Secured histories: computing group statistics on encrypted data while preserving individual privacy

Eleanor Rieffel, Jacob Biehl, William van Melle
FX Palo Alto Laboratory
{rieffel,biehl,billvm}@fxpal.com

ABSTRACT
As sensors become ever more prevalent, more and more information will be collected about each of us. A long-term research question is how best to support beneficial analysis of such data while preserving individual privacy. Awareness systems represent an emerging class of applications supporting both business and social functions that leverage pervasive sensors to detect and report end-user physical state, activities, and available communication channels. To buy into the system, however, users must be able to control how information about them is shared. We introduce “need to know” security in which an individual has full access to her own data, a third party processes the data without learning anything about the data values, and other users, such as analysts, learn only the desired statistics. Our novel privacy mechanism for time series data gives users a high level of control over their individual data while allowing storage of data and computation of summary statistics to take place on untrusted machines. The mechanism supports computation of simple statistics across multiple users whose data have been encrypted under distinct keys. We designed key structures and extensions to provide a family of efficient non-interactive “need to know” protocols for time series data. We implemented the mechanism and integrated it with MyUnity, a prototype awareness system.

Categories and Subject Descriptors
H.3.4 [Info. Storage and Retrieval]: Systems and Software—Current awareness systems; D.4.6 [Software Engineering]: Operating Systems—Security and Protection

General Terms
Security

Keywords
privacy; awareness; access control; cloud computing; homomorphic encryption

1. INTRODUCTION
As sensors become ever more prevalent, more and more information will be collected about each of us. This wealth of data has many benefits such as advancing medicine and public health, improving software and services through user pattern analysis, and enabling each of us to gain greater insight into our own habits and tendencies. At the same time, the potential for misuse of such data is significant, and simply the possibility that such data are being collected can “lessen opportunities for solitude and chill curiosity and self development” [7]. A long-term research question is how best to support beneficial uses while inhibiting less desirable effects.

Currently analysts, such as medical researchers, usually have full access to data for a group of individuals from which they compute the statistics they want. In some cases, analysts use a third party service provider to store, or even process, such data. Either the third party has access to all data or the data are encrypted, in which case the third party does no processing. An interesting research question is how to provide mechanisms to support “need to know” security in which an individual has full access to her own data, the third party learns nothing about the data but can nevertheless contribute to the processing, and the analyst learns only the desired statistics.

Our interest in “need to know” security arose in connection with MyUnity, a prototype awareness system [8]. MyUnity collects data from a variety of sources and displays summary presence states, such as “in office” or “with visitor,” computed from the received data. MyUnity was deployed in a small research lab and has been in use by over 30 people for more than a year. To avoid concerns about misuse, the system did not store any data. Users expressed interest, however, in seeing personal trends, activity patterns of coworkers, and data pooled across groups of users. At the same time, users expressed concern about misuse if their data were stored. Most of the patterns users were interested in seeing could be obtained by averaging. For this reason, we looked at “need to know” security for cases in which, at each time step, each member of a group of users has a value (i.e., presence state) to contribute, and the group would like to provide only an aggregate view of those values to people outside their group.

In this paper, for the simple case of a summing operation, we provide a family of efficient “need to know” protocols, with varying privacy guarantees, in which an untrusted third party can perform the bulk of the computation. Our protocols enable a single user to request encrypted sums
over arbitrary subsets of her own data from the third party that she can then decrypt using her own key or keys. More interestingly, our protocols allow each user to encrypt under her own key, but nevertheless enable a third party to compute an encryption of a sum across multiple users without decrypting any of the values or further interacting with the individuals, and provide means for an outsider, such as an analyst, to decrypt the encrypted sum. We have implemented the basic protocol, integrated it with MyUnity, and evaluated its baseline performance.

We also significantly extend the basic protocol to a series of increasingly sophisticated non-interactive protocols, each of which provides every user with more control over access to her own data. In the simplest of these protocols, anyone who can decrypt the sum can decrypt the individual values. In a more sophisticated protocol, the ability to decrypt a sum does not imply the ability to decrypt the individuals’ values. Yet more sophisticated schemes guard against $k$-collisions. These schemes can be nested to support a hierarchy in which nodes at higher levels can decrypt statistics across all nodes below them, but cannot decrypt any partial statistics, including individual values.

All of our protocols use a symmetric-key, additively homomorphic encryption scheme\cite{8}. A critical property of this encryption scheme is that it enables summation of values encrypted under different keys, a property not easy to obtain from existing public key homomorphic encryption schemes. Additional advantages of using this symmetric key scheme as the base—as opposed to using a public key scheme—include its simplicity and efficiency. The more sophisticated protocols combine this symmetric-key homomorphic encryption scheme with extensions of Chaum’s DC-nets\cite{9} to provide stronger privacy guarantees. “Need to know” security complements differential privacy, which limits what can be learned from the statistics.

The most significant contributions of this paper include:

- Recognition of the need for “need to know” access control structures for the statistical analysis of data from multiple individuals in which:
  - each individual encrypts her presence values under her own key,
  - a third party can compute encrypted statistics over data encrypted using multiple keys without learning anything about individual data values or the statistic computed, and
  - entities equipped with the appropriate keys can decrypt the group statistics without learning partial statistics or individual values.

- A family of non-interactive protocols that provide “need to know” security for sums of data values.

- Application of a symmetric-key homomorphic encryption scheme to support privacy in an established awareness system.

Section 2 describes the MyUnity system and the concerns and desires of MyUnity users that inspired this work. Sections 3-5 define the problem, present our schemes in the context of the MyUnity system, and discuss our preliminary implementation and efficiency evaluation. We then review related work (Section 6), discuss open problems (Section 7), and give our overall conclusions (Section 8).

