Secure and Verifiable Outsourcing of Large-Scale Biometric Computations

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

Keith B. Frikken
Computer Science and Software Engineering
Miami University

Abstract

Cloud computing services are becoming more prevalent and readily available today, bringing to us economies of scale and making large scale computation feasible. Security and privacy considerations, however, stand on the way of fully utilizing the benefits of such services and architectures. In this work we address the problem of secure outsourcing of large-scale biometric experiments to a cloud or grid in a way that the client can verify that with very high probability the task was computed correctly. We conduct thorough theoretical analysis of the proposed techniques and provide implementation results that indicate that our solution imposes modest overhead.

1 Introduction

Cloud computing enables on-demand access to computing and data storage resources, which can be configured to meet unique constraints of the clients and utilized with minimal management overhead. The recent rapid growth in availability of cloud services makes such services attractive and economically sensible for clients with limited computing or storage resources who are unwilling or unable to procure and maintain their own computing infrastructure. One of the largest possibilities that the cloud enables is computation outsourcing, when the client can utilize any necessary computing resources for its computational task. It has been suggested that the top impediment on the way of harnessing the benefits of cloud computing to the fullest possible extent is security and privacy considerations that prevent clients from placing their data or computations on the cloud (see, e.g., survey [1]). While in general sensitive data can be protected by the means of encryption, computation using the data encrypted via the traditional means becomes impossible (at least from a practical perspective). Furthermore, the clients no longer have direct control over the outsourced data and computation and there is a lack of transparency in the current cloud services. The cloud provider can be incentivized to delete rarely accessed data or skip some of the computations to conserve resources (for financial or other reasons), which is especially true for volunteer-based computational clouds. Furthermore, unintentional data or computation corruption might also take place for a variety of reasons including malware, security break-ins, etc. From that perspective, it is important for the clients to be able to verify the correctness of the result of the outsourced computational task. The verification mechanism should be such that it does not require the client to perform the computation comparable in size to the outsourced task itself. Secure and verifiable outsourcing of certain types of computation is therefore the focus of this work.

The main motivation for this work comes from the extensive amount of computation involved in biometric research that, due to memory and processing power constraints, inevitably pushes the computation on a computational cloud or grid. The sensitive nature of the data makes its protection throughout the computation a necessary requirement, and to be able to rely on the outcome of the computation, the result needs to be verified. While we present the developed techniques in the context of biometric data processing, our results can be used for any type of computation of similar structure.

The computation, secure and verifiable outsourcing of which we address in this work, can be described as follows. In biometric research, evaluation of a new recognition algorithm amounts to running the algorithm on a very large number of biometric images. Given a data set D of biometric images, first each image needs to be processed to extract the features; we refer to the result as the biometric template. Next, the distance between each two templates in D is computed; this is called "all pairs" computation. The result allows users to gather distribution (and any necessary statistical) information about the quality of biometric matching for impostor and authentic comparisons (images corresponding to different subjects are termed imposter and images corresponding to the same subjects are termed authentic). The accuracy of the result depends on the size and quality of images in D, which can often consist of tens (or even hundreds) of thousands of biometrics. This volume of the computation cannot be performed on a single machine and needs to be partitioned and run by a computational grid or cloud.

To help the users with the task of running such large scale experiments, Bui et al. [6] developed a level of abstraction and corresponding end-to-end computing system, called BXGrid, which eliminates the need for users to be effective at configuring and using grid computing systems, relational databases, distributed filesystems, etc. With BXGrid, the computations can be run using four simple abstractions:

- $\mathsf{Select}(R)$: select a set of images and metadata from the repository based on requirements R (such as subject gender, location, etc.).
- $\mathsf{Transform}(S, F)$: apply transformation function F (such as feature extraction) to each member of a given set S.
- $\mathsf{AllPairs}(S, F)$: compare all members of given set S using function F, producing matrix M, where each element $M[i][j] = F(S[i], S[j])$.
- $\mathsf{Analyze}(M, C)$: extract statistical (or other quality metric) data from matrix M and store the result in C.

In this work, we extend this framework by integrating security protection in the above computations to ensure that it can be placed on a cloud comprised of untrusted machines and the correctness of the computation is verifiable by the client. Throughout this work, we assume that the client is capable of performing work linear in the number biometrics in D, $|D|$, but computation exceeding this linear bound (e.g., quadratic complexity of $\mathsf{AllPairs}$) is beyond the client's capabilities. For that reason, we concentrate on the computation corresponding to $\mathsf{AllPairs}$ and $\mathsf{Analyze}$ functionalities. Furthermore, because secure comparison of biometric templates has been a subject of prior research (see, e.g., [11, 5, 3] and others), most of this work is dedicated to techniques for computation verification. These techniques are independent of the mechanisms for securing the computation (such as those based on secret sharing or homomorphic encryption) and can be used with any suitable solution for protecting privacy of the data. For concreteness, we utilize unconditionally secure multi-party computation techniques based on secret sharing to secure the computation and use such techniques in the implementation.

Our contributions. We design a mechanism for verification of outsourced $\mathsf{AllPairs}$ computation and provide its rigorous analysis, which will allow the client to set the security parameters as to achieve the desired probability of misbehavior detection (Section 3). We also design a mechanism for verifying the distance distribution computation Analyze and likewise provide its rigorous analysis. Our full analysis is for a solution when the Hamming distance is used as the distance metric (Section 4), and consecutive modifications to the strategy and the analysis address the Euclidean distance (Section 5) and the set intersection cardinality (Section 6). Finally, we combine solutions for AllPairs and Analyze functionalities to result in a complete solution for the overall process for each distance metric and report on implementation results on the example of the Hamming distance metric.

2 Problem Description

Computation description. A client has a pre-selected large collection S of biometric templates, which are to be compared and analyzed. To accomplish the AllPairs functionality, the matrix M is partitioned across multiple computational servers such that each server receives a computational task which can be performed within its memory and computational capacity constraints. Let a server receive a job of the form of two sets S_1 and S_2 of n items each and its task is to perform comparisons of each pair of items $x \in S_1$ and $y \in S_2$, producing an $n \times n$ matrix as the output. When we refer to this matrix, we will assume that the items from the first set correspond to rows of the matrix and the items from the second set correspond to columns. The server is not assumed to follow the computation as prescribed, but it might be interested in attempting to avoid being detected that (some of) the computation was not performed. The computation is appropriately secured, so that the server does not learn any information about the data it handles, but has the description of the computation. Depending on how the secure computation is realized, a single task might take the form of multi-party computation, in which case it should be understood that when we refer to server's computation it will involve (interactive) computation by multiple entities. Also, because biometric comparisons amount to computing the distance between two items which normally consist of multiple elements, we will assume that each item consists of m elements.

To accomplish the Analyze functionality, the computational servers will need to post-process the distance values in the matrix to compute statistical information. In this work we propose that each server computes the number of times each particular distance value appeared among the n^2 computed distances. This information fully defines the distribution and will allow the client to compute any necessary statistics from it. (When the computation is distributed among multiple servers, the client can easily merge the returned data by simply adding the counts from multiple servers for each given distance value.) To compute the count for each distance, the server will obliviously compare a distance it computed for a given cell to all possible distance values and increment one of them that matched (without knowing which count was incremented). We will refer to the data structure that stores distribution information (i.e., an array of protected counts) as C. Note that the number of counts that C contains will depend on the range of distances information about which is being collected, which for two original biometrics of size m is assumed to be $[0, m]$. Let $C = \langle c_0, \ldots, c_{v-1} \rangle$, where v may be $m + 1$ or another value. Then the pseudo-code below shows how C is being updated during the computation of distance in each cell. Below, notation $[x]$ denotes that the value of x is not known by the server, the result of the equality testing operation ($a \overset{?}{=} b$) is a bit, and d_i is the distance value associated with count c_i. Initially, all c_i's are set to 0.

```
$[d] := \mathsf{dist}([x], [y])$;
for $i = 0, \ldots, \ell - 1$
    $[b] := ([d] \overset{?}{=} [d_i])$;
    $[c_i] := [c_i] + [b]$;
```

Note that because the server does not have access to the distances d_i associated with each c_i, the values of d_i do not have to equal to i or even be in the range $[0, v-1]$.

Security model. As mentioned above, our goal is to achieve secure and verifiable computation outsourcing. As the emphasis of this work is the design of computation verification techniques for the types of computations described above, throughout most of this paper we will assume that the computation can be carried out in a secure manner. That is, the server is unable to learn information about the data it handles, but might deviate from the computation by skipping a portion of it or returning incorrect results. In particular, we assume that the server computes fraction p of its task, where $0 \leq p \leq 1$, and attempts to manipulate the result so as to make the client believe that it computed its task as prescribed. The client's goal, then, is to devise a verification mechanism which detects the server's misbehavior even when the portion of skipped computation, $1-p$, is small. Let D denote the event that the client detects server's cheating (and $\overline{\mathsf{D}}$ the event that the client does not detect it). Then the client's goal is to achieve $\Pr[\mathsf{D}] \geq 1 - \frac{1}{2^\kappa}$ for the desired security parameter κ whenever p is above the specified threshold.

On the security definition. With the above formulation, the client will detect, with the desired probability, instances when the server performs (at most) p fraction of its task. Because a task consists of many sub-tasks, this fraction of work can, for instance, correspond to pn^2 computed distances out of assigned n^2. An alternative formulation of the problem is to suggest that each task is computed with probability p, where the probabilities for each sub-task are independent. (This can be meaningful in the context of data or task corruption, where each sub-task has a small probability of being corrupt.) In that case, the overall number of performed computations is not known, but is characterized by a distribution. As we show later in this work, the two definitions lead to almost identical analysis for the purposes of this work assuming that the task size n is large. For concreteness we base our analysis on the first formulation of server misbehavior.

3 Verification of Distance Computation

The main idea behind verifying AllPairs computation is to insert a number of fake items in the computational task at random positions when forming a job $\langle S_1, S_2, \mathsf{dist}\rangle$ to send to a computational server. The fake items need to be chosen only once for all tasks (and sub-tasks) and the distances between them are precomputed. Because all computation takes place on protected data, the server will not be able to distinguish fake items from original data. Upon job completion, the client receives the matrix, reconstructs the results, and compares the distances between the fake items computed by the server with the values that it expects. Because the server does not know a priori which values will be checked, it will have to compute the distances honestly. (If in the specific implementation the matrix M is not returned to the client, then the server will send a commitment to all computed values in M using, e.g., the Merkle hash tree approach of [10]. Then the client will challenge the server on certain cells of the matrix and the server must show that correct distances were included in the commitment.)

