Privacy in Plain Sight: Using Cloud-based Office Tools for Private Information Retrieval

Chris Wacek and Wenchao Zhou, and Micah Sherr

ABSTRACT

This paper introduces the first private information retrieval (PIR) implementation that relies entirely on a free web-based office suite as the database backend. Our PIR scheme requires no hosted services other than Google Docs and therefore inherits the reliability, robustness, access control mechanisms, and well-defined interfaces of Google's public cloud infrastructure. We describe numerous architectural challenges and solutions for constructing a PIR backend using only Google Spreadsheet, and present our open-source implementation that achieves *symmetric* PIR—i.e., the cloud does not learn which spreadsheet cell is retrieved by the client, and the client learns only the requested cell.

1. INTRODUCTION

Users sometimes need to query a database without revealing to the database operator which records are of interest. For example, Alice may want to query a medical database to learn information about her health condition without disclosing her ailment to the database operator. The Strawman Solution—transferring the database in its entirety—is guaranteed to protect the user's privacy, but incurs unacceptable communication costs. In 1995, Chor et al. [3] introduced information theoretic techniques for more efficient *private information retrieval* (PIR) that rely on $k \geq 2$ database replicas; Kushilevitz and Ostrovsky [11] originated computational PIR variants that, under standard cryptographic assumptions, provide PIR using a single database server and incur less communication cost than transferring the entire database. As described in more detail below, a significant and growing body of subsequent work has proposed increasingly efficient means of PIR.

This increased efficiency has led to PIR's emergence as a practical tool for enhancing privacy in real systems. To illustrate, PIR has been applied to (i) location-based systems [6] to allow users to query for nearby services while maintaining their location privacy, (ii) e-commerce systems [8] to provide tiered pricing models with record-level granularity, (iii) the DNS system to permit more private domain lookups [16], and (iv) anonymity networks [19] to allow users to query onion router information [4, 22] without disclosing to a directory service the routers of interest.

In all previous work of which we are aware, PIR schemes require specialized databases that have been customized to support the relevant PIR protocol. In other words, each service that wishes to utilize PIR must either provide and maintain its own infrastructure or pay a provider to host the customized database backend. This paper poses the question: *can we build a PIR backend using only the (free) public cloud infrastructure*[1]*?* That is, we assess the feasibility of implementing an efficient PIR scheme [17] using only free web-based office suites.

1.1 Advantages of PIR on the Public Cloud

The public cloud infrastructure has several features that make it particularly well-suited as PIR backends. First, the services themselves are highly robust and distributed, obviating the need to separately develop, maintain, and secure a robust service infrastructure. Hardware maintenance, data backup and restoration, security best practices (e.g., software patching), and hosting costs (power, cooling, network connectivity, etc.) are all handled by the public cloud operator.

Second, public cloud infrastructures also often include user management and access control features. For example, Google Drive implements a simple access control system in which read, write, and "comment" permissions can be assigned on a per-document basis to users or groups. Both the enforcement of these policies and the maintenance of user accounts is handled by the public cloud provider.

Additionally, since the services are widely used and in most cases are protected by SSL/TLS (i.e., HTTPS), the act of using a service is not in-and-of-itself suspicious or even noteworthy. While this may not be of concern for all use cases, it is useful in situations in which a user who desires to maintain the privacy of their database queries may similarly want to maintain privacy about *the fact that they are making queries at all.* The importance of this point should not be understated. Many proposed uses of private information retrieval focus on the desire to avoid exposing information about oneself (often focusing on medical aspects); however, the very act of searching within a specialized medical database can leak large amounts of information about the querier. By using a completely unremarkable data backend, we remove this concern.

Finally and perhaps most importantly, public cloud providers often offer their services for free. In this paper, we implement a PIR scheme using Google Spreadsheet, a component of their Google Drive/Docs online software suite.

