Collaborative, Privacy-Preserving Data Aggregation at Scale

Haakon Ringberg[*], Benny Applebaum[*], Michael J. Freedman[*], Matthew Caesar[†], Jennifer Rexford[*]
[*]Princeton University, [†]University of Illinois at Urbana-Champaign

ABSTRACT

Combining and analyzing data collected at multiple locations is critical for a wide variety of applications, such as detecting and diagnosing malicious attacks or computing an accurate estimate of the popularity of Web sites. However, legitimate concerns about privacy often inhibit participation in collaborative data-analysis systems. In this paper, we design, implement, and evaluate a practical solution for privacy-preserving collaboration among a large number of participants. Scalability is achieved through a "semi-centralized" architecture that divides responsibility between a *proxy* that obliviously blinds the client inputs and a *database* that identifies the (blinded) keywords that have values satisfying some evaluation function.

Our solution leverages a novel cryptographic protocol that provably protects the privacy of both the participants and the keywords. For example, if web servers collaborate to detect source IP addresses responsible for denial-of-service attacks, our protocol would not reveal the traffic mix of the Web servers or the identity of the "good" IP addresses. We implemented a prototype of our design, including an amortized oblivious transfer protocol that substantially improves the efficiency of client-proxy interactions. Our experiments show that the performance of our system scales linearly with computing resources, making it easy to improve performance by adding more cores or machines. For collaborative diagnosis of denial-of-service attacks, our system can handle millions of suspect IP addresses per hour when the proxy and the database each run on two quad-core machines.

1. INTRODUCTION

Many important data-analysis applications must combine and analyze data collected by multiple parties. Such distributed data analysis is particularly important in the context of security. For example, victims of denial-of-service (DoS) attacks know they have been attacked but cannot easily distinguish the malicious source IP addresses from the good users who happened to send legitimate requests at the same time. Since compromised hosts in a botnet often participate in multiple such attacks, victims could potentially identify the bad IP addresses if they combined their measurement data [34]. Cooperation is also useful for Web clients to recognize they have received a bogus DNS response or a forged self-signed certificate, by checking that the information they received agrees with that seen by other clients accessing the same Web site [29, 39]. Collaboration is also useful to identify popular Web content by having Web users—or proxies monitoring traffic for an entire organization—combine their access logs to determine the most frequently accessed URLs [1]. In this paper, we present the design, implementation, and evaluation of an efficient, privacy-preserving system that supports these kinds of data-analysis operations.

Today, these kinds of distributed data-analysis applications lack privacy protections. Existing solutions often rely on a trusted (typically centralized) aggregation node that collects and analyzes the raw data, thereby learning both the identity and inputs of participants. There is good reason to believe this inhibits participation. ISPs and Web sites are notoriously unwilling to share operational data with one another, because they are business competitors and are concerned about compromising the privacy of their customers. Many users are understandably unwilling to install software from Web analytics services such as Alexa [1], as such software would otherwise track and report every Web site they visit. Unfortunately, even good intentions do not necessarily translate to good security and privacy protections, only too-well demonstrated by the fact that large-scale data breaches have become commonplace [30]. As such, we believe that many useful distributed data-analysis applications will not gain serious traction unless privacy can be ensured.

Fortunately, many of these collaborative data-analysis applications have a common pattern, such as computing set intersection, finding so-called *icebergs* (items with a frequency count above a certain threshold), or identifying items that in aggregate satisfy some other statistical property. We refer to this problem as *privacy-preserving data aggregation* (PDA). Namely, each participant p_j has an input set of key-value tuples, $\langle k_i, v_{i,j} \rangle$, and the protocol outputs a key k_i if and only if some evaluation function $f(\forall j | v_{i,j})$ is satisfied. For example, the botnet anomaly-detection application is an instance of the iceberg problem where the goal is to detect keys that occur more than some threshold τ times across the n parties. In this scenario, the keys k_i refer to IP addresses, each value $v_{i,j}$ is 1, and f is defined to be $\sum_{j=1}^{n} v_{i,j} \geq \tau$ (implemented, in fact, as simply keeping a running sum per key). In other words, such a protocol performs the equivalent of a database join (union) across each participant's input (multi)set, and outputs those IP addresses that appear more than τ times. In our system, keys can either be arbitrary-length bitstrings or can also be drawn from a limited domain (*e.g.*, the set of valid IP addresses). However, we restrict our consideration of values to those drawn from a more restricted domain—such as an alphanumeric score from 1 to 10 or A to F—a limitation for privacy reasons we explain later. This f could as easily perform other types of frequency analysis on keys, such as median, mode, or dynamically setting the threshold τ based on the set of inputs—for example, if there exists some appropriate "gap" between popular and unpopular inputs—as opposed to requiring τ be set *a priori* and independent of the inputs.

Informally, PDA should provide two privacy properties: (1) *Keyword privacy* requires that no party should learn anything about k_i if its corresponding values do not satisfy f. (2) *Participant privacy* requires that no party should learn which key inputs (whether or not the key remains somehow blinded prior to satisfying f) belongs to which participant. In our example of collaborating DoS victims, keyword privacy means nobody learns the identity of good IP addresses or which Web sites they frequent, and participant privacy means a Web site need not worry that its mix of clients would be revealed. In our example of collaborating Web clients, the privacy guarantees mean that a Web user need not worry that other users know what Web sites he accesses, or whether he received a bogus DNS response or a forged certificate. We believe these privacy properties would be sufficient to encourage participants to collaborate to their mutual benefit, without concern that their privacy (or the privacy of their clients) would be compromised. Our goal, then, is to design a system that provably guarantees these properties, yet is efficient enough to be used in practice.

Ideally, we would like a system that can handle hundreds or thousands of participants generating thousands of key-value tuples. Unfortunately, fully-distributed solutions do not scale well enough, and fully-centralized solutions do not meet our privacy requirements. Simple techniques like hashing input keys [12, 2], while efficient, cannot ensure keyword and participant privacy. In contrast, the secure multi-party computation protocols from the cryptographic literature [42, 9, 25, 21, 11, 10, 20, 23, 3] would allow us to achieve our security goals, but are not practical at the scale we have in mind. [40] has a similar intent to our work, but provides much weaker privacy properties (*e.g.*, keys are known by the system) and was not evaluated in a distributed setting. Finally, few of these systems have ever been implemented [23, 13, 3], let alone operate in the real world [4] and at scale. So, a meta-goal of our work is to help bring multi-party computation to life.

In this paper, we *design, implement, and evaluate* a viable alternative: a "semi-centralized" system architecture, and associated cryptographic protocols, that provides privacy-preserving data aggregation without sacrificing efficiency. Rather than having a single aggregator node, the data analysis is split between two separate parties—a *proxy* and a *database*. The proxy plays the role of obliviously blinding client inputs, as well as transmitting blinded inputs to the database. The database, on the other hand, builds a table that is indexed by the blinded key. For each row of this table whose values satisfy f, the database shares this row with the proxy, who unblinds the key. The database subsequently publishes its non-blinded data for that key.

The resulting semi-centralized system provides strong privacy guarantees *provided that the proxy and the database do not collude*. In practice, we imagine that these two components will be managed either by the participants themselves that do not wish to see their own information leaked to others, perhaps even on a rotating basis, or even third-party commercial or non-profit entities tasked with providing such functionality. For example, in the case of cooperative DoS detection, ISPs like AT&T and Sprint could jointly provide the service. Or, perhaps even better, it could be offered by third-party entities like Google (which already plays a role in bot and malware detection [15]) or the EFF (which has funded anonymity tools such as Tor [7]), who have no incentive to collude. Such a separation of trust appears in several cryptographic protocols [6], and even in some natural real-world scenarios, such as Democrats and Republicans jointly comprising election boards in the U.S. political system. It should be emphasized that the proxy and database are not treated as *trusted parties*: we only assume that they will not collude. Indeed, jumping ahead, our protocol does not reveal sensitive information to either party.

Using a semi-centralized architecture greatly reduces operational complexity and simplifies the liveness assumptions of the system. For example, clients can asynchronously provide their key-value tuples without our system requiring any complex scheduling. Despite these simplifications, the cryptographic protocols necessary to provide strong privacy guarantees are still non-trivial. Specifically, our solution makes use of oblivious pseudorandom functions [27, 10, 16], amortized oblivious transfer [26, 17], and homomorphic encryption with re-randomization.

We formally prove that our system guarantees keyword and participant privacy. We first show a protocol that is robust in the *honest-but-curious* model (where, informally, each party can perform local computation on its own view in an attempt to break privacy, but still faithfully follows the protocol). Then, we show how, with a few modifications to our original protocol, to defend against *any coalition of malicious participants*. In addition, the protocols are robust in the face of collusion between either proxy/database and any number of participants.

The remainder of the paper is organized as follows. Section 2 defines our system goals and discusses why prior techniques are not sufficient. Section 3 describes our PDA protocols and sketches the proofs of their privacy guarantees. Section 4 describes our implementation, and Section 5 evaluates its performance. We conclude the paper in Section 6.

2. DESIGN GOALS AND STATUS QUO

This section defines our goals for practical, large-scale privacy-preserving data aggregation (PDA), and we discuss how prior proposals failed to meet these requirements. We then expand on our security assumptions and privacy definitions.

