A Survey of Verifiable Delegation of Computations

Rosario Gennaro

The City College of New York
rosario@cs.ccny.cuny.edu

CANS 2013, Paraty, Brasil
November 22, 2013
**Motivation**

Cloud computing, Small Devices, Large Scale Computation

**Generic Results for Verifiable Computation**

- Protocols that work for arbitrary computations
  - Interactive Proofs
  - Probabilistically Checkable Proofs
  - "Muggles" Proofs
  - Other Arithmetizations approaches (QSP)
  - Implementations (Pinocchio, Snark-for-C)

**Delegation of Memory**

- Homomorphic MACs
- Proofs of Retrievability
- Verifiable Keyword Search
# Talk Outline

## Motivation
Cloud computing, Small Devices, Large Scale Computation

## Generic Results for Verifiable Computation
Protocols that work for arbitrary computations
- Interactive Proofs
- Probabilistically Checkable Proofs
- "Muggles" Proofs
- Other Arithmetizations approaches (QSP)
- Implementations (Pinocchio, Snark-for-C)

## Delegation of Memory
- Homomorphic MACs
- Proofs of Retrievability
- Verifiable Keyword Search
<table>
<thead>
<tr>
<th>Motivation</th>
<th>Cloud computing, Small Devices, Large Scale Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic Results for Verifiable Computation</td>
<td>Protocols that work for arbitrary computations</td>
</tr>
<tr>
<td></td>
<td>• Interactive Proofs</td>
</tr>
<tr>
<td></td>
<td>• Probabilistically Checkable Proofs</td>
</tr>
<tr>
<td></td>
<td>• &quot;Muggles&quot; Proofs</td>
</tr>
<tr>
<td></td>
<td>• Other Arithmetizations approaches (QSP)</td>
</tr>
<tr>
<td></td>
<td>• Implementations (Pinocchio, Snark-for-C)</td>
</tr>
<tr>
<td>Delegation of Memory</td>
<td>• Homomorphic MACs</td>
</tr>
<tr>
<td></td>
<td>• Proofs of Retrievability</td>
</tr>
<tr>
<td></td>
<td>• Verifiable Keyword Search</td>
</tr>
</tbody>
</table>
Computing on Demand

Cloud Computing

- Businesses buy computing power from a service provider

Advantages

- No need to provision and maintain hardware
- Pay for what you need
- Easily and quickly scalable up or down

Trust Issues

- Transfer possibly confidential data to computing service provider
- Trust computation is performed correctly without errors
- Malicious or benign
Computing on Demand

Cloud Computing
- Businesses buy computing power from a service provider

Advantages
- No need to provision and maintain hardware
- Pay for what you need
- Easily and quickly scalable up or down

Trust Issues
- Transfer possibly confidential data to computing service provider
- Trust computation is performed correctly without errors
- Malicious or benign
Computing on Demand

Cloud Computing

- Businesses buy computing power from a service provider

Advantages

- No need to provision and maintain hardware
- Pay for what you need
- Easily and quickly scalable up or down

Trust Issues

- Transfer possibly confidential data to computing service provider
- Trust computation is performed correctly without errors
- Malicious or benign
Small Devices

- Small devices outsourcing complex computing problems to larger servers
  - Photo manipulations
  - Cryptographic operations

- Same issues:
  - Confidentiality of data
  - Correctness of result
Small Devices

- Small devices outsourcing complex computing problems to larger servers
  - Photo manipulations
  - Cryptographic operations

- Same issues:
  - Confidentiality of data
  - Correctness of result
Large Scale Computations

- Network-based computations
  - SETI@Home
  - Folding@Home

- Users donate idle cycles
  - Known problem: users return fake results without performing the computation
  - Increases their ranking

- Needed a way to efficiently weed out bad results
  - Currently use redundancy
Large Scale Computations

- Network-based computations
  - SETI@Home
  - Folding@Home

- Users donate idle cycles
  - Known problem: users return fake results without performing the computation
  - Increases their ranking

- Needed a way to efficiently weed out bad results
  - Currently use redundancy
Large Scale Computations

- Network-based computations
  - SETI@Home
  - Folding@Home

- Users donate idle cycles
  - Known problem: users return fake results without performing the computation
  - Increases their ranking

- Needed a way to efficiently weed out bad results
  - Currently use redundancy
Verifiable Computation

- The client sends a function $F$ and an input $x$ to the server

- The server returns $y = F(x)$ and a proof $\Pi$ that $y$ is correct. Verifying $\Pi$ should take less time than computing $F$. 
Verifiable Computation

- The client sends a function $F$ and an input $x$ to the server.
- The server returns $y = F(x)$ and a proof $\Pi$ that $y$ is correct. Verifying $\Pi$ should take less time than computing $F$. 
Interactive Proofs (GMR,B)

- An all powerful Prover interacts with a poly-time Verifier
  - Prover convinces Verifier of a statement she cannot decide on her own
  - Probabilist guarantee
  - All of PSPACE can be proven this way [LFKN,S]

- We want something different
  - A scaled back version of this protocols for efficient computations
  - A powerful but still efficient prover: its complexity should be as close as possible to the original computation
  - A super-efficient Verifier: ideally linear time
Interactive Proofs (GMR,B)

- An all powerful Prover interacts with a poly-time Verifier
  - Prover convinces Verifier of a statement she cannot decide on her own
  - Probabilist guarantee
  - All of \text{PSPACE} can be proven this way [LFKN,S]

- We want something different
  - A scaled back version of this protocol for efficient computations
  - A powerful but still efficient prover: its complexity should be as close as possible to the original computation
  - A super-efficient Verifier: ideally linear time
Muggles Proofs (GKR)

- Poly-time Prover interacts with a quasi-linear Verifier
  - Refines the proof that $\text{IP} = \text{PSPACE}$ to efficient computations

- For a log-space uniform NC circuit of depth $d$
  - Prover runs in $\text{poly}(n)$
  - Verifier runs in $O(n + \text{poly}(d))$
  - Interactive ($O(d \cdot \log n)$ rounds)
  - Unconditional Soundness
Muggles Proofs (GKR)

- Poly-time Prover interacts with a quasi-linear Verifier
  - Refines the proof that IP=PSPACE to efficient computations

- For a log-space uniform NC circuit of depth $d$
  - Prover runs in $\text{poly}(n)$
  - Verifier runs in $O(n + \text{poly}(d))$
  - Interactive ($O(d \cdot \log n)$ rounds)
  - Unconditional Soundness
Optimizations and Implementations (CMT,T)

- **Prover** can be implemented in $O(S \log S)$
  - Where $S$ is the size of the circuit computing the function
  - $O(S)$ for circuits with a regular wiring pattern

- Implementation tests show that for the regular wiring pattern case the prover is less than 10x slower than simply computing the function.