1.2 Goals and Contributions

This paper does not propose any cryptographic or performance advancements to PIR. We rely on the PIR scheme in-

[1]Our use of the term "public cloud infrastructure" is restricted to web-based office suites such as Google Docs or Microsoft's Office 365.

troduced by Melchor and Gaborit [17] and inherit its threat model of an honest-but-curious database (Google Spreadsheet) that obeys the protocol (Google Code API) but wishes to learn which datum (spreadsheet cell) is of interest to the user. (Conversely, the user's goal is to learn the value of a desired spreadsheet cell without revealing the identity of that cell to the web service.) Additionally, since the scheme by Melchor and Gaborit [17] provides *symmetric* private information retrieval, we aim to prevent the client from learning *any* additional information about the content of the spreadsheet other than the value of the requested cell.

It is unlikely that popular web-based office suites were designed with PIR backends in mind. In Section 3, we enumerate many of the challenges of implementing a PIR scheme on a web-based spreadsheet with limited support for advanced datatypes, structured storage, and stateful communication.

In this work, we contribute the design and open-source implementation of the Melchor and Gaborit PIR protocol for the public cloud infrastructure. We do not introduce any cryptographic novelty. Rather, our contributions center on the architectural design and implementation of a PIR scheme on what at first blush may appear as an incompatible backend. Our goals are to show the feasibility of implementing PIR on unmodified web-based office suites and promote that medium as an advantageous alternative to dedicated and specialized database backends.

The remainder of the paper is organized as follows. In the next section, we review the related work. We discuss challenges in porting PIR schemes to web-based office suites in Section 3. Section 4 reviews the PIR construction by Melchor and Gaborit [17] that serves as the basis of our cloud-based implementation. We present and evaluate our implementation in Sections 5 and 6, respectively. Section 7 discusses the limitations of our techniques. We conclude in Section 8.

2. RELATED WORK

Chor et al. [3] introduced the private information retrieval problem in 1995. The goal of a PIR scheme is to allow the user to retrieve an item from a b-bit database without revealing to the database which item was requested. Chor et al. prove that perfect *information theoretic* security in a single database setting is only possible through the inefficient Strawman strategy of transferring all b bits to the client. However, Chor et al. also demonstrate a more efficient information theoretic PIR scheme that uses multiple noncolluding database replicas and achieves a communication cost that is sublinear in b. Their scheme guarantees that no single database server receives a complete set of information about what the user is requesting [3]. Subsequent work [20] has shown that information theoretic approaches that utilize multiple non-colluding servers are more efficient than single-server PIR techniques that rely on computational assumptions (described next).

Based on standard cryptographic assumptions, *computational PIR* (CPIR) protocols use cryptographic techniques to disguise the queries made by the users to the server. The reliance on standard assumptions (in particular, invertible functions) allows CPIR to work in single-server settings. Several techniques have been shown to have sublinear communication cost in the size of the database (a requirement to improve over the trivial solution). These techniques include trapdoor permutations [12], Φ-Hiding [1], and homomorphic encryption [2, 11]. In this paper, we assume a single cloud provider and therefore focus on CPIR approaches.

In 2007, Sion and Carbunar evaluated a number of existing CPIR protocols and showed that while these protocols are sublinear in *communication* cost, they are extremely *computationally* expensive. This disparity lead them to the conclusion that given relative communication speeds and computational capacity, the trivial protocol was likely to outperform any CPIR protocol existing, and in fact, any one likely to exist [21].

However, Melchor and Gaborit proposed a new protocol shortly thereafter based on linear algebra and the addition of random noise to lattices. Their protocol showed holistic performance improvements of several orders of magnitude, and appears to make computational PIR a feasible solution [17]. They later generalized the algorithm to support larger database elements [18]. Work by Olumfin and Goldberg in 2012 showed empirically that the Melchor-Gaborit scheme was an order of magnitude more efficient than the trivial scheme, given the network bandwidth available to the current average consumer [20]. The same work showed a number of multi-server information theoretic PIR schemes which were 1-2 orders of magnitude more efficient yet.

Symmetric PIR. The critical requirement for a private information retrieval scheme is that the database be incapable of identifying the requested datum. An extension of PIR called *symmetric PIR* (SPIR) [15] requires not only that the query be hidden from the database, but that the user *only learns information about their intended query.* Several of the existing methods for PIR support SPIR, so long as queries to the database are rate-limited. (Without rate-limiting, a client can simply enumerate the database by issuing a query for each item in the database.) We note that the trivial solution for private information retrieval (i.e., transferring the entire database), fails the stricter definition of SPIR.

