Implementations of probabilistic proofs for verifiable outsourcing: survey and next steps

Srinath Setty
Microsoft Research

(Thanks to Michael Walfish for some of the slides.)
without executing f, can check that: “y = f(x)”
more generally: “prover knows w s.t. y = f(x,w)”

Motivation: Third-party computing
• Cloud computing, distributed ledger technologies (DLTs)

Requirements:
• Efficiency (client CPU, communication, server CPU, etc.)
• Privacy of w (zero-knowledge, desirable in some applications)
A naïve implementation of the theory results in outrageous costs

Thousands of CPU years to verifiably execute even simple computations

What do we need?
Practicality (as real people understand the term) in addition to efficiency and privacy for w
Good news
• Running code; cost reductions of \(10^{20}\) vs. theory
• Compilers from C to protocol entities
• Stateful computations; remote inputs, outputs
• Concretely efficient verifiers

Bad news
• Extreme expense: \(10^6\)x overhead vs. native
• Programming model is clumsy
• Useful only for special-purpose applications
Note: There are pragmatic alternatives

Replication [Castro & Liskov TOCS02]
Far less expensive, but it does not support privacy for w

Trusted hardware such as Intel SGX
Far less expensive, but requires significant trust
No formal security guarantees
Hard (or impossible) to reason about end-to-end security
Rest of this talk

Summary of state of the art implementations

Reality check with a performance evaluation

Next steps
Common framework in state of the art systems

front-end
(program translator)

C program
main(){
  ...
}

arithmetic circuit
(non-det. input)

back-end
(probabilistic proof protocol)

verifier
\( x \leftrightarrow y, \pi \)

prover

interactive proof [GKR08]
interactive argument [IKO07]
non-interactive argument [Groth10, Lipmaa12, GGPR12]

General “processor”
Custom circuit
<table>
<thead>
<tr>
<th></th>
<th>Interactive proofs</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[GKR08, CMT12, …]</td>
<td>Interactive [IKO07, SBW11, SMBW12, …]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-interactive [Groth10, Lipmaa12, GGPR12, …]</td>
</tr>
<tr>
<td>Circuit type</td>
<td>Deterministic</td>
<td>Non-deterministic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-deterministic</td>
</tr>
<tr>
<td>#Rounds</td>
<td>Lots</td>
<td>Two</td>
</tr>
<tr>
<td></td>
<td></td>
<td>One</td>
</tr>
<tr>
<td>Assumptions</td>
<td>None</td>
<td>Simple, falsifiable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-standard</td>
</tr>
<tr>
<td>Prover expense</td>
<td>10 to 100x</td>
<td>$10^6x$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$10^6x$</td>
</tr>
<tr>
<td>Verifier setup</td>
<td>0 or (10 to 100x)</td>
<td>$10^6x$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$10^6x$</td>
</tr>
<tr>
<td>Zero-knowledge</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Hardware impl.</td>
<td>Yes</td>
<td>Non-aemenable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-aemenable</td>
</tr>
</tbody>
</table>

All recent implementations use the QAP encoding [GGPR12]
Attempt 1: Use PCPs that are asymptotically short

[ALMSS92, AS92] [BGHSV05, BGHSV06, Dinur07, BS08, Meir12, BCGT13]

This does not meet the efficiency requirements (because $|\text{PCP}| > \text{running time of } f$).
Attempt 2: Use arguments or CS proofs

But the constants seem too high ...
Attempt 3: Use long PCPs interactively [IKO07, SMBW12, SVPBBW12]

Achieves simplicity, with good constants ... ... but pre-processing is required (because $|q_i| = |v|$) ... and prover’s work is quadratic; address that shortly
Attempt 4: Use long PCPs non-interactively

Query processing now happens “in the exponent” ...
... pre-processing still required (again because $|q_i| = |v|$)
... prover’s work still quadratic; addressing that soon
# Recap

<table>
<thead>
<tr>
<th>who</th>
<th>efficient (short) PCPs</th>
<th>arguments, CS proofs</th>
<th>arguments w/ preprocessing</th>
<th>SNARGs w/ preprocessing</th>
</tr>
</thead>
<tbody>
<tr>
<td>who</td>
<td>ALMSS92, AS92, BGVH, Dinur, ...</td>
<td>Kilian92, Micali94</td>
<td>IKO07, SMBW12, SVPBBW12, SBVBPW13</td>
<td>GGPR12, BCIOP13, ...</td>
</tr>
<tr>
<td>what</td>
<td>classical PCP</td>
<td>commit to PCP by hashing</td>
<td>commit to long PCP using linearity</td>
<td>encrypt queries to a long PCP</td>
</tr>
<tr>
<td>security</td>
<td>unconditional</td>
<td>CRHFs</td>
<td>linearly HE</td>
<td>knowledge-of-exponent</td>
</tr>
<tr>
<td>why/why not</td>
<td>not efficient for V</td>
<td>constants are unfavorable</td>
<td>simple</td>
<td>simple, non-interactive</td>
</tr>
</tbody>
</table>

