Introduction and overview of verifiable computation

(≈ delegation of computation
≈ succinct arguments
≈ execution integrity)

Bar-Ilan Winter School on Verifiable Computation
Class 1
January 4, 2016

Michael Walfish
Dept. of Computer Science, Courant Institute, NYU
without executing \( f \), check that: “\( y = f(x) \)”

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
without executing $f$, check that: “$y = f(x)$”

Classic and fundamental problem
- Many applications (cloud computing, information retrieval, untrusted hardware supply chain, etc.). Generalizes to verifying assertions.

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.
Many of the variants are addressable in theory, with probabilistic proof protocols.

![Diagram of a Prover-Verifier Protocol](image)

\[ \text{verifier (randomized)} \quad \text{``claim: } f(x) = y \text{''} \quad \text{prover} \]

Citations here and throughout connect to the references at the end of the slides.
Many of the variants are addressable in theory, with probabilistic proof protocols

Are probabilistic proof protocols practical?

REJECT
Good news:

- Running code; cost reductions of $10^{20}$ vs. theory
- Compilers from C to verifiable computations
- Concretely efficient verifiers

Equivocal news:

- Small computations, extreme expense, etc.
- Useful only for special-purpose applications

So, lots of work left … and high payoff:
this is a good opportunity for you!

Below is a list of published implementations of probabilistic proofs. See [WB15] for a partial survey.

SBW11  CMT12  SMBW12  TRMP12  SVPBBW12  SBVBPW13  VSBW13  PGHR13  Thaler13  BCGTV13  BFRSBW13  BFR13  DFKP13  BCTV14a  BCTV14b  BCGGMTV14  FL14  KPPSST14  FGP14  WSRHBW15  BBFR15  CFHKKNPZ15  CTV15  KZMQCPPs15  WHGSW15
Rest of this session

(1) Landscape, history, and synopsis of the area

(2) Syllabus (for the 10 classes on verifiable computation)

(3) Technical preliminaries
Landscape: 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
A brief history of verifiable computation via probabilistic proofs

- Interactive Proofs [GMR85], Arthur-Merlin [Babai85]
- PCPs [BFLS91]

“In this setup, a single reliable PC can monitor the operation of a herd of supercomputers working with possibly extremely powerful but unreliable software and untested hardware.”

A brief history of verifiable computation via probabilistic proofs

- Interactive Proofs [GMR85], Arthur-Merlin [Babai85]
- PCPs [BFLS91] ("a single reliable PC can monitor…")
- PCP theorem [ALMSS92, AS92]
- Efficient arguments [Kilian92]
- CS proofs [Micali94]

"we aim at obtaining certificates ensuring that no error has occurred in a given execution of a given algorithm on a given input...This question is quite crucial whenever we are confident in the design of a given algorithm ... but less so in the physical computer that runs it."

—Micali, Computationally Sound Proofs, 2000
A brief history of verifiable computation via probabilistic proofs

- Interactive Proofs [GMR85], Arthur-Merlin [Babai85]
- PCPs [BFLS91] (“a single reliable PC can monitor…”)
- PCP theorem [ALMSS92, AS92]
- Efficient arguments [Kilian92]
- CS proofs [Micali94] (“certified computation”)

- Interactive proof with polynomial prover [GKR08]
- Efficient argument with simple PCP [IKO07]
- Non-interactive verifiable computation [GGP10] (coins “VC”)

- Challenges to the view that “this is theory-only” (2011—) [WB15]
- Theoretical innovation ongoing: SNARG/SNARK [GW11, Groth10, Lipmaa12, GGPR12, BCCT13, BCCGLRT14], 2-msg delegation [KRR14], …
Synopsis of the research area

front-end
(program translator)

C program

arithmetic circuit

main()
{
...
}

back-end
(probabilistic proof protocol)

prover

verifier

x

y, proof
• 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)

verifier

x

y, proof

prover

interactive proof

interactive argument

non-interactive argument

(CS proof, SNARG, SNARK)
main()
{
  ...
}

C program  arithmetic circuit

front-end
(program translator)

back-end
(probabilistic proof protocol)

verifier

x

prover

y, proof
front-end
(program translator)

C program → arithmetic circuit

main()
{
...

}

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

• 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?