- Protocol remains highly interactive
  - Interaction can be removed via the Fiat-Shamir heuristic (random oracle model).
Optimizations and Implementations (CMT,T)

- Prover can be implemented in $O(S \log S)$
  - Where $S$ is the size of the circuit computing the function
  - $O(S)$ for circuits with a regular wiring pattern

- Implementation tests show that for the regular wiring pattern case the prover is less than 10x slower than simply computing the function.

- Protocol remains highly interactive
  - Interaction can be removed via the Fiat-Shamir heuristic (random oracle model).
Optimizations and Implementations (CMT,T)

- Prover can be implemented in $O(S \log S)$
  - Where $S$ is the size of the circuit computing the function
  - $O(S)$ for circuits with a regular wiring pattern

- Implementation tests show that for the regular wiring pattern case the prover is less than 10x slower than simply computing the function.

- Protocol remains highly interactive
  - Interaction can be removed via the Fiat-Shamir heuristic (random oracle model).
The IP=PSPACE result yielded a surprising consequence: any computation can be associated with a (very long) proof which can be queried in only a constant number of locations (...AMLSS, AS, ...)

The Prover commits to this proof using a Merkle tree and then the Verifier queries it and verifies the openings (K)

- Note that now we have an argument with a computational soundness guarantee

This protocol can also be made non-interactive using the random oracle (M) or strong extractability assumptions about the hash function used in the protocol (DL,BCCT,GLR)

- Main bottleneck: still the Prover’s complexity $O(S^{1.5})$
The IP=PSPACE result yielded a surprising consequence: any computation can be associated with a (very long) proof which can be queried in only a constant number of locations (...AMLSS, AS, ...)

The Prover commits to this proof using a Merkle tree and then the Verifier queries it and verifies the openings (K)

Note that now we have an argument with a computational soundness guarantee

This protocol can also be made non-interactive using the random oracle (M) or strong extractability assumptions about the hash function used in the protocol (DL,BCCT,GLR)

Main bottleneck: still the Prover’s complexity $O(S^{1.5})$
The IP=PSPACE result yielded a surprising consequence: any computation can be associated with a (very long) proof which can be queried in only a constant number of locations (...AMLSS, AS, ...)

The Prover commits to this proof using a Merkle tree and then the Verifier queries it and verifies the openings (K)

- Note that now we have an argument with a computational soundness guarantee

This protocol can also be made non-interactive using the random oracle (M) or strong extractability assumptions about the hash function used in the protocol (DL,BCCT,GLR)

- Main bottleneck: still the Prover’s complexity $O(S^{1.5})$
The IP=PSPACE result yielded a surprising consequence: any computation can be associated with a (very long) proof which can be queried in only a constant number of locations (...AMLSS, AS, ...)

The Prover commits to this proof using a Merkle tree and then the Verifier queries it and verifies the openings (K)

- Note that now we have an argument with a computational soundness guarantee

This protocol can also be made non-interactive using the random oracle (M) or strong extractability assumptions about the hash function used in the protocol (DL,BCCT,GLR)

Main bottleneck: still the Prover’s complexity $O(S^{1.5})$
Arithmetization

- Turn a circuit computation into a set of polynomial equations
  - Replace each gate with a quadratic polynomial
  - Check these polynomial identities in a randomized fashion by checking them on random points
  - Use error-correcting encodings to make sure that the proof is *locally checkable* (i.e. to reduce the number of random queries to the proof)

- Can we use different arithmetizations?
  - Avoid composing long PCP proofs with compressing hash functions for a more direct way to get short proofs
  - Linear Prover complexity?

- Groth showed a different approach
  - Polynomial equations are verified in the exponent (using bilinear maps over a cyclic group)
  - A Diffie-Hellman type of assumption prevents the Prover from cheating
  - Proof is very compact without using Merkle trees
  - Drawback: quadratic prover complexity and a quadratic CRS
  - Lipmaa shows how to reduce those to quasilinear
Arithmetization

- Turn a circuit computation into a set of polynomial equations
  - Replace each gate with a quadratic polynomial
  - Check these polynomial identities in a randomized fashion by checking them on random points
  - Use error-correcting encodings to make sure that the proof is *locally checkable* (i.e. to reduce the number of random queries to the proof)

- Can we use different arithmetizations?
  - Avoid composing long PCP proofs with compressing hash functions for a more direct way to get short proofs
  - Linear Prover complexity?

- Groth showed a different approach
  - Polynomial equations are verified in the exponent (using bilinear maps over a cyclic group)
  - A Diffie-Hellman type of assumption prevents the Prover from cheating
  - Proof is very compact without using Merkle trees
  - Drawback: quadratic prover complexity and a quadratic CRS
  - Lipmaa shows how to reduce those to quasilinear
Arithmetization

- Turn a circuit computation into a set of polynomial equations
  - Replace each gate with a quadratic polynomial
  - Check these polynomial identities in a randomized fashion by checking them on random points
  - Use error-correcting encodings to make sure that the proof is *locally checkable* (i.e. to reduce the number of random queries to the proof)

- Can we use different arithmetizations?
  - Avoid composing long PCP proofs with compressing hash functions for a more direct way to get short proofs
  - Linear Prover complexity?