(Thanks to Rafael Pass.)
Final attempt: apply linear query structure to GGPR’s QAPs

Addresses the issue of quadratic costs.
PCP structure implicit in GGPR. Made explicit in [BCIOP13, SBVBBW13].

[Groth10, Lipmaa12, GGPR12]
- Standard assumptions
- Amortize over batch
- Interactive

- Non-falsifiable assumptions
- Amortize indefinitely
- Non-interactive, ZK, ...

"Zaatar" [SBVBBW13]
Interactive argument [IKO07]

"Pinocchio," "libsnark"
SNARG, zk-SNARK with pre-processing
Pre-processing avoidable in theory [BCCT13, BCTV14b, CTV15]

QAPs

plaintext queries

queries in exponent

[PGHR13, BCTV14a]
[Groth10, BCCT12, GGPR12]

[SBVBBW13]
C program \rightarrow \text{arithmetic circuit (non-det. input)} \rightarrow \text{QAPs [GGPR12]} \rightarrow \text{prover} \rightarrow \text{verifier}
State of the art front-ends

“General” processor

- Verbose circuits (costly)
- Good amortization
- Great programmability

Custom circuits

- Concise circuits
- Amortization worse
- How is programmability?

Circuit is unrolled CPU execution

Each line translates to gates

\[ \text{BCGTV13, BCTV14a, BCTV14b, CTV15} \]
Front-ends trade off performance and expressiveness

<table>
<thead>
<tr>
<th>Concrete</th>
<th>Applicable Computations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Costs</td>
<td></td>
</tr>
<tr>
<td>Lower</td>
<td>special-purpose</td>
</tr>
<tr>
<td></td>
<td>pure</td>
</tr>
<tr>
<td></td>
<td>stateful</td>
</tr>
<tr>
<td></td>
<td>general loops</td>
</tr>
<tr>
<td></td>
<td>function pointers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Special-purpose</th>
<th>Pure</th>
<th>Stateful</th>
<th>General Loops</th>
<th>Function Pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thaler CRYPTO13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMT, TRMP ITCS, Hotcloud12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pepper, Ginger NDSS12, Security12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trueset, Zerocash Security14, Oakland15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zaatar Euros13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pinocchio Oakland15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geppetto Oakland15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pantry SOSP13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Custom circuit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>General “processor”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCTV Security14 BCGTV CRYPTO13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proof-carrying data CRYPTO14, Eurocrypt15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Short PCPs Eurocrypt17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Front-ends trade off performance and expressiveness. Concrete costs lower the tradeoff, while higher costs make it more difficult to achieve a balance. Higher (still theory) costs represent the highest level of performance and expressiveness. Applicable computations include special-purpose, pure, stateful, general loops, and function pointers. Custom circuits offer a better tradeoff between performance and expressiveness. General "processor" refers to an idealized computational model that balances performance and expressiveness. Proof-carrying data and short PCPs are advanced techniques used in this context.
Summary of common framework:

Front-end (program translator):
- main()
- "custom circuit"

Back-end (argument variants):
- verifier
- prover
- QAPs
- x -> y, π

"general processor"
Summary of state of the art implementations

Reality check with a performance evaluation

Next steps
Quick performance study

Back-end: libsnark i.e., BCTV’s optimized Pinocchio implementation

Front-ends: implementations or re-implementations of

- Zaatar (Custom circuit) [SBVBPW Eurosys13]
- BCTV (General processor) [Security14]
- Buffet (Custom circuit) [WSRHBW NDSS15]
Landscape of front-ends (again)