2.1 Design Goals

In the privacy-preserving data aggregation (PDA) problem, a collection of participants (or *clients*) may autonomously make observations about *values* (v_i) associated with *keys* (k_i). These observations may be, for example, the fact that an IP address is suspected to have performed some type of attack (through DoS, spam, phishing, and so forth), or the number of participants that associate a particular credential with a server. The system jointly computes a two-column input table T. The first column of T is a set comprised of all unique keys belonging to all participants (the *key column*). The second column is comprised of a value $\mathsf{T}[k_i]$ that is the aggregation or union of all participant's values for k_i (the *value column*). The system then defines a particular function f to be evaluated over each row's value(s). For simplicity, we focus our discussion on the simple problem of over-threshold set intersection for f: If clients' inputs of the form $\langle k_i, 1\rangle$ are aggregated as $\mathsf{T}[k_i] \leftarrow \mathsf{T}[k_i] + 1$, is $\mathsf{T}[k_i] \geq \tau$?

A practical PDA system should provide the following:

- ***Keyword privacy:*** We say a system satisfies *keyword privacy* if, given the above aggregated table T, at the conclusion of the protocol all involved parties learn only the keys k_i whose corresponding aggregate value $\mathsf{T}[k_i] \geq \tau$. In addition, we might also have parties learn the values $\mathsf{T}[k_i]$, *i.e.*, the entire value column of T, even if the corresponding keys remain unknown. We discuss later why we may reveal the keyless value column (a histogram of frequencies in the over-threshold set intersection example) in addition to those over-threshold keys.

- ***Participant privacy:*** We say a system satisfies *participant privacy* if, at the conclusion of the protocol, nobody can learn the inputs $\{\langle k_i, v_{i,j}\rangle\}$ of participant p_j other than p_j himself (except for information which is trivially deduced from the output of the function). This is formally captured by showing that the protocol leaks no more information than an ideal implementation that uses a trusted third party. This convention is standard in secure multi-party computation; further details can be found in [14].

- ***Efficiency:*** The system should scale to large numbers of participants, each generating and inputting large numbers of observations (key-value tuples). The system should be scalable both in terms of the network bandwidth consumed (communication complexity) and the computational resources needed to execute the PDA (computational complexity).

- ***Flexibility:*** There are a variety of computations one might wish to perform over each key's values $\mathsf{T}[k_i]$, other than the sum-over-threshold test. These may include finding the maximum value for a given key, or checking if the median of a row exceeds a threshold. Rather than design a new protocol for each function f, we prefer to have a single protocol that works for a wide range of functions.

- ***Lack of coordination:*** Finally, the system should operate without requiring that all participants coordinate their efforts to jointly execute some protocol at the same time, or even all be online around the same time. Furthermore, no set of participants should be able to prevent others from executing the protocol and computing their own results (*i.e.*, a liveness property).

As we discuss next, existing approaches fail to satisfy one or more of these goals.

2.2 Limitations of Existing Approaches

Having defined these five goals for PDA, we next consider several possible solutions from the literature. We see that prior secure multi-party computation protocols achieve strong privacy at the cost of efficiency, flexibility, or ease of coordination. On the other hand, simple hashing or network-layer anonymization approaches fail to satisfy our privacy requirements. Our protocol, which leverages insights from both approaches, combines the best of both worlds. Table 1 summarizes the discussion in this section.

Set-Intersection Protocols. Freedman *et al.* [11] proposed a specially-designed secure multi-party computation protocol to compute set intersection between the input lists of two parties. It represented each party's inputs as the roots of an encrypted polynomial, and then had the other party evaluate this encrypted polynomial on each of its own inputs. While asymptotically optimized for this setting, a careful protocol implementation found two sets of 100 items each took 213 seconds to execute (on a 3 GHz Intel machine) [13]. Kissner and Song [20] extended and further improved this polynomial-based protocol for a multi-party decentralized setting, yet their computational complexity remains $O(n\ell^2)$ and communication complexity is $O(n^2\ell)$, where n is the number of participants and ℓ is the number of input elements per party. Furthermore, after a number of pairwise interactions between participants, the system needed to coordinate a group decryption protocol between all parties. Hence, this prior work on set-intersection faces scaling challenges on large sets of inputs or participants, and it also requires new protocol design for each small variant of the set-intersection or threshold set-intersection protocol.

Secure Multi-Party Computations using Garbled Circuits. In 1982, Yao [42] proposed a general technique for computing any two-party computation privately, by building a "garbled circuit" in which one party encodes the function to be executed and his own input, and the other party obliviously evaluates her inputs on this circuit. Very recently, the Fairplay system [23, 3] provided a high-level programming language for automatically compiling specified functions down into garbled circuits and generating network protocol handlers to execute them. While such a system would provide the privacy properties we require and offer the flexibility that hand-crafted set-intersection protocols lack, this comes at a cost. These protocols are even more expensive in both computation and communication, requiring careful coordination as well.

Hashing Inputs. Rather than building fully decentralized protocols—with the coordination complexity and quadratic overhead (in n) this entails—we could aggregate data and compute results using a centralized server. One approach is to simply have clients first hash their keys before submitting them to the server (*e.g.*, using SHA-256), so that a server only sees $H(k_i)$, not k_i itself [2]. While it may be difficult to find a pre-image of a hash function, brute force attacks are still always possible: In our collaborating intrusion detection application, for instance, a server can simply compute the hash values of all four billion IP addresses and build a simple lookup table. Thus, while certainly efficient, this approach fails to achieve either of our privacy properties. An alternative that prevents such a brute-force attack would be for all participants (clients) to coordinate and jointly agree on some secret key s, then use instead a *keyed* pseudorandom function on the input key, *i.e.*, $F_s(k_i)$. This would satisfy keyword privacy, until a single client decides to share s with the server, a brittle condition for sure.

Network Anonymization through Proxying. In the previous proposal, the server received inputs directly from clients. Thus, the server was always able to associate a row of the database with a particular client, whether or not its key is known. One solution would be to simply proxy a client's request through one or more intermediate proxies that hides the client's identity (*e.g.*, its own IP address), as done in onion routing systems such as Tor [7]. Of course, this solution still does not achieve keyword privacy.

Although the prior approaches have their limitations, they also offer important insights that inform our design. First, a more centralized aggregation architecture avoids distributed coordination and communication overhead. Second, proxying can add participant privacy when interacting with a server. And third, a keyed pseudorandom function (PRF) can provide keyword privacy. Now, the final insight to our design is, *rather than have all participants jointly agree on the PRF secret s, let it be chosen by and remain known only to the proxy*. After all, the proxy is already trusted not to expose a client's identity to the server (database), so let's trust it not to expose this secret s to the database as well. Thus, prior to proxying (roughly) the tuple $\langle F_s(k_i), v_i \rangle$, the proxy executes a protocol with a client to *blind* its input key k_i with F_s. This blinding occurs in such a way that the client does not learn s and the proxy does not learn k_i.[1] This completes the loop, having a proxy play a role in providing both keyword and participant privacy, while the database offers flexibility in any computation over a key's values $\mathsf{T}[k_i]$ and scalability through traditional replication and data-partitioning techniques (*e.g.*, consistent hashing [19]).

2.3 Security Assumptions and Definitions

We now motivate and clarify some design decisions related to our security assumptions and privacy definitions. Roughly speaking, our final protocol defends against *malicious participants* and non-colluding *honest-but-curious* databases and proxies.

[1] We note that oblivious pseudorandom function evaluation had been previously used in the set intersection context in [10] and [16].

	Keyword	Participant			Lack of
Approach	Privacy	Privacy	Efficiency	Flexibility	Coordination
Private Set Intersection	**Yes**	**Yes**	Poor	No	No
Garbled-Circuit Evaluation	**Yes**	**Yes**	Very Poor	**Yes**	No
Hashing Inputs	No	No	**Very Good**	**Yes**	**Yes**
Network Anonymization	No	**Yes**	**Very Good**	**Yes**	**Yes**
This paper	**Yes**	**Yes**	**Good**	**Yes**	**Yes**

Table 1: Comparison of proposed schemes for privacy-preserving data aggregation

Honest-but-curious parties. In our model, both proxy and database are expected to act as *honest-but-curious* (also called *semi-honest*) participants. That is, each party can perform local computation on its own view in an attempt to break privacy, but is assumed to still faithfully follow the protocol when interacting with other parties. We believe this model is very appropriate for our semi-centralized system architecture. In many deployments, the database and proxy may be well-known and trusted to act on their good intentions to the best of their abilities, as opposed to simply another participant amongst a set of mutually distrustful parties. Thus, other than fully compromising a server-side component and secretly replacing it with an actively malicious instance, data breaches are not possible in this model, as participants never see privacy-comprising data in the first place. In addition, the honest-but-curious model is one of the two standard security models in multi-party computation protocols—the other being the (obviously stronger) assumption of full malicious behavior. Unfortunately, security against fully malicious behavior comes at a great cost, as each party needs to prove at each step of the protocol that it is faithfully obeying it. For example, the proxy would need to prove that it does not omit any submitted inputs while proxying, nor falsely open blinded keys at the end of the protocol; the database would need to prove that it faithfully aggregates submitted values, and doesn't omit any rows in T that satisfy f. These proofs, typically done in zero-knowledge, greatly complicate the protocol and impact efficiency.