- Groth showed a different approach
  - Polynomial equations are verified in the exponent (using bilinear maps over a cyclic group)
  - A Diffie-Hellman type of assumption prevents the Prover from cheating
  - Proof is very compact without using Merkle trees
  - Drawback: quadratic prover complexity and a quadratic CRS
  - Lipmaa shows how to reduce those to quasilinear
Quadratic Span Programs (GGPR)

QSPs add a single quadratic step to the computation, instead of checking several quadratic equations (one for each gate)

- To check that all the wires in the circuits are correct it just requires a linear test (*span program*)
- This would be too much work for the verifier (same as the size of the circuit)
- Build two copies of the "checking" span program and test them against each other
- A QSP is defined by two sets of polynomials \( V = \{ v_1, \ldots, v_{n+m} \} \), \( W = \{ w_1, \ldots, w_{n+m} \} \) and a target polynomial \( t \)
  - We say that a QSP \((V, W, t)\) computes a Boolean function \( F \) of \( n \) inputs if and only if
  - For all \( x = (x_1, \ldots, x_n) \) s.t. \( F(x) = 1 \)
  - \( t \) divides the product of a linear combination of subsets of \( V \) and \( W \)
    - \( t | \left( \sum_{i=1}^{n} a_i v_i \right) \cdot \left( \sum_{i=1}^{n} b_i w_i \right) \)
    - where \( a_i = b_i = 0 \) if \( x_i = 0 \)
Quadratic Span Programs (GGPR)

QSPs add a single quadratic step to the computation, instead of checking several quadratic equations (one for each gate)

- To check that all the wires in the circuits are correct it just requires a linear test (*span program*)
- This would be too much work for the verifier (same as the size of the circuit)
- Build two copies of the "checking" span program and test them against each other
- A QSP is defined by two sets of polynomials $V = \{v_1, \ldots, v_{n+m}\}$, $W = \{w_1, \ldots, w_{n+m}\}$ and a target polynomial $t$
  - We say that a QSP $(V, W, t)$ computes a Boolean function $F$ of $n$ inputs if and only if
  - For all $x = (x_1 \ldots x_n)$ s.t. $F(x) = 1$
  - $t$ divides the product of a linear combination of subsets of $V$ and $W$
    - $t | (\sum_{i=1}^{n} a_i v_i) \cdot (\sum_{i=1}^{n} b_i w_i)$
    - where $a_i = b_i = 0$ iff $x_i = 0$
Quadratic Span Programs (GGPR)

QSPs add a single quadratic step to the computation, instead of checking several quadratic equations (one for each gate)

- To check that all the wires in the circuits are correct it just requires a linear test *(span program)*
- This would be too much work for the verifier (same as the size of the circuit)
- Build two copies of the "checking" span program and test them against each other
- A QSP is defined by two sets of polynomials \( V = \{v_1, \ldots, v_{n+m}\} \), \( W = \{w_1, \ldots, w_{n+m}\} \) and a target polynomial \( t \)
  - We say that a QSP \((V, W, t)\) computes a Boolean function \( F \) of \( n \) inputs if and only if
  - For all \( x = (x_1 \ldots x_n) \) s.t. \( F(x) = 1 \)
  - \( t \) divides the product of a linear combination of subsets of \( V \) and \( W \)
    - \( t | (\sum_{i=1}^{n} a_i v_i) \cdot (\sum_{i=1}^{n} b_i w_i) \)
    - where \( a_i = b_i = 0 \) iff \( x_i = 0 \)
Quadratic Span Programs (GGPR)

QSPs add a single quadratic step to the computation, instead of checking several quadratic equations (one for each gate)

- To check that all the wires in the circuits are correct it just requires a linear test (*span program*)
- This would be too much work for the verifier (same as the size of the circuit)
- Build two copies of the "checking" span program and test them against each other
- A QSP is defined by two sets of polynomials $V = \{v_1, \ldots, v_{n+m}\}$, $W = \{w_1, \ldots, w_{n+m}\}$ and a target polynomial $t$
  - We say that a QSP $(V, W, t)$ computes a Boolean function $F$ of $n$ inputs if and only if
  - For all $x = (x_1 \ldots x_n)$ s.t. $F(x) = 1$
  - $t$ divides the product of a linear combination of subsets of $V$ and $W$
    - $t|(\Sigma_{i=1}^n a_i v_i) \cdot (\Sigma_{i=1}^n b_i w_i)$
    - where $a_i = b_i = 0$ iff $x_i = 0$
Quadratic Span Programs (GGPR)

QSPs add a single quadratic step to the computation, instead of checking several quadratic equations (one for each gate)

- To check that all the wires in the circuits are correct it just requires a linear test (*span program*)
- This would be too much work for the verifier (same as the size of the circuit)
- Build two copies of the "checking" span program and test them against each other
- A QSP is defined by two sets of polynomials $V = \{v_1, \ldots, v_{n+m}\}$, $W = \{w_1, \ldots, w_{n+m}\}$ and a target polynomial $t$
  - We say that a QSP $(V, W, t)$ computes a Boolean function $F$ of $n$ inputs if and only if
  - For all $x = (x_1 \ldots x_n)$ s.t. $F(x) = 1$
  - $t$ divides the product of a linear combination of subsets of $V$ and $W$
    - $t | (\sum_{i=1}^{n} a_i v_i) \cdot (\sum_{i=1}^{n} b_i w_i)$
    - where $a_i = b_i = 0$ iff $x_i = 0$
In a preprocessing stage the Verifier publishes the values $g^{s_i}, g^{v_i(s)}, g^{w_i(s)}$ and $g^{t(s)}$ for a secret random value $s$.

On input $x$ the server finds the coefficients $a_i, b_i$ and polynomial $h$ such that

$$t \cdot h = (\sum_{i=1}^{n} a_i v_i) \cdot (\sum_{i=1}^{n} b_i w_i)$$

Using the values produced by the Verifier the Prover can evaluate in the exponent the above equation at the point $s$.

Verifier checks the equation using bilinear maps.

Efficiency:

- The verifier is linear to prepare the input; constant time to verify the result.
- Prover is quasi-linear - the polylog overhead comes from doing polynomial division to compute $h$.

