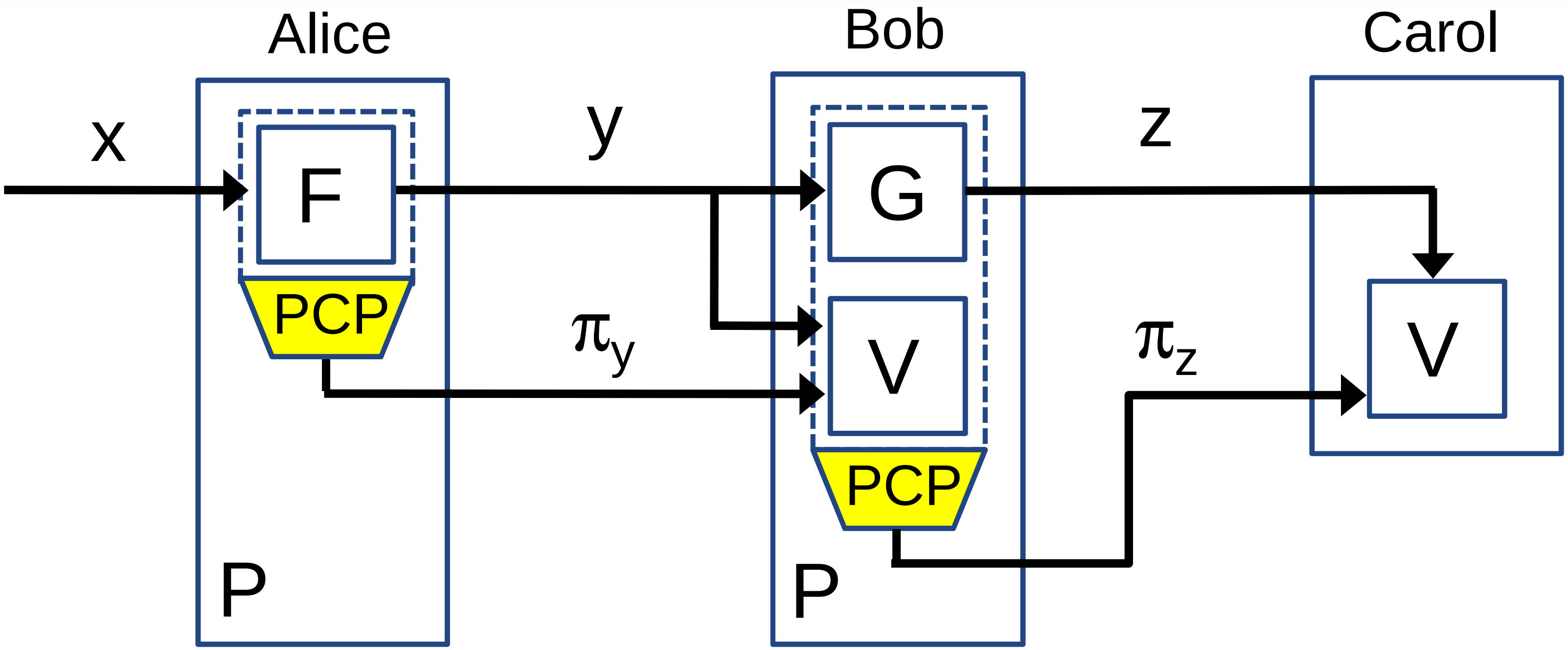
Soundness vs. proof of knowledge



Need proof of knowledge:



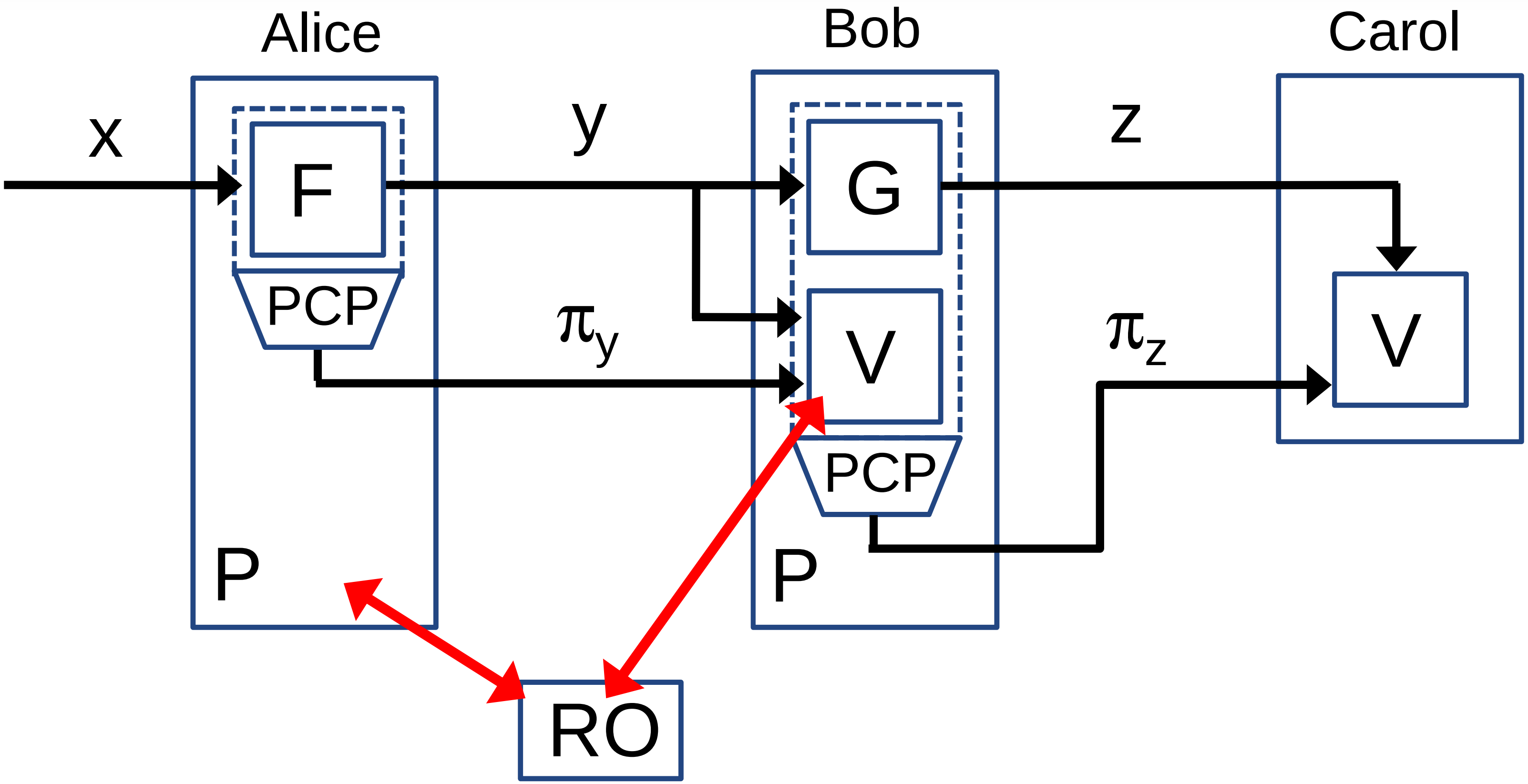
Must use PCPs for compression



- Probabilistically Checkable Proofs (PCPs) used to generate concise proof strings.

