Secure and Verifiable Computation Outsourcing

Marina Blanton
Department of Computer Science and Engineering
University of Notre Dame

Summer School on Secure and Oblivious Computation and Outsourcing
May 2016
Motivation

- **Computation outsourcing** to a cloud comes with security risks
  - sensitive information might be revealed to third parties
  - computation might be corrupt or skipped resulting in invalid output

- Different **security objectives** might be desired
  - we might want to **protect confidentiality of client’s sensitive data** used in the computation
  - we might also want to **verify integrity of outsourced computation**

- The focus of this talk is primarily on the second security objective
• Application 1: Large-scale biometric computations
  – computation consists of distance computations as well as distribution computation
    • can be applicable to other domains
  – we would like to achieve both data confidentiality protection and computation integrity verification
• **In biometric research**, there is a need to test effectiveness of new algorithms on large data sets.

• After image selection and feature extraction, **two types of computations take place**
  
  – **AllPairs**($S, F$) compares all members of set $S$ pair-wise using function $F$, producing matrix $M$, where $M[i][j] = F(S[i], S[j])$
  
  – **Analyze**($M, C$) extracts statistical data from matrix $M$ and stores the result in $C$

• Because of the volume of the computation, the tasks are outsourced to computational nodes.
• Three types of distance computation $F$ are of interest:
  – Hamming distance, Euclidean distance, and set intersection cardinality

• Each item consists of $m$ elements and the distance is in the range $[0, \sigma]$

• To enable any type of statistical analysis, outsourced computation for Analyze consists of producing raw distribution data
  – it forms a vector $C' = (c_0, \ldots, c_{v-1})$, where $c_i$ stores the number of times the distance was $d_i$
  – $[d] = \text{dist}([x], [y])$;
    for $i = 0, \ldots, v - 1$ do $[b_i] = ([d] \xrightarrow{?} [d_i])$; $[c_i] = [c_i] + [b_i]$;
Large-Scale Biometric Computations

- **Computation overview** (without verification):

  \[ S' = S_1' + S_2' + \ldots + S_1'' + S_2'' \]

  **Client**

  **Output**

  \[ C = C_1 + C_3 \]

  **Tasks**

  1. \[ S_1' + S_1'' = M_1 + C_1 \]
  2. \[ S_2' + S_2'' = M_2 + C_2 \]
  3. \[ S_1' + S_1'' = M_3 + C_3 \]
  4. \[ S_2' + S_2'' = M_4 + C_4 \]
Security Model

• Computation should take place on protected data

• A server might deviate from the computation by skipping a portion of it or returning incorrect results
  – the server computes fraction $p$ of its task, where $0 \leq p \leq 1$
  – the server attempts to manipulate the result to pass result verification
  – the server’s work is bounded by fraction $p$ of the assigned task
  – probability of cheating detection should be at least $1 - 1/2^\kappa$ for security parameter $\kappa$ when $p$ is below the client-chosen threshold
• The main idea behind verifying AllPairs computation is as follows:
  – when forming a computational task $\langle S_1, S_2, \text{dist} \rangle$, the client inserts fake, random elements at random positions into the sets
  – the fake items are chosen once for all tasks and their distances are precomputed
  – because the data is protected, the server is unable to distinguish fake items from other items
  – fake items are required because real items might not have enough uncertainty
  – to form $S_1$, the client uses $n_1$ fake and $n - n_1$ real items
  – similarly, $S_2$ contains $n_2$ fake and $n - n_2$ real items
• **One possibility** for the client is to verify all $n_1n_2$ distances in the result
  
  – all fake items are placed at random locations
  
  – the client stores all $n_1n_2$ distances precomputed
  
  – verification cost is then $O(n_1n_2)$
  
  – it is not possible for all $n_1n_2$ fake distances to be distributed uniformly at random
  
  – the probability of cheating detection can be determined for different ways of computing $pn^2$ distances

• A solution of cost $O(n_1 + n_2)$ can be devised that has a higher probability of cheating detection
• A better approach is as follows
  
  – let \( n_1 = n_2 \) and denote the fake items in \( S_1 \) and \( S_2 \) as \( F_1 \) and \( F_2 \)
  