Security: requires a Diffie-Hellman type of assumption which assumes that the prover cannot divide in the exponent.
The QSP protocol

- In a preprocessing stage the Verifier publishes the values $g^{s_i}$, $g^{v_i(s)}$, $g^{w_i(s)}$ and $g^t(s)$
  - for a secret random value $s$.
- On input $x$ the server finds the coefficients $a_i$, $b_i$ and polynomial $h$ such that
  - $t \cdot h = (\sum_{i=1}^{n} a_i v_i) \cdot (\sum_{i=1}^{n} b_i w_i)$
- Using the values produced by the Verifier the Prover can evaluate in the exponent the above equation at the point $s$
  - Verifier checks the equation using bilinear maps

**Efficiency:**
- The verifier is linear to prepare the input; constant time to verify the result
- Prover is quasi-linear - the polylog overhead comes from doing polynomial division to compute $h$

**Security:** requires a Diffie-Hellman type of assumption which assumes that the prover cannot divide in the exponent.
The QSP protocol

- In a preprocessing stage the Verifier publishes the values $g^{s^i}$, $g^{v_i(s)}$, $g^{w_i(s)}$ and $g^t(s)$
  - for a secret random value $s$.
- On input $x$ the server finds the coefficients $a_i$, $b_i$ and polynomial $h$ such that
  - $t \cdot h = (\sum_{i=1}^{n} a_i v_i) \cdot (\sum_{i=1}^{n} b_i w_i)$
- Using the values produced by the Verifier the Prover can evaluate \textit{in the exponent} the above equation at the point $s$
  - Verifier checks the equation using bilinear maps

**Efficiency:**

- The verifier is linear to prepare the input; constant time to verify the result
- Prover is \textit{quasi-linear} - the polylog overhead comes from doing polynomial division to compute $h$

**Security:** requires a Diffie-Hellman type of assumption which assumes that the prover cannot divide in the exponent.
The QSP protocol

- In a preprocessing stage the Verifier publishes the values $g^{s^i}$, $g^{v_i(s)}$, $g^{w_i(s)}$ and $g^{t(s)}$
  - for a secret random value $s$.
- On input $x$ the server finds the coefficients $a_i$, $b_i$ and polynomial $h$ such that
  - $t \cdot h = (\sum_{i=1}^{n} a_i v_i) \cdot (\sum_{i=1}^{n} b_i w_i)$
- Using the values produced by the Verifier the Prover can evaluate *in the exponent* the above equation at the point $s$
  - Verifier checks the equation using bilinear maps

**Efficiency:**
- The verifier is linear to prepare the input; constant time to verify the result
- Prover is *quasi-linear* - the polylog overhead comes from doing polynomial division to compute $h$

**Security:** requires a Diffie-Hellman type of assumption which assumes that the prover cannot divide in the exponent.
The QSP protocol

- In a preprocessing stage the Verifier publishes the values $g^{s_i}$, $g^{v_i(s)}$, $g^{w_i(s)}$ and $g^{t(s)}$
  - for a secret random value $s$.
- On input $x$ the server finds the coefficients $a_i$, $b_i$ and polynomial $h$ such that
  - $t \cdot h = (\sum_{i=1}^{n} a_i v_i) \cdot (\sum_{i=1}^{n} b_i w_i)$
- Using the values produced by the Verifier the Prover can evaluate in the exponent the above equation at the point $s$
  - Verifier checks the equation using bilinear maps

**Efficiency:**
- The verifier is linear to prepare the input; constant time to verify the result
- Prover is quasi-linear - the polylog overhead comes from doing polynomial division to compute $h$

**Security:** requires a Diffie-Hellman type of assumption which assumes that the prover cannot divide in the exponent.
Pinocchio (PGHR)

- An end-to-end toolchain that compiles a subset of C into QSPs
- Proof size is 288 bytes regardless of what it is being computed
- Verification time is 10ms
- Prover complexity still not quite there in practice
  - About 60 times faster than previous proposals
  - Can run some lightweight computations

SNARKs-for-C (BCGTV)
Implementation Results

Pinocchio (PGHR)
- An end-to-end toolchain that compiles a subset of C into QSPs
- Proof size is 288 bytes regardless of what it is being computed
  - Verification time is 10ms
  - Prover complexity still not quite there in practice
    - About 60 times faster than previous proposals
    - Can run some lightweight computations

SNARKs-for-C (BCGTV)
Implementation Results

Pinocchio (PGHR)
- An end-to-end toolchain that compiles a subset of C into QSPs
- Proof size is 288 bytes regardless of what it is being computed
- Verification time is 10ms
- Prover complexity still not quite there in practice
  - About 60 times faster than previous proposals
  - Can run some lightweight computations

SNARKs-for-C (BCGTV)
- Given a C program, they produce a circuit whose satisfiability encodes the correctness of execution of the program.
  - First the C program is compiled into machine code for TinyRAM
  - Then the TinyRAM code is compiled into a circuit
Implementation Results

Pinocchio (PGHR)
- An end-to-end toolchain that compiles a subset of C into QSPs
- Proof size is 288 bytes regardless of what it is being computed
- Verification time is 10ms
- Prover complexity still not quite there in practice
  - About 60 times faster than previous proposals
  - Can run some lightweight computations

SNARKs-for-C (BCGTV)
- Given a C program, they produce a circuit whose satisfiability encodes the correctness of execution of the program.
  - First the C program is compiled into machine code for TinyRAM
  - Then the TinyRam code is compiled into a circuit
- A QSP is built for this circuit
  - Use the generic concept of Linear Interactive Proof
  - Could plug a more efficient LIP if one is found
- Slightly less efficient for the Verifier
  - Proof size: 322 bytes
  - Verification time dependent on x (from 103ms to 5s for long inputs)
- A bit more efficient for the Prover
- Were able to handle a Traveling Salesman Decider on a 200-nodes graph
  - Still it took almost 3 hours...
Implementation Results

Pinocchio (PGHR)
- An end-to-end toolchain that compiles a subset of C into QSPs
- Proof size is 288 bytes regardless of what it is being computed
- Verification time is 10ms
- Prover complexity still not quite there in practice
  - About 60 times faster than previous proposals
  - Can run some lightweight computations

SNARKs-for-C (BCGTV)
- Given a C program, they produce a circuit whose satisfiability encodes the correctness of execution of the program.
  - First the C program is compiled into machine code for TinyRAM
  - Then the TinyRam code is compiled into a circuit
- A QSP is built for this circuit
- Use the generic concept of Linear Interactive Proof
- Could plug a more efficient LIP if one is found
- Slightly less efficient for the Verifier
  - Proof size 322 bytes
  - Verification time dependent on $x$ (from 103ms to 5s for long inputs)
- A bit more efficient for the Prover
- Were able to handle a Traveling Salesman Decider on a 200-nodes
Implementation Results