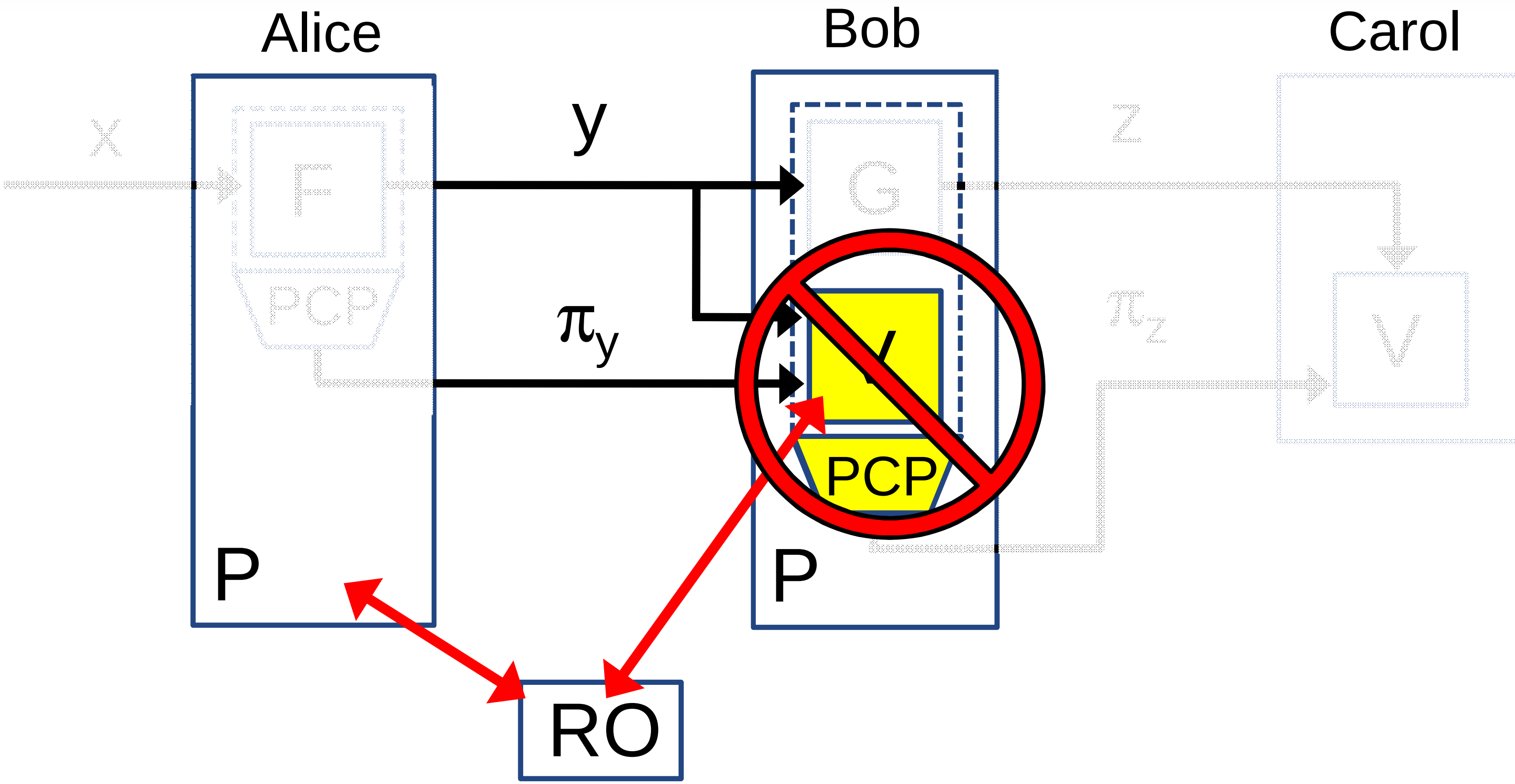
(And there is evidence this is inherent [Rothblum Vadhan 09].)