2. OVERVIEW OF MYUNITY

The past few years have seen a rapid expansion of technologies that fuse physical sensing capabilities with social and communication software. One such system is MyUnity\cite{6}, an awareness system for the workplace that increases workers’ awareness of their colleagues’ physical presence, current activities, and preferred communication channels. The MyUnity prototype has been in daily use by over 30 active participants for over a year.

MyUnity was designed to aid workers in building group awareness. MyUnity collects data from sources such as cameras, Bluetooth device sensors, mouse and keyboard activity, network connectivity, IM availability, and the employee calendar. The data are aggregated and summarized into one of five presence states: in office, in building, has visitor, actively online remotely, connected online but inactive. A sixth state indicates that the system has insufficient data on the user. Users of the system run a client that visually displays the presence states for colleagues in photo tiles within an awareness dashboard (see Figure 1). The color of each tile indicates the user’s presence state. Purple indicates that a person has visitors in her office. Green indicates that the person is in her office, while yellow indicates that the person is in the building. Bright blue indicates active remote activity, while faded blue indicates online connectivity. Figure 2 provides a high level overview of how MyUnity collects and fuses information from multiple sources. The system computes, at regular intervals, the current presence state for its user from the data it receives. It streams the current presence states to active clients for visual display.

MyUnity was designed to support a variety of user privacy
preferences, and allows each user to configure what information is collected on her behalf. For instance, if a user does not want a camera in her office, this data feed can be left out, and a presence state can still be computed for her. The fusion rules used in MyUnity adapt to missing information by degrading they system’s resolution of the user’s state. For example, if the user’s office does not have a camera and a colleague is visiting her, the system will report only ‘In the Building’ unless she is also activity using her computer, in which case it will report ‘In Office.’ This behavior was valued and easily understood by its users. Of the more than 30 people using the system over the course of a year, most used all of the sensor and data sources supported by the system. Four users chose not to have a camera in their office, not to be tracked using bluetooth, or both.

In addition to displaying presence states for individual users, the interface displays presence information for groups. The current implementation includes tiles for the admin group, the support group, and the MyUnity research group. These presence states are computed on the client from the presence states of the members of the group, so the group presence state is available only to users who have access to the presence states of all of the individuals in the group. Currently the ‘most present’ state for anyone in the group is used as the presence state for the group.

2.1 Whether to store user data?

In its initial deployment, MyUnity did not store historical data about users’ presence states so as to avoid concerns about unintended or unauthorized use. MyUnity users have consistently requested, however, trend data about themselves or their colleagues. Users provided many examples for when historical data about a co-worker would be useful in supporting their daily job activities. For instance, several users explained that trend data could suggest when a colleague would likely be back after lunch. Others expressed interest in having histories for self reflection on their own behavior. Users also expressed interest in seeing summaries for a group, while not necessarily needing summaries for individuals within the group. For example, a user may wish to know when is generally a good time to discuss a non-urgent but complex problem with someone in the system support group. Conversely, members of a group were often more comfortable sharing summary data about the group as a whole than they were sharing their own individual activity patterns with people outside their group.

Users also expressed concerns about misuse of such data. A frequent worry was that it might be used as part of an evaluation of performance. Workers tend to trust close colleagues with detailed presence data, but are less trusting of management and co-workers outside their project. For instance, workers would prefer to share detailed presence histories or trends with their team members, but not with management or co-workers outside their team. Users were more comfortable with managers obtaining summaries that aggregate presence histories over all members of their team. This structure is largely motivated by how workers perceive the data being used; i.e., co-workers would utilize the data to support richer interaction and collaboration, while management would use it to review performance.

The feedback from users indicates significant value in storing historical data, but only if it is secured and equipped with an appropriate access control structure. A user may want a moment-by-moment history her own activity, but her colleagues may need a sense only of her general work patterns (and that may be all she wishes to share). A user outside a project team may be interested only in summary statistics for that team, and such statistics may be all the team wishes to share with outsiders. Furthermore, different users had very different feelings about what they did and did not want done with their data. The bulk of this paper explains the mechanisms through which we meet these needs.

3. SECURED HISTORIES: PROBLEM DEFINITION

This section provides a more precise definition of what constitutes a “need to know” security protocol, and describes the system architecture and cryptographic background material used in the construction of our protocols.

We define the problem abstractly in terms of

- n participants, each of whom has a value, such as a presence state, to contribute at each time step,
- an analyst who wishes to obtain a statistic over these values at each time step, and
- an honest, but curious third party who contributes to performing the computation of the statistic without learning anything about the values.

There may be one or more analysts. Analysts may be researchers, as in the medical and user study cases, or may be one or more of the participants, as in the MyUnity case.

We wish to capture more precisely the conditions for “need to know” security: that an individual has full access to her own data, and may obtain help from the third party to analyze it, that individuals do not have access to each other’s data unless they explicitly share privileges, that the third party learns nothing about the data values, and that the analyst can obtain statistics about a group of individuals with help from the third party but learns nothing more about the data values than what can be deduced from the statistic. A “need to know” security protocol for sums with respect to time series data contains the following algorithms:

- **Setup:** Establishes public parameters and constants used by all parties in the protocol.
- **Key Generation:** Establishes the key structure used in the protocol. It is run once, prior to any of the data generation time steps.

Figure 2: Schematic overview of the architecture of the MyUnity awareness application.
• Encrypt: At each time step, each individual encrypts her values under her own key or keys and sends the encrypted values to the third party for storage.

• Compute Encrypted Stat: The third party can compute an encryption of the sum over any specified set of data.

• Decrypt Individual Stat: Any individual with access to individual A's keys can decrypt the sum over an arbitrary subset of individual A's values.

• Decrypt Group Stat: An analyst with the appropriate set of keys can decrypt the sum over all values, at a given time step, for a group of users.