Pinocchio (PGHR)
- An end-to-end toolchain that compiles a subset of C into QSPs
- Proof size is 288 bytes regardless of what it is being computed
- Verification time is 10ms
- Prover complexity still not quite there in practice
  - About 60 times faster than previous proposals
  - Can run some lightweight computations

SNARKs-for-C (BCGTV)
- Given a C program, they produce a circuit whose satisfiability encodes the correctness of execution of the program.
  - First the C program is compiled into machine code for TinyRAM
  - Then the TinyRam code is compiled into a circuit
- A QSP is built for this circuit
  - Use the generic concept of *Linear Interactive Proof*
  - Could plug a more efficient LIP if one is found
- Slightly less efficient for the Verifier
  - Proof size 322 bytes
  - Verification time dependent on $x$ (from 103ms to 5s for long inputs)
- A bit more efficient for the Prover
  - Were able to handle a Traveling Salesman Decider on a 200-nodes graph
  - Still it took almost 3 hours...
Implementation Results

**Pinocchio (PGHR)**
- An end-to-end toolchain that compiles a subset of C into QSPs
- Proof size is 288 bytes regardless of what it is being computed
- Verification time is 10ms
- Prover complexity still not quite there in practice
  - About 60 times faster than previous proposals
  - Can run some lightweight computations

**SNARKs-for-C (BCGTV)**
- Given a C program, they produce a circuit whose satisfiability encodes the correctness of execution of the program.
  - First the C program is compiled into machine code for TinyRAM
  - Then the TinyRam code is compiled into a circuit
- A QSP is built for this circuit
  - Use the generic concept of *Linear Interactive Proof*
  - could plug a more efficient LIP if one is found
- Slightly less efficient for the Verifier
  - Proof size 322 bytes
  - Verification time dependent on $x$ (from 103ms to 5s for long inputs)
- A bit more efficient for the Prover
  - Were able to handle a Traveling Salesman Decider on a 200-nodes graph
  - Still it took almost 3 hours ...
Implementation Results

Pinocchio (PGHR)

- An end-to-end toolchain that compiles a subset of C into QSPs
- Proof size is 288 bytes regardless of what it is being computed
- Verification time is 10ms
- Prover complexity still not quite there in practice
  - About 60 times faster than previous proposals
  - Can run some lightweight computations

SNARKs-for-C (BCGTV)

- Given a C program, they produce a circuit whose satisfiability encodes the correctness of execution of the program.
  - First the C program is compiled into machine code for TinyRAM
  - Then the TinyRam code is compiled into a circuit
- A QSP is built for this circuit
  - Use the generic concept of *Linear Interactive Proof*
  - could plug a more efficient LIP if one is found
- Slightly less efficient for the Verifier
  - Proof size 322 bytes
  - Verification time dependent on $x$ (from 103ms to 5s for long inputs)

- A bit more efficient for the Prover
  - Were able to handle a Traveling Salesman Decider on a 200-nodes graph
  - Still it took almost 3 hours...
Implementation Results

Pinocchio (PGHR)
- An end-to-end toolchain that compiles a subset of C into QSPs
- Proof size is 288 bytes regardless of what it is being computed
- Verification time is 10ms
- Prover complexity still not quite there in practice
  - About 60 times faster than previous proposals
  - Can run some lightweight computations

SNARKs-for-C (BCGTV)
- Given a C program, they produce a circuit whose satisfiability encodes the correctness of execution of the program.
  - First the C program is compiled into machine code for TinyRAM
  - Then the TinyRam code is compiled into a circuit
- A QSP is built for this circuit
  - Use the generic concept of Linear Interactive Proof
  - could plug a more efficient LIP if one is found
- Slightly less efficient for the Verifier
  - Proof size 322 bytes
  - Verification time dependent on \( x \) (from 103ms to 5s for long inputs)
- A bit more efficient for the Prover
  - Were able to handle a Traveling Salesman Decider on a 200-nodes graph
    - Still it took almost 3 hours ...
Outsourcing Your Data

- Up to now we have considered the case of a client sending $F$ and $x$ to the server
  - Client’s limitation is in computing time
  - Cannot compute $F$ on its own

- What if the client’s limitation is storage?
  - Client stores large quantity of data $D$ with the server
  - later queries $F$ on $D$ and receives back $F(D)$

- Previous approaches do not work: they require the client to know the input
Outsourcing Your Data

- Up to now we have considered the case of a client sending $F$ and $x$ to the server
  - Client’s limitation is in computing time
  - Cannot compute $F$ on its own

- What if the client’s limitation is storage?
  - Client stores large quantity of data $D$ with the server
  - later queries $F$ on $D$ and receives back $F(D)$

- Previous approaches do not work: they require the client to know the input
Outsourcing Your Data

- Up to now we have considered the case of a client sending $F$ and $x$ to the server
  - Client’s limitation is in computing time
  - Cannot compute $F$ on its own

- What if the client’s limitation is *storage*?
  - Client stores large quantity of data $D$ with the server
  - later queries $F$ on $D$ and receives back $F(D)$

- Previous approaches do not work: they require the client to know the input
Homomorphic Message Authenticators (GW)

- Client stores $D = D_1, \ldots, D_n$ and $t_i = MAC_k(D_i)$.
  - Client only stores the short key $k$

- Later the client submits $F$
  - Server returns $y = F(D)$ and $t$
  - Client accepts if and only if $t = MAC_k(y)$
  - Verification time may be as long as computing $F$ – focus on storage and bandwidth

- Original idea uses homomorphic encryption
  - Mostly of theoretical interest

- New ideas use "traditional" crypto (CF, GN)
  - Much more efficient
  - But only work for "shallow" circuits
**Homomorphic Message Authenticators (GW)**

- Client stores $D = D_1, \ldots, D_n$ and $t_i = MAC_k(D_i)$.
  - Client only stores the short key $k$.

- Later the client submits $F$.
  - Server returns $y = F(D)$ and $t$.
  - Client accepts if and only if $t = MAC_k(y)$.
  - Verification time may be as long as computing $F$ – focus on storage and bandwidth.

- Original idea uses homomorphic encryption.
  - Mostly of theoretical interest.

- New ideas use "traditional" crypto (CF, GN).
  - Much more efficient.
  - But only work for "shallow" circuits.
Homomorphic Message Authenticators (GW)

- Client stores $D = D_1, \ldots, D_n$ and $t_i = MAC_k(D_i)$.
  - Client only stores the short key $k$.