For that reason, to form the sets S_1 and S_2, the client uses $n - n_1$ real items and n_1 fake items at random positions to create S_1. Similarly, S_2 consists of $n - n_2$ real items and n_2 fake items at random positions. Here n_1 and n_2 are parameters which will be set to obtain $\Pr[\mathsf{D}] \geq 1 - \frac{1}{2^\kappa}$. The exact strategies for assigning the values to the fake items and verifying the result can differ. We present one approach that leads to rather simple analysis and has attractive properties.

The solution is as follows: For a task of square size $n \times n$ we set $n_1 = n_2$. The client generates n_1 fakes items $x_1, \ldots, x_{n_1}$ and $n_2 (= n_1)$ fake items $y_1, \ldots, y_{n_2}$ such that the distance between each x_i, y_i is chosen uniformly from the range $[0, m]$ and then items x_i, y_i are created with that distance. (We assume that any distance in the range $[0, m]$ can be generated.) The x_i's (y_i's) are placed at random locations in S_1 (resp., S_2). To verify the result, the client extracts the distances corresponding to items x_i, y_i for each i from the returned by server matrix and compares the result to the values it generated.

Let $n_1 \ll n$ and $n_2 \ll n$ (which will hold under our assumption that n is large). The manner in which the server computes the pn^2 distances can affect the probability of its cheating being detected. In what follows, we analyze three server's strategies:

1. *The server computes the distances in the entire matrix row.* That is, if the server computes a distance in any given row, it will compute all other distances in that row. This means that the server computes the distances corresponding to pn rows. Suppose for the current analysis that the server's cheating is always detected (with probability 1) if it skipped the computation of a cell which the client checks. In this case, the probability that the server's behavior remains undetected corresponds to the probability that all n_1 fake items from S_1 fell into pn computed rows.

Given that $n_1 = n_2 \ll n$, we can use binomial distribution for computing $\Pr[\mathsf{D}]$ which assumes that elements are drawn from a set with replacement (i.e., the probability that a fake item falls within the set of computed distances always remains p, regardless of how many fake items have previously fallen within either set). We then obtain:

$$\Pr[\mathsf{D}] = (1-p) + p(1-p) + p^2(1-p) + \cdots + p^{n_1-1}(1-p) = 1 - p^{n_1} = 1 - \Pr[\overline{\mathsf{D}}]. \tag{1}$$

When, however, n is small, the experiment should be modeled using hyper-geometric distribution, where elements are drawn from a set without replacement. Therefore, the previous equation becomes: $\Pr[\mathsf{D}] = (1-p)+p\frac{(1-p)n}{n-1}+p\frac{pn-1}{n-1}\frac{(1-p)n}{n-2}+\cdots+p\frac{pn-1}{n-1}\cdots\frac{pn-n_1+2}{n-n_1+2}\frac{(1-p)n}{n-n_1+1}$, which using the properties of hyper-geometric distribution can be rewritten as: $\Pr[\mathsf{D}] = 1-\binom{np}{n_1}/\binom{n}{n_1} = 1-\frac{(np)!(n-n_1)!}{(np-n_1)!n!}$. While this equation more accurately determines the probability, for the purposes of this work it is sufficient to consider binomial distribution as a close approximation that simplifies the analysis and we therefore use the expression in equation 1.

We now can complete the analysis by taking into account the fact that the server can avoid the detection by guessing the correct distance for a cell that the client checks. The probability that the server's misbehavior is not detected after verifying a single cell is $p + (1-p)\frac{1}{m+1} = \frac{pm+1}{m+1}$ (i.e., the server computed the distance or did not compute and guessed it). This gives us that detection probability after checking n_1 cells (at distinct rows unpredictable to the server) is

$$\Pr[\mathsf{D}] = 1 - \left(\frac{pm+1}{m+1}\right)^{n_1}. \tag{2}$$

The same applies to the case when the server computes the cells column-wise instead of row-wise since $n_1 = n_2$.

2. *The server computes partial rows and columns.* In this case, the server computes cells corresponding to $p_r n$ partial rows and $p_c n$ partial columns in a consistent way, where $p = p_r p_c$, and a cell is computed if both its row and column are among the computed $p_r n$ and $p_c n$, respectively. With this server's strategy, the client will not be able to detect misbehavior after checking a single cell with probability $p_r p_c + (1 - p_r p_c)\frac{1}{m+1}$. We therefore obtain that, using similar analysis with replacement, after checking n_1 cells at distinct rows and columns $\Pr[\mathsf{D}]$ is the same as in equation 2.

3. *The server computes distances at random cells.* The server chooses pn^2 matrix cells at random out of n^2 and computes their values. Similar to the above cases, the probability that detection is not successful after a single check is $p + (1-p)\frac{1}{m+1}$, and $\Pr[\mathsf{D}]$ is the same as before. Hence, in any case the probability of detection is $1 - (\frac{pm+1}{m+1})^{n_1}$, and this property guided our choice of

computation verification approach. Furthermore, if we set $n_1 \geq \frac{\kappa(m+1)}{(1-p)m}$, then the adversary will be undetected with probability $\Pr[\overline{\mathsf{D}}] = (\frac{pm+1}{m+1})^{n_1} = (1 - \frac{(1-p)m}{m+1})^{n_1} \leq e^{-\frac{n_1(1-p)m}{m+1}} = e^{-\kappa}$.

While the above analysis is applicable to any type of computational tasks when pair-wise computations are performed $n \times n$ times, the structure of each individual computation can be such that it permits server's misbehavior to be undetected. In our case, the computed function corresponds to the computation of the distance between two items. Because each item consists of multiple elements, the server might attempt to reduce its workload by computing partial distance using a subset of the elements and then adjust the results to avoid detection. We next show that our solution also effectively combats this server's strategy resulting in necessary detection success probability. Suppose now the server computes p fraction of each distance and computes this for each cell of the matrix. Since there are m elements in each biometric, the server computes the distance corresponding to pm elements of its choice and needs to guess the remaining portion of the distance. Because the distances that the client checks are distributed uniformly at random across the range $[0, m]$, the distances corresponding to $(1-p)m$ elements that the server did not compute in the checked cells will be distributed uniformly at random across the range $[0, (1-p)m]$. This means that the probability that the server's behavior is not detected after checking a single cell is $\frac{1}{(1-p)m+1}$ and the probability of detecting the server's behavior after checking all n_1 cells is $\Pr[\mathsf{D}] = 1 - \left(\frac{1}{(1-p)m+1}\right)^{n_1}$. This quantity is at least as large as the value computed in equation 2 for any $p \leq 1$ and m (and is strictly larger when $p < 1$), and therefore is not attractive for the adversary. In addition, if the server combines this strategy with one of the possibilities listed above, it still will not be able to reduce the probability of detection below the value listed in equation 2. For instance, if the server computes partial distances in $p'n$ rows using $p''m$ elements, such that $p' \cdot p'' = p$, the probability that misbehavior is not detected after checking a single cell is when either the server computed the partial distance and guessed the remaining portion or did not compute the distance and guessed it entirely: $p' \frac{1}{(1-p'')m+1} + (1-p')\frac{1}{m+1}$. It is clear that the success probability of this hybrid strategy lies between the computed probabilities for the individual misbehavior approaches and becomes equal to the probabilities of one of the individual approaches when $p' = 1$ or $p'' = 1$ (while keeping the overall amount of work the same with $p' \cdot p'' = p$).

4 Verification of Statistics Computation for Hamming Distance

We are now interested in providing a secure implementation of the $\mathsf{Analyze}$ functionality. The client now employs a different method for verifying that the distribution computation was performed correctly. In the following description we use Hamming distance as the distance metric for computing distances between two items. Modifications to this method that apply to other distance metrics are provided in the next sections.

4.1 First attempt

As before, the server receives a job of size $n \times n$. The client inserts a number of fake items in the computation to aid the verification process, but it also inserts additional fake elements into both real and fake items. We have m elements in each original item, where each element is a bit (i.e., the Hamming distance between any two items is in the range $[0, m]$). Suppose that the client inserts a new $(m+1)$st element in each item, such that distances between real and fake items fall outside of the original range $[0, m]$. In particular, if $x = \langle x_1, \ldots, x_m \rangle$ corresponds to a real item, the client modifies it to $x = \langle x_1, \ldots, x_m, 0 \rangle$. To form a fake item y, the client sets its elements to

$\langle y_1, \ldots, y_m, m+1\rangle$, where bits $y_1, \ldots, y_m$ are chosen at random or according to any other desired distribution. The server will be unable to distinguish two types of values because the elements are not accessible to the server in the clear.

The above means that the distances between two real items remain unmodified, but all distances between a real and fake items will now lie in the range $[m+1, 2m+1]$. Also, if during the computation of the Hamming distance we directly apply the XOR to a pair of elements, then the distance between two fake items will lie in the range $[0, m]$. In our case, however, secure computation of the XOR operation is performed using arithmetic operations, where $a+b-2ab$ is used to implement $a \oplus b$. Therefore, the distance between two fake items will lie in the range $[-2(m+1)m, -2(m+1)m+m]$. We can setup the secure computation over non-negative integers in such a way that the range $[-2(m+1)m, -2(m+1)m+m]$ has no overlap with $[0, 2m+1]$.

This allows us to verify the correctness of server's computation using the number of computed distances that fall in each range. In particular, the server compiles the distance distribution data by comparing each computed distance to $2m+2$ values from 0 to $2m+1$ and incrementing the count for the distance that matched. The client's verification consists of adding the counts corresponding to the distances in the range $[0, m]$ and the counts corresponding to the distances in the range $[m+1, 2m+1]$. Afterwards, the client compares the aggregate counts to their expected values (i.e., $(n-n_1)(n-n_2)$ and $(n-n_1)n_2 + n_1(n-n_2)$, respectively), and the server's computation is considered correct if both of them match.

Now suppose that the server computes pn^2 distances and would like to avoid being detected. In this case, instead of trying to guess the locations of fake items, the server can simply return $2m+2$ counts, such that counts c_0 through c_m add to $(n-n_1)(n-n_2)$ and counts c_{m+1} through c_{2m+1} add to $(n-n_1)n_2 + n_1(n-n_2)$. Because it is reasonable to assume that the server knows (or can guess sufficiently well) the values of n_1 and n_2, which are set by the client to achieve a certain level of security, the server can always be successful in avoiding the detection. This means that a different solution is needed.

4.2 Improved solution

To improve the security properties of the above solution, we employ two ideas: (i) (protected) distances used for computing statistics are given to the server in randomized order and (ii) the client verifies a larger number of aggregate counts. By itself, the first modification still results in insufficiently high detection probability, but in combination with the second it leads to the client's ability to achieve a desired level of protection.

In detail, both client's procedures for preparing the data and verifying the computation change. Before the computation is sent to the server, the client adds extra k elements to each real item (with m original elements). The $m+k$ elements are randomly permuted, but consistently across all items. Let $i_1, \ldots, i_k$ denote positions of extra elements. All artificial elements are set to 0 in real items.

