# String Discovery for Private Analytics

Istemi Ekin Akkus
MPI-SWS
iakkus@mpi-sws.org

Ruichuan Chen
Bell Labs / Alcatel-Lucent
ruichuan.chen@alcatel-lucent.com

Paul Francis
MPI-SWS
francis@mpi-sws.org

Technical Report MPI-SWS-2013-006
November 2013

## Abstract

A number of research systems enable analysts to aggregate user data that is distributed across user devices while preventing online tracking and providing users with differential privacy guarantees. These systems rely on pre-defined string values to release relevant user data in a controlled fashion. Unfortunately, many string values (e.g., tags in a photo application) may not be easily predicted. Existing private aggregation systems that can be used to *discover* strings for private analytics purposes exhibit serious shortcomings, such as heavy client-side operations and an inability to deal with malicious clients supplying incorrect data. In this paper, we present a practical and privacy-preserving string discovery system that provides analysts with previously unknown strings and limits the effects of malicious clients, while supporting a variety of user devices with varying computation and bandwidth resources. To achieve these goals, our system employs the exclusive-OR (XOR) operation as its crypto primitive, and utilizes a novel method to determine the equivalence of two XOR-encrypted strings without revealing them. We present our design, analyze its privacy properties and evaluate its feasibility. Our results show that our system outperforms the closest system by several orders of magnitude for client-side computations and one order of magnitude for server-side computations.

## 1 Introduction

Tracking users on the web benefits analysts and data aggregators. For instance, web publishers learn more about their users via web analytics, and aggregators help them monetize their content by mediating between publishers and advertisers. This arrangement comes with a price for user privacy: Aggregators can build profiles about individual users [45] with the sensitive information they obtain. Users' trust in aggregators not to misuse this information has been violated in many cases [8–10]. As a response, researchers and industry have proposed methods to detect and prevent tracking [3, 4, 6, 11, 13, 15, 45]. While these methods protect privacy, they significantly reduce the benefits of analytics by limiting the information analysts get about their users.

Recent proposals avoid this inherent trade-off by providing analysts with the same aggregate analytics data they obtain from data aggregators today, but without tracking [16, 23, 24, 33, 37]. The common approach utilized in these systems is to store the user data at the user device and release it with the help of a client software in a controlled fashion, using either proxies [16, 23, 24, 33], or restricted interfaces [37]. While aggregating user data, these systems add differentially-private noise [27, 28] to protect individual user privacy.

One disadvantage of these systems, however, is that they require a set of pre-defined string values that are relevant to the user data they want to aggregate, such that the client only releases data re-

garding those values. These string values are either used as potential answers in analysts' queries over user data [16, 23, 24], or as counter names [33, 37]. This requirement on the string values limits the applicability of these systems. For instance, imagine a scenario where the developer of a photo gallery application would like to gather statistics about the tag values its users are assigning to their photos. For better functionality, the developer allows the users to enter free text as tag values. In this case, defining a list of potential tag values in advance may be difficult or impossible.

Some previous systems [17, 20] do not suffer from this drawback because they provide private data aggregation with *unknown strings*, but have other limitations that make them unsuitable for analytics scenarios that are similar to our example. First, and most importantly, these systems either employ general-purpose secure multiparty computation protocols [20], or rely on expensive cryptographic operations [17], such as oblivious transfer [43], public-key cryptography or zero-knowledge proofs [32]. These operations put a significant burden on the clients, whose computing resources may be limited in large-scale, distributed environments such as the web. For example, users increasingly access the web via mobile devices [5, 7], which have limited capabilities compared to personal computers.

Second, these systems cannot detect whether an adversarial client participating in the data aggregation is trying to manipulate the results. The purpose of this manipulation may be to breach the privacy of other clients, and/or to reduce the utility of results by supplying incorrect data. In an environment in which there are millions of clients who cannot be generally trusted to provide correct data, this issue significantly reduces the suitability of these systems for private analytics.

Third, these systems are designed for aggregating and correlating network events across big organizations (e.g., ASes). This specialization limits the length of strings these systems can aggregate because of the underlying cryptographic operations. For example, Sepia [20] assumes a string size of 32-bits (i.e., an IP address). In our example scenario, the strings can be much longer. Furthermore, these systems only deal with one type of string, which may be limiting in a web setting (e.g., many different applications with different string types).

Finally, to reveal an unknown string, these systems require that its count is above a threshold [17], similar to k-anonymity [51]. This property opens the attack in which an adversary creates $k-1$ instances of a rare string (using adversarial clients as above) to expose the existence of a user. In fact, previous private analytics systems [16, 23, 24, 33, 37] preferred using differential privacy in part to defeat this trivial attack.

In this paper, we present the design and evaluation of a practical and private string discovery system. Our system can discover unknown strings belonging to a wide-range of *string types* (e.g., photo tags, applications installed, products viewed, websites visited), and provide analysts with string values that can be used in various analytics systems. While achieving this goal, our system places very little overhead on the clients.

In our system, the user data resides at a user device running our client software. The client periodically participates in string discovery procedures run by the aggregator (the entity providing the string discovery service), and supplies encrypted strings using a low-cost (XOR) form of encryption. With the help of two honest-but-curious proxies, the aggregator aggregates encrypted strings. The aggregator then provides the analysts with discovered strings of various string types.

Care must be taken while discovering a string. From a client's perspective, rare strings shared by few clients may leak privacy, and thus, should not be discovered. For example, the tag "Alex Finkelmeier getting drunk" is rarer than "Mom's birthday", and can leak a client's identity if discovered. From an analyst's perspective, the discovered strings are presumably more useful if they belong to a relatively large client population. In our photo application example, the developer may only be interested in tags that are used by a substantial number of clients. For these reasons, our system discovers a string only if there are *sufficient* clients: strings with fewer clients than a *discovery threshold* are not discovered.

Using a fixed threshold value, however, is problem-

atic due to the $k-1$ sybils attack described above. Note that just adding differentially-private (or any kind of) noise does not address this attack: if the threshold is applied before adding noise, $k-1$ sybils attack would still succeed, because the mere reporting of a string (even with a noisy count) would indicate that there exists *at least* one client with that string: if not, the string would not have passed the threshold, noise would not have been added to its count, and it would not have been reported. Our system deals with this problem by employing differential privacy mechanisms to add noise to a string's count *before* applying the threshold, and thus, producing uncertainty in the adversary's view: is it a client or is it just noise? In other words, only strings whose differentially-private noisy counts pass the threshold are discovered.

To determine whether a particular string should be discovered with our noise-before-threshold mechanism, we first need to count the number of clients with that string value. The key challenge here is to count the clients without revealing their strings, which in turn requires determining the equivalence of encrypted strings. One approach would be to rely on heavy crypto operations (e.g., oblivious transfer [43]) as suggested by Applebaum et al. [17]; however, this approach creates problems with low-power clients. A low-cost alternative would be for the clients to hash their strings with a secure function using a secret shared with a component (e.g., SHA-1 with a secret salt). Unfortunately, a component could easily run a fake client to learn the secret and utilize rainbow tables to determine the existence of clients with specific strings. Our system uses a *blind comparison* method to distinguish (encrypted) client strings and count them without knowing their actual values. This method utilizes low-cost primitives (XOR and hash), and significantly reduces the burden on the clients by avoiding expensive crypto operations. This blind comparison also enables our system to detect malicious clients and limit their effects on the string counts without violating the privacy of honest clients.

The contributions of this paper are as follows. To the best of our knowledge, we propose the first practical and private analytics system that enables analysts to discover previously unknown string values with differentially-private counts. We describe and analyze a novel design that creates little overhead on the client, but nevertheless provides a method to distinguish and count distinct strings without revealing them. Finally, we demonstrate our system's feasibility using simulations with generated data sets that are based on the distributions of real-world data: website popularity from Quantcast [12] and recent search phrases from a large search engine.

The next section presents our assumptions and goals. We give an overview of our system in Section 3, and describe the primitives we use in Section 4. Sections 5, 6 and 7 present the details of our system, detection of malicious clients trying to skew the string counts and optimizations, respectively. We analyze our system's privacy properties in Section 8 and evaluate its feasibility in Section 9. We discuss related work in Section 10, and conclude in Section 11.

# 2 Assumptions & Goals

## 2.1 Components & Trust Assumptions

There are three types of components in our system: client, aggregator and proxies. These components already exist in today's aggregation infrastructure, except for the proxies. Proxies, however, have been widely proposed for aggregation purposes [17, 23, 24, 33], and we also adopt this approach.

**Client.** The client is a piece of software that stores user data locally, similar to other systems [16, 23, 24]. The client participates in the aggregator's string discovery procedures by sending encrypted strings it has. Besides the strings, the client also keeps some metadata regarding the *string types*. A *generic string type* may be of interest to many analysts. Examples of generic string types include websites visited, extensions installed and search phrases. An *analyst-specific string type* may be useful to only one or a few analysts. An example of an analyst-specific string type is the tag values in a specific application (e.g., tags in a photo app). How exactly the client obtains these

data is outside our scope; however, we note that the browser already sees much of these data.

The client typically runs on a user device, but may also run on a different, trusted device. We assume that the user trusts the client to safeguard the data it stores and its operation, just as users trust their browsers for operations such as certificate management or TLS connections. We do not protect against the case in which a client's device is infected with malware: such malware can violate user privacy in many ways our system does not cover. We also assume that fake clients can be run by other components to violate an honest client's privacy. A client can also act maliciously against the aggregator and try to skew string counts by sending the same string multiple times.

**Aggregator.** The aggregator is an entity that provides the string discovery service. This service reports previously unknown strings and their differentially-private counts. These strings may be used by analysts that want to query distributed user data with other systems, such as [16, 23, 24]. We assume that the aggregator is honest-but-curious: it follows the protocol and does not collude with the proxies. However, it may operate fake clients to try to link or deanonymize client strings.

**Proxies.** The proxies act as anonymizing proxies between clients and the aggregator, and enable the aggregation of encrypted user data and discovery of strings. They also help the aggregator in detecting and limiting malicious clients trying to skew the counts. We assume that the proxies are also honest-but-curious: They do not lie about string counts, and do not collude with the aggregator nor with each other; however, like the aggregator, they can run fake clients to link or deanonymize client strings.

Although our assumption about honest-but-curious aggregator and proxies is weaker than a more general model, in which these entities could be malicious, we think that it reflects the reality on the Internet: The aggregator operates a business by providing string discovery service for analysts. The proxies can be operated by independent companies and/or privacy watchdogs. All of these entities would put their non-collusion statement in their privacy policies, making them legally liable. Furthermore, any entity not following the protocol would risk losing reputation and customers. Previous systems also make similar assumptions [16, 17, 23, 24, 26, 33].

Finally, we assume the aggregator and the proxies are not impersonated, and all point-to-point connections use TLS (i.e., messages cannot be eavesdropped upon and modified in flight).