We will, however, present a protocol that is robust against any coalition of *malicious participants*. After all, the same trust assumptions that hold for the proxy and database does not extend to the potentially large number of participants.

Security against coalitions. Another important aspect of security is the ability to preserve privacy even when several adversarial players try to break security by sharing the information they gained during the protocol. In this aspect, we insist on providing security against any coalition of an arbitrary number of participants together with the database. This is essential as otherwise the database can perform a Sybil attack [8], *i.e.*, create many dummy participants and use their views, together with his own view, to reveal sensitive information. Similarly, we require security against any coalition of the proxy and the participants. On the other hand, in order to have an efficient and scalable system, we are willing to tolerate vulnerability against a coalition of the database and the proxy, which could otherwise break participant and keyword privacy.

Releasing the value column. Our protocol releases those keys whose values satisfy f, but the database also learns the entire value column ($\mathsf{T}[k_i], \forall i$), even though it learns no additional information about the corresponding k_i's. This asymmetric design was chosen as revealing all $\mathsf{T}[k_i]$ may be seen as a privacy violation.

That said, in other settings it may be acceptable to release the entire value column, so that all parties see identical information. This also serve another practical purpose, as it may be hard to fully specify f *a priori* to collecting clients' inputs. For example, how should an anomaly detection system choose the appropriate frequency threshold τ? In some attacks, 10 observations about a particular IP address may be high (*e.g.*, suspected phishing), while in others, 1000 observations may be necessary (*e.g.*, for bots participating in multiple DoS attacks). Furthermore, a dataset may naturally expose a clear gap between frequency counts of normal and anomalous behavior; the very reason data operators like to "play" with raw data in the first place.

We also note that the acceptable set of input values and the system's security assumptions has some bearing here. If the domain $\mathcal{D}$ of possible values is large, a client can try to "mark" a key k by submitting it together with an uncommon value $w \in \mathcal{D}$. If a value column that somehow includes w is revealed, the client can discover other clients' values for that same key. That said, a similar problem exists when the value column is not released and one is concerned about collusions between a client and database (who can search for the $\mathsf{T}[k]$ that includes w). This problem does not arise when the domain is relatively small (*e.g.*, when values are grades over some limited scale).

We mention that this asymmetry and/or security issue can be completely eliminated by first having participants encrypt their values under the public keys of both proxy and database, and by then using additional cryptographic protocols for the aggregation of the values. While these tools are relatively expensive, the structure of our system allows us to employ them only for the two-party case (for the proxy and database) which results in a significant efficiency improvement over other more distributed solutions.

3. OUR PDA PROTOCOL

In this section, we describe our protocol and analyze its security. Section 3.1 describes a simplified version of the protocol that achieves somewhat weaker security properties. This version will be extended to support a stronger notion of security in Section 3.2. Our protocol employs several standard cryptographic tools (*e.g.*, public-key encryption schemes, pseudorandom functions, and the oblivious evaluation of a pseudorandom function). We will elaborate on these tools and suggest concrete instantiations in Section 3.3. More details about the extended protocol and sketches of formal security proofs are given in the Appendix.

3.1 The Basic Protocol

Our protocol consists of four basic steps (see Figure 1). In the first two steps, the proxy interacts with the participants to collect the blinded keys together with their associated values encrypted under the database's public-key, and then passes these encrypted values on to the database. Then, in the third step, the DB aggregates the blinded keys together with the associated values in a table and decides which rows should be revealed according to a predefined function f. Finally, the DB asks the proxy to unblind the corresponding keys. Since the blinding scheme F_s is not necessarily invertible, the revealing mechanism uses some additional information that is sent during the first phase.

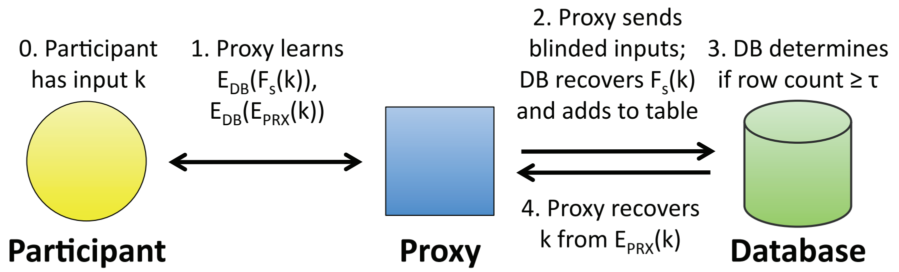

0. Participant has input k
1. Proxy learns $E_{DB}(F_s(k))$, $E_{DB}(E_{PRX}(k))$
2. Proxy sends blinded inputs; DB recovers $F_s(k)$ and adds to table
3. DB determines if row count ≥ τ
4. Proxy recovers k from $E_{PRX}(k)$
Participant
Proxy
Database

Figure 1: High-level system architecture and protocol. F_s is a keyed hash function whose secret key s is known only to the proxy.

- **Parties**: Participants, Proxy, Database.
- **Cryptographic Primitives**: A pseudorandom function F, where $F_s(k_i)$ denotes the value of the function on the input k_i with a key s. A public-key encryption E, where $E_{\mathrm{K}}(x)$ denotes an encryption of x under the public key K.
- **Public Inputs**: The proxy's public key PRX, the database's public key DB.
- **Private Inputs.** *Participant:* A list of key-value pairs $\langle k_i, v_i \rangle$. *Proxy:* key s of PRF F and secret key for PRX; *Database:* secret key for DB.

1. Each participant interacts with the proxy as follows. For each entry $\langle k_i, v_i \rangle$ in the participant's list, the participant and the proxy run a sub-protocol for oblivious evaluation of the PRF (OPRF). At the end of this protocol, the proxy learns nothing and the participant learns only the value $F_s(k_i)$ (and nothing else, not even the key s of the PRF). The participant computes the values $E_{\text{DB}}(F_s(k_i))$, $E_{\text{DB}}(v_i)$, and $E_{\text{DB}}(E_{\text{PRX}}(k_i))$, and it sends them to the proxy. (The last entry will be used during the revealing phase.) The proxy adds this triple to a list and waits until most/all participants send their inputs.

2. The proxy randomly permutes the list of triples and sends the result to the DB. The DB uses its private key to decrypt all the entries of each triple. Now, it holds a list of triples of the form $\left\langle F_s(k_i), v_i, E_{\text{PRX}}(k_i) \right\rangle$. The DB inserts these values into a table which is indexed by the (blinded) key $F_s(k_i)$. At the end, the DB has a table of entries of the form $\left\langle F_s(k_i), \mathsf{T}[k_i], \mathsf{E}[k_i] \right\rangle$, where $\mathsf{T}[k_i]$ is (in general) a list of all the v_i's that appeared with this key (or simply the number of times a client inputted k_i in the case of threshold set intersection), and $\mathsf{E}[k_i]$ is a list of values of the form $E_{\text{PRX}}(k)$.

3. The DB uses some predefined function f to partition the table into two parts: R, which consists of the rows whose keys should be revealed, and H, which consists of the rows whose keys should remain hidden. Then, it sends all the rows of R to the proxy.

4. The proxy goes over the received table R and replaces all the encrypted $E_{\text{PRX}}(k_i)$ entries with their decrypted key k_i. Then it publishes the updated table.

Variants. One may consider several variants in which different information is released to the participants by the database. For example, it is possible to release only the *keys* k_i which are chosen by the function f without the corresponding values $\mathsf{T}[k_i]$. On the other extreme, the DB can release more data by publishing the pairs $(k_i, \mathsf{T}[k_i])$ for the k_i's that are selected by f, together with the values $\mathsf{T}[k_i]$ of the keys that were not selected by f without the corresponding keys (*i.e.*, the entries $\mathsf{T}[k_i]$ of the table H). This might be useful to the participants and, in some scenarios, the additional information might not constitute a privacy violation (in the "real-world" sense). Consider, for example, the case where the values are always one, *i.e.*, the participants only want to increment a counter for some key. In this case, the table R simply consists of keys and their frequencies, and H is simply a frequency table of all the unrevealed keys.

Security Guarantees. We claim that this protocol guarantees privacy against the following attacks:

Coalition of honest-but-curious participants. Consider the view of an honest-but-curious participant during the protocol. Due to the security of the OPRF sub-protocol, a single participant sees only a list of pseudorandom values of the form $F_s(k_i)$, and therefore it learns nothing beyond the output of the protocol (formally, this view can be easily simulated by using truly random values). The same holds for a coalition of participants.

In fact, this protocol achieves a reasonable level of security against malicious participants as well. Recall that the interaction of the proxy with a participant is completely *independent* of the inputs of other participants. Hence, even if the participants are malicious, they still learn nothing about the data of other honest participants. Furthermore, even malicious participants will be forced to choose their inputs *independently* of the inputs of other honest participants. For example, they cannot duplicate the input of some other honest participant. (Similar security definitions were also considered in [26, 16].) However, malicious participants can still violate the *correctness* of the above protocol. This issue will be fixed in the extended protocol.

Honest-but-curious proxy. The proxy's view consists of three parts: (1) the view during the execution of the OPRF protocol—this gives no information due to the security of the OPRF; (2) the tuples that the participants send—these values are encrypted under the DB's key and therefore reveal no information to the proxy; and (3) the values that the DB sends during the last stage of the protocol—these are just key-value pairs (encrypted under the proxy's key) that should be revealed anyway, and thus they give no additional information beyond the actual output of the protocol.