To form fake items, the client first chooses a small integer ℓ, which will be used as a security parameter. The client next chooses ℓ values larger than m; each value will be used to increase the distance between certain fake items and real items. For concreteness, and without loss of generality, let these values be $m+1, \ldots, m+\ell$. Then to create a fake item, the client first chooses a distance d at random from the range $[m+1, m+\ell]$. Next, the client chooses k values $d_1, \ldots, d_k$ such that $\sum_{i=1}^{k} d_i = d$, and sets the element at position i_j in the fake item to d_j for $j = 1, \ldots, k$ and all original elements to 0. This setup will imply that the distance between this fake item and any real item will be in the range $[d, d+m]$, and the distances between any fake item and any real time will always be in the range $[m+1, 2m+\ell]$. The client also records the number of times each d was used

Input:

- the range of distances $[l_r, u_r]$ between two real items in data structure C;
- the range of distances $[l_f, u_f]$ between real and fake items in data structure C;
- the maximum distance d_o that the original elements of real items contribute to the distance between real and fake items.

Algorithm steps:

1. The client adds the counts corresponding to the distances between l_r and u_r using $C[l_r]$ through $C[u_r]$ and compares the result to $(n - n_1)(n - n_2)$.
2. The client computes the expected distribution of the distances between real and fake items as follows: for each $i = l_f, \ldots, u_f$, the client sets $c_i = \sum_{j=\max(l_f, i-d_o)}^{\min(u_f-l_f+1, i)} (c_j^1 s_{i-j}^2 + c_j^2 s_{i-j}^1)$. The client then compares each c_i with $C[i]$.

Figure 1: Description of client's computation of the expected statistics.

in a fake item in the set S_1 and the set S_2, respectively (both formed in the above described way). Let the counts be denoted c_i^j, where $i \in [m+1, m+\ell]$ and $j \in [1, 2]$.

Before the client will be able to verify the computation performed by the server using sets S_1 and S_2, the client needs to compute additional information as follows: For each real item in S_1, the client computes the number of bits sets to 1 in that biometric (i.e., its Hamming weight) and counts the number of instances of each distance across all real items. Let $s_0^1, \ldots, s_m^1$ denote the distribution of the distances, where s_i^1 indicates the number of real items in S_1 with the Hamming weight of i. Similarly, the client produces similar information for the real items in S_2, and we denote the computed values by $s_0^2, \ldots, s_m^2$.

After the server receives sets S_1 and S_2 from the client, it computes the distances between all pairs. Afterwards, when the server produces statistics, it will need to compare each computed distance to the values in the range $[0, 2m+\ell]$. Since the comparisons are performed obliviously without the server knowing to what value a distance is being compared, the client randomizes the order in which all possible distances are communicated to the server. This means that the server will not be able to know what positions within the set of $2m+\ell$ values correspond to original distances from 0 to m and which correspond to artificial distances above m.

After the server returns statistics data to the client, the client performs the following:

1. The client adds the counts corresponding to distances between 0 and m and compares the result to $(n - n_1)(n - n_2)$.
2. The client computes the expected distribution of the distances above m as follows: for each $i = m+1, \ldots, 2m+\ell$, set $c_i = \sum_{j=\max(m+1, i-m)}^{\min(m+\ell, i)} (c_j^1 s_{i-j}^2 + c_j^2 s_{i-j}^1)$. The above represents the number of distances contributed by the intersection of fake items in S_1 with real items in S_2 and fake items in S_2 with real items in S_1. For example, $c_{m+1} = c_{m+1}^1 s_0^2 + c_{m+1}^2 s_0^1$, $c_{m+2} = c_{m+1}^1 s_1^2 + c_{m+2}^1 s_0^2 + c_{m+1}^2 s_1^1 + c_{m+2}^2 s_0^1$, etc. The client compares each c_i with the value returned by the server.

If all above checks succeed, the client treats the obtained statistics data as correct. We generalize the above procedure (which also will be used for other distance metrics) in the algorithm in Figure 1. We use notation $C[i]$ to denote the count returned by the servers for distance i. For the current solution, we use $[l_r, l_u] = [0, m]$, $[l_f, u_f] = [m+1, 2m+\ell]$, and $d_o = m$. Note that the last parameter d_o indicates that the original elements of real items can contribute the distance between 0 and m to the distances between real and fake items.

In order for distances between two fake items not to interfere with the counts being verified, we want to ensure that $\mathsf{dist}(x, y)$ between fake x and y are outside of the range $[0, 2m+\ell]$ and suggest the following: Because in secure computation modular arithmetic is normally used, the client chooses d_i's uniformly at random from $\mathbb{Z}_q$, when the arithmetic is carried out modulo q, while maintaining $\sum_{i=1}^{k} d_i \bmod q = d$. When $q \gg 2m + \ell$, with high probability all $n_1 n_2$ distances between two fake items will fall outside of the range $[0, 2m+\ell]$. If, however, $\mathsf{dist}(x, y)$ happens to be in $[0, 2m + \ell]$ for some x and y, the client can choose a different set of d_i's for x or y. We, however, allow $\mathsf{dist}(x, y)$ to be in $[0, 2m + \ell]$, in which case the client needs to adjust the counts it expects from the server and also needs to compensate for the error when using statistical data computed by the server. This incurs minimal overhead on the client, but allows to keep q low without increasing the server's load.

4.3 Analysis of misbehavior detection

We next analyze the server's success in avoiding being detected when it performs the fraction p of its task. (Now this fraction corresponds to server's work for computing distances and corresponding statistics.) We treat two options when (i) the server computes all distances between real and fake items correctly (by guessing the locations of fake items) and increases the count for distances between real and real values, and (ii) the server does not compute all distances between real and fake items and needs to increase counts both for distances between real and real and distances between real and fake items. In all cases we assume that the server knows (or can approximate well) the values of security parameters n_1, n_2, ℓ, and it also knows m.

Correct computation of statistics for distances between real and fake items. The server's goal is to identify locations of items that correspond to fake items (both for columns and rows) and compute their distance to real items, as the client distinguishes between distances between real-real and real-fake items. Suppose that, to maximize the coverage of fake items, the server computes $n - n_1$ cells in selected columns and $n - n_2$ cells in selected rows, where the total number of cells computed is pn^2. As before, let $n_1 = n_2$, in which case we obtain $pn^2 = (n - n_1)t$, where $t = t_1 + t_2$, t_1 is the number of computed rows, and t_2 is the number of computed columns. Let $t_1 = t_2$ since this is the best for the server strategy when $n_1 = n_2$. Now, given the value of p, we can compute $t_1 = t_2$ and then compute the probability that all n_1 rows/columns will appear among the computed t_1.

Let $p' = t_1/n = (np)/(2(n - n_1))$. The probability that the server computes rows that include all n_1 inserted rows is $1 - (p')^{n_1}$ and the probability that the server computes columns that include all n_1 inserted columns is also $1 - (p')^{n_1}$. Furthermore, the server's behavior can be undetectable only when it computed all distances between real and fake items and also guessed a location of a cell corresponding to a value between 0 and m among the statistics counts (which it needs to increment); the probability of its success in that case is $(m + 1)/(2m + 1 + \ell)$. We therefore obtain:

$$\Pr[\mathsf{D}] = 1 - \frac{m+1}{2m+1+\ell}(p')^{n_1+n_2} = 1 - \frac{m+1}{2m+1+\ell}\left(\frac{np}{2(n-n_1)}\right)^{2n_1} \tag{3}$$

Now notice that the server might also be able to compute correct distances between the items using only a fraction of elements during the computation of each distance. In particular, recall that each distance is computed over $m + k$ elements, where m of them correspond to the original data in real items, and the server can compute n^2 partial distance using $p(m + k)$ elements. When computing a distance between two real items, using any number of elements from 0 to $m + k$ will result in the outcome falling into the correct range $[0, m]$. For computing distances between real and fake values, however, first notice that all artificial k elements must be used in order to pass the client's check. That is, all k artificial elements should fall within $p(m+k)$ elements used by server. Because k can be comparable to m, we need to assume that the elements are drawn without replacement,

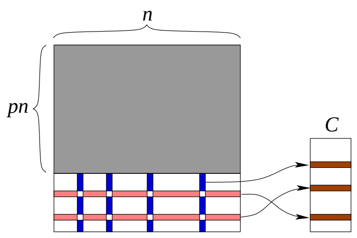


Figure 2: Illustration of incomplete computation while creating statistical data.

and using the properties of hyper-geometric distribution we obtain the probability of including all k elements in $p(m+k)$ elements:

$$\Pr[\overline{\mathsf{D}}] = \binom{m}{p(m+k)-k} \Big/ \binom{m+k}{p(m+k)} = \prod_{i=0}^{k-1} \frac{p(m+k)-i}{m+k-i} \tag{4}$$

Second, in order to pass the client's verification, the server needs to obtain the exact distances between real and fake items, which means that it also needs to have the elements with the value of 1 to be within the computed elements. Assuming that at least $\beta \cdot m$ of the original elements are set to 1 in each biometric and the elements are randomly (and consistently) permuted to eliminate any patterns, the above probability decreases to:

$$\Pr[\overline{\mathsf{D}}] \;=\; \binom{m-\beta m}{p(m+k)-\beta m-k} \Big/ \binom{m+k}{p(m+k)} = \prod_{i=0}^{\beta m+k-1} \frac{p(m+k)-i}{m+k-i} \tag{5}$$

The above assumes that $p(m+k) \geq \beta m + k$; otherwise, $\Pr[\overline{\mathsf{D}}] = 0$. We also note that while equation 3 tells us the minimum value n_1 should be set to for a desired bound on p and desired probability of detecting cheating, equations 4 and 5 tell us the minimum value the security parameter k should be set to for a given p and $\Pr[\mathsf{D}]$.

Incomplete computation of statistics for distances between real and fake items. Now suppose that the server does not try to guess the locations of fake items, but rather attempts to adjust statistics information to avoid detection. In this case, the server might have higher chances of being undetected (for the same p) and its success depends on the values of m and ℓ, as well as distribution of distances between real and fake items.