The “appropriate set of keys” with which the analyst can decrypt varies from protocol to protocol, as does the key structure.

A “need to know” protocol is secure if

• an “honest, but curious” third party can learn nothing about the data values,

• an analyst learns nothing about users’ values other than what she can deduce from the statistics, and

• each user learns nothing about other users’ values.

A “need to know” protocol is secure against \( k \)-collusions if for any set of \( k \) or fewer parties, whether consisting of individual participants, outsiders, or analysts, the colluding group cannot learn anything about an external members’ data other than what can be deduced from the colluding members’ data and the full statistic (if an analyst is part of the colluding group).

We present a series of increasingly sophisticated protocols, each of which provides users with more control over their own data. The first protocol, described in Section 3.1 is not quite a “need to know” protocol because anyone who can decrypt a group sum, such as the analyst, can also decrypt every user’s data values. Section 5.1 describes the first “need to know” protocol: in this protocol the ability to decrypt a group sum does not imply the ability to decrypt an individual’s value, or any partial sum. Under this scheme, however, certain pairs of individuals can collude to decrypt another individual’s value. We then describe a family of schemes that is secure against \( k \)-colluding parties. Finally we describe how these schemes can be nested to support a hierarchical structure in which higher levels in the hierarchy have access only to summary statistics across a group of users and do not have access to data about individual users. First we explain the architecture underlying all of these protocols, and review cryptographic material underlying the schemes.

3.1 Secured histories architecture

This section describes the system components and the high-level system architecture involved in our protocol.

The raw data sources, such as cameras, Bluetooth device sensors, and keyboard monitors send their data, along with metadata, such as a source ID and timestamp, to sensor input processors that process the data and send the processed data to the Feed Server. For example, the processors of the video feeds send on to the Feed Server only descriptions of behaviors or events observed, not the raw video streams. In some cases, raw data sources may talk directly to the Feed Server. The Feed Server forwards data to the appropriate fuser, which computes the presence states. In this paper, we are not concerned with how these communication channels are secured, only that they have been.

The following components play a role in our protocols, and their relationship is indicated in Figure 3.

TRUSTED (partial access to keys)

• Fusers: There is one fuser per individual with access to the key or keys used to encrypt its individual’s data. It computes its individual’s presence state from data received from the Feed Server, encrypts this presence state using the individual’s key or keys, and returns the encrypted presence state to the Feed Server which routes it to Encrypted Data Store.

• Client: A given individual may run multiple clients on different desktops or mobile devices. Each of those clients has access to that individual’s keys, and the keys for any other individuals who wish to share their historical presence data with her. A client decrypts, completes the computation of the statistics if necessary, and presents the information in the client interface. It also passes client queries from the client interface to the Feed Server.

UNTRUSTED (no access to keys)

• Feed Server: The Feed Server routes information between the various components of the system. For example, the Feed Server routes all information used to compute an individual’s presence state to her fuser. It also routes queries from the client to the Encrypted Data Computation Engine.

• Encrypted Data Store: The Encrypted Data Store stores the encrypted data, together with its metadata. It also keeps a list of missing data ranges. When aggregated results are returned, it includes a list, often empty, of any missing data. The Encrypted Data Store may also store results from computations made by the Encrypted Data Computation Engine.

• Encrypted Data Computation Engine: The Encrypted Data Computation Engine performs computations on encrypted data and returns the results to the Feed Server to be sent to the clients. The Encrypted Data Computation Engine may also carry out a caching strategy in which the results of certain computations are computed ahead of time and stored.

Instead of having one fuser per individual, in situations where a group of individuals trust each other, they can share a fuser. The untrusted components could reside in a public cloud. More than one of each of the untrusted components may be needed to support a large organization.

3.2 Cryptographic background

To provide the functionality described above, we require an encryption primitive in which (i) encryption is fast, (ii) values encrypted by the same party can be aggregated in encrypted form, and (iii) values encrypted by multiple parties under different keys can be aggregated in encrypted form. Property (i) is required due to the high frequency with which awareness readings are taken. Properties (ii) and (iii) are required for the untrusted party to aid in the computation of
individual and group presence statistics, respectively. Most homomorphic cryptosystems that can provide us with property (ii) are based on expensive public-key primitives that violate property (i), and do not easily support property (iii).

To this end, we make use of a symmetric-key additively homomorphic cryptosystem recently proposed by Castelluccia et al. [8]. In this cryptosystem, sensor readings are assumed to be integer values, and pseudorandom function families are used to generate per-user pseudorandom values using a nonce $n_t$ that includes the timestamp and other information. We will use the notation $X_i$ for a user, where $i$ is an index over the user population. Encryption is simply addition mod $M$, and decryption is subtraction mod $M$: an individual $X_i$ with key $k_i$ encrypts her value $v_i$ at time $t$ by evaluating the pseudorandom function $g_{k_i}$ at the nonce $n_t$ and adding it to $v_i$ to obtain

$$c_i = v_i + g_{k_i}(n_t) \mod M.$$  

To decrypt, she computes $g_{k_i}(n_t)$ and subtracts it from $c_i$:

$$v_i = c_i - g_{k_i}(n_t) \mod M.$$  

This cryptosystem is parametrized by a pseudorandom function (PRF) family, which consists of a collection $F_{\lambda} = \{f_s\}$ of functions indexed by the security parameter $\lambda$. The functions $f_s$ are functions from $\lambda$ bits to $\lambda$ bits, $f_s : \{0,1\}^\lambda \to \{0,1\}^\lambda$. Since provably secure pseudorandom functions are very slow, Castelluccia et al. [8] advocate using keyed hash functions such as HMAC [4, 16] followed by a length-matching hash function $h$ that does not need to be collision resistant, but must have uniform output upon uniform input. Applying this hash function $h$ ensures, for example, that if at least one of the blocks is indistinguishable from random, then the output of the composition of $h$ with $f_s$ is indistinguishable from random. Applying $h$ would be unnecessary if we were using a provably secure pseudorandom function.