Applicable computations

<table>
<thead>
<tr>
<th>concrete costs</th>
<th>special-purpose</th>
<th>pure</th>
<th>stateful</th>
<th>general loops</th>
<th>function pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>lower</td>
<td>Thaler</td>
<td>CRYPTO13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CMT, TRMP</td>
<td>ITCS, Hotcloud12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pepper, Ginger</td>
<td>NDSS12, Security12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Trueset, Zerocash</td>
<td>Security14, Oakland15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Zaatar</td>
<td>Eurosys13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Geppetto</td>
<td>Oakland15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pinocchio</td>
<td>Pantry</td>
<td>SOSP13</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Buffet</td>
<td>Buffet</td>
<td>NDSS15</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BCTV</td>
<td>BCTV</td>
<td>Security14</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BCGTV</td>
<td>BCGTV</td>
<td>CRYPTO13</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Proof-carrying data</td>
<td>CRYPTO14, Eurocrypt15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Short PCPs</td>
<td>Eurocrypt17</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Higher

Highest (still theory)
Quick performance study

Back-end: libsnark i.e., BCTV’s optimized Pinocchio implementation

Front-ends: implementations or re-implementations of:

- Zaatar (Custom circuit) [SBVBPW Eurosys13]
- BCTV (General processor) [Security14]
- Buffet (Custom circuit) [WSRHBW NDSS15]

Evaluation platform: cluster at Texas Advanced Computing Center (TACC)

Each machine runs Linux on an Intel Xeon 2.7 GHz with 32GB of RAM.
(1) What are the verifier’s costs?

(2) What are the prover’s costs?

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Proof length</td>
<td>288 bytes</td>
</tr>
<tr>
<td>V per-instance</td>
<td>$6 \text{ ms} + (</td>
</tr>
<tr>
<td>V pre-processing</td>
<td>$</td>
</tr>
<tr>
<td>P per-instance</td>
<td>$</td>
</tr>
<tr>
<td>P’s memory requirements</td>
<td>$O(</td>
</tr>
</tbody>
</table>

(|C|: circuit size)

(3) How do the front-ends compare to each other?

(4) Are the constants good or bad?
How does the prover’s cost vary with the choice of front-end?

Extrapolated prover execution time, normalized to Buffet
All of the front-ends have terrible concrete performance.

Extrapolated prover execution time, normalized to native execution.
Summary of concrete performance

- Front-end: generality brings a concrete price (but better in theory)
- Verifier’s “variable costs”: genuinely inexpensive
- Prover’s computational costs: **near-total disaster**
- Memory: creates scaling limit for verifier and prover
One option: where do we find a place?

Caution!
- Proof generation takes many minutes
- Needs trusted setup
- Prover needs queries that are many GBs

- Anonymous credentials: Cinderella [Oakland16]
- Anonymity for Bitcoin: Zerocash [Oakland14]
- Location-private tolling [Security09]: Pantry [SOSP13]

Another option: try to motivate theoretical advances
Summary of state of the art implementations

Reality check with a performance evaluation

Next steps: We need 3-6 orders of magnitude speedup
Wish list (1): front-end

• More efficient reductions from programs to circuits

• Inexpensive floating-point operations (to target domains such as deep learning, machine learning, ...)

• Better handling of state
### Status quo: systems that handle external state

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Generality</td>
<td>Any circuit</td>
<td>Specific</td>
<td>Any circuit</td>
</tr>
<tr>
<td>Prover expense</td>
<td>O(k log(</td>
<td>D</td>
<td>))</td>
</tr>
<tr>
<td>Concrete expense</td>
<td>$10^6$ to $10^8x$</td>
<td>$10^6$ to $10^8x$</td>
<td>$10^6$ to $10^8x$</td>
</tr>
</tbody>
</table>

VSQLE [S&P17] recently proposed an approach based on polynomial commitments, but it also opens the entire database inside circuit.

- Bottom line: handling state adds additional expense.
Wish list (2): back-end

• Construct short PCPs that are efficient
  Ben-Sasson et al. [EUROCRYPT17] have taken steps toward this, but concrete costs are quite high

• Endow IKO’s arguments with more properties or lower costs
  Reuse the verifier’s setup work beyond a batch
  Make the protocol zero knowledge

• Add zero-knowledge inexpensively to GKR’s protocol

• Improve GGPR’s QAPs or the cryptography used to query it
Wish list (3): Mission-critical applications

• Verifiable database with support for industrial-grade features: multiple users, concurrency, indexes, etc.

• Screaming performance for the prover e.g., tons of the

Lots of other ideas needed; we don’t know what they are!

• Why? DBs process financial transactions worth trillions of dollars. Connections with emerging distributed ledgers.
Conclusions and takeaways

- Exciting area with good news and bad news
  - Lots of progress, but ...
  - ... extreme expense in general-purpose systems
- Overheads rooted in QAPs and circuit representation
- Theoretical breakthroughs are needed

- Incentive: the potential is huge, especially with emerging distributed ledgers
  (http://www.pepper-project.org/)