Because there is a structure to distances in the matrix, we first examine the case when the server fully computes distances and statistics information for a number pn of rows and modifies the statistics information to compensate for the remaining distances. First, note that if the server computes pn rows correctly, it will need to guess (i) the number of fake rows among the remaining rows (red rows in Figure 2), (ii) the locations of the counts in C associated with distances between each of uncomputed fake item in S_1 and real items in S_2 (i.e., locations in C associated with red cells in Figure 2), (iii) the locations of the counts in C associated with the distances between n_2 fake items in S_2 and uncomputed real items in S_1 (locations in C associated with blue columns in Figure 2), and (iv) at least one location in C corresponding to a distance between 0 and m (for increasing the count corresponding to white cells in Figure 2). Next, because the server might be able to estimate (i) with good accuracy from the values of pn and n_1 and the probability of guessing (iv) is high, we have to rely on (ii) and/or (iii) being difficult. Suppose that the server guessed (i) correctly, and its value is $t = (1-p)n_1$. A naive strategy for the server would be to guess one location in C for a distance in the range $[0, m]$ and increment the corresponding count by $((1-p)n - t)(n - n_2)$ (thus covering all white cells) and then guess the locations in C corresponding to all red and blue cells correctly and update the counts. We, however, notice that the server can reduce the amount of information that needs to be guessed by reusing information associated with some cells. For instance, the server can take one of the computed rows (which with high probability corresponds to a real item in S_1) and copy it $(1-p)n-t$ times into uncomputed rows (or just replacing the value of $[b_k]$ with $((1-p)n-t+1)[b_k]$ when updating the counts for that row). This would remove the need for the server to guess the locations in C associated with the blue cells in Figure 2. Alternatively, the server can compute the distances in the first cell of each row (which with high probability correspond to a real item in S_2) and copy the distance contained in that cell to all distances in the row. This will remove the need to guess locations of the distances contained in the red cells, but all locations associated with the blue cells need to be guessed, as well as at least one count associated with a distance in the range $[0, m]$ needs to be decremented to compensate for incorrectly updated blue cells distances. The previous, row copying, strategy appears to be superior for the server in that it minimizes the number of values to be guessed. In particular, for each fake item x_i in S_1 or S_2, the distance between that item and a real item y_i in S_2 or S_1, respectively, will be equal to $s_i + s_j$, where s_i is the sum of the elements of x_i and s_j is the sum of the elements in y_j. In the worst for the client case, all original biometrics have the same number of 1's in them, which means that all distances between a single fake element and real elements will have the same value. This also means that all distances corresponding to the cells in a single blue column or a single red row will be stored in a single location in C. And, in order to update statistics information about an uncomputed row or column, the server will need to guess a single location in C. Therefore, our goal is to set the security parameters in such a way that, regardless of what strategy the server employs, it will have to guess enough locations in C so that its probability of success is sufficiently low.

Because the server's success is maximized when it has to guess only the locations in C associated with the distances in red rows, we perform the analysis based on this case. In what follows, let γ be a confidence security parameter, which will guide the values of other security parameters to ensure that the client's security guarantees hold with probability at least $1-\gamma$. Now let us look at a hyper-geometric experiment that models the number of fake (red) rows falling within the uncomputed region of $(1-p)n$ rows. If we denote a random variable that follows this distribution by X and its outcome by t, we obtain: $\Pr[X = t] = \binom{(1-p)n}{t}\binom{pn}{n_1-t}/\binom{n}{n_1}$. Because the client will want the outcome of X to be above a certain threshold, we change this equation to the following:

$$\Pr[X \geq t] = \sum_{i=t}^{n_1} \binom{(1-p)n}{i}\binom{pn}{n_1-i}/\binom{n}{n_1} \geq 1-\gamma \tag{6}$$

Then given the value of γ and t, the client will be able to compute the necessary value of parameter n_1 from equation 6. To determine the value of t, we first need to compute how many distinct locations in C the server must guess to achieve the desired probability of cheating detection $\Pr[\mathsf{D}]$, after which we will use that value, s, to compute t.

The server can guess s out of $2m+\ell+1$ locations in C correctly with probability:

$$\Pr[\overline{\mathsf{D}}] = \frac{s}{2m+1+\ell} \cdot \frac{s-1}{2m+\ell} \cdots \frac{1}{2m+1+\ell-(s-1)} = 1/\binom{2m+1+\ell}{s}. \tag{7}$$

Therefore, given $\Pr[\overline{\mathsf{D}}] = 1 - \Pr[\mathsf{D}]$, the client computes s and will need to set $\ell \geq s$. Next, we determine how many fake items from S_1 (red rows) need to fall within the uncomputed region to result in s distinct distance values (where a row is conservatively assumed to contribute one distance) with desired probability of success. That is, we express the probability that t (fake) items, each with a randomly chosen distance d in the range $[1, \ell]$, will result in s out of ℓ distinct values. By increasing the value of t, the success probability increases, and the client will choose the minimum t that gives probability at least $1-\gamma$.

In what follows, let X denote a random variable associated with an experiment of throwing t balls into ℓ bins and follow the distributions of the number of bins with at least one ball in them. First notice that when $t = s$, each ball must land in a new bin:

$$\Pr[X \geq s] = 1 \cdot \frac{\ell-1}{\ell} \cdot \frac{\ell-2}{\ell} \cdots \frac{\ell-s+1}{\ell} = \frac{\prod_{i=1}^{s-1}(\ell-i)}{\ell^{s-1}}. \tag{8}$$

When $t = s+1$, the first s balls can still land in unique bins, but now one ball can also land in a previously occupied bin. We therefore obtain:

$$\begin{aligned}\Pr[X \geq s] &= 1 \cdot \frac{\ell-1}{\ell} \cdot \frac{\ell-2}{\ell} \cdots \frac{\ell-s+1}{\ell} \cdot 1 + 1 \cdot \frac{1}{\ell} \cdot \frac{\ell-1}{\ell} \cdots \frac{\ell-s+1}{\ell} + \\ &+ 1 \cdot \frac{\ell-1}{\ell} \cdot \frac{2}{\ell} \cdot \frac{\ell-2}{\ell} \cdots \frac{\ell-s+1}{\ell} + \cdots + 1 \cdot \frac{\ell-1}{\ell} \cdots \frac{\ell-s+2}{\ell} \cdot \frac{s-1}{\ell} \cdot \frac{\ell-s+1}{\ell} = \\ &= \frac{\prod_{i=1}^{s-1}(\ell-i)}{\ell^{s-1}} \left(1 + \sum_{i=1}^{s-1} \frac{i}{\ell}\right)\end{aligned} \tag{9}$$

Similarly, for $t = s+2$, we have:

$$\Pr[X \geq s] = \frac{\prod_{i=1}^{s-1}(\ell-i)}{\ell^{s-1}} \left(1 + \frac{1}{\ell}\sum_{i=1}^{s-1} i + \frac{1}{\ell^2}\sum_{i=1}^{s-1}\sum_{j=1}^{s-1}(i \cdot j)\right). \tag{10}$$

Therefore, by increasing the value of t, the client will find a value that satisfies $\Pr[X \geq s] \geq 1-\gamma$ and will further use that value of t to compute the value of n_1 using equation 6. Note that by increasing the value of parameter ℓ, the value of n_1 will be reduced.

Now consider the possibility that the server computes partial distances using $p(m+k)$ elements for all cells and then corresponding statistics using the computed distances. We next show that if in doing so the server misses at least one artificial element, its probability in avoiding detection will be low (and can be controlled by appropriately setting the parameter ℓ). Thus, the client should set the value of k so that

$$1 - \binom{m}{(1-p)(m+k)} \Big/ \binom{m+k}{(1-p)(m+k)} = 1 - \frac{m!(p(m+k))!}{(m+k)!(p(m+k)-k)!} \geq 1-\gamma. \tag{11}$$

This expression corresponds to the probability that at least one of the k elements will fall within the skipped $(1-p)(m+k)$ elements.

Now suppose that at least one artificial element was not used during distance computation. This means that the computed distances between two real items will be in the correct range and will pass verification. Similarly, the distances between two fake items with high probability will fall outside of the checked range, as intended. The distances between real and fake items, however, with high probability will also fall outside of the checked range and the counts corresponding to their true values will need to be updated. (Note that if some partial distances between fake and real items fall within the checked range, the server will have to additionally guess the locations of the counts in which erroneous distances fell and decrease the counts. We therefore conservatively assume that computed distances between fake and real items fall outside the checked range, and the client's detection probability will be at least as large as the computed value.) If the server knows the distribution of the number of original elements set to 1 in a biometric and sufficiently well estimates the number of fake items with any given sum of its elements d, the server will be able to update the counts correctly if it can guess the location of the appropriate counts in C. This is, however, difficult because a large number of locations needs to be guesses. Furthermore, for each unique count difference, the exact locations in C must be determined. In the worst for the client case, the server needs to update all locations with the same information, in which case the server has to guess $u + \ell - 1$ locations (out of $2m + \ell + 1$) in C, where $u = \max_i(||x_i||) - \min_i(||x_i||) + 1$ corresponds to the difference between the smallest and the largest number of original elements set to 1 in a biometric. Because in the best for the server case, it does not need to distinguish between the $u + \ell - 1$ locations, the detection probability is:

$$\Pr[\mathsf{D}] \geq 1 - \prod_{i=0}^{u+\ell-2} \frac{u + \ell - 1 - i}{2m + \ell + 1 - i} = 1 - \prod_{i=1}^{u+\ell-1} \frac{i}{2m - u + 2 + i}. \tag{12}$$

If for a particular biometric representation u is small, the value of ℓ can be increased to achieve the desired $\Pr[\mathsf{D}]$.

4.4 Combining verification of distances and statistics

When the client needs to verify computation of both the distances and statistics, it can combine the techniques of Sections 3 and 4.2 as follows: The client adds fake items and artificial elements to both real and fake items as described in Section 4.2 with the difference that, instead of setting the original elements of fake items to 0, they are set according to the description in Section 3 (to maintain uniform distribution of the distances for the cells being checked by the client). Then to verify the computation of the distances (i.e., cells of the matrix), the client as before checks $n_1 = n_2$ distances between pairs of fake items. The verification obviously needs to take into account the offset introduced by the values assigned to artificial elements. To verify computed statistics data, the client proceeds as before, with the exception that now to produce expected counts for the distances in the range $[m + 1, 2m + \ell]$ the client needs to perform $(n_1 + n_2)nm$ instead of $nm + \ell$ work. The distances between real and fake items will, as before, fall into the range $[m + 1, 2m + \ell]$ and the distances between two fake items will, as before, fall outside of the range $[0, 2m + \ell]$ with high probability. The computation is summarized in Figure 3. Steps 1, 2, and 5 of task creation are performed offline once for all tasks. Step 1 of task verification can be performed during task execution.

To compute the security parameters n_1, k, and ℓ, we can use the previously established bounds. In particular, the client's success probability in detecting cheating in computing the distances is at least the probability in equation 2, which on input κ gives us the value of $n_1 = n_2$. To detect cheating in computing statistics with the desired probability, the client computes (i) n_1 using equation 3, (ii) k using equation 5, (iii) n_1 and ℓ using equation 6 and consecutive analysis (equations 7 through 10), (iv) k using equation 11, and (v) ℓ using equation 12. For all of n_1, k, and ℓ the highest computed value should be used.