- Later the client submits $F$:
  - Server returns $y = F(D)$ and $t$;
  - Client accepts if and only if $t = MAC_k(y)$;
  - Verification time may be as long as computing $F$ – focus on storage and bandwidth.

- Original idea uses homomorphic encryption:
  - Mostly of theoretical interest.

- New ideas use "traditional" crypto (CF, GN):
  - Much more efficient;
  - But only work for "shallow" circuits.
Client stores $D = D_1, \ldots, D_n$ and $t_i = MAC_k(D_i)$.
- Client only stores the short key $k$

Later the client submits $F$
- Server returns $y = F(D)$ and $t$
- Client accepts if and only if $t = MAC_k(y)$
- Verification time may be as long as computing $F$ – focus on storage and bandwidth

Original idea uses homomorphic encryption
- Mostly of theoretical interest

New ideas use "traditional" crypto (CF,GN)
- Much more efficient
- But only work for "shallow" circuits
Proofs of Retrievability (JK)

- Client stores a large file $F$ with the server and wants to make sure that it can be retrieved without downloading the entire thing (e.g. auditing)
  - Client sends a short challenge $c$
  - Server responds with a short answer $a$
    - avoid reading the entire file to produce the answer

- A possible solution (A+, SW)
  - Encode the file $F$ using an error correcting code $F' = Encode(F)$
  - Store each block $F'_i$ with a linearly homomorphic MAC
    $t_i = MAC_k(F'_i)$
  - The client queries a small number ($\ell$) of the blocks $F_{i_1} \ldots F_{i_\ell}$ and also sends $\ell$ random coefficients $\lambda_1, \ldots, \lambda_\ell$
  - The server sends back $\phi = \Sigma_j \lambda_j F_{i_j}$ and $t = \Sigma_j \lambda_j t_j$
  - The client accepts if and only if $t = MAC_k(\phi)$

- The scheme is very efficient
  - Linearly homomorphic MACs can be built from basic universal hash functions
  - Minimal storage overhead due to the error-correction expansion
  - Query complexity is quadratic in the security parameter
Proofs of Retrievability (JK)

- Client stores a large file $F$ with the server and wants to make sure that it can be retrieved without downloading the entire thing (e.g. auditing)
  - Client sends a short *challenge* $c$
  - Server responds with a short *answer* $a$
    - avoid reading the entire file to produce the answer

- A possible solution (A+,SW)
  - Encode the file $F$ using an error correcting code $F' = \text{Encode}(F)$
  - Store each block $F'_i$ with a *linearly homomorphic MAC* $t_i = MAC_k(F'_i)$
  - The client queries a small number ($\ell$) of the blocks $F_{i_1} \ldots F_{i_\ell}$ and also sends $\ell$ random coefficients $\lambda_1, \ldots, \lambda_\ell$
  - The server sends back $\phi = \sum_j \lambda_j F_{i_j}$ and $t = \sum_j \lambda_j t_j$
  - The client accepts if and only if $t = MAC_k(\phi)$

- The scheme is very efficient
  - Linearly homomorphic MACs can be built from basic universal hash functions
  - Minimal storage overhead due to the error-correction expansion
  - Query complexity is quadratic in the security parameter
Proofs of Retrievability (JK)

- Client stores a large file $F$ with the server and wants to make sure that it can be retrieved without downloading the entire thing (e.g. auditing)
  - Client sends a short challenge $c$
  - Server responds with a short answer $a$
    - avoid reading the entire file to produce the answer

- A possible solution (A+,SW)
  - Encode the file $F$ using an error correcting code $F' = Encode(F)$
  - Store each block $F'_i$ with a \textit{linearly homomorphic MAC}
    \[ t_i = MAC_k(F'_i) \]
  - The client queries a small number ($\ell$) of the blocks $F_{i_1} \ldots F_{i_\ell}$ and also sends $\ell$ random coefficients $\lambda_1, \ldots, \lambda_\ell$
  - The server sends back $\phi = \Sigma_j \lambda_j F_{i_j}$ and $t = \Sigma_j \lambda_j t_j$
  - The client accepts if and only if $t = MAC_k(\phi)$

- The scheme is very efficient
  - Linearly homomorphic MACs can be built from basic universal hash functions
  - Minimal storage overhead due to the error-correction expansion
  - Query complexity is quadratic in the security parameter
Verifiable Keyword Search (BGV)

- Client stores a large text file $F = w_1, \ldots, w_n$ with the server
  - Client sends a keyword $w$
  - Server responds with yes/no
  - how can we efficiently verify the answer?

- Can be handled by Merkle trees
  - $O(\log n)$ complexity (time/bandwidth)
  - Can we do better?

- Encode the file as the polynomial $F(X) = \Pi_i (X - w_i)$
  - Note that $F(w) = 0$ if and only if $w \in F$

- Problem reduces to efficiently verifying the computation of a large degree polynomial.
Verifiable Keyword Search (BGV)

- Client stores a large text file \( F = w_1, \ldots, w_n \) with the server
  - Client sends a keyword \( w \)
  - Server responds with yes/no
  - how can we efficiently verify the answer?

- Can be handled by Merkle trees
  - \( O(\log n) \) complexity (time/bandwidth)
  - Can we do better?

- Encode the file as the polynomial \( F(X) = \Pi_i(X - w_i) \)
  - Note that \( F(w) = 0 \) if and only if \( w \in F \)

- Problem reduces to efficiently verifying the computation of a large degree polynomial.
Verifiable Keyword Search (BGV)

- Client stores a large text file $F = w_1, \ldots, w_n$ with the server
  - Client sends a keyword $w$
  - Server responds with yes/no
  - how can we efficiently verify the answer?

- Can be handled by Merkle trees
  - $O(\log n)$ complexity (time/bandwidth)
  - Can we do better?

- Encode the file as the polynomial $F(X) = \Pi_i (X - w_i)$
  - Note that $F(w) = 0$ if and only if $w \in F$

- Problem reduces to efficiently verifying the computation of a large degree polynomial.
Verifiable Keyword Search (BGV)

- Client stores a large text file $F = w_1, \ldots, w_n$ with the server
  - Client sends a keyword $w$
  - Server responds with yes/no
  - how can we efficiently verify the answer?