Honest-but-curious coalition of proxy and participants. The above argument generalizes to the case where the proxy colludes with honest-but-curious participants. Indeed, the joint view of such coalition reveals nothing about the inputs of the honest participants.

Honest-but-curious database. The DB sees a blinded list of keys encrypted under his public key DB, without being able to relate the blinded entries to their owners. For each blinded key $F_s(k_i)$, the DB also sees the list of its associated values $\mathsf{T}[k_i]$ and encryptions of the keys under the proxy's key $E_{\text{PRX}}(k)$. Finally, the DB also sees the output of the protocol. The values $F_s(k_i)$ and $E_{\text{PRX}}(k)$ bear no information due to the security of the PRF and the encryption scheme. Hence, the DB learns nothing but the value table of the inputs (*i.e.*, the $\mathsf{T}[k_i]$'s for all k_i's).[2]

3.2 The Full-Fledged Protocol

In the following, we describe how to immunize the basic protocol against stronger attacks.

Honest-but-curious coalition of participants and database. A careful examination of the previous protocol shows that it is vulnerable against such coalitions for two main reasons.

First, a participant knows the blinded version $F_s(k_i)$ of its own keys k_i, and, in addition, the DB can associate all the values $\mathsf{T}[k_i]$ to their blinded keys $F_s(k_i)$. Hence, a coalition of a participant and a DB can retrieve all the values $\mathsf{T}[k_i]$ that are associated with a key k_i that the participant holds, even if this key *should not be revealed* according to f. To fix this problem, we modify the first step of the protocol. Instead of using an OPRF protocol, we will use a different sub-protocol in which the participant learns nothing and the proxy learns the value $E_{\text{DB}}(F_s(k_i))$ for each k_i. This solves the problem as now that participant himself does not know the blinded version of his own keys. To the best of our knowledge, this version of encrypted-OPRF protocol (abbreviated EOPRF) has not appeared in the literature before. Fortunately, we are able to construct such a protocol by carefully modifying the OPRF construction of [10] and combining it with El-Gamal encryption (see Section 3.3).

Second, we should eliminate subliminal channels, as these can be used by participants and the database to match the keys of a participant to their blinded versions (that were forwarded to the DB by the proxy). Indeed, public-key encryption schemes use randomness (in addition to the public key) to encrypt a message, and this randomness can be used as a subliminal channel. To solve this problem, we use an encryption scheme that supports re-randomization of ciphertexts; that is, given an encryption of x with randomness b, it should be possible to recompute an encryption of y under fresh randomness b' (without knowing the private key). Now we eliminate the subliminal channel by asking the proxy to re-randomize the ciphertexts—$E_{\text{DB}}(F_s(k_i))$, $E_{\text{DB}}(v_i)$, and $E_{\text{DB}}(E_{\text{PRX}}(k_i))$—which are encrypted under the DB's public key (at Step 1). Furthermore, we should be able to re-randomize the *internal* ciphertext $E_{\text{PRX}}(k_i)$ of the last entry as well (we will show that this can be achieved through variant of El-Gamal encryption).

A coalition of malicious participants. As we already observed, malicious participants can violate the correctness of our protocol. Specifically, they might try to submit ill-formed inputs. Recall that the participant sends to the proxy triples $\langle a, b, c\rangle$, where in an honest execution we have $a = E_{\text{DB}}(F_s(k_i)), b = E_{\text{DB}}(v_i)$ and $c = E_{\text{DB}}(E_{\text{PRX}}(k_i))$ for some k_i and v_i. However, a cheating participant might provide an inconsistent tuple, in which $a = E_{\text{DB}}(F_s(k_i))$ while $c = E_{\text{DB}}(E_{\text{PRX}}(k_i'))$ for some $k_i' \neq k_i$. We can prevent such an attack by asking the proxy to apply a consistency check to R in the last step of the protocol and to make sure that $E_{\text{PRX}}(k_i')$ and $F_s(k_i)$ match. The proxy omits all the inconstant values (if there are any) and asks the DB to check again if the corresponding row should be revealed after the omission. (This modification suffices as long as the function f is local, *i.e.*, it is applied to each row separately. See appendix for more details.)

Another thing that a cheating participant might do is to replace b with some "garbage" value $b' = E_{\text{DB}}(v')$ for which he does not know the plaintext v' (while this might not seem to be beneficial in practice, we must prevent such an attack in order to meet our strong definitions of security). To prevent such attack, we ask the participant to provide a zero-knowledge proof that shows that he knows the plaintext v to which that b decrypts. As seen in the next section, this does not add too much overhead.

Finally, our sub-protocol for the EOPRF should be secure against malicious participants in the following sense: a malicious participant should not be able to generate a blinded value $E_{\text{DB}}(F_s(k_i))$ for a key k_i that he does not know.

In the appendix, we show that our modifications guarantee full security against malicious participants.

3.3 Concrete Instantiation of the Cryptographic Primitives

In the following section, we assume that the input keys are represented by m-bit strings. We assume that m is not very large (*e.g.*, less than 192–256); otherwise, one can hash the input keys and apply the protocol to resulting hashed values.

Public Parameters. Our implementation mostly employs Discrete-Log based schemes. In the following, g is a generator of a multiplicative group $\mathbb{G}$ of prime order p for which the decisional Diffie-Hellman (DDH) assumption holds. We publish (g, p) during initialization and assume the existence of algorithms for multiplication (and thus also for exponentiation) in $\mathbb{G}$. We let $\mathbb{Z}_p$ denote the field of integers modulo p, the set $\{0, 1, \ldots, p-1\}$ with multiplication and addition modulo p. We will let $\mathbb{Z}_p^*$ denote the multiplicative group of the invertible elements $\mathbb{Z}_p$.

El-Gamal Encryption. We will use El-Gamal encryption over the group $\mathbb{G}$. The private key is a random element a from $\mathbb{Z}_p^*$, and the public key is the pair $(g, h = g^a)$. To encrypt a message $x \in \mathbb{G}$, we choose a random b from $\mathbb{Z}_p^*$ and compute $(g^b, x \cdot h^b)$. To decrypt the ciphertext (A, B), compute $B/A^a = B \cdot A^{-a}$ (where $-a$ is the additive inverse of a in $\mathbb{Z}_p$). In case we wish to "double-encrypt" a message $x \in \mathbb{G}$ under two different public-keys (g, h) and (g, h'), we will choose a random b from $\mathbb{Z}_p^*$ and compute $(g^b, x \cdot (h \cdot h')^b)$. This ciphertext as well as standard ciphertexts can be re-randomized by multiplying the first entry (resp. second entry) by $g^{b'}$ (resp. $h^{b'}$) where b' is chosen randomly from $\mathbb{Z}_p^*$. Finally, a zero-knowledge proof for knowing the decryption of a given ciphertext is described in [36]. The scheme adds only 3 exponentiations and does not increase the overall round complexity as it can be applied in parallel to the EOPRF protocol.

Naor-Reingold PRF [27]. The key s of the function $F_s : \{0,1\}^m \to \mathbb{G}$ contains m values $(s_1, \ldots, s_m)$ chosen randomly from $\mathbb{Z}_p^*$. Given m-bit string $k = x_1 \ldots x_m$, the value of $F_s(k)$ is $g^{\prod_{x_i=1} s_i}$, where the exponentiation is computed in the group $\mathbb{G}$.

Oblivious-Transfer [31, 26]. To implement the sub protocol of Step 1, we will need an additional cryptographic tool called Oblivious Transfer (OT). In an OT protocol, we have two parties: sender and receiver. The sender holds two strings (α, β), and the receiver has a selection bit c. At the end of the protocol, the receiver learns a *single* string: α if $c = 0$, and β if $c = 1$. The sender learns nothing (in particular, it does not know the value of the selector c).

3.3.1 *The Encrypted-OPRF protocol*

Our construction is inspired by a protocol for oblivious evaluation of the PRF F, which is explicit in [10] and implicit in [25, 26]. We believe that this construction might have further applications.

[2] Formally, we define a functionality in which this additional information is given to the database as part of its output. See the appendix for details.

- **Parties**: Participant, Proxy.
- **Inputs.** *Participant:* m-bit string $k = (x_1 \dots x_m)$; *Proxy:* secret key $s = (s_1, \dots, s_m)$ of a Naor-Reingold PRF F.

1. Proxy chooses m random values $u_1, \dots, u_m$ from $\mathbb{Z}_p^*$ and an additional random $r \in \mathbb{Z}_p^*$. Then for each $1 \leq i \leq m$, the proxy and the participant invoke the OT protocol where proxy is the sender with inputs $(u_i, s_i \cdot u_i)$ and receiver uses x_i as his selector bit. That is, if $x_i = 0$, the participant learns u_i and otherwise it learns $s_i \cdot u_i$. The proxy also sends the value $\hat{g} = g^{r/\Pi u_i}$. (These steps can be done in parallel.)