4.5 Implementation

To evaluate the performance of our solution, we conducted experiments by implementing secure and verifiable AllPairs (Section 3) and Analyze (Section 4.2) functionalities for the Hamming distance. We first present the results for AllPairs distance computation outsourcing.

To securely outsource a task, the client sends shares of S_1 and S_2 to a number of computational servers N who jointly compute the result using secure multi-party computation (SMC) techniques based on Shamir secret sharing [18]. Due to simplicity of function $\mathsf{dist}(x, y) = \sum_{i=1}^{m}(x_i + y_i - 2x_iy_i)$, Given p, κ, and γ, compute $n_1 = n_2$, k, and ℓ.

To create an outsourced task:

1. Create n_1 pairs of m-element fake items $\langle x_i, y_i \rangle$ so that $\mathsf{dist}(x_i, y_i)$ is a randomly chosen value from $[0, m]$ for $i = 1, \ldots, n_1$.
2. Append k elements to each fake item, where the element values are chosen as follows: for each fake item, choose d at random from $[m+1, m+\ell]$ and set the values of the artificial elements to randomly chosen d_j's such that $\sum_{j=1}^{k} d_j = d$. Compute $\mathsf{dist}(x_i, y_{i'})$ for each i and i'; if some value falls in the range $[0, 2m+\ell]$, choose new d_j's for x_i or $y_{i'}$. Store n_1 values $\mathsf{dist}(x_i, y_i)$. Permute the elements in each item using a random permutation π_0.
3. Given two sets S'_1 and S'_2 of size $n - n_1$ comprised of m-element biometrics, append k artificial elements to each item, which are set to 0. Permute the elements in each item using the same permutation π_0.
4. Form S_1 of size n by appending the x_i's to the set S'_1 and permuting the result using random permutation π_1. Similarly, form S_2 using S'_2 and the y_i's and permuting the resulting set using random permutation π_2. Save S_1 and S_2.
5. Create (protected) distances $[i]$ for $i = 0, \ldots, 2m+\ell$, permute this set using random permutation π_3.
6. Send S_1, S_2, and $\pi_3([0]), \ldots, \pi_3([2m+\ell])$ to a server for secure $\mathsf{AllPairs}$ and $\mathsf{Analyze}$ computation, where the distances $\pi_3([0]), \ldots, \pi_3([2m+\ell])$ will be used to compute C.

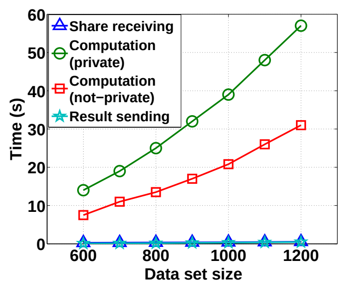
To verify an outsourced task:

1. Compute expected C as follows:
   (a) for $i = m+1, \ldots, 2m+\ell$ set $C[i] = 0$ and also set $C[0\text{–}m] = (n - n_1)^2$.
   (b) for $i = 1, \ldots, n_1$, compute $w_1 = \mathsf{dist}(x_i, z)$ and increment $C[w_1]$ by 1 for each $z \in S'_1$; compute $w_2 = \mathsf{dist}(z, y_i)$ and increment $C[w_2]$ by 1 for each $z \in S'_2$.
2. For each $i = 1, \ldots, n_1$ compare stored $\mathsf{dist}(x_i, y_i)$ to returned $M[\pi_1(x_i)][\pi_2(y_i)]$ and output failure if the values disagree.
3. Add returned counts $C[\pi_3^{-1}(0)]$ through $C[\pi_3^{-1}(m)]$ and compare them to computed $C[0\text{–}m]$. For each $i = m+1, \ldots, 2m+\ell$ compare returned $C[\pi_3^{-1}(i)]$ to the expected $C[i]$. If at least one check fails, output failure.

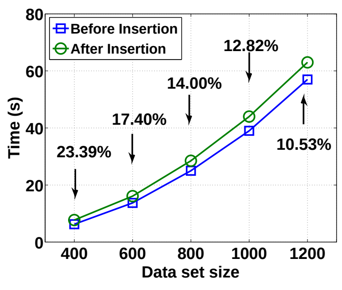
Figure 3: Description of preparation and verification procedures for an outsourced task.

where x_i denotes the ith element of x, and properties of this type of SMC techniques, each server can compute its share of $\mathsf{dist}(x, y)$ locally and return the result to the client. In particular, with (t, N)-linear secret sharing, a secret is split in N shares, where any $t+1$ shares can be used to reconstruct it, while combining t or less shares information-theoretically reveals no information about the secret. For parties that follow the computation, it is required that $t < N/2$. Each secret s is represented by a random polynomial $f_s(x)$ of degree t, where $f_s(0) = s$. Party P_i, for $i = 1, \ldots, N$. obtains its share $f_s(i)$. With this representation, addition, multiplication, or, more generally, any linear combination of secret shared values is carried out locally with no communication. Multiplication of two secret shared values, on the other hand, is interactive and involves multiplying two shares locally, after which they are randomized and re-shared. This temporarily raises the degree of the polynomial representation to $2t$, after which it is reduced back to t, and this is the reason for $2t < N$ requirement. It is important to notice that a (possibly multi-variate) polynomial of degree k can also be evaluated locally, as long as $kt < N$. This fact is exploited in our implementation, where $\mathsf{dist}(x, y)$ computation is represented as a polynomial of degree 2 over variables x_i, y_i for $i = 1, \ldots, m$ and $t < N/2$. This means that the servers compute the shares of $\mathsf{dist}(x, y)$ without any interaction (and the result is represented by a polynomial of degree $2t$), and the client uses $N = 2t + 1$ shares it receives from the servers to reconstruct the result.

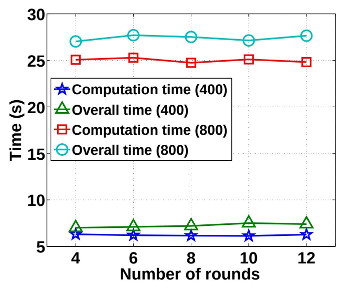
Our implementation used Java-based SEPIA library [7] for the underlying communication and elementary operations on secret shares with one client and three computational servers for a singe task. In particular, SEPIA handled establishment of SSL connections between the client and the



(a) task computation and communication time



(b) server's task computation time



(c) server's performance for a large task

Figure 4: Performance of secure and verifiable outsourcing of Hamming distance computations.

servers and between each pair of servers. We used $m = 1000$ with a varying number n of items per task and arithmetic modulo a 15-bit prime. The value of m was chosen to be comparable with the length of iris code representation.[1] The value of n was chosen so that n^2m shares would fit the server's memory for performance reasons (and if the client's task is of a larger size, it will be partitioned into sub-tasks of appropriate size). The client and each server were 2.4 GHz Linux machines with 12 GB of memory on a 1Gbps LAN. We used (1, 3) Shamir secret sharing with $N = 3$ servers.

Figure 4 reports the results of our experiments. Figure 4(a) measures the time to transmit $2nm$ shares to a single server, the computation taken by a single server to compute shares of n^2 distances, and the time for a server to send n^2 shares back to the client. For comparison, we also plot the time taken to execute a task without privacy protection in a similar environment on shorter representations of the elements and implementing XOR operations directly. (Note that in secure execution, a single task is run by N machines, while in insecure computation only a single server is needed.) The plot shows notable for secure function evaluation performance. Also, while our LAN experiment represents a best-case scenario for communication delay, communication is not expected to be a dominating factor even for significantly slower networks.

Figure 4(b) reports on the servers' overhead caused by the addition of fake items to make the computation verifiable. The curves show the time to compute tasks of size n and $n + n_1$ for fixed $n_1 = 50$ and variable n. We compute n_1 according to equation 2. For a sample setup of $\Pr[\mathsf{D}] \geq 0.99$ when $p \leq 0.95$, we obtain $n_1 = 90$, and for $\Pr[\mathsf{D}] \geq 0.95$ with $p \leq 0.9$ we obtain $n_1 = 29$; we then choose $n_1 = 50$ as a medium value. Because the overhead consists of $2nn_1 + n_1^2$ distance computations, it is clear that it will constitute a smaller fraction of the task as the value of n increases.

Finally, Figure 4(c) reports on time to securely compute a very large task. The plot shows average time per sub-task when the overall task is computed using sub-tasks of size $n = 400$ and $n = 800$. It is clear from the figure that the time per sub-task is constant. The overall time is slightly higher than the computation time and includes overhead such as key establishment and also depends on the worst out of N (vs. average) communication and computation. From all of the plots, we see that the techniques are efficient and incur a modest overhead.

To evaluate the performance of secure and verifiable statistics computation $\mathsf{Analyze}$, we start with computing all necessary parameters. As summarized at the end of Section 4.4, we need to compute the parameters n_1, k, and ℓ using $m = 1000$, variable n, and desired values for $\Pr[\mathsf{D}]$, γ, and p. First, notice that in the five parts for computing these parameters, computation of k

[1]For the purposes of analysis in this work, the exact value of m plays insignificant role assuming that it is large enough. This will become clear from the discussion below.

Security setting	Computed parameters							
	k	ℓ	n_1					
	any n	any n	$n = 200$	$n = 400$	$n = 600$	$n = 800$	$n = 1000$	$n = 2000$
$p \le 0.9$, $\Pr[\mathsf{D}] \ge 0.95$, $\gamma \le 0.05$	28	1	27	28	28	28	29	29
$p \le 0.95$, $\Pr[\mathsf{D}] \ge 0.95$, $\gamma \le 0.05$	57	1	51	55	56	57	57	58
$p \le 0.95$, $\Pr[\mathsf{D}] \ge 0.99$, $\gamma \le 0.01$	87	1	73	81	84	85	86	88

Table 1: Values of parameters k, ℓ, and n_1 for verification of Hamming distance-based statistics with $m = 1000$.

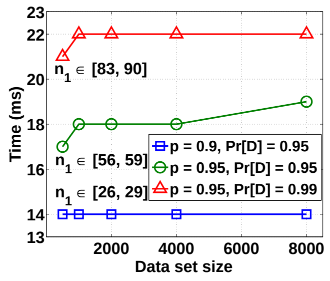
(parts (ii) and (iv)) is independent of either n_1 or ℓ. Furthermore, equation 11 will result in strictly higher value of k than equation 5 even for very conservative values of β. Therefore, it is sufficient to consider only equation 11 and compute k as a function of p, m, and γ. We also note the value of m has a low impact on the value of k (i.e., similar to computing the number of artificial items n_1, approximation by binomial distribution can be used to compute the number of artificial elements k when m is large, in which can k is independent of m). We compute the value of k and other parameters for three distinct settings of security parameters $\Pr[\mathsf{D}]$, γ, and p, which are given in Table 1.