Leveraging the Public Cloud Infrastructure. We briefly remark that we are not the first to consider using services such as Google Spreadsheets for unorthodox uses. Grossman [7] argues that modern spreadsheets constitute expressive modeling languages, and proposes a set of "spreadsheet engineering practices." More recently, in their announcement of Google Apps Script [5], Google highlights scripts that allow one to play Hangman and Sudoku. Lenssen proposes a number of Google Apps scripts for Google Drive, including the use of Google Spreadsheets as a content management system [14]. This is the first work that we are aware in which a private information retrieval backend is implemented using a cloud-based office suite.

3. CHALLENGES

Customized private information retrieval databases are often highly specialized and developed for a particular task. They are often developed in an open programming environment such as C on a POSIX platform, and implemented as either a standalone service or a modification to existing database systems.

Hosting a PIR database using web-based office suites offers many advantages: robustness, scalability, user management, software patching, amongst others (see Section 1.1 for a fuller discussion). However, web-based office suites were very likely not intended as PIR backends. The software-as-a-service (SaaS) public cloud infrastructure supports a much more limited set of functionalities—i.e., those provided by the cloud applications themselves. Migrating the same server-side functionality as is possible using traditional programming languages to cloud services (in particular, Google Drive) poses a number of challenges, many of which we highlight below.

3.1 Constrained Semantics

The design decisions for the semantics of the programming and execution environment on a cloud infrastructure are tuned for cloud services. They are not designed to assist (and sometimes even contradict!) the desired semantics for PIR services:

* *Stateless communication:* In a private information retrieval scheme, a client communicates with the server, in potentially multiple rounds, to extract the desired information. Cloud services often expose an API that allows such communication. However, unlike in a traditional client-server model, the communication between the client and the server is often stateless: each datagram sent by a client to the service will be considered without regard for the previous one.
  For any private information retrieval scheme that deploys multiple rounds of communication and relies on a set of internally maintained system state, stateless communication essentially requires that state be maintained on the service in an ad-hoc fashion, which may increase complexity or simply not be possible.

* *Structured storage:* Persistent data stored on the cloud service must be stored in a format and layout which is supported by the service. For example, if using Google Drive as a backend, all persistent data must be stored in a Document or a Spreadsheet. Both of these options assume text-based contents, and none of them are ideal for supporting PIR operations that involve bit operations and matrix calculations. More generally, it imposes the requirement that PIR services be designed to manage data in a (hopefully efficient) way that is compatible with the constrained storage provided by the cloud service.

* *Publicly accessible data:* A cloud-based data store such as Google Spreadsheet operates by default in a trivial PIR mode in which all of the stored data (i.e., the contents of the spreadsheet) are sent to the client. This semantic poses a challenge for symmetric PIR in which the client should learn no information about the database other than the value of a queried item. A symmetric PIR scheme therefore requires that the data be obfuscated so that clients who are not the data owner cannot discern the data's plaintext.

3.2 Limited Access to Resources

Cloud services are usually executed in a sandbox. These sandboxes (1) limit the functions accessible by users and (2) share physical resources (such as memory) with other services.

* *Limited function invocation:* In a traditional client-server model, server side functions can be called by the client more or less arbitrarily. However, in a cloud service, server side execution sometimes needs to be triggered by a standard service action. For example, while Google Drive provides a Javascript execution environment, there is no API for calling arbitrary functions. Instead, functions may be called by adding a *trigger* to a spreadsheet cell that calls a function whenever the cell's data is modified.

* *Constrained execution:* Many CPIR schemes are computationally expensive, and hence the public cloud infrastructure must support potentially expensive operations. In particular, even in cases in which the cloud application offers a feature rich programming environment, the infrastructure may throttle or even abort expensive operations.