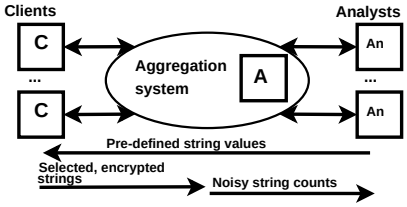


Figure 1: An overview of existing private analytics systems.

## 2.2 Functionality Goals

Figure 1 shows an overview of existing private analytics systems [16, 23, 24, 33, 37], which provide analysts with differentially-private aggregate analytics information. However, they require analysts to pre-define a list of potential string values, which can be difficult for many analytics scenarios (e.g., tags). Our main functionality goal is to discover unknown strings and report them to the analysts with their noisy counts.

Our system should scale well, both on the server and the client side. There are potentially millions of clients with tens of millions of strings. Thus, operations to discover strings should be fast. Additionally, the clients can run across a diverse set of user devices from smartphones to powerful desktop computers, possessing different computation and bandwidth resources. To support such a diverse client population, the client-side operations should not incur much overhead.

Finally, our system should detect malicious clients and limit their effect on the string counts. An inaccurate count may cause our system's utility to suffer, because the discovered strings may not be beneficial for the analysts. It can also cause the discovery threshold to be ineffective, and thus, can be a privacy

issue: in order to infer honest clients' strings, other components may run fake clients and skew string counts.

### 2.3 Privacy Goals

Our system has two main privacy goals: (i) Only discover strings reported by a sufficient, noisy number of clients, and (ii) Report discovered strings with differentially-private counts.

Our first goal requires us to define the term 'sufficient'. Unfortunately, this definition varies: one analyst may be interested only in strings reported by at least 1K clients, whereas another may set a much smaller threshold. Nevertheless, the system should impose a threshold of some kind. Our goal is not to find a 'one-size-fits-all' threshold, but rather *to find a method to only discover the strings that satisfy some 'sufficient' definition.* In other words, any string not satisfying the condition must not be revealed to any component—not even to the aggregator.

A subgoal of this goal is that the method to discover strings should not reveal the string values until they are deemed discovered. Furthermore, a discovered string should not reveal any information about any other string. For example, guessing a common string value should not leak any information about a rare string during the discovery process.

To satisfy our second goal, we need to add differentially-private noise to the counts of the discovered strings. At the same time, no component should be able to learn the noise-free count of a string—the components should be oblivious to the total noise added to a string count.

At all times during this process, the system should ensure that clients participate in string discovery procedures in an anonymous and unlinkable fashion. This participation should be anonymous, such that given a string value or a string type, no component should be able to associate it with a client. The participation should also be unlinkable, such that given two string values or string types, no component should be able to tell if they come from the same client.

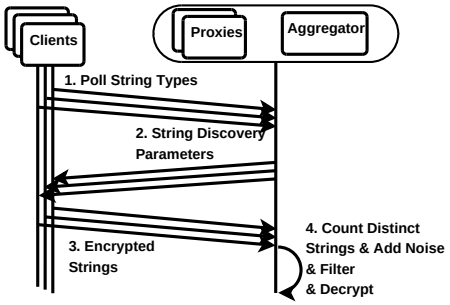


Figure 2: Overview of our system's operation.

## 3 System Overview

Figure 2 shows an overview of our system's operation. All string discovery procedures are periodically run by the aggregator. The aggregator handles all interactions with the analysts, who may express their interest in learning strings of a specific string type. The aggregator controls access to the discovered strings. For example, strings of an analyst-specific string type can only be learned by that analyst.

To participate in discovery procedures, clients periodically send encrypted polling requests to the aggregator with their string types (step 1 in Figure 2). Our system uses exclusive-OR (XOR) as the crypto primitive, similar to SplitX [23] (§4.1). Using XOR enables our system to support low-power and low-bandwidth client devices, significantly reducing the burden on the clients compared to previous systems.

After the polling request is XOR-encrypted, it is sent via the proxies to provide clients with anonymity and unlinkability regarding their string types. In other words, the aggregator cannot associate clients with string types, and cannot determine if any two requests come from the same client. Because of the encryption, the proxies also cannot learn a client's string types.

After receiving the polling request for a string type, the aggregator returns the associated string discovery parameters to the client via the proxies (step 2). These parameters include the string type, the discovery period and the $\epsilon$ value that will be used for the differentially-private noise (§5.1). The discovery pe-

riod is used by the aggregator to synchronize the beginning and end times of string discovery procedures for many string types. This synchronization enables our system to detect malicious clients (§6) as well as provide some additional privacy guarantees for clients by grouping aggregation of multiple string types (§8). Like the polling requests, the discovery parameters are also XOR-encrypted before being sent.

After obtaining the discovery parameters, the client retrieves the strings associated with the string type from its local database. Each distinct string is XOR-encrypted by the client before being returned for aggregation (step 3) (§5.2).

Besides reducing the burden on the clients compared to previous systems, using XOR as the crypto primitive also enables our system to aggregate encrypted client strings without revealing them (step 4). This aggregation utilizes a low-cost comparison method that only reveals if any two XOR-encrypted strings are equivalent (§4.2). With this method, our system counts distinct strings, adds differentially-private noise and applies the discovery threshold—all without learning the actual string values (§5.3). Once it is determined which encrypted strings have counts above the threshold and should be revealed, our system decrypts those strings. The aggregator then reports those strings along with their associated noisy counts to the appropriate analysts (not shown in Figure 2).

Note that all the components in the system should be oblivious to the total noise added to each string count, such that no single component can determine the noise-free count of a string. Our system ensures this property by employing two proxies: each proxy compares and counts the strings of approximately half of the clients in parallel, and adds noise to discovered strings independently, making the final noise added to a given string value oblivious to all components (§5.4).

To ensure the accuracy of the counts, our system checks for duplicate strings that may have been reported by malicious clients (§6) before proceeding with the above counting protocol. This duplicate detection utilizes the same low-cost comparison method, and limits a malicious client's effect on the string counts without violating the privacy of honest clients.

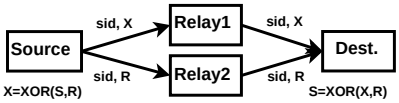


Figure 3: Splitting and joining. String $S$ is split into $\{X, R\}$. These split messages are then sent over two relays with the same *sid*, such that the destination can pair matching $X$ and $R$ values, and join them to obtain $S$.

The next section presents our system's building blocks, including the XOR-encryption and our blind comparison method. Section 5 provides the details of our system's operation.

# 4 Building Blocks

We first describe the low-cost XOR-encryption that enables our system to support a variety of clients, including resource-constrained mobile devices. We then introduce our novel blind comparison method that determines the equivalence of two XOR-encrypted strings without revealing their values. Finally, we give some background on differential privacy.

## 4.1 XOR-Encryption: Splitting and Joining

Our system utilizes XOR as the underlying crypto primitive, similar to SplitX [23]. Splitting is the equivalent of encryption, and joining is the equivalent of decryption. These operations enable a source to anonymously send a string to a destination using two different relays, without the relays learning the string because of the encryption. At the same time, the crypto operations for the source and destination are low-cost, because they are based on XOR (see Figure 3).

When a source wants to send a string ($S$) to a destination, it splits $S$ to obtain two split messages, $X$ and $R$. Let $L$ be the length of $S$. The source first

generates a random string, $R$, of length $L$ using a Pseudo Random Number Generator (PRNG) with a secure *seed*, and then encrypts $S$ with $R$:

$$\begin{aligned} R &= PRNG(seed, L) \\ X &= S \oplus R \end{aligned}$$

The source also generates a *split identifier* (*sid*). The split identifier is a large value (e.g., 128 bits) and ensures the two parts of the split message are uniquely paired by the destination with high probability. The source sends $X$ and $R$ to the relays:

$$\begin{aligned} Source \rightarrow Relay_1 : sid, X \\ Source \rightarrow Relay_2 : sid, R \end{aligned}$$

The relays then forward the split messages to the destination. Borrowing notation from [23], we denote the split message pair $\{X, R\}$ as $\underline{S}$ (underlined $S$), and write this action as:

$$Source \xrightarrow[Relay_2]{Relay_1} Destination : \underline{S}$$

The destination then decrypts the split messages to obtain $S$:

$$S = X \oplus R$$

For efficiency purposes, the source can send the $\langle seed, L \rangle$ tuple instead of $R$. The destination can first generate $R$ using the $\langle seed, L \rangle$, and then obtain $S$ as above.

### 4.2 Blind Comparison via pairwise-XOR Hash ($PXH$)

To count distinct string values without revealing them, our system utilizes a blind comparison method to determine the equivalence of any two XOR-encrypted messages.

Consider two strings $S_i$ and $S_j$ with corresponding split message pairs $\{X_i, R_i\}$ and $\{X_j, R_j\}$ and split identifiers $sid_i$ and $sid_j$, respectively. Recall that the split messages are held by two different relays (i.e., $\text{Relay}_1$ holds $X_i$ and $X_j$, and $\text{Relay}_2$ holds $R_i$ and $R_j$). Let $H$ be a secure hash function (e.g., SHA-1). For $\text{Relay}_1$ and $\text{Relay}_2$ respectively, we define the pairwise-XOR hash ($PXH$) operation as:

$$\begin{aligned} PXH_{Relay_1}(sid_i, sid_j) &:= H(X_i \oplus X_j) \\ PXH_{Relay_2}(sid_i, sid_j) &:= H(R_i \oplus R_j) \end{aligned}$$

Remember that $X_i = S_i \oplus R_i$ and $X_j = S_j \oplus R_j$. Therefore, the above equations can be rewritten as:

$$\begin{aligned} PXH_{Relay_1}(sid_i, sid_j) &= H((S_i \oplus R_i) \oplus (S_j \oplus R_j)) \\ PXH_{Relay_2}(sid_i, sid_j) &= H(R_i \oplus R_j) \end{aligned}$$

If $S_i = S_j$, then $PXH_{Relay_1} = PXH_{Relay_2} = H(R_i \oplus R_j)$. Thus, by comparing $PXH_{Relay_1}$ and $PXH_{Relay_2}$, our system can *blindly* determine whether $S_i$ and $S_j$ are equivalent.

The secure hash ensures that neither string can be obtained, even if the other string is guessed (e.g., a common string).

### 4.3 Differential Privacy

We rely on differential privacy for adding noise to (encrypted) string counts. After adding noise, our system decrypts the strings whose noisy counts pass the discovery threshold. A computation, $C$, provides $\epsilon$-differential privacy [27,28,30] if the following inequality is satisfied for all data sets, $D_1$ and $D_2$, that differ on one record, and for all outputs of $O \subseteq Range(C)$:

$$\Pr[C(D_1) \in O] \leq \exp(\epsilon) \times \Pr[C(D_2) \in O] \qquad (1)$$

Namely, the probability of a computation producing a given output is almost independent of the existence of any individual record in the data set. This property is achieved by adding noise to the output of the computation using the privacy parameter $\epsilon$: the smaller $\epsilon$, the greater the privacy.