Next, we consider parameters ℓ and n_1 computed in parts (i), (iii), and (v). First, notice that the value $\ell = 1$ satisfies all analyses with $m = 1000$ (namely, the probability in equation 12 is very high even with the lowest $u = 1$, the probability in equation 7 is very low with $\ell = s = 1$, and the probability in equation 8 is 1 with $\ell = s = t = 1$). This gives us $t = 1$ for the purposes of equation 6, and all that remains is to compute the higher value of n_1 using equations 3 and 6. Equation 3 gives us a very low n_1 for any set of parameters, and we therefore use the higher value from equation 6. Such values of n_1 are shown in Table 1 as a function of data set size n. As can be seen from the table, the value of n_1 increases slowly with the value of n and approaches data set independent values, which were computed for the purpose of AllPairs computation verification experiments. Also note that the values of k with $m = 1000$ are very similar to values of n_1 with $n = 1000$.

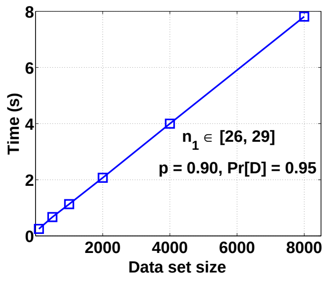
The results of our experiments are given in Figures 5 and 6, where the former reports on the client's computation and communication overhead and the latter shows the server's performance. For the client, we measured all components necessary for preparation and verification of an outsourced task. Figure 5(a) measures the client's preparation time for generating the fake items to be inserted into a data set. This is a one-time cost for all possible tasks to be outsourced. We present the client's overhead for the three security settings in Table 1 which differ by the values of supported p and guaranteed $\Pr[\mathsf{D}]$ and γ. Note that the time is very low and grows slowly with the size of the task, as the value of n_1 increases before reaching the ceiling for large n.

Figure 5(b) reports on the client's time for preparing a task for outsourcing. It includes reading the input data from a locally stored file and inserting the fake items into (random) predefined locations of the data set, where the former amounts to the majority of that time. The curve in the plot corresponds to the security setting with $p = 0.9$, $\Pr[\mathsf{D}] = 0.95$, and $\gamma = 0.95$ and therefore n_1 in the range $[26, 29]$ increasing with the value of n. It is clear from the plot that the time is linear in the data set size, and was the same in our experiments for the three security setting (i.e., the value of n_1 does not play a major role in the task preparation time).

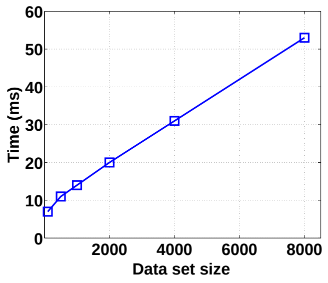
Figure 5(c) reports on the client's overhead for computing the expected statistics to be used during verification and which can be carried out while the servers are working on the task. Recall that to compute the expected statistics, the client needs to compute the Hamming weight of each (real) biometric in its data sets, which necessitates a single round of traversing of all of the items



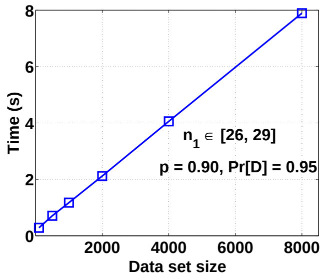
(a) one-time precomputation



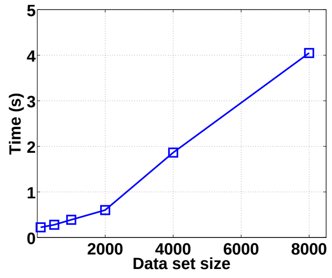
(b) task preparation time



(c) expected statistics computation



(d) task preparation and verification time



(e) communication overhead

Figure 5: Client's performance of secure and verifiable outsourcing of Hamming distance-based statistics computation and communication.

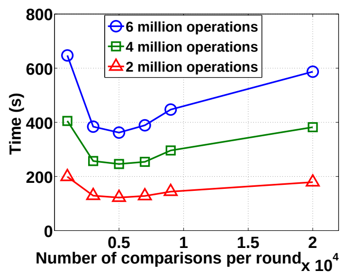
(which are already in the memory after task preparation). Thus, the time in Figure 5(c) grows linearly in the size of data set. Once again, we report the results for the first security setting in Table 1 and performance for other security settings is the same.

The next Figure 5(d) presents the client's overall time for preparing a task and verifying the server's computation, which combines the times in Figures 5(a)–(c) with task verification time. The task verification time includes (i) reconstruction of the secret shares received from the servers and (ii) comparing the result with the expected statistics. Note that both (i) and (ii) depend only on the size of C and are independent of the data set size n. Therefore, we observed a constant task verification time around 15 msec. From the figures, we can conclude that the client's overhead is dominated by the task preparation time, which is linear in n and takes on the order of a couple of seconds for data sets of a few thousands of items.

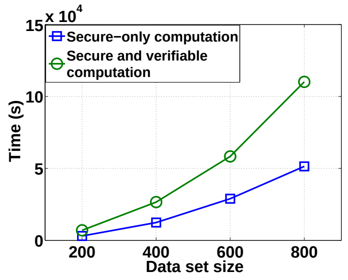
Finally, Figure 5(e) measures the time that communication between the client and the servers takes, which consists of sending shares of the data sets and receiving shares of the resulting C. Since the size of C is independent of n, the time for receiving shares of the result also remains constant around 10–15 msec. Thus, the communication overhead primarily depends on the time spent transmitting shares of input data and therefore grows linearly with the data set size.

Unlike the distance computation for AllPairs functionality, in order to produce statistics data C the servers need to engage in interactive computation. In the implementation, we compute the distances the same approach as before, which results in each distance represented by a $2t$ degree polynomial. The servers then reduce the degree of each distance representation to t and engage in comparison operations as described in section 2. The fastest known realization of the equality testing in this framework is due to Catrina and de Hoogh [8], but because we build our prototype using the SEPIA library, our implementation relies on the equality testing available in SEPIA. This means that faster implementations are possible today.

The performance of any interactive secure multiparty protocol can be improved if the computation can be parallelized. In our context, all cells in the $n \times n$ matrix can be processed in parallel.



(a) performance with varying number of operations per synchronization round



(b) server's task execution time

Figure 6: Server's performance of secure and verifiable outsourcing of Hamming distance-based statistics computation.

In addition, all $|C| = 2m + \ell$ comparison operations per cell in the matrix can also be carried out in parallel. We therefore utilize the ability to carry out operations in parallel in SEPIA. SEPIA provides only a limited way to perform parallel computation, in that operations are performed in rounds[2] and all computation and communication within a single round are synchronized. That is, each server waits until it receives all of the intermediate results necessary for the next step for all operations within the same synchronization round, and then proceeds with the computation for the next step. In other words, synchronization throughout each round is mandatory even if it is not required by the computation itself, and the overall computation is performed in sequential rounds. We experimented with SEPIA to find out the number of operations per synchronization round that would minimize the servers' runtime. The experiments were performed using computation that involved a few million comparison operations, all of which could be carried out in parallel. Performing 10^7 comparisons is roughly equivalent to computing statistics C for $n = 100$ and $|C| = 1000$. The results of our experiments are given in Figure 6(a) for $N = 3$ servers. As can be seen from the figure, the computation time is the lowest when 5000 comparison operations are used per synchronization round. We, however, would like to note that this result is specific to the SEPIA library and faster performance can be achieved by using tools with more flexible parallelization options.

The time that it takes the servers to carry out an outsourced task is shown in Figure 6(b). We report the servers' runtime for two settings: secure computation of C and secure and verifiable computation of C. The former guarantees that the servers do not learn information about the data they handle, and the latter additionally uses the techniques from Section 4.2 to ensure that correctness of the computation can be verified by the client. We use the parameters for the first security setting with $p \leq 0.9$, $\Pr[\mathsf{D}] \geq 0.95$, and $\gamma \leq 0.05$ from Table 1. The overhead introduced by the addition of fake elements amounts to the fraction k/m for tasks of all sizes n. The overhead due to the addition of fake items amounts to the fraction n_1/n, which decreases as n grows. Finally, expanding the size of C from m to $2m + \ell$ doubles the work associated with processing each pair of items.

[2]This notion of a round differs from the traditional definition used in SMC that corresponds to elementary sequential interactive operations.

5 Verification of Statistics Computation for Euclidean Distance

In this section, we use the Euclidean distance as the distance metric for computing distances between each pair of items. As before, the client attempts to verify that the distribution computation was performed correctly by the server(s). This time, each item x is represented as an m-dimensional vector $\langle x_1, x_2, \ldots x_m \rangle$, where each element $x_i \in [0, t]$ for $1 \le i \le m$. Unlike the Hamming distance computation which is carried out following the formula exactly, in this metric, we have the server compute the distribution of squared distance and send the result back to the client. That is, we define $\mathsf{dist}(x, y) = \sum_{i=1}^{m} (x_i - y_i)^2$. The client will then either produce mapping between regular and squared distances as it forms a task assignment for the server (recall that the client supplies protected distances used to collect distribution information), takes the square root of each returned result, or operates directly on squared distances.

5.1 Preliminary solution

Our first solution is very similar to that used for the Hamming distance. To aid the verification process, the client, as before, inserts fake items into the computation and also inserts fake elements into the real and fake items. In detail, before the computation is sent the the server, the client inserts extra k fake elements into each real item. The resulting $m + k$ elements are randomly permuted, but consistently across all items. As before, we denote the positions of these extra elements by $i_1, \ldots, i_k$. All artificial elements are set to 0 in the real items.

To form fake items, the client first chooses a small integer ℓ as the security parameter. Then, the client chooses ℓ values larger than mt^2 with each of them being used to increase the distance between real and fake items. For concreteness, we set these values to $mt^2 + 1$, $mt^2 + 2$, $\ldots$, $mt^2 + \ell$. To form a fake item, the client first randomly chooses a distance d out of these ℓ candidates. Next, the client chooses k fake elements at random from $\mathbb{Z}_q$, denoted by d_i for $1 \le i \le k$, so that the constraint $\sum_{i=1}^{k} {d_i}^2 = d$ is satisfied. In more detail, the client chooses the first $k - 1$ values of d_i's uniformly at random from $\mathbb{Z}_q$, sets $(d_k)^2$ to $d - \sum_{i=1}^{k-1} {d_i}^2$, and computes d_k. For a prime q, there is about 50% probability that square root computation fails (i.e., $(q-1)/2$ values from $\mathbb{Z}_q$ are quadratic nonresidues). In this case, the client chooses a new d_{k-1} at random and tries again until d_k is successfully found. Finally, the client sets the remaining m (original) elements in that item to 0. To aid producing the expected statistics for computation verification purposes, the client records the number of times each d was used in a fake item in the set S_1 and S_2, respectively. Let the counts be denoted c_i^j, where $i \in [mt^2 + 1, mt^2 + \ell]$ and $j \in [1, 2]$.