4. BACKGROUND: PIR ALGORITHM

Our public cloud implementation of CPIR is based on the lattice-hiding construction proposed by Melchor and Gaborit [17]. In this section, we provide a high level overview of the algorithm. For reference, we also provide brief versions of Melchor's and Galorit's query generation, response encoding, and response decoding procedures *for the standard client-server model* in Figure 1. Our implementation of the protocol for web-based office suites is described in Section 5.

In the PIR scheme by Melchor and Gaborit, a query is comprised of n matrices, one for each element in the database. These matrices are created by disturbing a random base $N \times 2N$ matrix with hard and soft noise. Hard noise is used for only one of the query matrices – the one corresponding to the index of the desired element. The parameter N can be considered a security parameter which makes it difficult to discern the N undisturbed columns of the original matrix that consists of $2N$ columns. The problem is provable NP-hard by reducing from the Punctured Code Problem [17]. The computational difficulty[2] for breaking the PIR scheme is $O(\binom{2N}{N})$ (i.e., the combinatorial number for selecting N elements from $2N$ elements), which is at least $O(2^{2N})$.

Response calculation splits each element into l_0 bits and multiplies the pieces into the query matrices, returning a reply vector of length N. Using knowledge of the original base matrix and the noise added, the client is able to filter out the soft noise and extract the requested element.

Under this scheme, database elements have a maximum size in bits, which must be equal to $N \times l_0$. Aguilar-Melchor and Gaborit later proposed an extension of their system which detaches the size of each database element from the N and l_0 parameters [18]. We chose not to implement this

[2]Melchor and Gaborit also showed that their PIR scheme is unlikely to be vulnerable to lattice based attacks.

Query Generation *Given l_0, N, n; set $q = 2^{2l_0}$, $p = 2^{3l_0}$. Let i_0 be the element of interest. All random matrices are generated over Z/pZ.*

1. Generate M as the concatenation of two random $N \times N$ matrices $[A \mid B]$.
2. For each $i \in 1 \dots n$ compute $M_i'' = P_i \times M$, where P_i is a random invertible $N \times N$ matrix
3. Generate a random $N \times N$ matrix Δ over Z/pZ, and $n-1$ random matrices $D_{i \dots n, i \neq i_0}$ over $\{1, -1\}$
4. Generate a random matrix D_{i_0} over $\{1, -1\}$ with each diagonal set to q
5. For each $i \in \{1 \dots n\}$ compute M_i' by adding $D_i \times \Delta$ to the second half (B) of M_i'' and permuting the columns.
6. Send $\{M_1' \dots M_n'\}$ to the database.

Response Encoding *Given $\{M_1' \dots M_n'\}$ and database elements $m = \{m_1 \dots m_n\}$. Let M_{ij} refer to the j^{th} row of M_i.*

1. For each element m_i in the database:
   a. Split m_i into N l_0-bit integers $\{m_{i1} \dots m_{iN}\}$.
   b. Construct the vector $v_i = \sum_{j=1}^{N} m_{ij} M_{ij}$
2. Return $V = \sum_{j=1}^{n} v_i$

Response Decoding *Given V, Δ, $M = [A \mid B]$. Let V_1, V_2 denote equal segments of V such that $|V_1| = |V_2|$ and $V = V_1 | V_2$.*

1. Apply the reverse permutation from step 5 of query generation to V.
2. Calculate $E = V_2' - V_1' A^{-1} B$
3. Compute $E' = E\Delta^{-1}$
4. For $e_j \in E' = [e_1 \dots e_n]$, compute
$$m_{i_0 j} = \begin{cases} e_j - (e_j \mod q) & \text{if } e_j \mod q < \frac{q}{2} \\ e_j - (e_j \mod q - q) & \text{if } e_j \mod q >= \frac{q}{2} \end{cases}$$
5. $[m_{i_0 1} \mid \dots \mid m_{i_0 n}]$ are the bits of the desired element.

Figure 1: Procedure for generating PIR queries *(top)*, encoding responses *(middle)*, and decoding responses *(bottom)* using the method proposed by Melchor and Gaborit [17].

extension. Instead we suggest that a Google Spreadsheet-based PIR system is better suited to supporting relatively small elements (given the natural limitations on the number of bytes supported by each cell). As such, while we evaluate several different element sizes, we do so by adjusting the parameters l_0 and N rather than by using the slightly more complex extended method.