  – for each \( i \)th element of \( F_1 \) and \( F_2 \) we choose a random distance from \([0, \sigma]\) and create two fake items with that distance
  
  – as before, the items are placed at random locations in \( S_1 \) and \( S_2 \)
  
  – we now check only \( n_1 = n_2 \) distances in the returned matrix
The probability that the server’s cheating is not detected after verifying a single cell is

- \( p + (1 - p)\frac{1}{\sigma+1} \) if the server computes entire rows (or columns)
  - \( pn \) complete rows or columns were computed
- \( prpc + (1 - prpc)\frac{1}{\sigma+1} \) if the server computes partial rows and columns
  - \( prn \) partial rows and \( pcn \) partial rows were computed with \( p = prpc \)
  - a cell is computed if both its row and column were among the computed ones
- \( p + (1 - p)\frac{1}{\sigma+1} \) if the server computes distances at random cells
  - \( pn^2 \) cells were computed
• In either case, the probability of cheating detection after checking $n_1$ cells is

\[
\Pr[D] = 1 - \left( \frac{p\sigma + 1}{\sigma + 1} \right)^{n_1}
\]

• If we set $n_1 \geq \frac{\kappa(\sigma+1)}{(1-p)\sigma}$, cheating is undetected with probability

\[
\Pr[\overline{D}] = \left( \frac{p\sigma + 1}{\sigma + 1} \right)^{n_1} = \left( 1 - \frac{(1 - p)\sigma}{\sigma + 1} \right)^{n_1} \leq e^{- \frac{n_1(1-p)\sigma}{\sigma+1}} = e^{-\kappa}.
\]
Verification of Distribution Computation

- Recall that distribution computation produces a vector $C$ with counts.
- Now the verification mechanism depends on the distance computation.
- For the Hamming distance we can do the following:
  - In addition to inserting fake items into the task, the client inserts a fake element into both real and fake items.
  - The fake element is used to separate the distance ranges between two real items and real and fake items.
  - Distances between two real items remain in the range $[0, \sigma]$.
  - Distances between real and fake items are in the range $[\sigma + 1, 2\sigma + 1]$.
  - The client adds the distances in each range and compares them to the expected values.
This strategy is not yet enough

- if the server knows (or can guess) $n_1$ and $n_2$, it will be able to pass the verification

We employ two ideas to mitigate the problem

- (protected) distances for computing the distribution are given to the server in a randomized order
- the client verifies a larger number of aggregate counts
- the client also inserts a number of fake elements in each item instead of only one
An improved solution is as follows:

- the client adds \( k \) fake elements to the original \( m \) elements of each real item and creates fake items with \( m + k \) elements
- the \( m + k \) elements are randomly permuted, but consistently across all items
- the client chooses a small integer \( \ell \) and \( \ell \) values larger than the maximum distance \( \sigma \)
  - let these \( \ell \) values be \( \sigma + 1, \ldots, \sigma + \ell \)
- the client forms real and fake items so that the distance between
  - two real items is in the range \([0, \sigma]\)
  - a real and fake items is in the range \([\sigma + 1, 2\sigma + \ell]\)
Improved solution (cont.)

- for any given fake item, its distance to a real item is in the range 
  \[ [d, d + \sigma] \] for fixed \( d \in [\sigma + 1, \sigma + \ell] \)
- for each \( d \in [\sigma + 1, 2\sigma + \ell] \), the client compiles statistics and records the expected counts
- at the time of computation verification
  - the client compares the computed counts for distances in 
    \([\sigma + 1, 2\sigma + \ell]\) to their expected values
  - the client also compares the aggregate count for distances in \([0, \sigma]\) to 
    \((n - n_1)(n - n_2)\)
Hamming Distance Distribution Verification

- This solution is realized for the Hamming distance as follows
  - for \(m\) original elements, \(\sigma = m\) and original distances between real items are in \([0, m]\)
  - the extra \(k\) elements are all set to 0 in real items
  - the original \(m\) are set to 0 in fake items
  - to create the extra \(k\) elements in a fake item, the client
    - selects \(d\) at random from \([m + 1, \ldots, m + \ell]\)
    - selects \(k\) random values \(d_i\) such that \(\sum_i^k d_i = d\) and sets each fake element to \(d_i\)
      - this deviates from data representation, but is hidden from the server
The Hamming distance solution (cont.):

- the expected counts for each distance in \([m + 1, 2m + \ell]\) can be determined by knowing
  - the Hamming weight of each original item
  - and the number of fake items that used each \(d\) from \([m + 1, m + \ell]\) for its fake elements

- distances between two fake items could
  - be forced to lie outside the range \([0, 2m + \ell]\)
  - overlap with the range, but be compensated for
Hamming Distance Distribution Verification

- The above solution can be shown to meet the necessary security requirements.