By forming the items in S_1 and S_2 as described above, the distances between two real items will fall into the range $[0, mt^2]$, the distances between a real and a fake items will fall into the range $[mt^2 + 1, 2mt^2 + \ell]$, and the distances between two fake items could be anywhere in $\mathbb{Z}_q$. Because the range of distances between two fake items might now overlap with the range of distances between two real (or real and fake) items, for verification purposes, the client needs to precompute the distribution of distances between two fake items that fall into the range $[0, 2mt^2 + \ell]$, and subtract it from the statistics returned by the server. Notice that the fake items could be reused for each task without lowering the detection probability, thus the overhead associated with this computation should be treated as a one-time overhead.

Before the client is able to verify the computation performed by the server using S_1 and S_2, the client needs to compute additional information as follows: for each real item x in S_1 and S_2, the client computes the sum of squares of its elements $\sum_{i=1}^{m} x_i^2$, and counts the number of instances of each occurred value across all items. Let s_i^1 (s_i^2) denote count of the number of items with computed value i in S_1 (S_2, respectively). After acquiring this distribution, the client computes the expected statistics and verifies the results returned by the servers using the algorithm in Figure 1 with the following inputs:

- $[l_r, u_r] = [0, mt^2]$
- $[l_f, u_f] = [mt^2 + 1, 2mt^2 + \ell]$
- $d_o = mt^2$

If all the checks in the algorithm succeed, the client treats the obtained statistics data as correct.

Next, notice that if we now compute the security parameters using the security analyses in Section 4.3 for the Hamming distance with respect to the client's ability to detect skipped computation, the necessary security guarantees will hold. (The only obvious exception is that we replace the total number of distances $2m + \ell + 1$, i.e., the size of C, with $2mt^2 + \ell + 1$ in equations 3, 7 and 12.) In particular, for a given p, $\Pr[\mathsf{D}]$ and γ, the values of the parameters n_1, ℓ, and k are determined from equations 6 through 12. There are, however, two major differences from the setting for the Hamming distance that influence the values of these parameters that the client needs to compute:

1. The number of dimensions m in biometrics that rely on the Euclidean distance (such as faces) is significantly lower, normally not exceeding 50.

2. If the server misses at least a single fake item in the computation (either from S_1 or S_2), it will have to correctly adjust several counts in C. Recall that in the case of the Hamming distance it was realistic to assume that all original biometric items might have the same Hamming weight (e.g., $m/2$), in which case all distances between a given fake item and all real items from the other set would be the same. In the case of the Euclidean distance, however, it is not realistic to assume that all $\sum_{m=1}^{m} x_i^2$ for each x would result in exactly the same value.

A direct consequence of item 2 above is that $u > 1$ for the purposes of equation 12, where u is the now the number of unique values $\sum_{i=1}^{m} x_i^2$ across all original x in S_1 or all original x in S_2. Furthermore, when the server needs to guess locations in C corresponding to s fake rows (red rows in Figure 2), the number of locations to guess now is at least $s + u$. This changes equation 7 to $\Pr[\overline{\mathsf{D}}] \leq 1/\binom{2mt^2+\ell+1}{s+u}$, which means that lower s and ℓ can be used in the case of the Euclidean distance than in the case of the Hamming distance for comparable sizes of C. This gives us that $\ell = 1$ will be sufficient for the Euclidean distance as well, and we obtain that the client will be able to use the same values of n_1 and ℓ as given in Section 4.5 and compute k from equation 11.

5.2 Improved solution

Now notice that, while the above solution meets the security goals, it can be become inefficient due to the large size of C and therefore a large number of comparisons per pair of items. In particular, this happens when m and/or t are not small. Under such circumstances, the client might be interested in learning aggregate statistics, where the computed distances are rounded with the desired precision or, more generally, the distances are placed in specific ranges and the aggregate count for the entire range is reported instead of individual counts for each distance in the range. While the client can clearly compute this information using the current solution, having the server compile the necessary information directly will result in significant reduction of its computational load and therefore the speed with which a task is performed. In what follows, we describe a modified strategy that allows us to improve the computational load of the servers while still maintaining security.

When the client sends to the server a task in the form of two sets S_1 and S_2, it now supplies a list of ranges $[l_i, u_i]$ for $1 \leq i \leq v$ which are not known to the servers. The server's task becomes to compute the number of distances between the elements of S_1 and S_2 that fall within each range. The sets S_1 and S_2 themselves are formed as previously described. Therefore, within the v different ranges, some ranges will correspond to the distances between real and real items, and others will corresponds to the distances between real and fake items.

To compute the count for each range, the server will obliviously compare a computed distance to all possible ranges and increment one of them that matched. As before, the counts are stored in $C = \langle c_0, c_1, \ldots, c_{v-1} \rangle$, where c_i corresponds to the number of distances in the range $[l_i, u_i]$. Initially, all c_i's are set to 0 and are updated as follows, where b_1 and b_2 are bits:

```
$[d] := \mathsf{dist}([x], [y]);$
for $i = 0, \ldots, v-1$
    $[b_1] := ([l_i] \overset{?}{\leq} [d]);$
    $[b_2] := ([d] \overset{?}{\leq} [u_i]);$
    $[c_i] := [c_i] + [b_1][b_2];$
```

By adjusting the granularity of the ranges, the client has a lot of flexibility to express its preference. That is, the ranges can aggregate a different number of distances, and the client can use fine granularity for the regions that convey more information to the client and coarse granularity for other regions.

he above solution reduces both the server's work and the size of C. This in particular means that the security analysis must be revisited to ensure that the parameters provide the necessary guarantees. As described in Section 5.1, however, the same security parameters are sufficient in the case of the Euclidean distance with smaller $|C|$ than in the case of the Hamming distance. That is, because u (which now corresponds to the number of ranges in which values $\sum_{i=1}^{m} x_i^2$ fall for all original x) will still be larger than 1, the same security parameters still should be sufficient, which the client will need to verify.

This setting, however, provides new opportunities for security enhancements. In particular, the client can use ranges that are overlapping. In our context, this can be valuable for the distances between real and fake items, where the need to update a single count by a cheating server can now translate into the need to update (and therefore correctly guess) multiple locations in C.

5.3 Combining verification of distances and statistics

The techniques for combining verification of distance computation in AllPairs and statistics computation in Analyze remain largely unchanged from the techniques for the Hamming distance. That is, the algorithm in Figure 3 can be used with minimal changes. In particular, during creation of an outsourced task, the client generates n_1 pairs of m-element fake items with the distance uniformly distributed in the range $[0, mt^2]$. It then chooses the values of d in step 2 from the range $[mt^2+1, mt^2+\ell]$ and creates protected distances in step 5 for $i = 0, \ldots, 2mt^2+\ell$. During verification of an outsourced task, the range of distance between two real items in steps 1(a) and 3 changes from $[0, m]$ to $[0, mt^2]$ and the range of distances between real and fake items in steps 1(a) and 3 changes from $[m+1, 2m+\ell]$ to $[mt^2+1, 2mt^2+\ell]$.

6 Verification of Statistics Computation for Set Intersection Cardinality

We next proceed with the description of our solution for statistics verification when the set intersection cardinality is used as the distance metric. We now assume that each original item is composed of m elements with each of them in the range $[0, t]$. Given S_1 and S_2, the server is to compute the cardinality of set intersection of elements in x and y, $|x \cap y|$, for each $x \in S_1$ and $y \in S_2$ and compile the distribution of the distances in the form of C.

6.1 Preliminary solution

Similar to prior solutions, the client proceeds with inserting $n_1 + n_2 = 2n_1$ fake items into the computation and k fake elements into the real and fake items to aid the verification process. Note that unlike the previous distance metrics, the fake elements are not required to be positioned consistently across all items. All artificial elements are set to 0 in the real items.

To generate fake items, the client produces $2n_1$ values larger than t and assigns each of them to a single fake item so that each fake item obtains a unique value. We use d_i to denote the value assigned to the ith fake item. To form the ith fake item, the client randomly chooses a value d from the range $[0, k-1]$, and sets d randomly chosen elements of it to 0 and sets the remaining $m + k - d$ elements to d_i. Each resulting real or fake item is now a multiset, and we assume that the distance computation function will work properly on such items. The client also records the number of times each d was used in a fake item in the set S_1 and the set S_2, respectively, and we denote such counts by c_d^j, where $d \in [0, k-1]$ and $j \in [1, 2]$.

This setup gives us that the distances between any two real items will fall into the range $[k, m+k]$, the distances between any real and fake items will fall into the range $[0, k)$, and the distances between any two fake items will fall into the range $[0, k)$. Because of the overlap of the last two ranges, the client will need to precompute the statistics for the distances between any two fake items, add it to the expected statistics for the distances between real and fake items, and then compare the result to the statistics returned by the server. The rest of the verification process uses the algorithm in Figure 1 with the inputs:

- $[l_r, u_r] = [k, m+k]$
- $[l_f, u_f] = [0, k-1]$
- $d_o = 0$

As far as security analysis goes, note that parameter k now serves the role of both k and ℓ in Section 4.3, and the size of C is $m + k$ instead of $2m + 1 + \ell$. Also, the value of m is relatively small in biometric types that use this distance metric (e.g., fingerprints). Furthermore, the distances between a single fake item and all real items in the other set are always the same depending merely on the value of d for that fake item, which means that equation 7 does not change. This means that for given m and k, the quantity in equation 7 might not be sufficiently low when $s = 1$, which means that higher s and therefore higher n_1 than what is reported in Section 4.5 might be necessary.

With respect to the analysis for the value of k (equation 12), we have that when the server skips at least one of the artificial elements during its computation, the verification associated with the distances between real items will be successful, but the counts associated with the distances in the range between 0 and $k-1$ will need to be updated by the server to pass verification. Skipping one artificial element will cause some, but not all, of the distances in that range to decrease depending on the value of the element at that position (this affects distances between real and fake items as well as between two fake items). This will result in at least two incorrect counts due to the distances between real and fake items (the count for distance $k-1$ will need to be increased and the count for distance 0 will need to be decreased in the best for the server scenario when all counts consistently shift down), the locations of which must be guessed correctly. This will also invalidate any number of counts from this range due to the distances between two fake items, and we should expect the server to have to guess the locations and update at least half of them with the exact differences. Equation 12 then becomes

$$\Pr[\mathsf{D}] \geq 1 - \prod_{i=0}^{u-1} \frac{1}{m+k-i},$$

where u can be set to $k/2$ or to a more conservative lower value.