As discussed in Section 2, this lattice-based method is one of the few computational private information retrieval schemes with sufficiently low computational complexity for reply generation to be feasible. This is particularly important for cloud services with limited computational resources (in particular, popular free web-based office suites).

5. PIR IN THE CLOUD

We implement a proof-of-concept cloud-based private information retrieval system using Google Spreadsheet as the cloud service. Google Spreadsheet offers a structured data store combined with a well-documented execution platform

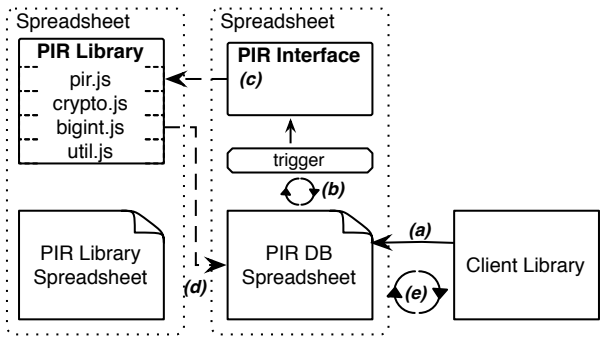

Figure 2: The elements involved in computing a PIR response.

(Javascript) and an API that permits client communication. We choose Google Spreadsheet over Google Document because it provides a level of structure. (It also still constrains our ability to store arbitrary data.) Our prototype implements a symmetric private information retrieval scheme, under some assumptions discussed in Section 7. Our implementation is available as free and open-source software, and may be downloaded from `http://db.tt/Bsvd37hh`.[3]

In the remainder of this section, we first present an overview of our private information retrieval service (Section 5.1), followed by more detailed descriptions of how information is stored and maintained (Section 5.2), and how a PIR query is processed given the data model (Section 5.3).

5.1 Overview

Our prototype PIR implementation consists of two Google Spreadsheets as shown in Figure 2. Generally, a Google Spreadsheet can be considered as a data store which holds multiple *sheets* of elements, where each element is a text-based string. We make use of a spreadsheet (i.e., the "PIR DB") to maintain the database uploaded by the data owner.

To support private information retrieval from the database, we use another Google Spreadsheet (i.e., the "PIR Library") as the container of a Google Apps Script project that implements the PIR algorithms described in Section 4. The PIR Library spreadsheet contains no data within its sheets and is used only as a container for the actual server-side PIR implementation. The implemented PIR algorithms are written as Javascript code that can be called from the associated spreadsheet via custom functions, directly from the script interface, or on events via *triggers*.

There are two user roles in our PIR system: the database owner and the query account. The owner creates both spreadsheets, and inserts the data (in encrypted form; see Section 7.1) into the data sheet in the PIR DB spreadsheet. The query account is publicly accessible by any user who has been granted access to use the PIR service. A user interacts with the hosted PIR service through a lightweight interface formated as a "query" sheet in the PIR DB spreadsheet (shown in Figure 3). A user submits its information retrieval request by writing the matrices obtained from the query generation to the query sheet (highlighted in yellow in Figure 3). Upon receiving the matrices, the server-side PIR implementation is then triggered, and, once the compu-

[3]The source code is temporarily and anonymously housed at DropBox for author-blind review.