2. The participant multiplies together the values received in the OT stage. Let M denote this value. Then, it computes $\hat{g}^M = (g^{\Pi_{x_i=1} s_i})^r = F_s(k)^r$. Finally, the participant chooses a random element a from $\mathbb{Z}_p^*$ and encrypts $F_s(k)^r$ under the public key $\text{DB} = (g, h)$ of the database. The participant sends the result $(g^a, F_s(k)^r \cdot h^a)$ to the proxy.

3. The proxy raises the received pair to the power of r', where r' is the multiplicative inverse of r modulo p. It also re-randomizes the resulting ciphertext.

Correctness. Recall that $\mathbb{G}$ has a prime order p. Hence, when the pair $(g^a, F_s(x)^r \cdot h^a)$ is raised to the power of $r' = r^{-1}$, the result is $(g^{ar'}, F_s(k) \cdot h^{ar'})$, which is exactly $E_{\text{DB}}(F_s(k))$. Thus, the protocol is correct.

Privacy. All the proxy sees is the random tuple $(u_1, \dots, u_m, r)$ and $E_{\text{DB}}(F_s(k)^r)$. This view gives no additional information except of $E_{\text{DB}}(F_s(k))$. (Formally, the view can be perfectly simulated given $E_{\text{DB}}(F_s(k))$.) On the other hand, we claim that all the participant sees is a sequence of random values and therefore it also learns nothing. Indeed, the participant sees the vector $(s_1^{x_1} \cdot u_1, \dots, s_m^{x_m} \cdot u_m)$, whose entries are randomly distributed over $\mathbb{G}$, as well as the value $\hat{g} = (g^{1/\Pi u_i})^r$. Since r is randomly and independently chosen from $\mathbb{Z}_p^*$, and since $\mathbb{G}$ has a prime order p, the element $\hat{g}$ is also uniformly and independently distributed over $\mathbb{G}$. The protocol supports security against malicious participants (in the sense that was described earlier) as long as the underlying OT is secure against a malicious receiver.

3.3.2 *Implementing Oblivious Transfer*

In general, oblivious transfer is an expensive public-key operation (*e.g.*, it may take two exponentiations per single invocation). In the above protocol, then, we execute an OT protocol for each *bit* of the participants input k (which would result, for example, in 64 exponentiations just to input a single IP address). However, Ishai *et al.* [17] show how to reduce the amortized cost of OT to be as fast as matrix multiplication. This "batch OT" protocol uses a standard OT protocol as building block. We implemented this batch OT protocol on top of the basic OT protocol of [26].[3]

3.4 Efficiency of our Protocol

In both the basic and extended protocol, the round complexity is constant, and the communication complexity is linear in the number of items. The protocol's computational complexity is dominated by cryptographic operations. For each m-bit input key, we have the following amortized complexity: (1) The participant who holds the input key computes 3 exponentiations in the basic protocol (respectively 8 in the extended protocol), as well as $O(m)$ modular multiplication / symmetric-key operations in both versions. (2) The proxy computes 5 exponentiations in the basic protocol (resp. 12 in the extended protocol) and $O(m)$ modular multiplication / symmetric-key operations. (3) The database computes 3 exponentiations in the basic protocol (resp. 5 in the extended protocol).

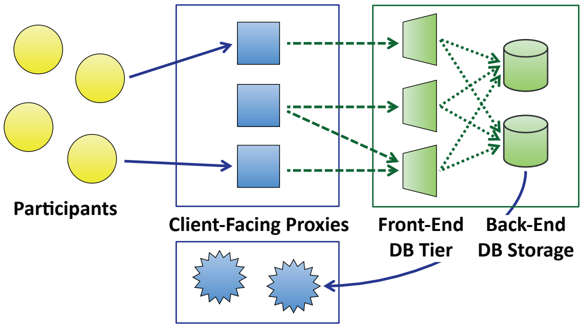


Figure 2: Distributed proxy and database architecture

4. DISTRIBUTED IMPLEMENTATION

In our system, both the proxy and database logical components can be physically replicated in a relatively straightforward manner. In particular, our design can scale out horizontally to handle higher loads, by increasing the number of proxy and/or database replicas, and then distributing requests across these replicas. Our distributed architecture is shown in Figure 2. Our current implementation covers all details described in the basic protocol, as well as some security improvements of the extended version (*e.g.*, including the EOPRF, but not ciphertext re-randomization, proofs of knowledge, or the final consistency check).

4.1 Proxy: Client-Facing Proxies and Decryption Oracles

One administrative domain can operate any number of proxies. Each proxy's functionality may be logically divided into two components: handling client requests, and serving as decryption oracles for the database when a particular key should be revealed. None of these proxies need to interact, other than having all client-facing proxies use the same secret s to key the pseudorandom function F and all decryption-oracle proxies use the same public/private key PRX. In fact, these two proxies play different logical roles in our system and could even be operated by two different administrative domains. In our current implementation, all proxies register with a single group membership server, although a distributed group membership service could be implemented for additional fault tolerance [5, 41].

To discover a client-facing proxy, a client contacts this group membership service, which returns a proxy IP address in round-robin order (this could be replaced by any technique for server selection, including DNS, HTTP redirection, or a local load balancer). To submit its inputs, a client connects with this proxy and then executes an amortized Oblivious Transfer (OT) protocol on its input batch. This results in the proxy learning $\left\langle E_{\text{DB}}(F_s(k_i)), E_{\text{DB}}(v_i), E_{\text{DB}}(E_{\text{PRX}}(k_i)) \right\rangle$ for each input tuple, which it pushes onto an internal queue. (While Section 3.3 only described the use of ElGamal encryption, its special properties are only needed for $E_{\text{DB}}(F_s(k_i))$; the other public-key operations can

[3] The "batch OT" protocol also has a version which preserves security against a malicious receiver. This increases the number of multiplications by a multiplicative factor, but does not affect the number of expensive public-key operations.

be RSA, which we use in our implementation.) When this queue reaches a certain length (10,000 in our implementation), the proxy randomly permutes (shuffles) the items in the queue, and sends them to a database server.

The database, upon determining that a key k_i's value satisfies f, sends $E_{\text{PRX}}(k_i)$ to a proxy-decryption oracle. The proxy-decryption oracle decrypts $E_{\text{PRX}}(k_i)$ and returns k_i to the database for storage and subsequent release to other participants in the system.

4.2 Database: Front-end Decryption and Back-end Storage

The database component can also be replicated. Similar to the proxy, we separate database functionality into two parts: the *front-end* module that handles proxy submissions and decrypts inputs, and a *back-end* module that acts as a storage layer. Each logical module can be further replicated in a manner similar to the proxy.

The servers comprising the front-end database tier do not need to interact, other than being configured with the same public/private keypair DB. Thus, any front-end database can decrypt the $E_{\text{DB}}(F_s(k_i))$ input supplied by a proxy, and the proxies can load balance input batches across these database servers.

The back-end database storage, on the other hand, needs to be more tightly coordinated, as we ultimately need to aggregate all $F_s(k_i)$'s together, no matter which proxy or front-end database processed them. Thus, the back-end storage tier partitions the keyspace of all 1024-bit strings over all storage nodes (using consistent hashing [19]). All such front-end and back-end database instances also register with a group membership server, which the front-end servers contact to determine the list of back-end storage nodes. Upon decrypting an input, the front-end node determines which back-end storage node is assigned the resulting key $F_s(k_i)$, and sends the tuple $\left\langle F_s(k_i), v_i, E_{\text{PRX}}(k_i)\right\rangle$ to this storage node.

As these storage nodes each accumulate a horizontal portion of the entire table T, they test the value column for their local table to see if any keys satisfy F. For each such row, the storage node sends the tuple $\left\langle F_s(k_i), T[k_i], E_{\text{PRX}}(k_i)\right\rangle$ to a proxy-decryption oracle.

4.3 Prototype Implementation

Our design is implemented in roughly 5,000 lines of C++. All communication between system components—client, front-end proxy, front-end database, back-end database storage, and proxy-decryption oracle—is over TCP using BSD sockets. We use the GnuPG library for large numbers (bignums) and cryptographic primitives (*e.g.*, RSA, ElGamal, and AES). The Oblivious Transfer protocol (and its amortized variant) were implemented from scratch, and comprised a total of 625 lines of code. All RSA encryption used a 1024-bit key, and ElGamal used a corresponding 1024-bit group size. AES-256 was used in the batch OT and its underlying OT primitive. The back-end database simply stores table rows in memory, although we plan to replace this with a durable key-value store (*e.g.*, BerkeleyDB [28]).

5. PERFORMANCE EVALUATION

We wish to evaluate our system along three primary dimensions. (a) Given fixed computing resources, what is the throughput of our system as a function of the size of the input set? (b) What are the primary factors limiting throughput? And, (c) how does the throughput scale with increasing computing resources? In each case, we are concerned with both (1) how long it takes for clients to send key-value pairs to the proxy during the OT phase (*proxy throughput*) and (2) how long it takes for the DB to decrypt and identify keys with values that satisfy the function f (*DB throughput*). We have instrumented our code to measure both. For a given experiment requiring the proxy to process n keys, proxy throughput is defined as n divided by the time it takes between when the first client contacts any client-facing proxy and when the last key is processed by some client-facing proxy. Similarly, database throughput is defined as the number of keys processed between when the first client-facing proxy forwards keys to some DB front-end and when the DB back-end storage processes the last submitted keys.