In [8], the authors prove this scheme semantically secure:

**Theorem 3.1.** Assuming $F_{\lambda} = \{f_s : \{0,1\}^\lambda \to \{0,1\}^\lambda\}$ with $s \in \{0,1\}^\lambda$ is a PRF, and $h : \{0,1\}^\lambda \to \{0,1\}^\mu$ is uniformly distributed over $\{0,1\}^\mu$, the above construction is semantically secure.

They suggest using the simple hash function $h : \{0,1\}^\lambda \to \{0,1\}^\mu$ that partitions the $\lambda$-bit output of $f_s$ into length $\mu$ substrings and adds them together. It is straightforward to verify that this $h$ satisfies the uniformity condition required in their theorem. Thus, the security of their scheme, and of ours, reduces to the security of the PRF used. HMAC is proven in [4] to be a PRF provided the underlying compression function is a PRF and the iterated hash is weakly collision-resistant. In 2006, Bellare was able to remove the second condition [3]. Thus, the security rests on the extent to which the compression function underlying HMAC is a PRF. Other PRFs could be used in Castelluccia’s construction, and therefore in ours.

This cryptosystem provides the ability to homomorphically combine values that are encrypted under the same or different keys. Consider two individuals $X_1$ and $X_2$ with keys $k_1$ and $k_2$, respectively. Suppose these individuals wish to encrypt the values $v_1$ and $v_2$, respectively, at time $t$. Each encrypts by evaluating her pseudorandom function $g_{k_i} = h(f_{k_i})$ indexed by $k_i$ at $n_t$:

$$c_1 = v_1 + g_{k_1}(n_t) \mod M \quad (1)$$
$$c_2 = v_2 + g_{k_2}(n_t) \mod M \quad (2)$$

Given the aggregate ciphertext $c = c_1 + c_2$, an individual with access to both $k_1$ and $k_2$ can construct the sum $v = g_{k_1}(t) + g_{k_2}(t)$ and recover the aggregate value

$$r = v_1 + v_2 = c - r \mod M.$$  

In addition to faster encryption and the ability to combine values encrypted under different keys, this cryptosystem has smaller ciphertext sizes than public key alternatives, leading to more efficient storage and bandwidth usage. The efficiency gains in storage and bandwidth are particularly significant when the data are binary or come from a small, finite range of values.

**4. THE BASIC PROTOCOL**

This section describes a protocol for sums over time series data that satisfies all of the conditions of “need to know” security, except that in this protocol an analyst can decrypt all of the individuals’ values. Section 5 extends this protocol to obtain full “need to know” security for sums over time series data.

**4.1 Protocol description**
Recall from Section 2 that there are five presence states plus an additional state indicating that no information is available. The system represents the presence state for each user as a five-bit string, in which each bit corresponds to one of the five positive presence states. There are six legal presence values, 10000, 01000, 00100, 00010, 00001, and 00000, corresponding to in office, has visitor, in building, active online remotely, and connected online but inactive, and no information. We encrypt each bit of the five-bit string separately in order to support computation of statistics restricted to one presence state.

Each fuser computes, at regular intervals, the current presence state for its user and sends it to the Feed Server to be sent to the clients. Each fuser also encrypts, under the user’s key, each of the binary values making up the presence state, taking the timestamp concatenated with the presence state type (one of five values) as the nonce. At each time step, the fuser sends a record, consisting of a user ID and timestamp, both unencrypted, and five encrypted Boolean values, one for each presence state, to the Feed Server to be placed in Encrypted Data Store.

- **Setup:** (i) Establish a modulus \( M \) large enough for the application at hand. The modulus must be larger than the number of terms that would ever contribute to the computation of a single statistic. For many applications, \( M > NT \) works, where \( N \) is the number of users and \( T \) is the total number of timestamped entries per user per statistic. The bit length of encrypted values will be \( \mu = \lceil \log_2(M) \rceil \). (ii) Establish a pseudorandom function family \( f_S = \{f_S : \{0,1\}^\mu \rightarrow \{0,1\}^\lambda \} \), and choose \( \lambda \) according to the desired level of security. (iii) Establish a length-matching hash function \( h : \{0,1\}^\lambda \rightarrow \{0,1\}^\mu \).

- **Key generation:** Each individual \( X_i \) runs a key generation algorithm to obtain a key \( k_i \). The individual securely transmits the key to her fuser, say by SSL.

- **Encrypt:** Each individual \( X_i \) encrypts each of the five bits \( m_j \), for \( j = 1, \ldots , 5 \), of a presence state \( m \) as

\[
c_i = m_j + h(f_{X_i}(n_j)) \mod M,
\]

where the nonce \( n_j \) is taken to be the concatenation of the presence state type and the timestamp. We refer to \( r_j = h(f_{X_i}(n_j)) \) as a pad. The record that is transmitted includes a header, containing the user ID and timestamp transmitted in the clear, followed by the five ciphertexts \( c_j \).

- **Addition:** The Encrypted Data Computation Engine computes a ciphertext sum \( c \) by adding ciphertexts together modulo \( M \).

- **Decrypt:** A user with access to the keys for all users whose values contribute to a sum can decrypt a ciphertext \( c \) for a sum by computing the pads for all values and subtracting their sum from \( c \).

Sections 4.2 and 4.3 give example decryptions of a sum. In order to decrypt a sum, a user with access to the appropriate keys must also have access to the appropriate nonces. Because data are collected at regular intervals, users know which timestamps should contribute to the sum. The system rounds the timestamps to the nearest minute in order to make them predictable to the user. In order to handle missing data, the Encrypted Data Computation Engine sends the client a list of any expected triples (timestamp, user ID, presence type) that are missing from the sum. Since the system is robust, usually this list will be empty or very small. Because missing data are often bursty, in our implementation we specify a range of missing values with a pair of values indicating the start and end times for the missing values.