## 5 Design Details

In this section, we present our system's details. The discovery procedure starts with the client receiving the string discovery parameters (§5.1). Afterwards, encrypted strings are collected from the clients

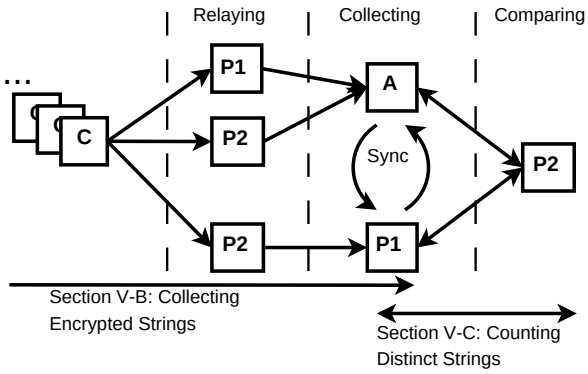


Figure 4: Mirror operation 1. The vertical, dashed lines separate the roles of each component: relaying, collecting, and comparing. The arrows labeled with section numbers represent the information flow in each step (Collecting Encrypted Strings (§5.2) and Counting Distinct Strings (§5.3)).

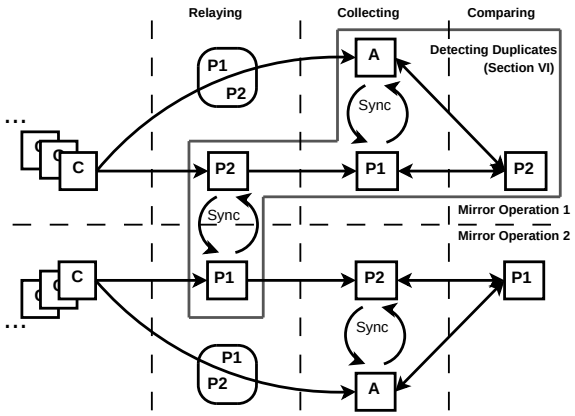


Figure 5: Depiction of our complete system. The horizontal, dashed line separates the mirror operations: $\text{Proxy}_1$ and $\text{Proxy}_2$ swap their roles for collecting and comparing across the line. Components involved in duplicate detection are shown within the rectangular shape (§6).

(§5.2). Finally, these strings are blindly compared and counted, after which differentially-private noise is added to the counts (§5.3).

Throughout these steps, there are three separate roles that each component can perform: relaying, collecting, and comparing. These roles are separated by the vertical, dashed lines in Figure 4. The aggregator only assumes the role of collecting, whereas the proxies assume all three roles (though not on the same data at the same time). Both proxies assume the role of relaying between the clients and the aggregator.

As stated in Section 3, to ensure that all components in the system are oblivious to the total noise added to a string count, our system employs two proxies, $\text{Proxy}_1$ and $\text{Proxy}_2$: $\text{Proxy}_1$ compares and counts the strings of roughly half the clients while $\text{Proxy}_2$ compares and counts the strings of the other half. Both proxies independently add differentially-private noise to the counts of the distinct strings. In other words, for half the clients, $\text{Proxy}_1$ and $\text{Proxy}_2$ swap their roles. We refer to this role swap as "mirror operations", which are shown in Figure 5 separated by the horizontal, dashed line. The mirror operations enable us to *obliviously* add noise to string counts (§5.4).

The mirror operations are generally concurrent. There are only two times there is any interaction between them: The first interaction is during the detection of duplicates (§6), requiring synchronization between the proxies that relay to the collecting proxy in their respective mirror operations (i.e., $\text{Proxy}_2$ in mirror operation 1 and $\text{Proxy}_1$ in mirror operation 2). The second interaction is at the end of the counting step, when the comparing proxies send independently discovered strings to the aggregator, which is present in both mirror operations.

For clarity, we describe our protocol's steps in mirror operation 1, in which $\text{Proxy}_2$ assumes the comparing role (Figure 4). The arrows in Figure 4 show the flow of information in the collection and counting steps: from clients to the collecting components ($\text{Proxy}_1$ and the aggregator), and between the collecting components and the comparing component ($\text{Proxy}_2$).

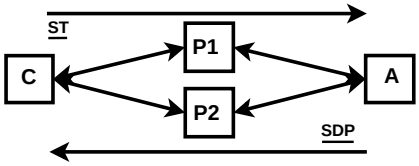


Figure 6: Initialization of string discovery. For each of its string types (ST), the client periodically sends a split request to the aggregator. The aggregator returns the associated string discovery parameters (SDP) after splitting them.

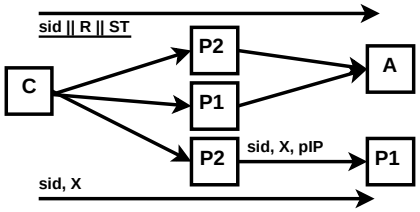


Figure 7: Collection of encrypted strings in mirror operation 1. The client splits each string $S$ and obtains $\{X, R\}$. Each $X$ is sent to the collecting component ($\text{Proxy}_1$) via the relaying ($\text{Proxy}_2$). The client also splits $sid, R, ST$ and sends it to the aggregator via the two relaying proxies.

## 5.1 Initializing String Discovery

To start participating in ongoing string discovery procedures, the client uses an anonymous polling mechanism similar to SplitX [23]. The client periodically polls the aggregator for *string discovery parameters* (SDP) by submitting each of its *string type*s (ST) present in its local database (Figure 6). For each $ST$, the client creates a separate request, splits it, and sends it to the aggregator using the proxies as a relay:

$$C \xrightarrow[P2]{P1} A : \underline{ST}$$

The aggregator retrieves the parameters associated with each $ST$, splits them and sends them to the client via the proxies:

$$A \xrightarrow[P2]{P1} C : \underline{SDP}$$

The client joins the split messages to obtain the parameters.

There are three parameters: $ST$, $DT_{End}$, and $\epsilon$. $ST$ represents the string type of the discovery procedure. $DT_{End}$ denotes the end time of the discovery (i.e., the last time a client can submit its strings). Discovery procedures are run in discovery epochs, such that their start and finish times are synchronized. A discovery procedure spans only one epoch, but can be repeated. $\epsilon$ defines the privacy parameter and is set to provide sufficient noise to the counts. Optionally, the aggregator can add a fourth parameter, which is a list of hashes of previously discovered strings, such that the client will only send strings not in this list. If there is no discovery procedure for a string type in the current epoch, its $SDP$ will be empty.

## 5.2 Collecting Encrypted Strings

The client records the $\epsilon$ value for the aggregator and retrieves all strings of type $ST$ from the local database. For each distinct string ($S$), the client creates a split message pair ($\{X, R\}$) and a split identifier ($sid$). These split messages, along with their $sid$, will need be sent to the collecting components: $\text{Proxy}_1$ will receive $sid$ and $X$, and the aggregator will receive $sid$ and $R$. The aggregator also needs to receive the $ST$ value. The $ST$ will be used to identify the type of each client string, and to group encrypted client strings into comparison lists (§5.3). The client uses the same collecting components throughout the current discovery epoch (i.e., the client participates in one mirror operation per epoch).

Figure 7 shows the details of the collection. To anonymously send $X$ to $\text{Proxy}_1$, the client uses $\text{Proxy}_2$ as a relay:

$$C \quad \rightarrow \quad P_2 : sid, X \tag{2}$$

$\text{Proxy}_2$ assigns the client a *pseudo IP address* ($pIP$) that is only valid for the current discovery epoch (i.e., will change in the next epoch). The $pIP$ value will

be used to distinguish strings from the same client in the duplicate detection (§6).

After attaching the $pIP$ value, Proxy$_2$ relays each $X$ value the client sends to Proxy$_1$:

$$P_2 \quad \rightarrow \quad P_1 : sid, X, pIP$$

To prevent the aggregator from linking the client with a particular $ST$ value, the $sid$, $R$ and $ST$ values are concatenated (shown as $||$), split and sent to the aggregator via both proxies:

$$C \xrightarrow[P_2]{P_1} A : \underline{sid||R||ST} \tag{3}$$

The aggregator then joins the split messages to obtain the $sid$, $R$ and $ST$ values.

## 5.3 Counting Distinct Strings & Comparison by a Third Party

At this point, the collecting components in mirror operation 1 (Proxy$_1$ and the aggregator) each have one split message of the XOR-encrypted string paired with its $sid$ value. To ensure that both components have the same set of $sid$ values, they exchange their $sid$ sets and discard any unpaired split messages. They then proceed with the counting process.

The counting process involves the computation and comparison of $PXH_A$ and $PXH_{P_1}$ values for each possible $\langle sid, sid \rangle$ tuple. Although the comparison is blind and does not reveal the strings being compared, knowledge about one string can be used to infer the other string, if $PXH_A$ and $PXH_{P_1}$ values are equal. We need to ensure that no component can exploit this property and deanonymize a client's string.

Recall that either of the collecting components can easily operate fake clients. For example, Proxy$_1$ can run a fake client to send a known string for a discovery procedure, and identify this encrypted string via its $sid$ value. If Proxy$_1$ has both the $PXH_A$ and $PXH_{P_1}$ values involving this known string and finds that it is equivalent to an honest client's string, Proxy$_1$ can deanonymize the honest client's string. Similarly, the aggregator can also perform the same attack, making it unsafe for either of these components to make the comparison.

As a result, the comparison must be performed by a third component, Proxy$_2$. Although Proxy$_2$ can also operate fake clients and send known strings with known $sid$ values, Proxy$_1$ and the aggregator can prevent it from launching the same attack: They share a random secret, $R_s$, valid for one discovery epoch, and modify the original $sid$ values in a deterministic, but pseudorandom fashion. Let $H$ be a secure hash function (e.g., SHA-1). To prevent Proxy$_2$ from identifying its strings via $sid$ values, they are overwritten as:

$$sid'_i = H(sid_i||R_s)$$

Another way for Proxy$_2$ to identify its strings is to use the $R$ values of encrypted strings. Consider that Proxy$_2$'s clients report two strings using $R_k$ and $R_m$ with $sid_k$ and $sid_m$, respectively. When Proxy$_2$ receives the $PXH_A(sid'_k, sid'_m)$, it can identify $H(R_k \oplus R_m)$ value, and deduce that $sid'_k$ and $sid'_m$ correspond to $sid_k$ and $sid_m$ (or vice versa).

To prevent Proxy$_2$ from exploiting the $R$ values, Proxy$_1$ and the aggregator need to modify the $PXH$ values, just like the $sid$ values, but without affecting the comparison result. They achieve this goal by first XORing the same shared secret, $R_s$, with the pairwise-XOR output, and then hash it:

$$\begin{aligned} P_1 \quad &: \quad PXH'_{P_1}(sid'_i, sid'_j) = H((X_i \oplus X_j) \oplus R_s) \\ A \quad &: \quad PXH'_A(sid'_i, sid'_j) = H((R_i \oplus R_j) \oplus R_s) \end{aligned}$$

By modifying both the $sid$ and $PXH$ values, Proxy$_1$ and the aggregator can prevent Proxy$_2$ from deanonymizing honest client strings by exploiting the comparison result.