Our experiments were run on multiple machines. The servers (proxy and DB) were run on HP DL160 servers (quad-core Intel Xeon 2 GHz machines with 4 GB RAM running CentOS Linux). These machines can perform a 1024-bit ElGamal encryption in 2.2 ms, ElGamal decryption in 2.5 ms, RSA encryption in 0.5 ms, and RSA decryption in 2.8 ms. Due to a lack of homogeneous servers, the clients were run on different machines depending on the experiment. The machines used for the clients were either (A) of the same configuration as the servers, or one of either (B) Sun SunFire X4100 servers with two dual-core 2.2 GHz Opteron 275 processors (four 64-bit cores) with 16GB RAM running CentOS, or (C) Dell PowerEdge 2650 servers with two 2.2 GHz Intel Xeon processors and 5 GB of memory, also running Linux.

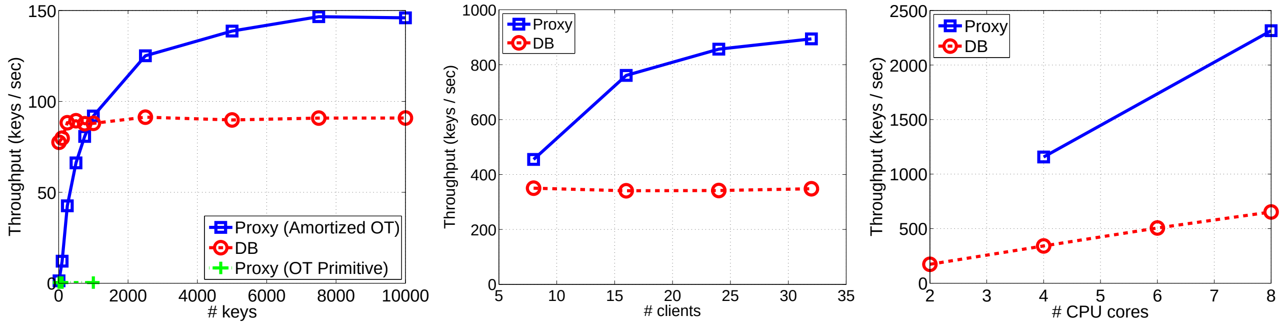
As noted in the introduction, our system can be used in different contexts. One of the most prominent is that of anomaly detection: specifically, networks collaborating to identify attacking IP addresses—*e.g.*, belonging to a botnet—with greater confidence. Modern botnets can range up to roughly 100,000 unique hosts [32], and we would like our system to be able to correlate suspicions of hundreds of participating networks within some numbers of hours. In order to support such a usage scenario, our implementation will need to be able to process millions of keys in the span of hours or many hundreds of keys per second. We will revisit the feasibility of our implementation for our supporting applications in Section 5.2, but these numbers should provide rough expectations for the throughput numbers to be presented in Section 5.1.

5.1 Scaling and Bottleneck Analysis

Effect of number of keys (Figure 3a). The input trace to our system is parameterized by the number of clients and by the number of keys they each submit. In Figure 3a, we measure the throughput of our system as a function of the number of keys. More precisely, we run a single client, a single proxy, and a single DB in order to measure single-CPU-core proxy throughput and single-CPU-core DB throughput. The top solid curve shows proxy throughput when the proxy and client utilize the amortized OT protocol, the middle dashed curve shows DB throughput, and the bottom partial curve shows proxy throughput when the proxy and client utilize only the standard OT primitive, which does not include our amortization-based extensions. The throughput of the OT primitive is exceedingly low (less than one key per second), which is why it was not evaluated on the full range of x-values.

Proxy throughput scales well with the number of incoming keys when the client and proxy utilize the amortized OT protocol. Throughput increases with increasing numbers of keys per batch, as the amortized OT calls the primitive OT a fixed number of k times regardless of the number of input keys n. With small n (*e.g.*, up to 1000), the cost of these calls to the primitive OT dominate overall execution time and leave the proxy underutilized. However, as the size of the input set increases, the cost of encrypting keys on the client becomes the primary bottleneck, which is the plot shows minimal increase in throughput above $n = 8000$.

DB throughput, on the other hand, does not scale with the num-



(a) Scaling number of keys (b) Scaling number of participants (c) Scaling number of proxy/DB replicas

Figure 3: *Scaling:* Effect of (a) number of keys, (b) number of participants, and (c) number of proxy/database replicas.

ber of keys. The reason for this is that the intensive work on the DB is decryption, which is performed in batch, and it is therefore entirely CPU limited. The DB becomes CPU limited at 10 keys and remains CPU limited at 10,000 keys (*i.e.*, latency goes up and throughput remains constant). We noted earlier that the machines on which the DB and proxy run require 2.5 ms per decryption. Since the DB has to perform 3 decryptions per key, the DB therefore has a maximum throughput of 135 keys per second on a single CPU core. Figure 3a shows that our DB implementation achieves throughput of roughly 90 keys per second.

The amortized OT protocol [17] introduces a trade-off between the message overhead and memory consumption. The memory footprint of this protocol per client-proxy interaction for n keys is $n \times 32 \times 2 \times 1024/8 = 8196n$ bytes (*i.e.*, we assume 32 bits per key, the 2 values for the OT primitive, 1024-bit keys, and 8 bits per byte). For $n = 10,000$ keys, for example, this requires 82 MB on both the proxy and the client. A proxy communicating with 100 clients would therefore require in excess of 8GB of memory. A user of the protocol could choose to execute the amortized OT protocol in stages, however, by sending k keys at a time, which would reduce the memory footprint. Our system is parameterized to support this, and because Figure 3a shows that there is little to gain from batch sizes in excess of 5,000 keys, the remainder of our experiments will use batch sizes of 5,000 keys.

Our architecture is designed to maximize throughput, not minimize latency. In fact, providing a meaningful measure of latency is challenging for multiple reasons: (a) the DB processes $f \overset{\text{def}}{=} \mathsf{T}[k_i] \geq \tau$ once every t seconds (*i.e.*, not upon arrival, which wouldn't make sense unless $\tau = 1$); (b) the proxy batches and randomly permutes/shuffles key-value pairs for security; and (c) the substantial benefit of the amortized OT (over the OT primitive: see again Figure 3a) is lost if the client submits only a 1 key-value pair, which is required for a "true" latency experiment. These qualifiers notwithstanding, Figure 3a does provide a form of "mean" latency. That is, a single client with 5000 keys would see mean proxy latency of 7.2 milliseconds per key and mean DB latency of 11.1 milliseconds per key.

Effect of number of participants (Figure 3b). Here we evaluate the throughput of our system as a function of the number of clients sending keys. In this experiment, we limit the proxy and DB to one server machine each. Four client-facing proxy processes are launched on one machine and four front-end DB processes are launched on the other. They can therefore potentially utilize all eight cores on these two machines. Figure 3b shows that the proxy scales well with the number of clients. Proxy throughput increases by nearly a factor of two between 8 and 32 clients. This signifies that, when communicating with a single client, a proxy spends a substantial fraction of its time idling. The four proxies in this experiment are not CPU limited until they handle 32 clients, at which time the throughput approaches 900 keys per second. The DB, on the other hand, is CPU-bound throughout. It has a throughput of about 350 keys per second, independent of the number of clients.

Global		Within amortized OT					
wait	encrypt	wait	pow	AES	arith	other	OT
60%	1%	0%	16%	4%	4%	6%	7%

Table 2: Breakdown of proxy resource usage

Global		Within amortized OT					
wait	encrypt	wait	pow	AES	arith	other	OT
0%	40%	31%	16%	2%	1%	3%	7%

Table 3: Breakdown of client resource usage

Effect of number of replicas (Figure 3c). Finally, we wish to analyze how our distributed architecture scales with the available computing resources. In this experiment, we provide up to 8 cores across two machines to each of the proxy and DB front-ends. While the proxy is evaluated on 64 clients, computing resource constraints meant that the DB is evaluated on 32 clients.

Both our proxy and DB scale linearly with the number of CPU cores allocated to them. Throughput for the DB with 2 cores when handling 32 clients was over 173 keys per second, whereas at 8 cores the throughput was 651 keys per second: a factor of 3.75 increase in throughput for a factor of 4 increase in computing resources. The proxy has throughput of 1159 keys per second when utilizing 4 cores and 2319 when utilizing 8 cores: an exact factor of 2 increase in throughput for an equal increase in computing resources. This clearly demonstrates that our protocol, architecture, and implementation can scale up to handle large data sets. In particular, our entire system could handle input sizes on the order of millions of keys in hours.

Micro-benchmarks. To gain a deeper understanding of the factors limiting the scalability of our design, we instrumented the code to account for how the client and proxy were spending their CPU cycles. While the DB is entirely CPU bound due only to decryptions (*i.e.*, its limitations are known), the proxy and client engage in the oblivious OT protocol whose bottlenecks are less clear. In Tables 2 and 3, we therefore show the fraction of time the client and proxy, respectively, spend performing various tasks needed for their exchange. In this experiment, we have a single client send keys to a single proxy at the maximum achievable rate.