### 4.2 Example: queries about an individual

A user can query the Encrypted Data Store about her own history, receiving encrypted values that she can decrypt using her key. In addition to receiving individual values, she can query the Encrypted Data Computation Engine to receive encrypted sums. For example, she may want to understand her typical daily presence pattern by dividing the day into fifteen minute intervals and requesting the totals of each presence state for each fifteen minute interval over the past three weeks. The Encrypted Data Computation Engine can compute and return the encrypted sum for each state in each interval. She can then decrypt each of these encrypted sums using her key and the nonces. The semantic security of the cryptographic construction used to encode each presence state ensures that the Encrypted Data Computation Engine cannot learn any information about the user’s presence states.

Because the presence states in our case are Boolean values, the variance can be computed directly from the average: \( V = A - A^2 \), where \( V \) is the variance and \( A \) is the average. To obtain the variance, the user decrypts the sum, divides by the number of values to obtain the average \( A \), and then uses the above formula to obtain the variance \( V \). In a non-Boolean case, the square of each value \( v_i^2 \) can be encrypted and stored to enable most of the computation of the variance to be done homomorphically by an untrusted party.

Instead of estimating her presence state pattern from the data over the last three weeks, she may wish to use data from the past six months, but weight the more recent data more heavily. She can request the encrypted weighted sum for any weighting with integer values (as long as the values are not so large as to make the computation overflow the modulus \( M \)). To decrypt, she will need to use the same weighting to sum the pads. As a simple example, she can request a weighted sum of two values, \( v_1 \) and \( v_2 \), in which the later value \( v_2 \) is weighted twice as much as the earlier value \( v_1 \). The Encrypted Data Computation Engine returns

\[
c = c_1 + 2c_2 \mod M,
\]

where \( v_1 \) and \( v_2 \) have been encrypted as \( c_1 = v_1 + r_1 \) and \( c_2 = v_2 + r_2 \) respectively. The user can decrypt by computing a similarly weighted sum of the pads, \( r = r_1 + 2r_2 \), and subtracting it from \( c \) to obtain

\[
v = v_1 + v_2 = c_1 + 2c_2 - r_1 - 2r_2 \mod M.
\]

### 4.3 Example: queries about a group of users

Suppose each of \( L \) team members sends her key to an analyst. At a given time point, and for a given presence type, all team members’ values are encrypted using the same nonce \( n \), a concatenation of the timestamp and the presence type. Each fuser encrypts its team member \( X_i \)’s value \( v_i \) by adding the pad \( r_i = h(f_{X_i}(n)) \) to \( v_i \) modulo \( M \):

\[
c_i = v_i + r_i \mod M.
\]
The analyst can request the sum of all of these ciphertexts from the Encrypted Data Computation Engine

\[ c = \sum_{i=1}^{L} v_i + \sum_{i=1}^{L} r_i \mod M. \]

The analyst can compute all of the pads \( r_i \) since she has all of the keys and knows all of the nonces. She can even compute the sum of all of the pads prior to receiving \( c \) from the Encrypted Data Computation Engine. She subtracts this sum, \( \sum_{i=1}^{L} r_i \), from \( c \) to obtain the total \( v = \sum_{i=1}^{L} v_i \).

The advantages of this approach over encrypting in a non-homomorphic way and having the client perform the computation after receiving and decrypting the data values include (i) more efficient bandwidth use, in that users can obtain a statistic without needing to obtain all of the values contributing to that statistic, and (ii) improved security, in that raw presence values are not needed in decrypted form.

The amount of computation required to decrypt the group statistic scales linearly with the number of values contributing to the statistic, since it is the computation of the pads that forms the bulk of the computation. The computation of these pads can, however, be computed prior to receiving the encrypted value. Thus, the part of the decryption that must take place after receiving an encrypted sum is constant; only one value, the sum of the pads, must be subtracted to decrypt. For this reason, a comparison of decryption times for sums between our protocol and public key based homomorphic encryption schemes is not straightforward. For large sums, the computation of the pad sum is expensive, but can be computed ahead of time, prior to receiving the encrypted sum. In contrast, the decryption time for public key homomorphic encryption schemes is constant in the number of elements in the sum, but the decryption can start only after the encrypted sum has been received. This decryption time is significantly longer than the single modular addition our approach applies after the encrypted value is received.

Section 5 describes protocols that achieve full “need to know” security and in which the cost of decrypting certain sums does not increase with the number of contributing terms because only a constant number of pads need to be computed to decrypt.

### 4.4 Application of the basic protocol

Our initial implementation supports the computation of a rough summary of a single user’s presence pattern from encrypted stored data. To obtain baseline efficiency estimates, we used presence data for one individual from roughly a three-week period. This test set consists of 31,568 records, collected once a minute, each with five encrypted values, for a total of 157,840 encrypted values. For the statistical summary of this person’s daily presence pattern, we aggregated the presence states over fifteen-minute intervals, and summed over the 21 days of data, to obtain histograms for each of the five presence states. We smoothed to further obscure the data and make it more visually appealing. Figure 4 shows the graph our system produced. Such graphs will be available to users in the next version of the MyUnity client dashboard. The colors are the same ones used on the tiles to indicate the presence states.

We implemented the core functionality in Java. Because the client is implemented in .NET, we ported some of this code to .NET. We used HMAC, as implemented in java cryptography library javax.crypto and in .NET. We used the de-fault security parameter of \( \lambda = 128 \). For convenience, we took \( M = 2^{52} \), so that the length of an encrypted value is 32 bits, the bit length of an unsigned integer in most programming languages, though we could have used a considerably smaller modulus. The bit-length of the encrypted data is an order of magnitude smaller than that needed by a public key homomorphic encryption scheme with a similar level of security. Thus, our protocol has more efficient storage and bandwidth usage than public key based solutions. We wrote our own length-matching hash function that splits a byte array into groups of four bytes and adds these together.

