Proof-Carrying Data and Hearsay Arguments from Signature Cards

Alessandro Chiesa and Eran Tromer
Motivation

System and protocol security often fail when assumptions about software, platform, and environment are violated.

<table>
<thead>
<tr>
<th>PLATFORM</th>
<th>SOFTWARE</th>
<th>ENVIRONMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>bug attacks</td>
<td>bugs</td>
<td>physical side channels</td>
</tr>
<tr>
<td>architectural side channels</td>
<td>trojans</td>
<td>tampering</td>
</tr>
<tr>
<td>hardware trojans</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
High-level goal

Ensure properties of a distributed computation when parties are mutually untrusting, faulty, leaky & malicious.
Example: 3-party correctness

\[ y \leftarrow F(x) \quad \text{Alice} \quad x, F \]

\[ y \rightarrow z \leftarrow G(y) \quad \text{Bob} \quad G \]

\[ z \rightarrow \text{Carol} \]

is \( z = G(F(x)) \) true?
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...
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.
Related work:
Secure multiparty computation \[\text{[GMW87][BGW88][CCD88]}\]

But:
- computational blowup is polynomial in the \textit{whole} computation, and not in the local computation
- does \textbf{not preserve} communication graph
- parties and computation must be \textbf{fixed in advance}
Generalizing:

The Proof-Carrying Data framework
Generalizing: distributed computations

Distributed computation?

Parties exchange messages and perform computation.

\[ m_1 \rightarrow m_2 \rightarrow m_3 \rightarrow m_4 \rightarrow m_5 \rightarrow m_6 \rightarrow m_7 \rightarrow m_{\text{out}} \]
Generalizing: arbitrary interactions

- Arbitrary interactions
  - communication graph over time is any DAG
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

How do we define correctness of a distributed computation?
C-compliance

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

**correctness** is a **compliance predicate** $C(\text{in},\text{code},\text{out})$ that must be locally fulfilled at every node.

Some examples:

- $C = “\text{none of the inputs are labeled secret}”$
- $C = “\text{the code was digitally signed by the sysadmin, and executed correctly}”$
- $C = “\text{the code is type-safe and the output is indeed the result of running the code}”$
Goals

Ensure $C$-compliance while respecting the original distributed computation.

- Allow for any interaction between parties
- Preserve parties’ communication graph
  - no new channels
- Allow for dynamic computations
  - human inputs, indeterminism, programs
- Blowup in computation and communication is local and polynomial
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.
Every node has access to a simple, fixed, stateless trusted functionality -- essentially, a signature card.

- **Signed-Input-and-Randomness (SIR) oracle**
Every node has access to a simple, fixed, trusted functionality -- essentially, a signature card.

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

  
  \[
  x \quad \xrightarrow{\text{input string}} \quad s \quad \xrightarrow{\text{length}} \quad SIR_{SK} \\
  r \quad \xleftarrow{\{0,1\}^s} \quad \sigma \quad \xleftarrow{\text{signature on } (x,r)} \quad VK
  \]

  \[\sigma \leftarrow \text{SIGN}(SK,(x,r))\]

  Similar assumptions: [Hofheinz Müller-Quade Unruh 05]
Sample application: type safety

\[ \text{C(} \text{in,code,out)} \text{ verifies that} \]
\[ \text{code is type-safe} \& \text{ out=code(in)} \]

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

- Type safety is very expressive
  - Can express any computable property
  - Extensive literature on types that can be verified efficiently
    (at least with heuristic completeness, which is good enough)
  - E.g., can do certain forms of confidentiality via IFC
Our results
Overview of Results

Proof-Carrying Data (PCD):
• C-compliance
• Aggregate proof strings to generate new ones
• Simpler interface hides implementation details

Assisted-Prover Hearsay-Arguments (APHA):
• Very strong variant of non-interactive CS proofs / arguments of knowledge (for NP)
• Proof system for a “single step”
Overview of Results

Need:

- Universal arguments (CS proofs) that are public-coin and constant-round [Barak Goldreich 02] [Micali 94]
- Signature schemes that are strongly unforgeable
generic (from UOWHFs): [Goldreich 04]
efficient: [Boneh Shen Waters 02]

Both exist if Collision Resistant Hash schemes exist.
Rest of this talk:

- Intuition on how to aggregate proofs in “F and G” example
Proof aggregation

$y = F(x)$

$z = G(y)$ and $\exists \pi_y : V("y = F(x)\), \pi_y) = 1$
Soundness vs. proof of knowledge