To continue, the aggregator first groups the $sid$ values regarding their $ST$ values and compiles *comparison lists*. A comparison list consists of either a generic string type (e.g., websites), or a few different, analyst-specific string types (e.g., photo app tags and health app tags). This mixing of analyst-specific string types provides clients with additional privacy guarantees regarding their string types (see §8 for details).

Figure 8 shows how distinct strings in each comparison list are counted. The aggregator sends each

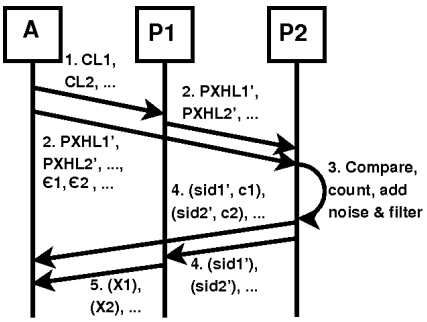


Figure 8: Counting and revealing distinct string values.

list to Proxy$_1$ (step 1):

$$A \rightarrow P_1 : CL_1, CL_2, ..., CL_n$$

Proxy$_1$ and the aggregator then compute $PXH'$ values for each possible $\langle sid'_i, sid'_j \rangle$ tuple in each comparison list, $CL_i$, and use the $\langle sid'_i, sid'_j \rangle$ tuple as the identifier for the $PXH'$ values. Let $PXHL'_{P_1,i}$ and $PXHL'_{A,i}$ represent the list of $PXH'_{P_1}$ and $PXH'_A$ results for $CL_i$. Proxy$_1$ then sends the $PXHL'_{P_1}$ to Proxy$_2$. Similarly, the aggregator sends the $PXHL'_A$ and $\epsilon$ values to Proxy$_2$ (step 2):

$$\begin{aligned} P_1 \rightarrow P_2 &: PXHL'_{P_1,1}, ..., PXHL'_{P_1,n} \\ A \rightarrow P_2 &: PXHL'_{A,1}, ..., PXHL'_{A,n}, \epsilon_1, ..., \epsilon_n \end{aligned}$$

Proxy$_2$ then determines the equivalence of the encrypted strings for each tuple by comparing $PXH'_{P_1}$ and $PXH'_A$ values, and creates *equivalence lists* consisting of $sid'$ values of equivalent strings. For example, if strings with $sid'_i$ and $sid'_j$ are equivalent, $sid'_i$ and $sid'_j$ are put in the same list.

From each equivalence list $EL_i$, Proxy$_2$ randomly selects a $sid'_i$ value as a *representative string* and records it with the count of equivalent strings. Proxy$_2$ then adds differentially-private (DP) noise to each count using the $\epsilon$ parameter received with the $PXHL'_A$, and discards the representative $sid'$ values whose noisy counts are below the discovery threshold (step 3).

Let $c_i$ be the noisy count of the representative string $sid'_i$ from equivalence list $EL_i$. Proxy$_2$ sends the representative $sid'$ values and their noisy counts to the aggregator whereas it sends only the representative $sid'$ values to Proxy$_1$ (step 4):

$$\begin{aligned} P_2 \rightarrow A &: \{\langle sid'_i, c_i \rangle, \langle sid'_j, c_j \rangle, \langle sid'_n, c_n \rangle\} \\ P_2 \rightarrow P_1 &: \{sid'_i, sid'_j, ..., sid'_n\} \end{aligned}$$

Proxy$_1$ sends the corresponding split messages of the encrypted strings (i.e., the $X$ values) to the aggregator (step 5):

$$P_1 \rightarrow A : \{sid'_i, X_i, sid'_j, X_j, ..., sid'_n, X_n\}$$

The aggregator joins the matching split messages (i.e., $X_i$ and $R_i$) and obtains the strings. In the future, it can ask clients not to send already discovered strings (§5.1).

## 5.4 Mirror Operation and Oblivious Noise

So far, we have described mirror operation 1 in Figure 5, in which Proxy$_2$ performs the comparison task for a client's strings. Recall that for other clients, Proxy$_1$ may perform the comparison task while Proxy$_2$ assuming the collecting role. Both proxies count, add noise and filter distinct strings independently. These strings are then sent to the aggregator.

It is possible that the aggregator receives a particular string from only one proxy. Imagine it receives a string with a noisy count from Proxy$_2$, but not from Proxy$_1$. If this single-noisy count is published, Proxy$_2$ can associate it with the string, subtract the noise it added, and get the string's noise-free count.

In this case, the aggregator may also add noise to the string's already noisy count and publish the double-noisy result. However, Proxy$_2$ may still be able to correlate some strings and their noisy counts: Assume a discovery threshold of 100. If no noise was added, each discovered string would have a total count of (at least) 200, because each proxy operates independently. A count below 200 (or 200 $\pm$ noise) would indicate that the threshold was crossed at only one proxy. Proxy$_2$ could use the order of the noisy counts it reported, associate them with the strings (or eliminate most possible cases), and obtain the noise-free count by removing its noise.

For these reasons, the aggregator only publishes a string value if it receives the string from *both* proxies, and publishes the sum of both noisy counts. This double-noisy count prevents each proxy from obtaining the noise-free count of any string: even if a proxy somehow removes its own noise, the count will still contain the other proxy's noise. As a result, all components will be oblivious to the final noise added to a string's count.

### 5.5 Other Issues & Discussion

**Preventing Traffic Analysis.** To prevent traffic analysis, proxies relay split messages after randomly ordering and delaying them. Each client sends a fixed number of strings for each string type (e.g., 25). If a client has more strings, it randomly selects enough strings to send. If a client does not have enough strings, it generates random strings as filler strings, and modifies the actual string type to indicate this modification (e.g., "websites_FS"). The aggregator filters these messages during synchronization with $\text{Proxy}_1$. Each string is a fixed size. If the string is not long enough, the client pads the string deterministically (e.g., with SHA-1 of string) before splitting.

**Generic vs. Analyst-specific.** For analyst-specific string types, the client prepends the string type to the string in its database. Therefore, even if the actual string value is the same for two analyst-specific string types, they will not be equivalent when compared, for instance, when the aggregator mixes multiple string types in a comparison list. An analyst-specific discovery procedure enables each analyst to obtain more specific information about its clients. A generic discovery procedure may produce strings that might go undiscovered within a specific client population. In a generic discovery, each client sends its strings once for many analysts rather than once per analyst-specific discovery, decreasing the number of strings the system handles. Additionally, each client's privacy loss would be reduced, because the number of discovery procedures each client would participate in would be reduced. We envision that both discovery types will be useful for analysts.

## 6 Detecting Duplicates

In our system, malicious clients can try to skew the counts of a string by sending the same string multiple times. This section describes how our system can detect such clients before distinct strings are counted (§5.3). For consistency, we again describe the protocol in mirror operation 1.

Figure 5 shows a rectangular shape around the components involved in this step. This step requires synchronization between the two mirror operations (i.e., the relaying $\text{Proxy}_2$ from mirror operation 1 and the relaying $\text{Proxy}_1$ from mirror operation 2). The high-level idea is to run the same blind comparison protocol described earlier, but this time only among all the strings received from the same client: Any equivalent strings found will be duplicates, because they come from the same client. As a result, the malicious client will be detected without revealing honest clients' strings.

Recall that the client uses $\text{Proxy}_2$ as a relay for sending $X$ values to $\text{Proxy}_1$. In this role, $\text{Proxy}_2$ attaches a *pseudo IP address* ($pIP$) for each client IP and forwards the $X$ values to the (collecting) $\text{Proxy}_1$ (Figure 7). Our duplicate detection protocol leverages the $pIP$ values assigned by the (relaying) $\text{Proxy}_2$ and works in two stages.

**Stage 1:** The (relaying) $\text{Proxy}_2$ in mirror operation 1 and the (relaying) $\text{Proxy}_1$ in mirror operation 2 exchange the real client IP addresses each has. If each client followed the protocol and used the same collecting components in the current discovery epoch, the intersection of both lists will be empty. If not, the clients with IP addresses appearing in both lists might have sent the same string using different collecting components. Strings that these clients sent are invalidated by sending the corresponding $pIP$ value to the collecting component. In mirror operation 1, the (relaying) $\text{Proxy}_2$ would send the $pIP$ values to the (collecting) $\text{Proxy}_1$, who would then discard the associated $X$ values. Alternatively, one proxy can be randomly selected to invalidate the strings, while the other does not.

**Stage 2:** After the strings coming from the clients detected in Stage 1 are invalidated, the collecting components ($\text{Proxy}_1$ and the aggregator) share a ran-

dom secret that is valid for one discovery epoch, $R_{sdd}$. This secret is different from $R_s$ used in the counting phase (§5.3), but has a similar purpose: to prevent Proxy$_2$ from correlating the strings it relayed to Proxy$_1$ during the collection of encrypted strings (§5.2) with the strings it checks for duplicates, and also from linking strings checked during the duplicate detection to the strings compared in the counting phase (§5.3). Proxy$_1$ and the aggregator then modify the $sid$ values, such that $sid'_i = H(sid_i||R_{sdd})$.

Additionally, Proxy$_1$ independently modifies the $pIP$ values it received from Proxy$_2$ and gets a $pIP \leftrightarrow pIP'$ mapping. For each $pIP'$, it sends the list of $sid'$ values, $sidL'$, to Proxy$_2$:

$$P_1 \rightarrow P_2 : pIP'_1, sidL'_1, ..., pIP_p, sidL'_p$$

The aggregator also independently modifies the $ST$ values to obtain the $ST \leftrightarrow ST'$ mapping. For analyst-specific string types, multiple $ST$ values can correspond to the same $ST'$ value. In other words, the aggregator mixes multiple analyst-specific string types into one list. The comparison between strings with different analyst-specific $ST$ values will not cause any problems, because the actual strings are prepended with the $ST$ value (§5.5). For each $ST'$, the aggregator sends the list of $sid'$ values to Proxy$_2$:

$$A \rightarrow P_2 : ST'_1, sidL'_1, ..., ST'_t, sidL'_t$$

Using both $pIP' \rightarrow sidL'$ and $ST' \rightarrow sidL'$ mappings, Proxy$_2$ divides the $sid'$ values into groups, each of which correspond to a unique $\langle pIP', ST' \rangle$ pair. The created groups are then sent to Proxy$_1$ and the aggregator, such that they can compute the $PXH'$ values within each group:

$$\begin{aligned} P_2 \rightarrow P_1 &: G_1, G_2, ..., G_n \\ P_2 \rightarrow A &: G_1, G_2, ..., G_n \end{aligned}$$

Note that Proxy$_1$ does not learn the $ST'$ values of the strings in each group; only that the strings might have different $ST'$ values. Similarly, the aggregator does not learn the $pIP'$ values of the strings in each group; only that the strings might have different $pIP'$ values (§8.5 & §8.6).