The timestamp uses the time rounded to the nearest minute with the following improvement to ensure that the timestamps are predictable. While the code requests a record every minute, the actual time the record is reported varies considerably. To handle this variation, the feed server compares the current rounded timestamp with the previous one and the subsequent one, and adjusts if two rounded timestamps collide. If adding a minute to one of the rounded timestamp values cannot resolve the collision, one record is discarded. Recall that the Encrypted Data Store of missing data in addition to the encrypted data records.

The server on which we ran our protocol is a virtualized Windows Server 2008 instance, hosted by a Citrix XenServer hypervisor, which was allocated four virtual CPUs with 8 GB of memory, an 80 GB virtual disk, and a 1 GB full duplex ethernet port. The underlying hypervisor CPU is an Intel Xeon E5450 running at 3.21 GHz. Our clients vary in their specifications, but the numbers we report are from an Intel 2.40 GHz dual core with 2 GB of RAM.

We made no attempt to optimize the code, so the following values are baseline upper bounds. The sums that go into the graph use fifteen-minute time buckets; thus each sum has approximately 2105 contributing values. On our server, each encryption takes roughly 2.33 milliseconds. Computation of all 480 sums, 96 fifteen-minute intervals per day for each of the five presence states, takes a total of 439 milliseconds, or approximately 0.92 milliseconds per sum. Computing the pads for decrypting all 480 sums is slow, taking about 11.5 seconds total, but these pads can be computed ahead of time, prior to receiving the encrypted sum. The final decryption takes 2.33 milliseconds per sum, or approximately 1.12 seconds for all sums contributing to the graph.

![Rhythmetrics](image)
5. SECURED HISTORIES: EXTENSIONS

As we saw in Section 4.4, the basic protocol of Section 4 enables an individual to make use of an honest, but curious third party to store her data and support computations on that data. Furthermore, that protocol also enables an analyst that is fully trusted by all individuals, and has access to all of their keys, to use the third party to aid in computing sums over data values from multiple individuals that have been encrypted under different keys. This section shows how to obtain full “need to know” security that enables an analyst to obtain the results of certain sums without having access to any of the individuals’ values. The full “need to know” protocols described in this section use a more sophisticated key structure than was used in the basic protocol, an extension of Chaum’s DC-nets. As a side benefit, decryption of group statistics is faster than for the basic protocol.

5.1 A “need to know” protocol

As mentioned in Section 2.1, MyUnity users are interested in sharing their presence histories or trend data with their closest colleagues, but prefer that only aggregate statistics be available to managers and employees outside their team. This trust structure leads to questions of how to support a different type of hierarchical access control structure than is usually considered in the literature, one in which higher levels in the hierarchy have access only to summary statistics across a group of users, but do not have access to data for individual users.

Imagine a project team that wants their manager to see only pooled data on the team members’ activities. The manager could see the pattern of availability of the group, for example, without learning anything about the pattern of any individual, other than what can be deduced from the statistics for the whole group. We describe a protocol in which $N$ team members $X_i$, at each time step, encrypt one value in such a way that the manager can decrypt the sum but not the individual values.

Ideally we would solve this problem by providing the manager with a key $k$ and the team members with keys $k_1, \ldots, k_N$, such that at each time step, and for each presence state, the pad computed from the manager’s key $k$ is the sum of the pads computed from the team members keys $k_1, \ldots, k_N$. We are not aware of a method for obtaining $n$ pseudorandom functions $g_1, \ldots, g_N$ and another pseudorandom function $f$ such that $f(x) = \sum g_i(x)$ for all $x$, with the property that the ability to compute $f$ does confer the ability to compute any $g_i$. For this reason, we take a less direct approach that uses an extension of Chaum’s DC-nets. Whether there is a more direct approach is an intriguing open problem.

The following algorithms constitute a “need to know” security protocol for sums with respect to time series data:

- **Setup:** Same as for the basic protocol (Section 4.4).
- **Key Generation and Sharing:** Each team member $X_i$ for $1 \leq i < N$, generates a key $k_i$, and the manager generates a key $k_0$. Each team member $X_i$ sends her key to individual $X_{i+1}$ where the indexing is modulo $N + 1$ and the manager is considered individual $X_0$.
- **Encrypt:** Each team member $X_i$ encrypts her value $v_i$ at time $t$ by adding the pad $r_{i-1} = h(k_{i-1}(n_i))$ and then subtracting the pad $r_i = h(k_i(n_i))$ from $v_i$, where all arithmetic is done modulo $M$:
  \[ c_i = v_i + r_{i-1} - r_i \mod M. \]

  All team members use the same nonce $n_t$, a concatenation of the timestamp with the presence state type.
- **Compute Encrypted Stat:** The third party can compute an encryption of the sum over any specified set of data.
- **Decrypt Individual Stat:** Anyone with access to individual $A$’s keys can decrypt the sum over an arbitrary subset of individual $A$’s values.
- **Decrypt Group Stat:** Anyone (e.g. the manager) with the two keys, $k_0$ and $k_N$, can decrypt the sum over all values, at a given time step, for the group of users. When the encrypted values $c_i$ at time $t$ are summed over the group, all of the pads cancel except for $r_0$ and $r_N$:
  \[ c = \sum_{i=1}^{N} c_i = v_1 + \cdots + v_N + r_0 - r_N \mod M. \]

  Anyone with keys $k_0$ and $k_N$ can compute pads $r_0$ and $r_N$ to decrypt $c$ to obtain the sum $v = \sum v_i$.