“ASIC”
“CPU”
A key trade-off is performance versus expressiveness.
A key trade-off is performance versus expressiveness

<table>
<thead>
<tr>
<th>concrete costs</th>
<th>applicable computations</th>
</tr>
</thead>
<tbody>
<tr>
<td>lowest</td>
<td>“regular” straight line pure stateful general loops function pointers</td>
</tr>
<tr>
<td>lowest</td>
<td>CMT++ Thaler13</td>
</tr>
<tr>
<td>lowest</td>
<td>CMT CMT12</td>
</tr>
<tr>
<td>lowest</td>
<td>Allspice VSBW13</td>
</tr>
<tr>
<td>lowest</td>
<td>Pepper SMBW12</td>
</tr>
<tr>
<td>lowest</td>
<td>Ginger SVPBBW12</td>
</tr>
<tr>
<td>lowest</td>
<td>Zaatar SBVBPW13</td>
</tr>
<tr>
<td>lowest</td>
<td>Pinocchio PGHR13</td>
</tr>
<tr>
<td>lowest</td>
<td>Geppetto CFH..PZ15</td>
</tr>
<tr>
<td>lowest</td>
<td>Pantry BFRSBW13</td>
</tr>
<tr>
<td>lowest</td>
<td>Buffet WSRBW15</td>
</tr>
<tr>
<td>highest</td>
<td>ASIC</td>
</tr>
<tr>
<td>highest</td>
<td>BCTV BCTV14b</td>
</tr>
<tr>
<td>highest</td>
<td>BCGTV BCGTV13</td>
</tr>
<tr>
<td>highest</td>
<td>Proof-carrying data &amp; bootstrapping BCTV14a, CTV15</td>
</tr>
</tbody>
</table>

[Your work here!]
The area is interdisciplinary:

- We care about interesting theory and concrete costs
- The area blends crypto, complexity theory, PL, systems
Lots of open problems and questions

- Unconditionally secure delegation for all of PSPACE (YTK $100)
- 2-msg delegation for \( \mathcal{NP} \) with standard assumptions (YTK)
- Publicly-verif. 2-msg delegation for \( \mathcal{P} \) with std. assumptions (YTK)
- Zero knowledge with standard assumptions that is inexpensive in practice
- More efficient reductions from programs to circuits
- More efficient encodings of execution traces
- Probabilistic proof protocols that do not require circuits
- Avoiding preprocessing/amortization in a way that is inexpensive in practice
- Special-purpose algorithms for outsourcing pieces of computations, which integrate with circuit verification
(1) Landscape, history, and synopsis of the area

(2) Syllabus (for the 10 classes on verifiable computation)

(3) Technical preliminaries
Our goal: motivate and equip you to do research in this area

How:

• Teach you some of the building blocks
• Expose you to the key results
• Provide you with pointers into the literature
Class 2: **Statistically sound delegation** (YTK)
- History
- Sum-check protocol, low-degree extensions
- Unconditionally secure delegation for low depth circuits

Classes 3 and 4: **Computationally sound delegation** (YTK)
- History of arguments and CS proofs
- PCP + hash paradigm, Fiat-Shamir heuristic
- The space of assumptions
- 2-msg delegation for computations in $\mathcal{P}$ (std. assumptions) …
- … and for “long input” computations
Class 5: *Interactive arguments with preprocessing* (MW)
- Linear PCPs
- Interactive arguments via linear PCPs
- The role of QAPs

Class 6: *Non-interactive arguments with preprocessing* (ET)
- SNARGs and (zk-)SNARKs based on linear PCPs
- Details of QAPs
- Refinements of QAPs
Class 7: **Program representations** (MW)
- Arithmetization: from programs to circuits (“ASIC approach”)
- Data-dependent control flow
- Expressiveness versus amortization versus performance
Classes 8 and 9: **TBA** (ET). Possible topics include:

- Application of “ASIC approach”: Zerocash [BCGGMTV14]
- “CPU approach” to circuits: TinyRAM [BCGTV13, BCTV14b]
- Permutation networks for RAM computations [BCGT13, BCGTV13, BCTV14b]
- Bootstrapping SNARKs [BCCT13] by composing QAPs, TinyRAM, and elliptic curve cycles [BCTV14a, CTV15]
- SNARKs without preprocessing, using short PCPs [BCCT12, BCCGLRT14]
- Progress in ongoing work implementing short PCPs
Class 10: Additional applications and summary (MW)

- External state
- MapReduce, face-matching, regression analysis, etc.
- Implementations of IPs (time permitting)
- Wrap-up
Classes at a glance (*numbers in blue* refer to class number)

**front-end**
(program translator)

- **main()**
- ...  
- ...