Proxy$_1$ and the aggregator compute the $PXH'$ values using $R_{sdd}$ for each possible $\langle sid'_i, sid'_j \rangle$ tuple in each group:

$$\begin{aligned} P_1 &: PXH'_{P1}(sid'_i, sid'_j) = H((X_i \oplus X_j) \oplus R_{sdd}) \\ A &: PXH'_A(sid'_i, sid'_j) = H((R_i \oplus R_j) \oplus R_{sdd}) \end{aligned}$$

They send the resulting $PXHL'_{P_1}$ and $PXHL'_A$ to Proxy$_2$:

$$\begin{aligned} P_1 \rightarrow P_2 &: PXHL'_{P_1,1}, PXHL'_{P_1,2}, ..., PXHL'_{P_1,n} \\ A \rightarrow P_2 &: PXHL'_{A,1}, PXHL'_{A,2}, ..., PXHL'_{A,n} \end{aligned}$$

For each $PXHL'$, Proxy$_2$ checks for duplicates (i.e., equivalent strings). If there are any duplicates, their $sid'$ values are reported to the aggregator.

Afterwards, Proxy$_1$ and the aggregator proceed to count the distinct strings (§5.3). During the counting phase, the aggregator modifies the $PXH'_A$ results involving the duplicates independent of Proxy$_1$, for instance, by XORing the pairwise-XOR with a random nonce. As a result, the comparison with other strings will not yield an 'equivalent' result, and the duplicates will not affect the counts of their respective strings.

### 6.1 Considering NATs

During this step, the IP address is used to identify a client. This assumption creates a bias in the counts, when many clients use the same IP address (e.g., home gateway, business firewall), and some duplicates may be legitimate. To decrease this bias, some duplicate strings may be randomly selected to be included in the counting phase depending on the aggregator's policy. Distinguishing such clients is outside the scope of this paper and left for future work.

## 7 Optimizations

Our blind comparison method is effective to learn if two encrypted strings are equivalent, but it requires $\frac{N \times (N-1)}{2}$ pairwise comparisons (i.e., $PXH$ operations) to count the distinct strings in a comparison list, where $N$ is the number of reported strings

in the list. Although $PXH$ utilizes low-cost operations (i.e., XOR and hash), the total cost can still be prohibitive. Here, we outline some optimizations to substantially lower this cost in practice without violating our privacy goals. In Section 9, we evaluate the effectiveness of these optimizations.

## 7.1 Short Hashes

One simple heuristic to reduce the number of $PXH$ operations is to distinguish different strings before they are even collected. The high-level idea is that the strings deemed different before the collection will not need to be pairwise compared. One way to achieve this separation is that the clients map their strings into a *bucket* ($B$) using a hash function with a *small* number of buckets (e.g., last byte of SHA-1). The clients send each string's $B$ value along its $ST$ value to the aggregator, who then compiles the comparison lists using the distinct $\langle ST, B \rangle$ tuples instead of just $ST$. As a result, fewer strings will be pairwise compared in each comparison list.

To determine the number of hash buckets, the aggregator starts a discovery procedure with one bucket and samples the collected strings. The strings in the sample are compared with each other. The number of distinct strings in the sample will give the aggregator an idea of how many distinct strings to expect, and determine how many buckets to use without creating a privacy issue. To build confidence, the aggregator can take more samples or increase the sample size. After selecting the number of buckets (e.g., 256), the aggregator starts a new discovery procedure and request clients to send their strings with the appropriate $B$ value.

We think this heuristic offers a reasonable trade-off between privacy and computational cost: The privacy loss is small, because with a small number of bucket values many distinct string values will map to the same bucket. In other words, there will be many hash collisions, and thus, the information that can be gained about the actual string by knowing its bucket will be small. On the other hand, as we show in Section 9, the number of $PXH$ operations will be significantly reduced, because strings with different bucket values will not need to be compared.

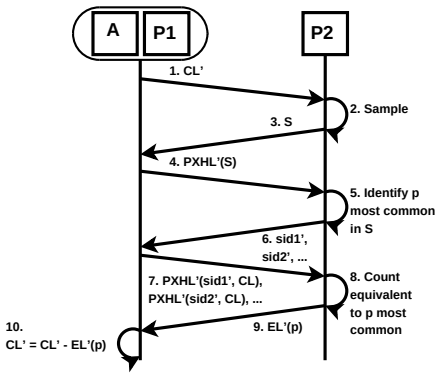


Figure 9: The operation of our sample-identify-count-filter optimization.

## 7.2 Sample-Identify-Count-Filter

Another heuristic is to use random samples and iteratively shorten the comparison lists. The high-level intuition is that, like many natural phenomena, the string distributions will show power law characteristics, and thus, a few strings will dominate in the comparison list. These few, very common strings can be identified using a small random sample. Afterwards, the strings that are equivalent to these identified strings can be filtered from the comparison list, significantly shortening it.

This process can continue iteratively until 1) enough strings are discovered (i.e., similar to top-k most common strings), 2) the counts of most recently discovered strings fall below a threshold (i.e., $k$ is a function of the string count in the top-k algorithm), 3) the number of remaining strings in $CL'$ falls below a threshold, or 4) until $\text{Proxy}_2$ does not discover any more new strings during step 8. When any of these conditions is satisfied, the counting protocol can stop: most common strings will have already been discovered, and the remaining undiscovered strings will probably have small counts. Stopping the counting protocol may cause false negatives, so that some strings above the threshold may go undiscovered; however, with increasing sample size, the probability of this event should decrease. Alternatively, the counting protocol can resume pairwise

comparing the remaining strings: at this point, pairwise comparison should be more affordable, because the comparison list will be much smaller.

Figure 9 shows one iteration in mirror operation 1: The collecting components ($\text{Proxy}_1$ and the aggregator) first send the comparison list with modified *sid* values ($CL'$) to the comparing component ($\text{Proxy}_2$) (step 1). $\text{Proxy}_2$ selects a random sample ($S$) (step 2) and sends it to $\text{Proxy}_1$ and the aggregator (step 3). $\text{Proxy}_1$ and the aggregator compute and send back $PXH'$ values for the strings in $S$ (step 4). $\text{Proxy}_2$ then identifies the distinct (encrypted) strings in $S$, and selects one representative $sid'$ value from each of the longest $p$ equivalence lists in the sample (i.e., most common $p$ distinct strings) (step 5). These $sid'$ values are sent to $\text{Proxy}_1$ and the aggregator (step 6), who compute $PXH'$ values for these $p$ strings with all other strings in the $CL'$ and send them to $\text{Proxy}_2$ (step 7). $\text{Proxy}_2$ counts all equivalent strings to each of these $p$ strings (step 8), stores them in equivalence lists, and sends them to $\text{Proxy}_1$ and the aggregator (step 9). Afterwards, they filter these $sid'$ values from the $CL'$ (step 10).

Our intuition about the privacy of this heuristic is the following: To filter equivalent strings and shorten the comparison list, $\text{Proxy}_1$ and the aggregator learn the comparison results between some strings. If one of these string values is known by them (e.g., because the string was sent by one of their fake clients), the comparison result may enable them to deanonymize honest clients' strings. However, $\text{Proxy}_1$ and the aggregator learn *many* $sid'$ values that are equivalent to *any* one of the $p$ common strings, and thus, cannot be certain which strings are actually equivalent. Furthermore, these comparison results are only for the $p$ most common strings in the random sample, which essentially reflects the string counts in the original $CL'$: to expose a string with a few clients, an adversary would need to send the string a huge number of times, such that it would become one of the $p$ most common strings in the random sample! We think that this heuristic is reasonable in terms of the reduced computational cost, and the difficulty and practicality of this potential attack.

# 8 Analysis

In this section, we present an analysis of our system. First, we informally show how our noise-before-threshold mechanism deals with the $k-1$ sybils attack, where $k$ represents the discovery threshold. We then place the adversary at each component, and describe how our system raises the bar for the adversary. We again assume mirror operation 1 in Figure 5.

## 8.1 Noise-before-Threshold

The discovery threshold $k$ is a known parameter. To learn the existence of a client with a rare string, an adversary can run $k-1$ fake clients and send the rare string in a discovery procedure. After counting and the addition of noise, if the noisy count passes the threshold, the string is discovered. According to differential privacy, the probability of producing a given output (e.g., the noisy count being $k$ or $k-1$, and thus, the string being discovered or not) is almost independent of the existence of any individual record in the data set (§4.3). In other words, whether the real client exists does not affect whether the string with $k-1$ sybils is discovered. For example, the client may exist, in which case the noise-free count would be $k$. However, the noise may be negative, making the noisy count go below the threshold, and thus, the string may not be discovered. On the other hand, the client may not exist, in which case the noise-free count would be $k-1$. However, the noise may be positive, making the noisy count pass the threshold, and thus, the string may be discovered.

For a pattern to emerge, the discovery procedure would need to be repeated. The number of repetitions depends on the $\epsilon$ value, with lower $\epsilon$ values requiring more repetitions. Akkus et al. showed via simulations that many repetitions (e.g., 100s) are needed to cancel out the noise [16]. Additionally, the aggregator may request the clients not send already discovered strings (§5.1), increasing the number of discovery procedures required for this attack to succeed, because the rare string will not be sent again for a while after the first discovery.

## 8.2 Client

A malicious client can lie in the strings it sends; however, each string count will change only by one. A client can send the same string twice to different collecting components, but the relaying proxies will determine the common IP addresses in the first stage of the duplicate detection and conservatively discard this client's strings. A client can send the same string multiple times with the same collecting components, but will be detected in the second stage of the duplicate detection (§6). False $ST$ or $B$ values for the heuristic in §7.1 will not affect any counts, because the strings will not be equivalent to others.

## 8.3 Relaying Proxies (Proxy$_1$ and Proxy$_2$)

**Discovery Initialization.** The client periodically polls the aggregator to participate in string discovery procedures. The polling requests are split and relayed over both proxies, preventing them from learning the string types of a client (§5.1).

**Collection.** Both proxies relay the $sid$, $R$ and $ST$ values to the aggregator; however, these values are split, and thus, illegible.

## 8.4 Relaying Proxy & Comparing Proxy (Proxy$_2$)

**Collection.** When Proxy$_2$ is used for the comparison task, it also relays the $sid$ and $X$ values to Proxy$_1$ after attaching a pseudo IP ($pIP$) to each client (Eqn 2 in §5.2). Proxy$_2$ also sees $\underline{sid||R||ST}$ values from the same client while relaying them to the aggregator (Eqn 3 in §5.2). It can determine the part of the random string used to split $sid$ by brute-forcing all $sid$ values of the same client; however, this part does not indicate any information about the parts used to split $R$ and $ST$, preventing Proxy$_2$ from learning them (§5.2).