At the highest level, we split the tasks performed into (a) waiting (called "wait"), (b) encrypting or decrypting values ("encrypt"), or (c) engaging in the amortized OT protocol. We further split work within the amortized OT protocol into time spent waiting, performing modulo exponentiations ("pow"), calling AES256, performing basic arithmetic such as multiplication, division, or finding multiplicative inverses ("arith"), calling the OT primitive ("OT"), and any other necessary tasks ("other") such as XOR'ing numbers, generating random numbers, allocating or de-allocating memory, etc.

Table 2 shows that when communicating with a single client, the client-facing proxy spends more than 60% of its time idling while waiting for the client—it is *more than* 60% because some part of the 7% of time spent within the OT primitive is also idle time. The 60% idle time is primarily due to waiting for the client to encrypt k_i and $F_s(k_i)$. The single largest computational expense for the proxy is performing powmods at 16%; the remaining non-OT tasks add up to 15%. In order to make the proxy more efficient, therefore, utilizing a bignum library with faster exponentiation and basic arithmetic would be advantageous.

The client also spends a non-trivial amount of time waiting—31% of total execution time—but substantially less than the proxy. It spends 40% of its time encrypting values. The reason this 40% does not match up with the 60% idle time of the proxy is because the proxy finishes its portion of the amortized OT before the client does its portion. That is, 20 out of the proxy's 60% idle time is due to the client processing data sent by the proxy in the last stage of the amortized OT protocol, and the remaining 40 is due to the client encrypting its values. A with the proxy, the client would benefit from faster exponentiations, but encryption is clearly the major bottleneck. We noted before that the GnuPG cryptographic library we use performed public-key operations in approximately 2.5–2.8 ms. On the same servers, we benchmarked the Crypto++ library to perform RSA decryption in only 1.2 ms, increasing speed by 130%. Crypto++ would also allow us to take advantage of elliptic curve cryptography, which would increase system throughput. In future work, we plan to modify our implementation to use this library.

5.2 Feasibility of Supporting Applications

In this section, we revisit several potential applications of our system. We consider our results in light of their potential demands on request rate: the number of requests per unit time that must be satisfied, the number of keys which must be stored in the system, and the number of participants.

Anomaly detection. Network operators commonly run systems to detect and localize anomalous behavior within their networks. These systems dynamically track the traffic mix—*e.g.*, the volume of traffic over various links or the degree of fanout from a particular host—and detect behavior that differs substantially from the statistical norm. For example, Mao *et al.* [24] found that most DDoS attacks observed within a large ISP were sourced by fewer than 10,000 source IPs, and generated 31,612 alarms over a four-week period (0.8 events per hour). In addition, Soule *et al.* [37] found that volume anomalies occurred at a rate of four per day on average, most of which involved fewer than several hundred source IPs. Finally, Ramachandran *et al.* [33] found were able to localize 4,963 Bobax-infected host IPs sending spam from a single vantage point. We envision our system could be used to improve accuracy of these techniques by correlating anomalies across ISP boundaries. We found our system could handle 10,000 IP addresses as keys, with a request rate of several hundred keys per second, even with several hundred participants. Given our system exceeds the requirements of anomaly detection, our system may enable the participants to "tune" their anomaly detectors to be more sensitive, and reduce false positive rates by leveraging other ISPs' observations.

Cross-checking certificates. Multiple vantage points may be used to validate authenticity of information (such as a DNS reply or ssh certificate [29, 39]) in the presence of "man-in-the-middle" attacks. Such environments present potentially larger scaling challenges due to the potentially large number of keys that could be inserted. According to [18], most hosts execute fewer than 15 DNS lookups per hour, and according to [35], ssh hosts rarely authenticate with more than 30 remote hosts over long periods of time. Here, we envision our system could simplify the deployment of such schemes, by reducing the amount of information revealed about clients' request streams. Under this workload (15 key updates per hour, with 30 keys per participating host), our system scales to support several hundred hosts with only a single proxy. Extrapolating out to larger workloads, our system can handle tens of thousands of clients storing tens of thousands of keys with under fifty proxy/database pairs.

Distributed ranking. Search tools such as Alexa and Google Toolbar collect information about user behavior to refine search results returned to users. However, such tools are occasionally labeled as *spyware* as they reveal information about the contents of queries performed by users. Our tool may be used to improve privacy of user submissions to these databases. It is estimated that Alexa Toolbar has 180,000 active users, and it is known that average web users browse 120 pages per day. Here, the number of participants is large, but the number of keys they individually store in the system is smaller. Extrapolating our results to 180,000 participants, and assuming several thousands of keys, our system can still process several hundred requests per second (corresponding to several hundred thousand clients) per proxy/database pair.

6. CONCLUSIONS

In this paper, we presented the design, implementation, and evaluation of a collaborative data-analysis system that is both scalable and privacy preserving. Since a fully-distributed solution would be complex and inefficient, our design divides responsibility between two independent parties—a proxy that obliviously blinds the client inputs and a database that identifies the (blinded) keys that have values satisfying an evaluation function. The functionality of both the proxy and the database can be easily distributed for greater scalability and reliability. Experiments with our prototype implementation show that our system performs well under increasing numbers of keys, participants, and proxy/database replicas. The performance is well within the requirements of our motivating applications, such as collaborating to detect the malicious hosts responsible for DoS attacks or to validate the authenticity of information in the presence of man-in-the-middle attacks.

As part of our ongoing work, we plan to evaluate our system in the context of several real applications—first through a trace-driven evaluation and later by extending our prototype to run these applications. In addition, we plan to explore opportunities to deploy our system in practice. A promising avenue is distributed Internet monitoring infrastructures such as NetDimes [38] and the new M-Lab (Measurement Lab) initiative [22]. We believe our system could lower the barriers to collaborative data analysis over the Internet, enabling a wide range of new applications that could improve Internet security, performance, and reliability.

7. REFERENCES

[1] Alexa the Web Information Company, 2009. `http://www.alexa.com/`.

[2] Allman, M., Blanton, E., Paxson, V., and Shenker, S. Fighting coordinated attackers with cross-organizational information sharing. In *HotNets* (November 2006).

[3] Ben-David, A., Nisan, N., and Pinkas, B. FairplayMP: A system for secure multi-party computation. In *Proc. ACM Computer and Communications Security Conference* (October 2008).

[4] Bogetoft, P., Christensen, D. L., Damgard, I., Geisler, M., Jakobsen, T., Krøigaard, M., Nielsen, J. D., Nielsen, J. B., Nielsen, K., Pagter, J., Schwartzbach, M., and Toft, T. Multiparty computation goes live. Cryptology ePrint Archive, Report 2008/068, 2008. `http://eprint.iacr.org/`.

[5] Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In *Proc. OSDI* (November 2006).

[6] Chor, B., Goldreich, O., Kushilevitz, E., and Sudan, M. Private information retrieval. *Journal of the ACM 45*, 6 (November 1998).

[7] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The second-generation onion router. In *Proc. 13th USENIX Security Symposium* (August 2004).

[8] Douceur, J. R. The Sybil attack. In *Proc. Intl. Workshop on Peer-to-Peer Systems* (March 2002).

[9] Fagin, R., Naor, M., and Winkler, P. Comparing information without leaking it. *Communications of the ACM 39*, 5 (1996), 77–85.

[10] Freedman, M. J., Ishai, Y., Pinkas, B., and Reingold, O. Keyword search and oblivious pseudorandom functions. In *Proc. Theory of Cryptography Conference* (February 2005).

[11] Freedman, M. J., Nissim, K., and Pinkas, B. Efficient private matching and set intersection. In *Advances in Cryptology — EUROCRYPT* (May 2004).

[12] Friend-of-a-Friend Project, 2009. `http://www.foaf-project.org/`.

[13] Garriss, S., Kaminsky, M., Freedman, M. J., Karp, B., Mazières, D., and Yu, H. Re: Reliable email. In *NSDI* (May 2006).

[14] Goldreich, O. *Foundations of Cryptography: Basic Applications*. Cambridge University Press, 2004.

[15] Google Safe Browsing for Firefox, 2009. `http://www.google.com/tools/firefox/safebrowsing/`.

[16] Hazay, C., and Lindell, Y. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *Proc. Theory of Cryptography Conference* (March 2008).

[17] Ishai, Y., Kilian, J., Nissim, K., and Petrank, E. Extending oblivious transfers efficiently. In *Advances in Cryptology — CRYPTO* (August 2003).

[18] Jung, J., Sit, E., Balakrishnan, H., and Morris, R. DNS performance and the effectiveness of caching. *IEEE/ACM Trans. Networking 10*, 5 (October 2002).

[19] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing* (1997).

[20] Kissner, L., and Song, D. Privacy preserving set operations. In *Advances in Cryptology — CRYPTO* (August 2005).

[21] Lindell, Y., and Pinkas, B. Privacy preserving data mining. In *Advances in Cryptology — CRYPTO* (August 2000).

[22] M-Lab: Welcome to Measurement Lab, 2009. `http://www.measurementlab.net/`.

[23] Malkhi, D., Nisan, N., Pinkas, B., and Sella, Y. Fairplay: A secure two-party computation system. In *Proc. USENIX Security* (August 2004).

[24] Mao, Z., Sekar, V., Spatscheck, O., van der Merwe, J., and Vasudevan, R. Analyzing large DDoS attacks using multiple data sources. In *SIGCOMM Workshop on Large Scale Attack Defense* (September 2006).

[25] Naor, M., and Pinkas, B. Oblivious transfer and polynomial evaluation. In *Proc. Symposium on Theory of Computing* (May 1999).