**back-end**
(probabilistic proof protocol)

<table>
<thead>
<tr>
<th>interactive proofs (IPs)</th>
<th>interactive args.</th>
<th>non-interactive args.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>8, 9</td>
</tr>
<tr>
<td>3, 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8, 9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| no | preproc | 10 | (QAPs) | 5 | (QAPs) | 6 |

* Indicates that the mechanism has been implemented

- 8, 9 bootstrapping (recursive use of the machinery)

- “CPU”
- “ASIC”
- (QAPs)
- * Indicates that the mechanism has been implemented
(1) Landscape, history, and synopsis of the area

(2) Syllabus (for the 10 classes on verifiable computation)

(3) Technical preliminaries
How are probabilistic proofs defined?

Completeness:

Soundness:

Efficiency:

Variants: computational soundness, non-deterministic languages, proof of knowledge, zero knowledge.
How are probabilistic proofs defined?

There are many definitions and variants; below is the general form. For details, consult a text ([AB09, Goldreich07, Micali94, BG02] are all extremely lucid). 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:**

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

The probability is taken over $V$’s random choices.

**Soundness:**

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

The probability is again over $V$’s random choices.

**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.
What language should we use for “correct program execution”?

- Boolean circuit satisfiability
- Arithmetic circuit satisfiability
- Non-deterministic (Boolean, arith.) circuit satisfiability

“Satisfiability” enters because there are implicit constraints. Sometimes it is easier to work with constraints explicitly.
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 \( \mathbb{F} \).

- A computation \( f \) is equivalent to constraints \( C \) if:

\[
\text{increment-by-one}
\begin{align*}
f(X) & \{ \\
& \quad Y = X + 1; \\
& \quad \text{return } Y;
\}
\end{align*}
\]
A convenient language: arithmetic constraint satisfiability

- System of equations in finite field $\mathbb{F}$.

- A computation $f$ is **equivalent** to constraints $C$ if:
  
  C is constraints over variables $(X, Y, Z)$ and field $\mathbb{F}$ s.t.
  (det case) $\forall x,y: y=f(x) \iff C(X=x,Y=y)$ is satisfiable
  (non-det. case) $\forall x,y: (\exists w \text{ s.t. } y=f(x,w)) \iff C(X=x,Y=y)$ is satisfiable

  Terminology: constraints $C$ are said to be an **arithmetization** of the computation $f$. 

increment-by-one

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

[equivalent] $\begin{cases} 
0 = Z - X, \\
0 = Z - Y + 1 
\end{cases}$
QuadConstraint\_\mathbb{F}

- Degree-2 constraints over finite field $\mathbb{F}$

- What do the constraints/gates below represent?

$$Y = (X_1 - X_2)Z$$

$$0 = (1 - Y)(X_1 - Z)(X_1 - 1)$$
Summary

- This is an exciting inter-disciplinary area:
  - Addresses a fundamental problem, using deep theory
  - There is still lots of work to be done …
  - …. but the potential is large (goes far beyond the cloud!)

- Central technical notions:
  - probabilistic proofs, circuits (constraints), program translators

- Many tradeoffs, properties, axes, facets
  - Ideally, we will help you understand them
  - Ideally, you will help improve them!
Acknowledgment: my collaborators in this research area

Andrew J. Blumberg (UT), Benjamin Braun (now at Stanford), Ariel Feldman (UChicago), Siddharth Garg (NYU), Max Howald (Cooper Union, NYU), Richard McPherson, Nikhil Panpalia (now at Amazon), Bryan Parno (MSR), Zuocheng Ren, Srinath Setty (now at MSR), abhi shelat (UVA), Justin Thaler (Yahoo Research), Victor Vu (now at Sandia), and Riad Wahby (now at Stanford).

(http://www.pepper-project.org)
References