Need proof of knowledge:

\[
\Pr\left[ \begin{array}{c}
P \\ \pi \\ V \\ 1 \\ \end{array} \right] \approx 1
\]

valid w

\[z = G(y)\text{ and } \exists_{\pi_y} V(\text{"y = F(x)"}, \pi_y) = 1\]

strong
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
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
Sketch of remaining constructions

Constructing APHAs:
• Start with universal arguments
• De-interactivize by replacing public-coin messages with oracle queries
• Add signature to statement to force witness query ($\approx$ [Chandran et al. 08])
• Prove a very strong PoK by leveraging the weak PoK of UA

Generalizing to PCD:
• Handle distributed-computation DAG, using APHA for the proofs along each edge.
• C-compliance: use fixed aggregation rule to reason about arbitrary computation by proving statements of the form:
  \[ C(\text{in}, \text{code}, \text{out})=1 \quad \text{&} \quad \text{“each input carries a valid APHA proof string”} \]
Discussion
Applications

Security design reduces to “compliance engineering”: write down a suitable compliance predicate $C$. 
Proof-Carrying Data: Conclusions and open problems

Contributions

• Framework for securing distributed computations between parties that are mutually untrusting and potentially faulty, leaky, and malicious.

• Explicit construction, under standard generic assumptions, in a “signature cards” model.

• Suggested applications.

Ongoing and future work

• Reduce requirement for signature cards, or prove necessity.

• Add zero-knowledge constructions.

• Achieve Practicality (PCPs are notorious for “polynomial” overheads).

• Identify and implement applications.
APHA systems

Construction sketch
What we have and what we need

Needed:

• Highly-compressing non-interactive arguments
• Proof-of-knowledge property that’s strong enough to prove “hearsay”:
  \textit{statements whose truth relies on previous arguments heard from others.}
• In the assisted prover model.

We call these \textbf{Assisted Prover Hearsay Arguments (APHA)} systems.
Start with universal arguments:
Efficient interactive arguments of knowledge with constant rounds and public coins.

- public coin: $r_1$ and $r_2$ are just random coins
- temporal dependence: $r_2$ is sent only after $\text{resp}_1$ is received
De-interactivize universal arguments: first try

Prover interacts with random oracle, not with verifier:
• obtains signed random strings
De-interactivize universal arguments

Prover interacts with SIR oracle, not with verifier:
- obtains random strings
- temporal ordering enforced by having each oracle query include the preceding transcript

≈ Fiat-Shamir heuristic
Enhance proof of knowledge

- Forces prover to get signature on witness
- Knowledge extractor finds witness by examining the queries → strong proof of knowledge

\[ x \in L \quad \text{and} \quad \sigma \text{ is a signature on a witness for } x \in L \]

\[ \sigma \]

Cf. [Chandran Goyal Sahai 08]
Can now do F and G example

We can now do the above example!

... how about proof-carrying data?
PCD systems
PCD systems are wrappers around APHA systems, with:

- Simpler interface for applications
  (no need to reason about theorems and proofs)
- Simpler proof-of-knowledge property
  (APHAs have a very technical “list extraction” definition)
- \textbf{C}-compliance
At every node, a party uses the **PCD prover** $P_{PCD}$:

Proofs are checked by the **PCD verifier** $V_{PCD}(C, VK, z_*, \pi)$ decides if $\pi$ is a convincing proof for $z_*$. 
PCD definition

PCD systems satisfy

- **efficient verifiability:**
  \[ \text{TIME}(V_{\text{PCD}}(C, \text{VK}, z, \pi)) = \text{polylog}(\text{time to make } \pi) \]
  (actually, much better... )

- **completeness via a relatively efficient prover:**
  if computation is $C$-compliant, then the proof output by
  prover convinces verifier. Moreover:
  \[ \text{TIME}(P_{\text{PCD}}(...)) = \text{poly}(\text{time to check } C) \]
  \[ + \text{polylog}(\text{time to generate inputs’ proofs}) \]

- **proof of knowledge:** from any convincing prover, can extract a
  whole $C$-compliant computation
• Having APHA systems, we can already do the above example
• How to generalize to C-compliance?
Adding **C**-compliance

- "Export" the choice of computation to be an input to a "fixed rule of aggregation"
Such fixed rule we call the **PCD machine**, and it depends on $C$.