**Comparison & Counting.** Proxy$_2$ is in a position to expose a client's string using the comparison result with a known string. Proxy$_2$ cannot correlate the strings it compares to the $pIP$ values it assigns to each client while relaying the $X$ values to Proxy$_1$, because Proxy$_1$ and the aggregator modify the $sid$ and the $PXH$ values. This modification also prevents Proxy$_2$ from identifying the strings its clients send, and thus, it cannot exploit the comparison result to deanonymize client strings (§5.3). Proxy$_2$ cannot create a known string and compare it with a client's string, because it does not receive individual split messages. Proxy$_2$ cannot obtain the exact count of a string, because the aggregator only publishes a string if it receives the same string value from both proxies, and thus, the final count will contain the other proxy's noise (§5.4).

**Duplicate detection.** Proxy$_1$ modifies the $pIP$ values to $pIP'$ values, preventing Proxy$_2$ from correlating the client IP addresses to the $pIP'$ values it receives for duplicate detection. Proxy$_2$ also cannot associate the number of strings received from each client to $pIP'$ values, because the clients send a fixed number of strings. Proxy$_2$ also receives $ST' \rightarrow sidL'$ mapping from the aggregator. Proxy$_2$ can run clients to send a specific number of duplicate strings for certain string types, such that these duplicates will signal Proxy$_2$ that a $sidL'$ belongs to a specific string type. However, these $sidL'$ lists have strings either for generic string types, which are present at every client, or for multiple analyst-specific string types, which creates uncertainty in Proxy$_2$'s guess for the string type of a $sidL'$ (§6). Thus, Proxy$_2$ cannot learn much about the (anonymous) clients. Proxy$_2$ also cannot exploit the comparison result to deanonymize a client's string, because strings from the same client are only compared with each other.

A *dishonest* Proxy$_2$ can relay fake duplicate strings from its own clients as if they come from an honest client, use the number of duplicates as a signature to correlate the $pIP'$ value to the client, and deduce the client's string if there are more duplicates than the signature. Because the client sends a fixed number of strings, Proxy$_1$ will notice the added fake strings. Proxy$_2$ can drop real client strings, but its possibilities to form a signature from the duplicate count will be reduced. To detect such a dishonest proxy, the aggregator can employ an auditing mechanism, similar to Akkus et al.'s proposal [16]: Clients would probabilistically send random nonces to Proxy$_1$ over

Proxy$_2$ and nonce reports to the aggregator over the proxies as if real strings. Proxy$_1$ and the aggregator would cooperate and expect to receive the nonces; if not, they suspect Proxy$_2$.

## 8.5 Collecting Proxy (Proxy$_1$)

**Collection.** Proxy$_1$ receives *sid* values after Proxy$_2$ relays them (Eqn 2 in §5.2). Proxy$_1$ also sees $\underline{sid||R||ST}$ values when it relays the split $R$ and $ST$ to the aggregator (Eqn 3 in §5.2). Proxy$_1$ can try to correlate the $pIP$ values to the clients by trying to determine the random string used to split *sid*, and correlating the split messages to the client. However, unlike the relaying Proxy$_2$, it sees the client directly only when receiving $\underline{sid||R||ST}$, and not both *sid* and $\underline{sid||R||ST}$. Thus, it cannot brute-force all *sid* values from the same client to determine the random string, and cannot correlate the $\underline{sid||R||ST}$ values to the $pIP$ values (§5.2).

**Comparison & Counting.** Proxy$_1$ receives the comparison lists ($CL$s) from the aggregator, but not their $ST$ values. Proxy$_1$ can run fake clients with certain string types and send strings, such that these strings would signal the string type of a $CL$. Proxy$_1$ then can correlate clients whose strings are present in that $CL$. However, a $CL$ can be for a generic string type, which is present at every client, rendering the knowledge of the $ST$ value less useful. On the other hand, the aggregator also mixes multiple analyst-specific string types into one $CL$, effectively creating uncertainty in Proxy$_1$'s guess for the string type of a $CL$ (§5.3). Furthermore, any uncertain information Proxy$_1$ obtains will be anonymous and short-lived, because Proxy$_2$ acts as an anonymizing proxy for the clients while relaying the $X$ values, and the $pIP$ values it assigns are only valid for one discovery epoch (§6).

After strings are discovered, Proxy$_1$ receives the $sid'$ values of the representative strings and sends the corresponding $X$ values to the aggregator. Proxy$_1$ knows if any two representative $sid'$ values are from the same client (i.e., they have the same $pIP$). However, it cannot correlate the $X$ values to the strings published by the aggregator (unless there are only a few strings, in which case the aggregator may not publish them). Additionally, strings for analyst-specific string types will only be available to specific analysts. Finally, if there are many discovered strings and many clients, the probability of two representative $sid'$ values being from the same client is low.

Proxy$_1$ cannot expose client strings via comparison results, because it does not learn them from Proxy$_2$, except for the optimization (§5.3).

**Sample-Identify-Count-Filter.** With this optimization, Proxy$_1$ learns some comparison results to shorten the comparison lists. These results, however, are for the most common $p$ strings. To expose a rare string, Proxy$_1$ would need to send it via fake clients and make its count in the sample surpass other common strings. Our duplicate detection raises the bar significantly and forces Proxy$_1$ to use more clients.

The strings in the sample are selected by Proxy$_2$. A *dishonest* Proxy$_1$ may replace the original split messages in the sample with messages of known strings; however, it cannot manipulate the matching split messages held by the aggregator, and thus, the comparison results will not be meaningful.

**Duplicate detection.** For each $pIP'$, Proxy$_1$ receives groups of $sid'$ values from Proxy$_2$ that differentiate between different string types a client has (§6). Proxy$_1$ already knows how many different string types an *anonymous* client has, because it knows the number of messages received from each client and the maximum number of messages a client sends. However, it does not know which string types a client has.

A *dishonest* Proxy$_1$ can include a known string's $sid'$ in the list of a $pIP'$ value. If Proxy$_1$ receives this $sid'$ in the same group as the client's strings, it can deduce the string type of this anonymous client. Generic string types will not leak any information, because every client has them. For analyst-specific string types, Proxy$_1$ has to correctly guess the client's string type; otherwise, the fake string will be in a different group by itself. A *dishonest* Proxy$_1$ can also include known strings for generic string types as if they are coming from an honest client. However, Proxy$_1$ never receives the duplicate detection result (§6). The aggregator can detect such a dishonest proxy by mapping the duplicate $sid'$ values to the original *sid* values and by cooperating with Proxy$_2$: Proxy$_2$ sees the original *sid* values while re-

laying them to $\text{Proxy}_1$, and can detect whether the original *sid* values were attached the same $pIP$ values. If not, $\text{Proxy}_1$ must have added them.

### 8.6 Aggregator

**Collection.** An aggregator using large values of $\epsilon$ for low noise can be detected by clients, privacy watchdogs and proxies who add noise using those values. All interactions between the clients and the aggregator are mediated by the proxies, providing clients with network anonymity (§5.1 & 5.2).

**Comparison & Counting.** When strings are discovered, the aggregator receives the corresponding $X$ values for the representative strings from $\text{Proxy}_1$; however, it does not know whether these $X$ values, (and thus, the strings) belong to one client. The aggregator cannot expose client strings via comparison results, because it does not learn them from $\text{Proxy}_2$, except for the optimization (§5.3).

**Sample-Identify-Count-Filter.** Like $\text{Proxy}_1$ (§8.5), the aggregator would need many clients to exploit this optimization.

**Duplicate detection.** The aggregator learns $sid'$ groups belonging to unique $\langle pIP', ST' \rangle$ tuples from $\text{Proxy}_2$, but does not learn whether any two groups belong to the same $pIP'$; thus, it cannot deduce a given client's string types. Running clients to send known strings is also not useful: Each client has a separate $pIP'$. Thus, honest clients' strings will be in different groups and will not be compared with the aggregator's fake strings, preventing it from exposing client strings.

## 9 Evaluation

In this section, we show the benefits of our optimizations through simulations, report on microbenchmark results, and evaluate our system's feasibility via example scenarios. To establish a point of reference, we use Applebaum et al. [17] rather than [20] or [25], because it is the most similar system to ours: it has centralized components, has some protection against malicious clients, assumes components can run clients, and is designed for large-scale environments (e.g., the web).

In Applebaum et al.'s system, there are three components: client, proxy and database. The client runs an encrypted oblivious transfer protocol with the proxy to obtain obliviously-blinded version of its keys (i.e., $F_s(k)$). It encrypts them with the database's public key (i.e., $E_{DB}(F_s k)$) and sends them to the database over the proxy. The client also sends its key in a double-encrypted form (i.e., $E_{DB}(E_{PRX}(k))$). The database decrypts the blinded keys collected from all clients and records each key's count. If a key's count is above a threshold, the double-encrypted key is decrypted first by the database, and then the proxy. Note that we leave out some operations (i.e., batched oblivious transfer [34]) to simplify the comparison.

### 9.1 Data Sets

In our simulations, we use data generated according to the distributions of two real-world data sets. In the first data set, the strings are website names from a snapshot of Quantcast's top 1M sites in April 2013 [12]. Excluding hidden website names, there are 996,934 websites ranked by their visitor counts. We label simulation data generated with this data set "quantcast".

In the second data set, the strings are anonymized search phrases obtained from a large search engine, whose name we cannot disclose for confidentiality. The data set covers 90 consecutive days within the last two years, and contains about 13 million unique search phrases. A user may have issued the same phrase more than once, but we conservatively assume that each occurrence of a phrase is from a unique client. We label simulation data generated with this data set "search_engine".

### 9.2 Benefits Of The Optimizations

Before we compare our system with Applebaum et al.'s system, we show the benefits of the optimizations in isolation.

**Hash buckets:** We plot the speedup as a function of the number of hash buckets used. The speedup is the

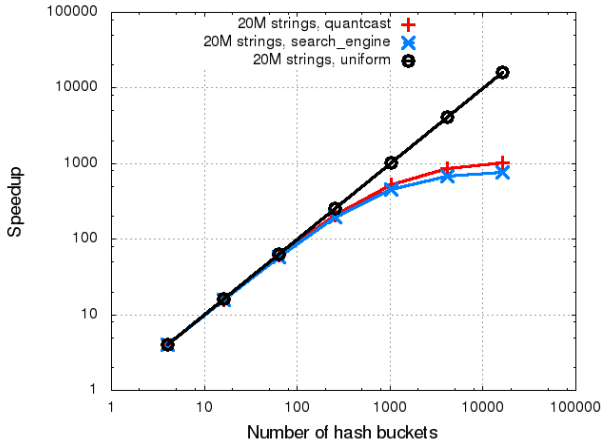


Figure 10: Speedup as a function of the number of hash buckets used.