  All of the players have two keys. The manager has keys $k_0$ and $k_N$, and each team member $X_i$ has keys $k_i$ and $k_{i-1}$, as shown in Figure 5. The key structure is designed to be a chain in which pads computed from the keys cancel in the desired way in a sum. Because the manager does not have any of the other keys, she cannot decrypt any subtotal, let alone any individual value $v_i$. To decrypt the sum, she needs to compute only two pads. For this reason, decryption of group sums is more efficient for this protocol than for the basic protocol of Section 4.4 in which the manager had to compute $N$ such terms. This protocol enables the decryption of a group statistic with the computation of only two pads at the expense of doubling the (fast) encryption time for individual values.

  In this protocol, while the manager cannot decrypt individual user’s values, two players can collude to decrypt...
another member’s data. Team members $X_{i-1}$ and $X_{i+1}$ can
together decrypt team member $X_i$’s value, where the man-
ger is considered team member $X_0$, and the indexing is
modulo $N + 1$. Each team member does not need to be able
to trust every other team member, but it must be possible
to construct a chain of trust in which team member $X_i$ trust
team members $X_{i-1}$ and $X_{i+1}$.

By giving other players the same keys as the manager,
this protocol supports cases in which multiple players can
decrypt only the statistic. This example illustrates that the
hierarchical structure of the keys does not necessarily cor-
directly with the access control structure, which is
determined by who receives which keys.

5.2 Generalizations

The protocol of Section 5.1 can be generalized to require
less trust among users. By increasing the number of keys, we
can construct schemes secure against $k$-collusion. Consider
team members $X_i$ for $1 \leq i \leq N$ and a manager $X_0$.

- **Setup:** Same as for the protocol of Section 5.1
  with a graph structure in which every team member
  and the manager has exactly $k + 1$ neighbors.

- **Key Generation and Sharing:** Team member $X_i$
generates keys $k_{ij}$ for every $j < i$ such that $X_j$
  and $X_i$ are neighbors. Team member $X_i$ shares key $k_{ij}$
  with neighbor $X_j$.

- **Encrypt:** Each team member $X_i$ encrypts her value
  $v_i$ at time $t$ by adding pad $r_{ij} = h(f_{ij}(t))$ for every
  neighbor $X_j$ with $j < i$ and subtracting the pad $r_{ij}$
  for every neighbor $X_j$ with $j > i$, where all arithmetic
  is done modulo $M$:

  $$c_i = v_i + \sum_{j \in subr(i)} (-1)^\chi(j) r_{ij} \mod M,$$

  where $\chi(j)$ is 0 for $j < i$ and 1 for $j > i$.

- **Compute Encrypted Stat:** Same as before.

- **Decrypt Group Stat:** In a sum of ciphertexts over
  all team members at a particularly time, all pads can-
  cell except for the $r_{ij}$ with $j = 0$. The manager $X_0$ has
  the keys necessary to compute these pads, and there-
  fore she can decrypt the sum over all values, at a given
time step, for the team of users.

The protocol of Section 5.1 can be generalized to handle
arbitrary multiple-level hierarchies with a tree structure in
such a way that nodes at higher levels can decrypt only summary
statistics over all leaf nodes below them, and cannot
decrypt any lower-level statistics, including individual val-
dues. The protocol of Section 5.1 and its generalization to
multiple-level hierarchies, support the contribution of values
for leaf nodes and not from players higher up in the hierar-
chy. To support situations in which some or all managers,
at various levels in the hierarchy, contribute values, we can
add the manager in as the first player below that manager.
Another generalization supports a structure in which a man-
ger gets access only to statistics, for an individual or a team,
with more coarse-grained time buckets than are available to
the individual or the team.

6. RELATED WORK

Several commercial and research systems support aware-
ness in organizations. Most provide awareness of a single
channel of information. Shared calendars, for example, en-
able workers to be aware of scheduled activities. Systems
such as Portholes [11] allow a worker to observe the activ-
ity of a co-worker via a switched, closed-circuit video feed.
Other research systems seek to provide information about
a person’s current state. Fogarty and Hudson’s toolkit [12]
used computer activity, ambient sound, and other sensors,
to predict a person’s level of interruptibility. Other systems
(e.g., [2] [10] [13] [14]), performed similar functions with dif-
ferent configurations of sensors.

Many of these systems do not save past state, and none
have adequate mechanisms for protecting and controlling
access to historical data. Shared calendars sometimes have
predefined, explicit mechanisms for how long data are re-
tained and how historical information can be accessed, and
chat clients often provide settings that control whether chat
logs are retained. Retention and reuse mechanisms for sens-
ing and video technologies are less developed; many systems,
such as Portholes, rely simply on management policy and
trust in the use of the system. While we applied our privacy
scheme to the MyUnity system, our work could be adapted
to work with other tools.

Castelluccia et al. [8] designed their symmetric homomor-
phic encryption scheme to aggregate data in wireless sensor
networks. In their setting, no data are stored, and a fixed
computation is carried out as the encrypted data traverse
the network. They have essentially one client, whereas we
have many. Applications of Castelluccia et al.’s scheme have
been limited to the wireless sensor network setting; thus, to
the best of our knowledge, our work is the first application
of their symmetric homomorphic encryption scheme outside
of the wireless sensor network area.

Benaloh et al. [5] develop a notion of ‘patient controlled
encryption’ that enables a patient to share decryption and
search capabilities to parts of her patient record. All pa-
ient records are encrypted so that administrators of health
data servers can learn nothing about the contents. Benaloh
et al. do not consider computations, other than search, on
the encrypted records. Molina et al. [15] consider a setting
close to ours. They study how to enable clinical research
without giving patient records to the researchers. In their
solution, caregivers, who have full access to patient records,
use multiparty computation with public key homomorphic
encryption to answer researcher aggregation queries.