- Can be handled by Merkle trees
  - $O(\log n)$ complexity (time/bandwidth)
  - Can we do better?

- Encode the file as the polynomial $F(X) = \Pi_i(X - w_i)$
  - Note that $F(w) = 0$ if and only if $w \in F$

- Problem reduces to efficiently verifying the computation of a large degree polynomial.
Verifiable Computation of Polynomials (BGV)

- Client stores a high degree polynomial $F(X) = \sum a_i X^i$
- Client sends a value $x$
- Server responds $y = F(x)$
- how can we efficiently verify the answer?

- Store the MAC $t_i = ca_i + r_i$
  - $r_i$ are computed pseudorandomly, i.e. $r_i = PRF_k(i)$
  - Client only stores random secret keys $c, k$
  - Let $R(X)$ be the polynomial defined by the $r_i$

- When the client queries the value $x$, the server returns
  - $y = \sum_i a_i x^i$ and $t = \sum_i t_i x^i$

- The client checks that $t = cy + R(x)$
  - Note that this requires $O(d)$ work where $d$ is the degree of the poly
  - This can be reduced if we use closed-form efficient PRFs
  - Knowledge of the key $k$ allows the computation of $\sum_i r_i x^i$ in $o(d)$ time
  - We know how to build them from Diffie-Hellman type of assumptions
Verifiable Computation of Polynomials (BGV)

- Client stores a high degree polynomial \( F(X) = \sum a_i X^i \)
  - Client sends a value \( x \)
  - Server responds \( y = F(x) \)
  - how can we efficiently verify the answer?

- Store the MAC \( t_i = ca_i + r_i \)
  - \( r_i \) are computed pseudorandomly, i.e. \( r_i = PRF_k(i) \)
  - Client only stores random secret keys \( c, k \)
  - Let \( R(X) \) be the polynomial defined by the \( r_i \)

- When the client queries the value \( x \), the server returns
  - \( y = \sum_i a_i x^i \) and \( t = \sum_i t_i x^i \)

- The client checks that \( t = cy + R(x) \)
  - Note that this requires \( O(d) \) work where \( d \) is the degree of the poly
  - This can be reduced if we use closed-form efficient PRFs
  - Knowledge of the key \( k \) allows the computation of \( \sum_i r_i x^i \) in \( o(d) \) time
  - We know how to build them from Diffie-Hellman type of assumptions
Verifiable Computation of Polynomials (BGV)

- Client stores a high degree polynomial \( F(X) = \sum a_i X^i \)
  - Client sends a value \( x \)
  - Server responds \( y = F(x) \)
  - how can we efficiently verify the answer?

- Store the MAC \( t_i = c a_i + r_i \)
  - \( r_i \) are computed pseudorandomly, i.e. \( r_i = PRF_k(i) \)
  - Client only stores random secret keys \( c, k \)
  - Let \( R(X) \) be the polynomial defined by the \( r_i \)

- When the client queries the value \( x \), the server returns
  - \( y = \sum_i a_i x^i \) and \( t = \sum_i t_i x^i \)

- The client checks that \( t = cy + R(x) \)
  - Note that this requires \( O(d) \) work where \( d \) is the degree of the poly
  - This can be reduced if we use closed-form efficient PRFs
  - Knowledge of the key \( k \) allows the computation of \( \sum_i r_i x^i \) in \( o(d) \) time
  - We know how to build them from Diffie-Hellman type of assumptions
Verifiable Computation of Polynomials (BGV)

- Client stores a high degree polynomial $F(X) = \sum a_i X^i$
  - Client sends a value $x$
  - Server responds $y = F(x)$
  - How can we efficiently verify the answer?

- Store the MAC $t_i = ca_i + r_i$
  - $r_i$ are computed pseudorandomly, i.e. $r_i = PRF_k(i)$
  - Client only stores random secret keys $c, k$
  - Let $R(X)$ be the polynomial defined by the $r_i$

- When the client queries the value $x$, the server returns
  - $y = \sum_i a_i x^i$ and $t = \sum_i t_i x^i$

- The client checks that $t = cy + R(x)$
  - Note that this requires $O(d)$ work where $d$ is the degree of the poly
  - This can be reduced if we use \textit{closed-form efficient} PRFs
  - Knowledge of the key $k$ allows the computation of $\sum_i r_i x^i$ in $o(d)$ time
  - We know how to build them from Diffie-Hellman type of assumptions
Keyword Search Optimizations (GPSS)

- Server has to read the entire file to answer queries
  - can we use our techniques together with some "indexing"?

- A simple "bucket-hashing" index
  - Partition words into $m$ buckets via hashing
  - Use polynomial scheme on each bucket
  - If $m \approx n$ we get expected constant size buckets

- Allows efficient updates
  - when adding or removing a word from a bucket, re-authenticate the entire polynomial associated with it.
  - Client keeps track of "state" using a "timestamp authentication scheme" (as in previous talk)
    - If using Merkle trees cost is $O(\log \ell)$ where $\ell$ is the number of updates

- Can encrypt document with additive homomorphic encryption
  - Server only computes linear operations
  - Using pseudo-random pseudonyms for the keywords $w_i = PRF_k(W_i)$ we get keyword privacy (e.g. previous talk)
  - No need to prepare a keyword-specific index as in SSE
Keyword Search Optimizations (GPSS)

- Server has to read the entire file to answer queries
  - can we use our techniques together with some "indexing"?

- A simple "bucket-hashing" index
  - Partition words into $m$ buckets via hashing
  - Use polynomial scheme on each bucket
  - If $m \approx n$ we get expected constant size buckets

- Allows efficient updates
  - when adding or removing a word from a bucket, re-authenticate the entire polynomial associated with it.
  - client keeps track of "state" using a "timestamp authentication scheme" (as in previous talk)
    - If using Merkle trees cost is $O(\log \ell)$ where $\ell$ is the number of updates

- Can encrypt document with additive homomorphic encryption
  - Server only computes linear operations
  - Using pseudo-random pseudonyms for the keywords $w_i = PRF_k(W_i)$ we get keyword privacy (e.g. previous talk)
  - No need to prepare a keyword-specific index as in SSE
Keyword Search Optimizations (GPSS)

- Server has to read the entire file to answer queries
  - can we use our techniques together with some "indexing"?