6.2 Improved solution

As shown above, when the server misses the computation corresponding to a single fake item (either a row or column) in the distance matrix, all it has to do to remain undetected is to correctly guess and update one location in C. When the size of C is not large, to guarantee that the probability of misbehavior detection is sufficiently high, the parameters will likely have to be increased resulting in larger overheads. For that reason, we propose an improved solution that remedies this problem.

To improve the probability of detection when the server skips the computation associated with a single fake item, we introduce the following modification to current strategy: the client defines a small integer ℓ as the security parameter which will correspond to the number of additional elements which in fake items will be from the range $[0, t]$. The ith fake item now consists of three components: (i) d elements with value 0, where d is now from the range $[0, k-\ell-1]$; (ii) ℓ elements with the values in the range $[0, t]$, which we term offset elements; and (iii) $m+k-\ell-d$ elements with values d_i greater than t. The purpose of these ℓ offset elements is to introduce differences in the distances between a single fake item and a number of real items. In particular, the values of the ℓ elements in the fake items will be such that they will overlap with certain elements in real items, causing the distances between a single fake item and real items to be in the range $[d, d+\ell]$ instead of always d. The values of the ℓ offset elements are set to 0 in all real items. For simplicity, we will use the same set of ℓ offset elements for each fake item.

Notice that the above setup does not change the range of distances between any pair of real items, $[k, m+k]$, and also the range of distances between a fake item and either fake or real item, $[0, k-1]$. The difference is that a single fake item can now affect the counts corresponding to $\ell+1$ distances in C. To guarantee that a single fake item does indeed affect $\ell+1$ locations in C, the values for the ℓ offset elements must be properly chosen. For instance, in the worst case, the values selected for the ℓ offset elements might not share any common elements with any real item in S_1 and S_2, resulting in no benefit from this approach. For that reason, when the client chooses candidate values for the offset elements, it must scan the datasets S_1 and S_2 to ensure that there are items in each set that have overlap size with the candidate set for the offset elements of everything between 0 and ℓ.

Before the client is able to verify the computations performed by the servers using S_1 and S_2, the client needs to compute additional information as follows: for all real items in sets S_1 and S_2, compute the number of items that share i values in common with the ℓ offset elements. Let s_i^1 (s_i^2) denote this number in S_1 (S_2, respectively) for $0 \leq i \leq \ell$. Now the client can produce the expected statistics by executing the algorithm in Figure 1 with the following inputs:

- $[l_r, u_r] = [k, m+k]$
- $[l_f, u_f] = [1, k-1]$

- $d_o = \ell$

Finally, the client needs to incorporate the precomputed statistics for the distances between each pair of fake items into the expected statistics, and compare it with the results returned by the server. If all the checks in the algorithm succeed, the client treats the obtained results as correct.

With this modified solution, certain portions of the security analysis change and we obtain that now equation 7 becomes $\Pr[\overline{\mathsf{D}}] \leq 1/\binom{m+k}{s+\ell}$. This means that even by setting ℓ to a low value such as 1 or 2, having $s = 1$ will be sufficient to meet the necessary security guarantees even when m is not large. This will imply that the value of parameter n_1 will not increase over the values reported in Section 4.5 for a range of security settings.

6.3 Combining verification of distances and statistics

The technique for combining verification procedures for distance and statistics computation for the set intersection cardinality metric differs from those for the Hamming and Euclidean distances. This time, the real items are generated in the same way as for verification of statistics computation, but there are small changes in the way the fake items are produced. In particular, instead of setting d elements to 0 in a given fake item, where d is chosen uniformly from a range $[0, k - \ell - 1]$, we coordinate the values of d in a pair of fake elements. That is, we create n_1 pairs of fake elements $\langle x_i, y_i \rangle$, where the distance between them a randomly chosen value from $[0, k - \ell - 1]$. This is accomplished by having $k - \ell - 1 - \mathsf{dist}(x_i, y_i)$ 0 elements in common between x_i and y_i. All other elements in both x_i and y_i are set as before. In other words, the values of d for x_i and y_i are chosen in a coordinated manner, but otherwise the process is the same as before. This gives us that the number of 0 elements among the fake items in either S_1 or S_2 is no longer guaranteed to be uniform in the range $[0, k - \ell - 1]$, but this does not pose a security problem for the following reason: Because the solution is designed in such a way that by missing the computation associated with a single fake item the server will have to guess multiple locations in C and this is what guarantees the necessary probability of detection, this will be true if any fake item has been missed, regardless of the overall distribution of the distances associated with the set of fake items in general. The procedure for verifying correctness of an outsourced task is therefore the same as for verifying the statistics computation as described in Section 6.2, followed by verification of the distances $\mathsf{dist}(x_i, y_i)$ for n_1 pairs of fake elements $\langle x_i, y_i \rangle$.

The modification we make for the purposes of verifying both distance and statistics computation does influence a portion of the security analysis that needs to be revisited. In particular, because the distances between two fake items are now in the range $[\ell, k - 1]$, the probability of detection of incorrect computation of distances in equation 2 now becomes $\Pr[\mathsf{D}] = 1 - \left(\frac{p(k-\ell-1)+1}{k-\ell}\right)^{n_1}$. If $(k - \ell - 1) < m$ and $\Pr[\mathsf{D}]$ is insufficiently high, either the value of n_1 or k can be increased until a desired probability of detection is achieved, where changing n_1 will have a significantly larger impact on the value of $\Pr[\mathsf{D}]$ than changing k.

7 Related Work and Conclusions

Research on verifiable or uncheatable computation was initiated in [13, 12] using redundant task execution and insertion of so-called ringers in search for rare events (in particular, performing inversion of one-way function). Consequently, Szajda et al. [19] extended the idea to optimization problems and sequential executions (while still relying on parallel and redundant task execution). Other publications in this direction include [9, 14, 15] that inject chaff sub-tasks or verify portions of the result for computations of certain structure (e.g., NP-complete problems); [16, 21] use redundant scheduling. Du et al. [10] suggest the use of commitment to the result of massively-parallel server's computation using a Merkle hash tree, where the client verifies the computation by challenging the server on a number of individual sub-tasks which must match the commitment. Finally, in [17] distributed checking is used where (possibly malicious) servers perform checks on each other. We note that often the techniques from the literature can achieve a high probability of cheating detection only when p is rather low (i.e., not close to 1). There are also a number domain-specific computation verification techniques [4, 2, 20] that exploit domain knowledge to verify the results efficiently. Such techniques are known for algebraic computations [4, 2] and linear programming [20]. This work is thus closer to general techniques, as it assumes a certain structure of the computation, but the developed techniques are applicable to different instantiations of the distance function. We note that the general solutions listed above would not work in the context of this work even for verifying AllPairs computation, as distance computation consists of many elements and should not be treated as an integral function.

In this work we develop techniques for verifiable outsourcing of large-scale biometric computations consisting of distance and statistical data computation and provide their rigorous security analysis. Our techniques for the distance computation are general and can be applied to any distance metric, while the techniques for statistics computation are distance-metric dependent and we treat several popular distance metrics such as the Hamming distance, Euclidean distance, and set intersection cardinality. We also provide experimental results that show that the overhead introduced by our techniques is reasonable.

References

[1] IT cloud services user survey, pt. 2: Top benefits & challenges. http://blogs.idc.com/ie/?p=210.

[2] M. Atallah and K. Frikken. Securely outsourcing linear algebra computations. In *ACM Symposium on Information, Computer and Communications Security*, pages 48–59, 2010.

[3] M. Barni, T. Bianchi, D. Catalano, M. Di Raimondo, R. Labati, P. Failla, D. Fiore, R. Lazzeretti, V. Piuri, F. Scotti, and A. Piva. Privacy-preserving fingercode authentication. In *ACM Workshop on Multimedia and Security (MM&Sec)*, pages 231–240, 2010.

[4] D. Benjamin and M. Atallah. Private and cheating-free outsourcing of algebraic computations. In *Annual Conference on Privacy, Security, and Trust (PST)*, pages 240–245, 2008.

[5] M. Blanton and M. Aliasgari. Secure computation of biometric matching. Technical Report 2009–03, Department of Computer Science and Engineering, University of Notre Dame, 2009.

[6] H. Bui, M. Kelly, C. Lyon, M. Pasquier, D. Thomas, P. Flynn, and D. Thain. Experience with BXGrid: A data repository and computing grid for biometrics research. *Journal of Cluster Computing*, 12(4):373–386, 2009.

[7] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium*, pages 223–240, 2010.

[8] O. Catrina and S. de Hoogh. Improved primitives for secure multiparty integer computation. In *International Conference on Security and Cryptography in Networks (SCN)*, pages 182–199, 2010.

[9] W. Du and M. Goodrich. Searching for high-value rare events with uncheatable grid computing. In *Applied Cryptography and Network Security*, pages 122–137, 2005.

[10] W. Du, J. Jia, M. Mangal, and M. Murugesan. Uncheatable grid computing. In *International Conference on Distributed Computing Systems*, pages 4–11, 2004.

[11] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft. Privacy-preserving face recognition. In *Privacy Enchancing Technologies Symposium (PETS)*, pages 235–253, 2009.

[12] P. Golle and I. Mironov. Uncheatable distributed computations. In *RSA Conference*, pages 425–440, 2001.

[13] P. Golle and S. Stubblebine. Secure distributed computing in a commercial environment. In *International Conference on Financial Cryptography*, pages 289–304, 2001.

[14] M. Goodrich. Pipelined algorithms to detect cheating in long-term grid computations. *Theoretical Computer Science*, 408(2–3):199–207, 2008.

[15] G. Karame, M. Strasser, and S. Capkun. Secure remote execution of sequential computations. In *International Conference on Information and Communications Security (ICICS)*, pages 181–197, 2009.

[16] H. Kim, J. Gil, C. Hwang, H. Yu, and S. Joung. Agent-based autonomous result verification mechanism in desktop grid systems. In *Agents and Peer-to-Peer Computing (AP2PC)*, pages 72–84, 2007.

[17] M. Kuhn, S. Schmid, and R. Watterhofer. Distributed asymmetric verification in computational grids. In *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–10, 2008.

[18] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[19] D. Szajda, B. Lawson, and J. Owen. Hardening functions for large scale distributed computations. In *IEEE Symposium on Security and Privacy*, pages 216–224, 2003.

[20] C. Wang, K. Ren, and J. Wang. Secure and practical outsourcing of linear programming in cloud computing. In *INFOCOM*, 2011.

[21] K. Watanabe, M. Fukushi, and S. Horiguchi. Collusion-resistant sabotage-tolerance mechanisms for volunteer computing systems. In *IEEE International Conference on e-Business Engineering (ICEBE)*, pages 213–218, 2009.