- We can treat two options:
  - the server computes all distances between real and fake vectors correctly (by guessing the locations of fake vectors) and increases the count(s) for distances between real and real items
  - the server doesn’t compute all distances between real and fake vectors and correctly increases the counts for distances between real and real and between real and fake items

- The probability of successful cheating should be negligible for any server’s strategy.
Hamming Distance Distribution Verification

• In the first case, it can be shown that

\[ \Pr[D] \geq 1 - \frac{m + 1}{2m + 1 + \ell} \left(1 - \sqrt{1 - p}\right)^{2n_1} \]

or

\[ \Pr[D] \geq 1 - \prod_{i=0}^{\beta m + k - 1} \frac{p(m + k) - i}{m + k - i} \]

where at least \( \beta m \) elements are set to 1 in each original biometric

– the first equation lets us set \( n_1 \) and the second equation lets us set \( k \)

• Analysis of the second case also places bounds on \( n_1, k, \) and \( \ell \)

• The highest of the computed values for each of \( n_1, k, \) and \( \ell \) should be used
A similar approach can be used for other distance metrics

- Euclidean distance
- set intersection cardinality

Experimental Results

- We evaluate performance of these techniques
  
  - task computation and communication time with and without data protection:

<table>
<thead>
<tr>
<th>Data set size</th>
<th>Share receiving</th>
<th>Computation (private)</th>
<th>Computation (non-private)</th>
<th>Result sending</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- securely computing the Hamming distance requires only a single interaction using secret sharing techniques
Experimental Results

• Sample parameters for distance verification
  
  – for \( \Pr[D] \geq 0.99 \) with \( p \leq 0.95 \), \( n_1 = 90 \)
  – for \( \Pr[D] \geq 0.95 \) with \( p \leq 0.9 \), \( n_1 = 29 \)
  – server’s task computation time for medium \( n_1 = 50 \):
Experimental Results

- Parameters for Hamming distance distribution verification with $m = 1000$ and varying task size $n$

<table>
<thead>
<tr>
<th>Security setting</th>
<th>Computed parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$k$</td>
</tr>
<tr>
<td></td>
<td>any $n$</td>
</tr>
<tr>
<td>$p \leq 0.9$, $\Pr[D] \geq 0.95$, $\gamma \leq 0.05$</td>
<td>28</td>
</tr>
<tr>
<td>$p \leq 0.95$, $\Pr[D] \geq 0.95$, $\gamma \leq 0.05$</td>
<td>57</td>
</tr>
<tr>
<td>$p \leq 0.95$, $\Pr[D] \geq 0.99$, $\gamma \leq 0.01$</td>
<td>87</td>
</tr>
</tbody>
</table>

- $\gamma$ is another security parameter that affects probability of cheating detection
Experimental Results

- Client’s performance
  - for tasks of size up to 8000, one-time precomputation is 14–22ms, expected distribution computation is 10–50ms, and task preparation and verification is 0–8s

- Server’s performance

\[
\begin{array}{c|c|c}
\text{Data set size} & \text{Secure-only computation} & \text{Secure and verifiable computation} \\
200 & \text{Time (s)} & \text{Time (s)} \\
400 & 5 & 10 \\
600 & 10 & 15 \\
800 & 15 & \times 10^4 \\
\end{array}
\]
• Application 2: Matrix multiplication
  – multiplication of two matrices $C = A \times B$ is outsourced
  – security objectives include
    • computation verifiability
    • data privacy protection
    • public verifiability
Two types of adversaries are considered

- a malicious adversary (server) who can arbitrary deviate from the prescribed computation