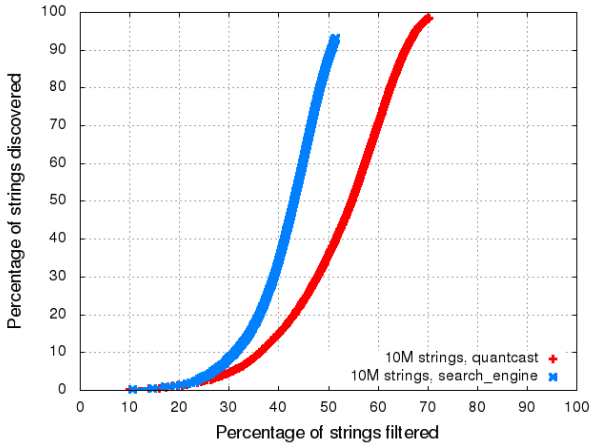


Figure 11: Percentage of discovered strings versus the percentage of strings that are filtered with our sample-identify-count-filter optimization with 99% confidence level, 3% margin of error, and $p = 20$. The discovery threshold is 100. The counting process stops after 10 rounds of no new discovered strings.

ratio of the time required to count the distinct strings without hash buckets to the time required with hash buckets distinguishing the strings. Essentially, it is the ratio between the number of $PXH$ operations the collecting components have to perform, without and with hash buckets. Let $N$ be the total number of strings, $n_i$ be the number of strings in hash bucket $i$, and $H$ be the number of hash buckets used. The speedup is given as:

$$S = \frac{\frac{N \times (N-1)}{2}}{\sum_{i=1}^{H} \frac{n_i \times (n_i - 1)}{2}}$$

Figure 10 shows that the speedup increases with the number of hash buckets as expected: Strings in different buckets need not be pairwise compared, because they are already deemed different; thus, the number of $PXH$ operations is reduced. After 256 buckets, however, the speedup starts to decline for the following reason: The power law characteristics of our data produce large counts for some strings, increasing their respective buckets' string counts. As a result, buckets with large counts start to dominate in the sum of $PXH$ operations, and thus, lower the speedup. By contrast, with a uniform distribution, the speedup does not decline after 256 buckets. Different total number of strings did not change the speedup values much, and thus, are not shown not to clutter the plot.

Using hash buckets results in some privacy loss, but it is quite small. For instance, with 256 buckets, the average number of distinct strings per bucket is about 3.9K for websites, and about 50.5K for search phrases. Thus, knowing a string's bucket value does not provide much information about it.

**Sample-Identify-Count-Filter:** We tested our optimization with 10M strings distributed according to our real-world data. The discovery threshold was set to 100. The sample size was computed according to 99% confidence level and 3% margin of error parameters, and the number of most common strings identified ($p$) was set to 20. We stopped counting after 10 consecutive rounds no new discovered strings.

Figure 11 shows the percentage of discovered strings versus the percentage of strings filtered with our optimization (i.e., common strings are identified and then filtered from the comparison list), validates our assumptions that the string distributions follow

a power law. For example, 10% of the strings above the threshold correspond to about 36% and 31% of all the strings in the Quantcast and search data sets, respectively. Our system discovered 97.5% and 93.3% of the strings that were above the threshold for these data sets, and the effective speedups are 335.5 and 219.6, respectively.

## 9.3 Microbenchmarks

We implemented our split/join and $PXH$ operations as well as Applebaum et al.'s operations, except for the batched oblivious transfer. All tests were run on a PC with an Intel I3 3.1 GHz CPU and 8GB RAM running Linux 3.2.42 kernel, and on a smartphone with a 1GHz CPU running Android 2.3.5.

We assume a string length of 100, including the padding used for short strings. The average and maximum length of the websites in Quantcast data is 17 and 63, respectively. A study of approximately 40 million queries found that around 97% of English search phrases contain up to 7 terms with an average of 3.1 terms [52]. Previous studies reported similar values [36, 48]. Assuming an average of 5.1 characters for an English word [2], a search phrase would be about 36 bytes. Note that previous approaches [17, 20] were assuming much shorter strings (e.g., 32-bits).

Applebaum et al. require the generation and verification of zero-knowledge proofs (ZKP) to ensure that a client supplies a 0 or 1 for a key, so that the total count is not distorted. However, it is not clear how such ZKPs would prevent malicious clients from sending the *same key multiple times*. Even if this attack could be prevented with ZKPs, the client would need to generate the proof, and the database would need to verify it, causing a high computational overhead for both.

The client and server microbenchmarks are written in Javascript and Java, respectively. Table 1 shows that our system's operations are quite fast, and the client overhead is several orders of magnitude less than Applebaum et al.'s system. Our $PXH$ operation can be performed about 1.1M times per second (i.e., XOR and SHA-1).

## 9.4 Example Scenarios

In this section, we analyze the memory, computational and bandwidth overhead of our system's components. The client bandwidth overhead is important because clients may possess limited resources. By contrast, the bandwidth overhead at the servers is not a big concern because bandwidth is generally cheap. For example, data outgoing from Amazon S3 is about $0.09 per GB up to 40TB, and even free when incoming [1].

For our simulations, we assume that each client sends 25 (real and filler) strings of length 100 for a discovery procedure. The split identifier and seed are 16 bytes each, and the string type, epoch end time and $\epsilon$ are 8 bytes each. Applebaum et al.'s system uses El Gamal encryption with a 1024-bit key [17].

**Memory Overhead.** Applebaum et al.'s batched oblivious transfer (BOT) achieves the highest throughput when the client uses 5K keys in a batch [17]. With 32-bit keys, the memory overhead is about 39MB at the client. To handle 100 clients, the proxy would require 3.8GB memory. Note that if the number of keys in a batch is decreased, the throughput of the BOT is significantly reduced (i.e., $< 1$ key per sec. [17]).

In our system, to send a total of 5K 100-byte strings, the client's memory overhead would be about 0.97MB. To handle 100 clients, the proxy/aggregator would require roughly 97MB.

**Computational Overhead.** To show the computational overhead of our system, we vary the number of strings in our simulation data from 10M to 100M, and plot the CPU time for discovering strings. In our system, $PXH$ operations consume CPU time. In Applebaum et al.'s system, the decryption of the obliviously-transferred key by the database consumes CPU time. Note the time for BOT of clients' keys is not included.

We report the computational overhead at the aggregator, because it is the bottleneck: to compute the $PXH$ values, it synchronizes with both proxies in both mirror operations. Each proxy's overhead would be half of the aggregator. We use both our optimizations: 256 hash buckets and random samples (99% confidence level, 3% margin of error, $p$=20).

Table 1: Microbenchmarks. String size of 100. Applebaum et al. use El Gamal crypto with a 1024-bit key. Operations per second. Client (PC) is Chromium, and Client (Phone) is Webkit. (g) = generation, (v) = verification.

| | Component | Splitting/ Encryption | Join/ Decryption | SHA-1 | ZKP |
|---|---|---|---|---|---|
| This paper | Client (PC) | 361,627 | 1,181,512 | 118,959 | - |
| | Client (Phone) | 2,922 | 22,695 | 1,761 | - |
| | Aggregator | 1,819,459 | 8,300,335 | 1,273,204 | - |
| Applebaum et al. [17] | Client (PC) | 21.54 | - | - | 3.22 (G) |
| | Client (Phone) | 0.52 | - | - | 0.08 (G) |
| | Database | - | 270.20 | - | 19.23 (V) |

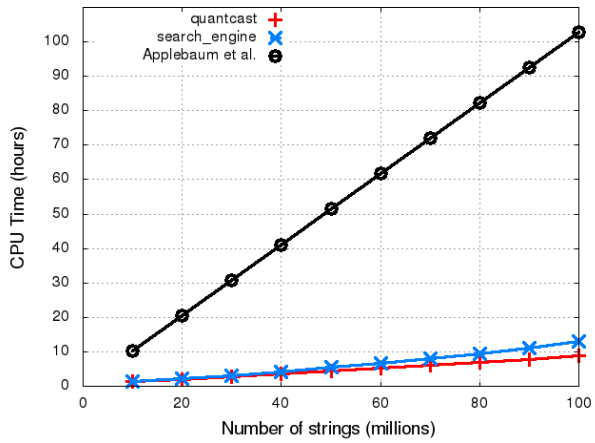


Figure 12: Total CPU time as a function of number of strings. Our system uses 256 hash buckets and the sample-identify-count-filter optimization with 99% confidence level, 3% margin of error, and $p = 20$. The discovery threshold is 100. For Applebaum et al., we assume that only the obliviously-transferred key is decrypted, and do not include the batched oblivious transfer time.

We also include the duplicate detection time.

Figure 12 shows that our system outperforms Applebaum et al.'s system by about one order of magnitude: For 100M strings, Applebaum et al. require about 103 CPU hours, whereas our system requires less than 9 and 13 CPU hours for the Quantcast and search data sets, respectively. Our system discovered almost all (99.995%) strings above the threshold.

**Bandwidth Overhead:** In Applebaum et al.'s system, the client sends the obliviously-blinded key encrypted with the database's public key (i.e., $E_{DB}(F_s(k))$) and the double-encrypted key (i.e., $E_{DB}(E_{PRX}(k))$) for each string. For 25 strings, the total bandwidth cost is about 9.38KB. The proxy receives client's encrypted keys and forwards them to the database. With 50M strings, the cost is about 35.8GB and 17.9GB for the proxy and database, respectively. Note that these numbers are optimistic, because they do not include the cost of the BOT and the ZKP.

In our system, the client sends 204 bytes for each string (i.e., a total of 4.98KB for 25 strings). The cost of the discovery initialization is negligible (i.e, 0.12KB per string type per poll). The total cost of a proxy (including the relaying and collecting roles in both mirror operations) for discovery initialization and collection of encrypted strings is about 12.54GB. The aggregator's total overhead is about 4.32GB. The duplicate detection costs about 54.04GB for each proxy and about 35.40GB for the aggregator for both

mirror operations in total. The biggest bandwidth cost is due to the $PXH$ operations. The cost for each proxy is about 1.35TB and 1.71TB for Quantcast and search data sets, respectively. The cost for the aggregator is about 0.90TB and 1.14TB for the corresponding data sets.

# 10 Related Work

To achieve anonymous communication, some systems use XOR [21, 49, 53] and matrix multiplication [35] as lightweight crypto operations. These systems do not focus on aggregating distributed user data with differential privacy.

Most database privacy research assumes a trusted database [27, 38, 51]. We refer the readers to a survey [31]. McSherry et al. [39] proposed discovering common payloads in network traces by iteratively incrementing the length of the queried string and choosing strings to increment regarding their differentially-private noisy counts. More recently, Chen et al. [22] proposed a method for publishing sequential data via variable-length n-grams while providing differential privacy. However, like the previous systems, these systems also assume a central database storing the data. In contrast, user data in our system resides on the user device in a distributed setting.

Some systems try to decrease the trust in the database by encrypting user data before it is stored, and enabling queries over the encrypted data [41, 46, 50]. However, these systems do not provide differential privacy for the results, and thus, allow a malicious analyst to obtain sensitive user data.

The trust in the storage entity can be also decreased by using more databases. Chow et al. [25] propose a two-entity model for privacy-preserving queries over distributed databases. One entity shares a secret with the databases to obfuscate results, and the other entity aggregates obfuscated data. However, if one database shares the secret with the aggregating entity (similar to the aggregator running fake clients), the privacy properties are lost. In contrast, our system's threat model allows the components to run their clients and prevents them from breaching privacy. Furthermore, Chow et al. assume that a database (a client in our system) supplies correct data, which may not be true in our analytics scenarios.