There is an extensive literature on standard hierarchical
access control structures via encryption. Atallah et al. [4]
provide an attractive mechanism for implementing standard
hierarchical access control structures for arbitrary directed
acyclic graphs. Their paper contains an extensive review of
prior work on hierarchical access control mechanisms. This
work does not consider computation on untrusted servers.

Rastogi and Nath [17] provide differentially private aggre-
gation of encrypted data. Differential privacy is concerned
with preventing the deduction of individual attributes from
data such as aggregate statistics. Differentially private me-
chanisms enable the release of useful data while meeting pri-
vacy guarantees. The standard setting in differential privacy
is a trusted curator, with access to all data, who implements
the mechanism and releases the resulting data. In many set-
ings, including ours, it is desirable that no one have access
to all of the data. Rastogi and Nath use the public key Pailler threshold homomorphic encryption to achieve differentially private aggregation without a trusted curator. Their decryption, unlike ours, is multiparty.

7. DISCUSSION

To deploy a system with the basic protocol in larger organizations, the only significant scaling issue is that the number of pad computations required to decrypt a sum increases linearly with the number of values contributing to the sum. While the protocols of Section 5 were designed to support full “need to know” security, they have the added benefit of reducing to a constant the number of pad computations needed to decrypt. For this reason, in large organizations in which statistics are desired over a large number of employees, it may be worth implementing the protocols of Section 6 even if standard hierarchical access control is desired and the manager is given all keys. The downside to doing so is that decryption of other statistics, for example statistics for a single individual, take more pad computations to compute. Doing so also increases the encryption time, but that is less of an issue, since encryption is fast. The particular application will determine whether this tradeoff is worthwhile. For situations in which the alternate hierarchical access control structure (managers can decrypt aggregate statistics but not individual data) is desired, this speedup comes for free.

A number of residual risks remain. We have not been concerned with protecting the integrity of the data. Because the encryption is homomorphic, it is malleable, so a separate mechanism would need to be introduced to protect against tampering with encrypted records. Castelluccia et al. [8] provide such a mechanism. We also do not guard against tampering at the sensor level. MyUnity users have control over most sensors and have the power to opt out of specific sensors, or the whole system. In other settings, tampering may be more of a concern.

A more significant remaining risk is that the release of aggregate statistics may enable a party to deduce individual values or trends. As mentioned in Section 6 differential privacy mechanisms address this threat. Our concern is orthogonal to that of differential privacy, and the simple structure of our schemes means that they could be combined with differential privacy techniques to support the computation of statistics with a differential privacy guarantee without the need for the individual contributors to share their individual values with anyone. We leave for future work how best to combine the two. Noise needs to be added by the participants, not the untrusted third party, and there are interesting questions as to what noise and how much should be added to obtain privacy guarantees while preserving efficiency and utility.

A risk associated with the protocols of Section 5 is that users can collude to decrypt another user’s values. Increasing the minimum number of users who can successfully collude comes at the expense of requiring each user to use more keys to encrypt each value. It would be good to have a protocol that does not require any trust among the team members. More specifically, an interesting open question is how to support a structure in which a manager can decrypt an aggregate of all team members values, but none of the individual values, and in which no group of entities can collude to decrypt any other individual’s values. A related question is whether it is possible to construct $n$-tuples of pseudorandom functions $\{f, g_1, \ldots, g_{n-1}\}$ such that $f(i) = g_1(i) + \cdots + g_{n-1}(i)$ for all positive integers $i$ and where the ability to compute any one of the functions does not imply the ability to compute any of the other functions. The existence of such a construction would enable an alternative mechanism in which a manager could decrypt the aggregate of values from $n-1$ employees, but not the individual values, and groups cannot collude to decrypt another individual’s value. It also reduces the number of keys a user uses to encrypt from two to one.

We know how to extend the protocol of Section 5.1 to arbitrary hierarchical tree structures. Many organizations do not follow a strict tree structure. An employee may be part of two different teams, for example. We know how to generalize this protocol to some non-tree hierarchies, but do not see how to generalize it to others. It is an interesting open question as to what types of non-tree structures can be supported by a similar mechanism.

We hope this work will inspire others to find “need to know” protocols for computations beyond the summation operations we present here.

8. CONCLUSIONS

We defined “need to know” security in which an individual has full access to her own data, a third party learns nothing about the data but can process the data without learning anything about the data values, and other users, such as analysts, learn only the desired statistics. Such trust structures exist in many settings, such as user studies, statistics on patient outcomes, and release of usage data from social networking sites, where there is a presumption of privacy for the individuals or individuals have the power to opt out of the data collection. We designed a practical protocol, that meets all but one of the criteria for a “need to know” protocol, and is useful in its own right. We implemented this protocol and integrated it with an established awareness system. We extended this protocol to obtain a full “need to know” protocol that is efficient and meets all of the criteria for a “need to know” protocol.

Our implementation shows that our protocols are practical in the setting of an awareness application for a small organization. We have not yet attempted to optimize our implementation. While the current implementation does not use a third party provider, the protocol’s properties mean that it could make use of commodity cloud services for computation on and storage of sensitive, private data. The untrusted components all can be pushed to federated or external resource providers, which is particularly important for scaling the system to a larger organization. Furthermore, the computation of the encrypted sum is easily parallelized, so it can be spread across different cloud nodes, or at least different threads. Similarly, the computation of the pad sum by the client is easily parallelized to different threads.

We hope this work will spur the development of a wide variety of “need to know” protocols that support other types of computation, more complex sharing structures, and improved efficiency and robustness.

9. ACKNOWLEDGMENTS

Many thanks to the MyUnity team for providing us with access to the code, some of their data, and graphics for inclusion in this paper.
10. REFERENCES