[26] Naor, M., and Pinkas, B. Oblivious transfer with adaptive queries. In *Advances in Cryptology — CRYPTO* (August 1999).

[27] Naor, M., and Reingold, O. Number-theoretic constructions of efficient pseudorandom functions. In *Proc. Symposium on Foundations of Computer Science* (October 1997).

[28] Oracle. Berkeley DB, 2009. `http://www.oracle.com/technology/products/berkeley-db/`.

[29] Poole, L., and Pai, V. S. ConfiDNS: Leveraging scale and history to improve DNS security. In *Proc. Workshop on Real, Large Distributed Systems* (November 2006).

[30] Privacy Rights Clearinghouse. A chronology of data breaches, January 2009. `http://www.privacyrights.org/ar/ChronDataBreaches.htm`.

[31] Rabin, M. How to exchange secrets by oblivious transfer. Tech. Rep. TR-81, Harvard Aiken Computation Laboratory, 1981.

[32] Rajab, M. A., Zarfoss, J., Monrose, F., and Terzis, A. My botnet is bigger than yours (maybe, better than yours): why size estimates remain challenging. In *HotBots* (Berkeley, CA, USA, 2007).

[33] Ramachandran, A., and Feamster, N. Understanding the network-level behavior of spammers. In *Proc. ACM SIGCOMM* (September 2006).

[34] Ringberg, H., Soule, A., and Caesar, M. Evaluating the potential of collaborative anomaly detection. Unpublished report, 2008.

[35] Schechter, S., Jung, J., Stockwell, W., and McLain, C. Inoculating SSH against address harvesting. In *Proc. Network and Distributed System Security Symposium* (February 2006).

[36] Schnorr, C.-P. Efficient signature generation by smart cards. *Journal of Cryptology 4*, 3 (1991), 161–174.

[37] Soule, A., Ringberg, H., Silveira, F., Rexford, J., and Diot, C. Detectability of traffic anomalies in two adjacent networks. In *Passive and Active Measurement* (April 2007).

[38] The DIMES Project, 2009. `http://www.netdimes.org/new/`.

[39] Wendlandt, D., Andersen, D. G., and Perrig, A. Perspectives: Improving SSH-style host authentication with multi-path probing. In *Proc. USENIX Annual Technical Conference* (2008).

[40] Xie, Y., Reiter, M. K., and O'Hallaron, D. Protecting privacy in key-value search systems. In *ACSAC: Proceedings of the 22nd Annual Computer Security Applications Conference* (Washington, DC, USA, 2006), pp. 493–504.

[41] Yahoo! Hadoop Team. Zookeeper. `http://hadoop.apache.org/zookeeper/`, 2009.

[42] Yao, A. C. Protocols for secure computations. In *Proc. Symposium on Foundations of Computer Science* (November 1982).

APPENDIX

Here, we describe the extended protocol of Section 3.2.

1. Each participant interacts with the proxy as follows. For each entry $\langle k_i, v_i \rangle$ in the participant's list, the participant and the proxy run a sub-protocol for encrypted oblivious evaluation of the PRF (EOPRF). At the end of this protocol, the participant learns nothing and the proxy learns only the value $E_{\text{DB}}(F_s(k_i))$. The participant sends the values $E_{\text{DB}}(E_{\text{PRX}}(k_i))$ and $E_{\text{DB}}(v_i)$ together with a proof of knowledge (POK) for knowing the plaintext of the last entry. If the POK succeeds, then the proxy re-randomizes the ciphertexts and adds the triple to a list. Otherwise, if the POK fails, the proxy ignores the triple.

2. Same as in the original protocol.

3. The DB builds the tables R and H as in the original protocol. For each row in R, the DB sends to the proxy the value $F_s(k_i)$ together with the corresponding list $\mathsf{E}[k_i]$ which supposedly contains ciphertexts of the form $E_{\text{PRX}}(k_i)$. The DB also re-randomizes these ciphertexts.

4. The proxy goes over the received table. For each entry of the received table, it decrypts all the values in the list $\mathsf{E}[k_i]$ and verifies that the plaintext corresponds to the blinded key $F_s(k_i)$. It reports inconsistencies to the DB and sends k_i if it appears in the list $\mathsf{E}[k_i]$.
5. For each row, the DB updates the list $\mathsf{T}[k_i]$ by omitting the values v_i for which inconsistencies were found. Then, it applies f again to the updated row, checks whether it should be released, and, if so, publishes the corresponding key k_i together with the updated list of values $\mathsf{T}[v_i]$. (The value k_i was given by the proxy as at least one of the ciphertexts in $\mathsf{E}[k_i]$ was consistent with the blinded key.)

We now sketch the proofs for the security of the protocol. First let us formally define the functionality we consider. Consider all submitted key-value pairs as a table, where each distinct key k_i is associated with a list $\hat{\mathsf{T}}[k_i]$ of all values v_i submitted with it. Let $\hat{\mathsf{R}}$ be the sub-table that consists of all the rows that should be revealed (according to f), and let $\hat{\mathsf{H}}$ be the table that contains all the other entries with the key column omitted. Our functionality outputs $\hat{\mathsf{R}}$ as a public value and $\hat{\mathsf{H}}$ as a private output for the DB. We prove that our protocol securely computes this functionality.

Honest but curious coalition of participants and a proxy. The joint view of the proxy and the honest-but-curious (HBC) participants contains the following: (1) the inputs (k_i, v_i) of the HBC participants and the public outputs $\hat{\mathsf{R}}$; (2) the information exchanged by the proxy and the HBC participants during the first stage; (3) the view of the proxy when interacting with other participants in the first stage, which consists of the proxy's view of the sub-protocols (EOPRF and POK) as well as triples of the ciphertexts $E_{\mathrm{DB}}(v_i)$, $E_{db}(F_s(k_i))$, and $E_{\mathrm{DB}}(E_{\mathrm{PRX}}(k_i))$; and (4) the table R sent by the DB to the proxy at the "revealing" phase of the protocol.

This view can be simulated, given the corresponding inputs (k_i, v_i) and the outputs $\hat{\mathsf{R}}$, as follows. Choose a random PRF key s, as well as public keys PRX and DB. Simulate (1) and (2) in the natural way (all the information needed for these computations is given). To simulate (3), use the simulators of the sub-protocols and generate garbage ciphertexts $E_{\mathrm{DB}}(0), E_{\mathrm{DB}}(0), E_{\mathrm{DB}}(0)$. To simulate (4), encrypt the values in $\hat{\mathsf{R}}$ under PRX and blind the keys under s.

Honest-but-curious coalition of participants and a DB. The joint view of the proxy and the HBC participants contains the following: (1) the inputs (k_i, v_i) of the HBC participants and the public outputs $\hat{\mathsf{R}}$; (2) the view of the HBC participants during the interaction with the proxy, which consists of the view of the sub-protocols (EOPRF and POK) as well as triples of ciphertexts $E_{\mathrm{DB}}(v_i)$, $E_{db}(F_s(k_i))$, and $E_{\mathrm{DB}}(E_{\mathrm{PRX}}(k_i))$; and (3) the view of the DB when interacting with the proxy, which consists of the tables R and H (encrypted under the DB's public key).

Given the corresponding inputs (k_i, v_i), the public output $\hat{\mathsf{R}}$, and the DB's private output $\hat{\mathsf{H}}$, we show how to simulate the above view. First, choose a random PRF key s, as well as public keys PRX and DB. Then, simulate (1) and (2) in the natural way (all the information needed for these computations is now given). It remains just to simulate R and H. The table R can be computed from $\hat{\mathsf{R}}$ and s. To simulate H, we should somehow add blinded values to $\hat{\mathsf{H}}$ (and encrypt the tuples under DB). We do this by building a key-value table for the inputs of the HBC participants. Then, for each row k_i, we choose a random consistent row in $\hat{\mathsf{H}}$ and add the value $F_s(k_i)$ as an additional blinded-key column. (A row is consistent with a key k_i if the list of values of the HBC's that are associated with k_i appear as part of the value list of the row in $\hat{\mathsf{H}}$.) Finally, for those rows which are left with no blinded key column, a random value is added.

Malicious coalition of participants. Let A be an adversarial strategy for a coalition of cheating participants. We construct a simulator that achieves the same "cheating" affect in the ideal-world. The simulator S chooses a key s for the PRF, as well as pairs of private/public keys for the DB and proxy. It provides these keys to A and executes A. For each iteration i, A generates a triple (a_i, b_i, c_i), together with a POK for knowing the plaintext encrypted in c_i. (In an honest execution $a_i = E_{\mathrm{DB}}(F_s(k_i)), b_i = E_{\mathrm{DB}}(E_{\mathrm{PRX}}(k_i))$, and $c_i = E_{\mathrm{DB}}(v_i)$, for some k_i and v_i.) The simulator S uses the POK to extract v_i; if the POK fails, then S ignores the triple. Finally, S checks (using all the above keys) whether a_i and b_i are consistent (*i.e.*, it decrypts a_i to a'_i, decrypts b_i to b'_i, and then verifies that $F_s(b'_i) = a_i$). If the check fails, S ignores the tuple. Otherwise, the simulator, which now knows both k_i and v_i, passes these entries to the trusted party.