Must use oracles for non-interactive proof of knowledge



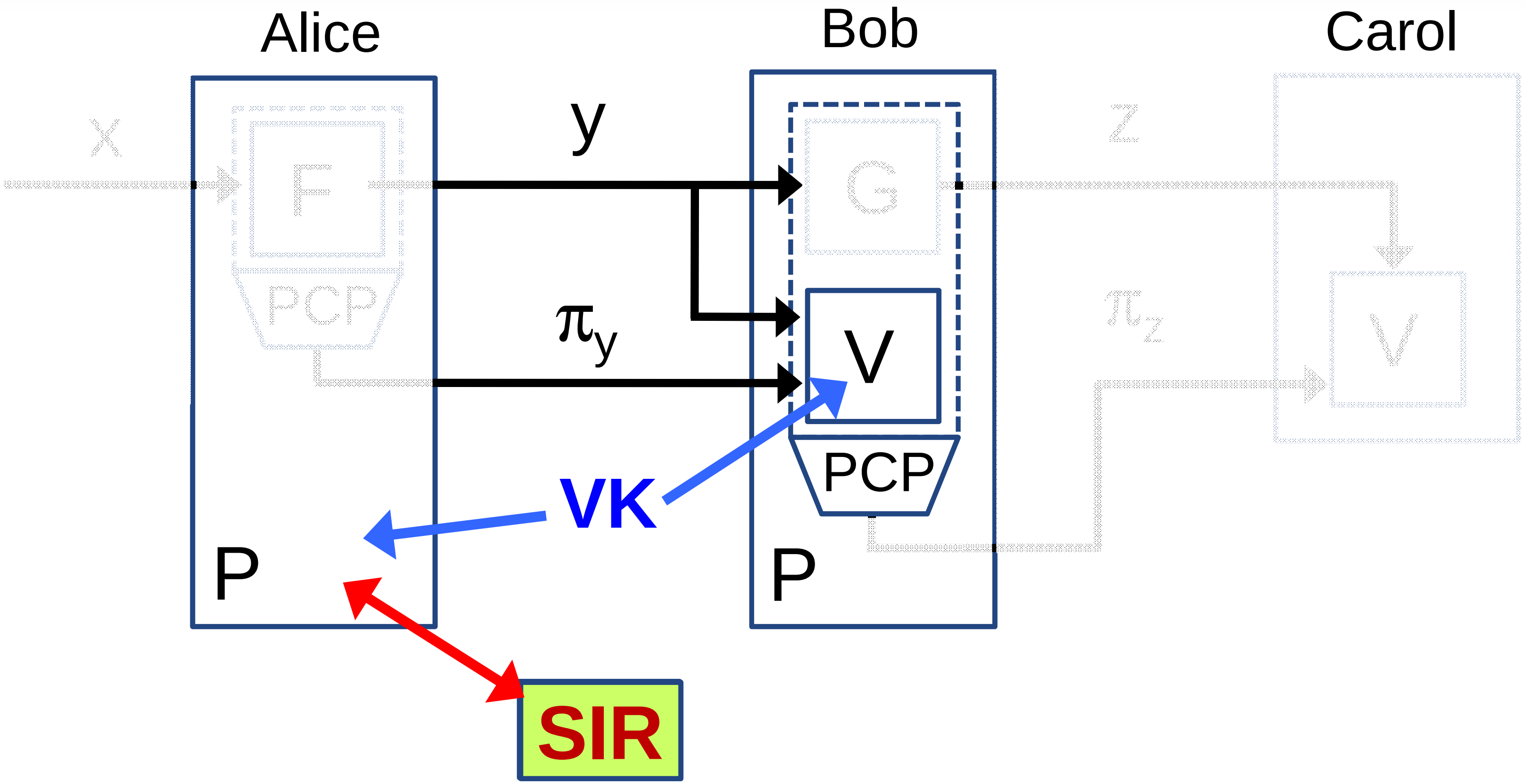
The only known construction of non-interactive proofs of knowledge is Micali's, using Merkle trees where the "hashing" is done using random oracle calls.

PCP vs. oracles conflict



- PCP theorem does **not** relativize [Fortnow '94], not even with respect to a RO [Chang et al. '92]
- this precluded a satisfying proof of security in [Valiant '08]

Our solution:
Public-key crypto to the rescue



Oracle signs answers using public-key signature:

- answers are verifiable without accessing oracle
- asymmetry allows us to break “PCP vs. oracle” conflict, and recursively aggregate proofs

Proof-Carrying Data: Conclusions and open problems

PCD offers a new approach to expressing and enforcing security properties in distributed systems:

- the system designer writes a compliance specification
- compliance is enforced in all subsequent computation, even if parties are untrusted and platforms are faulty and leaky

Established

- Formal framework
- Theoretical constructions

Ongoing and future work

- Detailed specifications, implementation and evaluation
- Achieve practicality ("polynomial time" PCP is not good enough!)
- Reduce assumptions, extend functionality
- Identify and realize applications

The road to PCD

Established: [Chiesa Tromer '10]

- Formal framework
- Explicit construction
  - "Polynomial time" - not practically feasible (yet).
  - Requires signature cards

Ongoing fundamental work:

- Reduce requirement for signature cards (or prove necessity)
- Extensions (e.g., zero knowledge)

Ongoing applicative work:

- Full implementation
- Practicality – improved algorithms
- Interface with complementary approaches: tie "compliance" into existing methods and a larger science of security
- Applications and "compliance engineering" methodology