- a rational adversary
  - it is neither honest nor malicious
  - it is economically motivated with the goal of maximizing its monetary rewards
  - it returns correct results if it performed the corresponding computation
• A **verifiable computation scheme** is defined by four polynomial-time algorithms
  
  - KeyGen($1^k$, $f$) $\rightarrow$ params
  - ProbGen($x$, params) $\rightarrow$ ($SK_x$, $EK_x$, $\sigma_x$)
  - Compute($EK_x$, $\sigma_x$) $\rightarrow$ $\sigma_y$
  - Verify($SK_x$, $\sigma_y$) $\rightarrow$ $y \cup \bot$

• **Security** requires that verification of an output other than $f(x^*)$ for the challenge input $x^*$ cannot succeed with a non-negligible probability
Verifiable Matrix Multiplications

- In the malicious security setting
  - input preparation ProbGen is linear in the input size
  - outsourced computation has the same $O(n^3)$ complexity as the regular (non-secure) matrix multiplication
  - output verification involves only a single modulo exponentiation
    - the overall work is linear in the output size
- Verification is regarded as more frequent than other operations
The setting relies on a bilinear map \( e : G_1 \times G_2 \rightarrow G_T \) with respective generators \( g_1, g_2, \) and \( g_T \).

At input preparation time, the delegator
- sends matrices \( A, B, X, \) and \( Y \) to the server
- \( X \) is of the form \( X_{ij} = g_1^{r_i A_{ij}} \) and \( Y \) is of the form \( Y_{ij} = g_2^{c_j B_{ij} + w_j d_i} \)

The server computes \( C = A \times B \) and \( D = X \times Y \) (using pairing operations) and returns \( C \) and \( \sum_i \sum_j D_{ij} \).

Verification is of the form \( g_T^{f(C)} = \sum_i \sum_j D_{ij} \).

Security relies on M-DDH and XDH assumptions.
• In the rational security setting
  – the verification computation itself is outsourced to the server
  – verification now involves a simple comparison
  – complexities of all other algorithms remain unchanged

• In addition to forming $X$ and $Y$, the delegator now also releases a new matrix $Z$ of the form $g_{T}^{r_{i}c_{j}}$
  – the matrix allows the server to produce a proof of correct computation
  – now additional randomness needs to be incorporated into $X$ and $Y$
Verifiable Matrix Multiplications

• Now at input preparation time, the delegator
  - prepares $X_{ij} = g_1^{r_i/m_jA_{ij}}$, $Y_{ij} = g_2^{c_jm_iB_{ij}+w_jd_i}$, and $Z_{ij} = g_T^{c_jr_i}$
  - computes expected keys for each cell and hashes all keys together

• The server now computes
  - $C = A \times B$, $D = X \times Y$
  - the expected keys using $Z$, $C'$, and $D$
  - the hash of all computed keys

• Verification simply consists of comparing the expected key to the key produced by the server

• Security relies on the fact that if the server computed $C$, it won’t corrupt it
We enhance the basic solution with a number of features

We put forward a new feature in the form of chained computation

- the overall computation is divided into stages and there is a key associated with each stage
- only if the computation in the current stage is performed correctly, the server can recover the correct key and proceed with next stage

Advantages of chaining

- allows for more efficient task verification
- allows for efficient identification of the first incorrectly computed stage
- allows honest servers to detect computation corruptions
Verifiable Matrix Multiplications

- **Chaining** can be naturally added to the rational adversary scheme
  - stage \( i \) computes the \( i \)th column of the product matrix \( C' \)
  - only when the \( i \)th key is produced, the inputs to the \( (i + 1) \)th sub-computation can be recovered
    - \( (i + 1) \)th stage inputs are masked (XORed) with a pseudo-random sequence computed using the \( i \)th key
    - extra (fixed) bits are added to the inputs to indicate successful decoding
  - Detection of the first faulty stage now involves \( \log n \) steps (server queries)
  - The constructions can also be modified to add data privacy and/or public verifiability
Conclusions

• Achieving verifiable computation is often non-trivial for different functions

• A variety of different techniques can be found in the literature

• Using rational adversaries is one way of improving performance of verifiable computation schemes