Secure Multiparty Computation (SMC) can also be used to aggregate private data from distributed databases. Burkhart et al. introduce Sepia [20], an SMC framework specialized for aggregating network events without a centralized entity. Input peers provide information to the system, and privacy peers run computations over them in a privacy-preserving way. Sepia's primitives are used to provide results for correlating network events and top-k queries [18, 19]. By using carefully optimized comparison operations, Sepia scales much better than other SMC frameworks; however, it is limited to small keys (32-bits) and a small number of participants (i.e., $< 100$), and thus, cannot be directly applied to analytics scenarios.

Rather than databases, user data can also reside on users' own devices. Anonygator [42] provides privacy properties while aggregating sensitive information; however, it assumes that the shared data is not going to leak privacy. P3 [40] is a privacy-preserving, distributed personalization system, but requires a method to determine which data is safe to supply for personalization. Both systems assume that the clients use an anonymizing network such as TOR [14]. In contrast, our system utilizes differential privacy for the discovered strings to protect privacy and uses centralized proxies for anonymity.

The common limitation of these systems is that they do not provide differential privacy (DP) for the participants. Although DP was originally designed for centralized databases [27], researchers have also attempted to provide DP in distributed settings. Some of these systems rely on complex cryptographic operations, putting too much computational overhead on the clients [29]. Other systems rely on distributed key distribution protocols, which reduce the computational overhead at the clients [44, 47], but suffer from churn in large-scale environments. To address the churn problem, Hardt et al. [33] propose that clients generate DP noise, such that available clients would compensate for the noise unavailable clients cannot add. The aggregation is performed by two honest-but-curious servers for scalability. All of

these systems [33, 44, 47] suffer from the attack where a malicious client skews the results arbitrarily. P4P [26] uses light-weight zero-knowledge proofs to prevent such clients; yet, the overheads are still too high to be practical.

To address both issues regarding scalability and malicious clients, recent distributed DP systems utilize pre-defined string values and use centralized entities. $\pi$Box [37] utilizes pre-defined counter names and leverages a trusted platform to restrict the interface how much and how often a malicious app instance (i.e., client) can update the counter value. The trusted platform adds DP noise to counter values before releasing them to developers (i.e., analysts). Other systems [16, 23, 24] provide DP in analytics scenarios and prevent tracking of users via centralized entities to distribute queries and add noise, either using a dedicated, honest-but-curious proxy [24], or existing entities such as the websites of publishers [16]. Compared to these systems using public-key cryptography, SplitX [23] scales much better by utilizing XOR as its crypto primitive, similar to our system. However, these systems [16, 23, 24] rely on the knowledge of pre-defined answer values to limit the effect of malicious clients as well as for adding DP noise blindly. Our system complements all above systems by discovering unknown strings to be used as counter names or answer values, while providing DP counts for those strings.

# 11 Conclusion

We presented a practical and privacy-preserving string discovery system that provides analysts with unknown strings and their noisy counts. To preserve the privacy of the clients, our system utilizes a discovery threshold as well as differential privacy. Our system utilizes XOR as its crypto primitive, reducing the client overheads significantly. To count client strings without revealing them, we use a novel blind comparison method that determines the equivalence of two XOR-encrypted client strings. While achieving these goals, our system also protects the accuracy of the string counts by detecting malicious clients submitting duplicate strings. Our evaluation shows that our system reduces the client computation overheads by several orders of magnitude compared to previous private aggregation systems. Simulations using real world data for website popularity and search phrases show that the computational overhead for server side computations is also reduced by about one order of magnitude.

# References


[1] Amazon S3 Pricing. http://aws.amazon.com/s3/pricing/. Aug 9, 2013.

[2] Average word length in English. http://www.wolframalpha.com/input/?i=average+word+length+in+english. Jul 8, 2013.

[3] BlueKai Consumers. http://bluekai.com/consumers_optout.php.

[4] BrightTag ONE-Click Privacy. http://www.brighttag.com/privacy/.

[5] ComScore: Mobile Will Force Desktop Into Its Twilight In 2014. http://www.businessinsider.com/mobile-will-eclipse-desktop-by-2014-2012-6. May 5, 2013.

[6] FTC Issues Final Commission Report on Protecting Consumer Privacy. http://www.ftc.gov/opa/2012/03/privacyframework.shtm.

[7] Internet Access Statistics. http://www.ons.gov.uk/ons/rel/rdit2/internet-access---households-and-individuals/2012-part-2/stb-ia-2012part2.html. May 5, 2013.

[8] Lawsuit accuses comScore of extensive privacy violations. http://www.computerworld.com/s/article/9219444/Lawsuit_accuses_comScore_of_extensive_privacy_violations.

[9] Privacy Lawsuit Targets Net Giants Over 'Zombie' Cookies. http:

//www.wired.com/threatlevel/2010/07/zombie-cookies-lawsuit.

[10] Quantcast Clearspring Flash Cookie Class Action Settlement. http://www.topclassactions.com/lawsuit-settlements/lawsuit-news/920.

[11] Quantcast Opt-Out. http://www.quantcast.com/how-we-do-it/consumer-choice/opt-out/.

[12] Quantcast Top Ranking International Websites. https://www.quantcast.com/top-sites.

[13] The Do Not Track Option: Giving Consumers a Choice. http://www.ftc.gov/opa/reporter/privacy/donottrack.shtml.

[14] Tor Project. https://www.torproject.org/.

[15] Web Tracking Protection. http://www.w3.org/Submission/web-tracking-protection/.

[16] I. E. Akkus, R. Chen, M. Hardt, P. Francis, and J. Gehrke. Non-tracking web analytics. In *ACM CCS*, 2012.

[17] B. Applebaum, H. Ringberg, M. J. Freedman, M. Caesar, and J. Rexford. Collaborative, Privacy-preserving Data Aggregation at Scale. In *PETS*, 2010.

[18] M. Burkhart and X. A. Dimitropoulos. Fast privacy-preserving top-k queries using secret sharing. In *ICCCN*, pages 1–7, 2010.

[19] M. Burkhart and X. A. Dimitropoulos. Privacy-preserving distributed network troubleshooting - bridging the gap between theory and practice. *ACM Trans. Inf. Syst. Secur.*, 14(4):31, 2011.

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

[21] D. Chaum. The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. *J. Cryptology*, 1(1):65–75, 1988.

[22] R. Chen, G. Ács, and C. Castelluccia. Differentially private sequential data publication via variable-length n-grams. In *ACM Conference on Computer and Communications Security*, pages 638–649, 2012.

[23] R. Chen, I. E. Akkus, and P. Francis. SplitX: High-Performance Private Analytics. In *SIGCOMM*, 2013.

[24] R. Chen, A. Reznichenko, P. Francis, and J. Gehrke. Towards Statistical Queries over Distributed Private User Data. In *NSDI*, 2012.

[25] S. S. M. Chow, J.-H. Lee, and L. Subramanian. Two-Party Computation Model for Privacy-Preserving Queries over Distributed Databases. In *NDSS*, 2009.

[26] Y. Duan, J. Canny, and J. Z. Zhan. P4P: Practical Large-Scale Privacy-Preserving Distributed Computation Robust against Malicious Users. In *USENIX Security Symposium*, pages 207–222, 2010.

[27] C. Dwork. Differential Privacy. In *ICALP*, 2006.

[28] C. Dwork. Differential Privacy: A Survey of Results. In *TAMC*, pages 1–19, 2008.

[29] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our Data, Ourselves: Privacy Via Distributed Noise Generation. In *EUROCRYPT*, pages 486–503, 2006.

[30] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating Noise to Sensitivity in Private Data Analysis. In *TCC*, pages 265–284, 2006.

[31] B. C. M. Fung, K. Wang, R. Chen, and P. S. Yu. Privacy-preserving data publishing: A survey of recent developments. *ACM Comput. Surv.*, 42(4), 2010.

[32] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on computing*, 18(1):186–208, 1989.

[33] M. Hardt and S. Nath. Privacy-aware personalization for mobile advertising. In *ACM CCS*, 2012.

[34] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *CRYPTO*, pages 145–161, 2003.

[35] S. Katti, J. Cohen, and D. Katabi. Information Slicing: Anonymity Using Unreliable Overlays. In *NSDI*, 2007.

[36] E. P. Lau and D. H.-L. Goh. In search of query patterns: A case study of a university OPAC. *Information Processing & Management*, 42(5):1316–1329, 2006.

[37] S. Lee, E. L. Wong, D. Goel, M. Dahlin, and V. Shmatikov. $\pi$box: a platform for privacy-preserving apps. In *NSDI*, 2013.

[38] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. l-Diversity: Privacy Beyond k-Anonymity. In *ICDE*, 2006.

[39] F. McSherry and R. Mahajan. Differentially-private network trace analysis. In *SIGCOMM*, pages 123–134, 2010.

[40] A. Nandi, A. Aghasaryan, and M. Bouzid. P3: A Privacy Preserving Personalization Middleware for Recommendation-based Services. In *HotPETS*, 2011.

[41] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

[42] K. P. N. Puttaswamy, R. Bhagwan, and V. N. Padmanabhan. Anonygator: Privacy and Integrity Preserving Data Aggregation. In *International Conference on Middleware*, 2010.

[43] M. O. Rabin. How to exchange secrets by oblivious transfer. Technical report, Technical Report TR-81, Harvard Aiken Computation Laboratory, 1981.

[44] V. Rastogi and S. Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *SIGMOD Conference*, pages 735–746, 2010.

[45] F. Roesner, T. Kohno, and D. Wetherall. Detecting and Defending Against Third-Party Tracking on the Web. In *NSDI*, 2012.

[46] E. Shi, J. Bethencourt, H. T.-H. Chan, D. X. Song, and A. Perrig. Multi-Dimensional Range Query over Encrypted Data. In *IEEE Symposium on Security and Privacy*, 2007.

[47] E. Shi, T.-H. H. Chan, E. G. Rieffel, R. Chow, and D. Song. Privacy-Preserving Aggregation of Time-Series Data. In *NDSS*, 2011.

[48] C. Silverstein, M. Henzinger, H. Marais, and M. Moricz. Analysis of a very large altavista query log. Technical report, 1998-014, Systems Research Center, Compaq Computer Corporation, 1998.

[49] E. G. Sirer, S. Goel, M. Robson, and D. Engin. Eluding carnivores: file sharing with strong anonymity. In *ACM SIGOPS European Workshop*, 2004.

[50] D. X. Song, D. Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In *IEEE Symposium on Security and Privacy*, 2000.

[51] L. Sweeney. k-Anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.

[52] M. Taghavi, A. Patel, N. Schmidt, C. Wills, and Y. Tew. An analysis of web proxy logs with query distribution pattern approach for search engines. *Computer Standards & Interfaces*, 34(1):162 – 170, 2012.

[53] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Dissent in Numbers: Making Strong Anonymity Scale. In *OSDI*, 2012.