- A simple "bucket-hashing" index
  - Partition words into $m$ buckets via hashing
  - Use polynomial scheme on each bucket
  - If $m \approx n$ we get expected constant size buckets

- Allows efficient updates
  - when adding or removing a word from a bucket, re-authenticate the entire polynomial associated with it.
  - client keeps track of "state" using a "timestamp authentication scheme" (as in previous talk)
    - If using Merkle trees cost is $O(\log \ell)$ where $\ell$ is the number of updates

- Can encrypt document with additive homomorphic encryption
  - Server only computes linear operations
  - Using pseudo-random pseudonyms for the keywords $w_i = PRF_k(W_i)$ we get keyword privacy (e.g. previous talk)
  - No need to prepare a keyword-specific index as in SSE
Keyword Search Optimizations (GPSS)

- Server has to read the entire file to answer queries
  - can we use our techniques together with some "indexing"?

- A simple "bucket-hashing" index
  - Partition words into $m$ buckets via hashing
  - Use polynomial scheme on each bucket
  - If $m \approx n$ we get expected constant size buckets

- Allows efficient updates
  - when adding or removing a word from a bucket, re-authenticate the entire polynomial associated with it.
  - client keeps track of "state" using a "timestamp authentication scheme" (as in previous talk)
    - If using Merkle trees cost is $O(\log \ell)$ where $\ell$ is the number of updates

- Can encrypt document with additive homomorphic encryption
  - Server only computes linear operations
  - Using pseudo-random pseudonyms for the keywords $w_i = PRF_k(W_i)$
    we get keyword privacy (e.g. previous talk)
  - No need to prepare a keyword-specific index as in SSE
Dynamic Storage

- A very important problem is how to deal with updates on the memory
  - without changing the secret state of the client, the server can always ignore updates
  - challenge: updates that do not require the client to re-authenticate large part of the server storage

- Merkle-trees allow to check individual memory locations which change over time
  - but not "global" verifications (proof of retrievability, verifiable keyword search)

- Some progress on dynamic proofs of retrievability (CW, SSP)
A very important problem is how to deal with updates on the memory without changing the secret state of the client, the server can always ignore updates. The challenge is updates that do not require the client to re-authenticate, large part of the server storage.

Merkle-trees allow to check individual memory locations which change over time, but not "global" verifications (proof of retrievability, verifiable keyword search).

Some progress on dynamic proofs of retrievability (CW, SSP).
A very important problem is how to deal with updates on the memory without changing the secret state of the client, the server can always ignore updates. Challenge: updates that do not require the client to re-authenticate large part of the server storage.

Merkle-trees allow to check individual memory locations which change over time but not "global" verifications (proof of retrievability, verifiable keyword search).

Some progress on dynamic proofs of retrievability (CW, SSP).
Multiple clients
- Protect information from the other clients
- Becomes secure multiparty computation with an added constraint
  - only one party has enough resources to compute the desired functionality
- Leverage successes in SMC.

General VC: Explore more realistic models of computation
- e.g. RAM

Explore more pragmatic approaches
- Weaker security guarantee that rules out most likely forms of attacks
  - e.g., program checking against bugs in the implementation
- Rational Agents (AM): pay the server for his work. Make sure reward is maximized when the server is correct.
Future Directions 1

- **Multiple clients**
  - Protect information from the other clients
  - Becomes secure multiparty computation with an added constraint
    - only one party has enough resources to compute the desired functionality
  - Leverage successes in SMC.

- **General VC**: Explore more realistic models of computation
  - e.g. RAM

- **Explore more pragmatic approaches**
  - Weaker security guarantee that rules out most likely forms of attacks
    e.g. program checking against bugs in the implementation
  - *Rational Agents (AM)*: pay the server for his work. Make sure reward is maximized when the server is correct.
Future Directions 1

- Multiple clients
  - Protect information from the other clients
  - Becomes secure multiparty computation with an added constraint
    - only one party has enough resources to compute the desired functionality
  - Leverage successes in SMC.

- General VC: Explore more realistic models of computation
  - e.g. RAM

- Explore more pragmatic approaches
  - Weaker security guarantee that rules out most likely forms of attacks
    - e.g. program checking against bugs in the implementation
  - *Rational Agents* (AM): pay the server for his work. Make sure reward is maximized when the server is correct.
Future Directions 1

- **Multiple clients**
  - Protect information from the other clients
  - Becomes secure multiparty computation with an added constraint
    - only one party has enough resources to compute the desired functionality
  - Leverage successes in SMC.

- **General VC**: Explore more realistic models of computation
  - e.g. RAM

- **Explore more pragmatic approaches**
  - Weaker security guarantee that rules out most likely forms of attacks
    - e.g. program checking against bugs in the implementation
  - *Rational Agents* (AM): pay the server for his work. Make sure reward is maximized when the server is correct.
Future Directions 1

- Multiple clients
  - Protect information from the other clients
  - Becomes secure multiparty computation with an added constraint
    - only one party has enough resources to compute the desired functionality
  - Leverage successes in SMC.

- General VC: Explore more realistic models of computation
  - e.g. RAM

- Explore more pragmatic approaches
  - Weaker security guarantee that rules out most likely forms of attacks
    - e.g. program checking against bugs in the implementation
  - *Rational Agents* (AM): pay the server for his work. Make sure reward is maximized when the server is correct.
Does the outsourcing of polynomials have larger applicability?
  - Alternatively, can we use the same idea of "closed form efficient" PRFs for other computations

A more efficient general result for memory outsourcing/homomorphic MACs

"Important" Computations, which would benefit from being outsourced:
  - Image processing
  - Crypto operations
Does the outsourcing of polynomials have larger applicability?
  - Alternatively, can we use the same idea of "closed form efficient" PRFs for other computations

A more efficient general result for memory outsourcing/homomorphic MACs

"Important" Computations, which would benefit from being outsourced:
  - Image processing
  - Crypto operations
Future Directions 2

- Does the outsourcing of polynomials have larger applicability?
  - Alternatively, can we use the same idea of "closed form efficient" PRFs for other computations

- A more efficient general result for memory outsourcing/homomorphic MACs

- "Important" Computations, which would benefit from being outsourced:
  - Image processing
  - Crypto operations