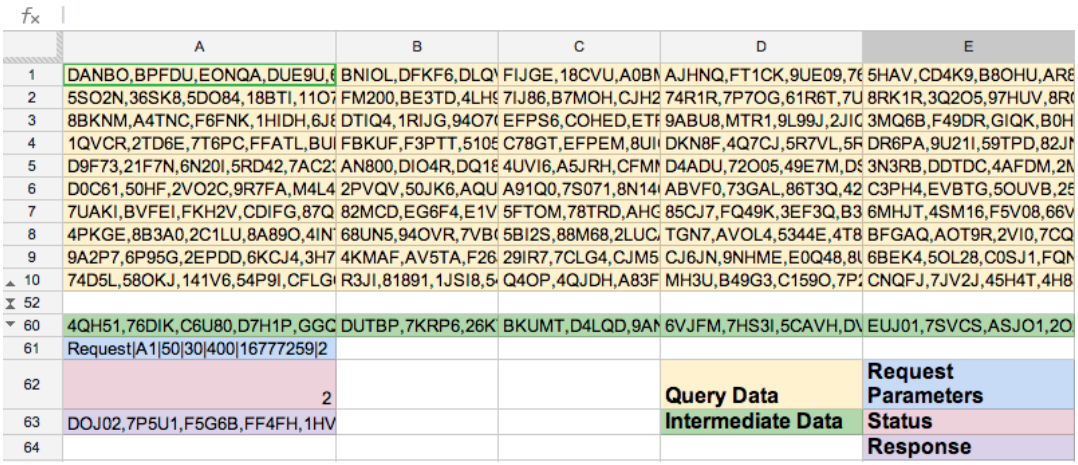

Figure 3: A screenshot depicting the various components of the query sheet (areas are denoted by fill color, with a legend in the lower right). A user submits its information retrieval request by writing the matrices obtained from the query generation (shown in yellow) along with control data (shown in blue), and checks and retrieves the query processing status and results by periodically reading from the corresponding elements (shown in red and purple respectively).

tation finishes, updates the query sheet to report the query processing status and results (highlighted in red and purple, respectively). The user then retrieves the query result by periodically reading from the corresponding elements.

While this section provides an overview of our prototype PIR implementation, we defer the discussion on how the challenges discussed in Section 3 are address in the following sections. We describe the data model and maintenance that enables symmetric PIR using *publicly accessible data* that are stored in *structured storage*. We then discuss how queries are processed with *stateless communications* and *limited function invocation*.

5.2 Data Model and Maintenance

Data model. To bridge the gap between the provided text-based storage in Google Spreadsheet and the requirement of frequent bit-based matrix operation, we encode database elements as comma-separated base-32 encoded strings. Given the maximum length restrictions within a single cell, a database element may be split into multiple cells.

In addition, to enable symmetric PIR, each database element is obfuscated so that users cannot discern the data's plaintext. We encrypt database elements using AES-128 in OFB mode prior to insertion into the data sheet. The reasons behind this are discussed more extensively in Section 7.

Once the database elements are loaded to the PIR DB spreadsheet, the corresponding sheet is *locked* by the data owner, to prevent the sheet from being modified by the query account or other unauthorized third party.

Access control. In Google Spreadsheet, access to the data contained within the data store is configurable at the spreadsheet level, where each user can be assigned *owner*, *edit*, and *view* permissions. Besides their intuitive effect on the access to the data elements stored in the spreadsheet, these permissions also affect the usage of Google Apps Script projects. (Recall that the server-side PIR implementation is constructed using such scripts.) More specifically, an Apps Script project may load an App Script project from another spreadsheet as a library and call functions from it. Under Spreadsheet's access policies, each user of the script must have at least *view* permissions on the linked spreadsheet in order to load the library.

	PIR DB	PIR Library
Database owner	*owner*	*owner*
Query account	*read*	*view*

Table 1: The permissions granted to each user.

To support symmetric PIR on Google Spreadsheet, the query account is given *edit* permissions on the database (i.e., the PIR DB spreadsheet), and *view* permissions on the PIR Library spreadsheet. The use of encryption effectively prevents the query account from meaningfully interpreting the (encrypted) database elements. On the other hand, *view* permissions on the PIR Libary allow the query account to access the spreadsheet, but not its associated Script project — a fact we leverage to protect the decryption key for the database elements.

We summarize the access control policies in Table 1.

5.3 Query Processing

A PIR query is comprised of three steps: query generation, response compution, and response extraction. The first and last elements are performed by the client, while responses are computed by the server. A high level view of the server side of this process is depicted in Figure 2.

Query generation. The private information retrieval scheme we adopted supports making queries by element index, e.g. *Obtain the fifth element in the database*. It does so by creating a $N \times 2N$ query matrix for every element in the database and crafting the one for the desired index (5 in this example) carefully to allow extraction later.

We developed a Ruby library that generates these query matrices according to the algorithm. As part of this development, we extended the Ruby Matrix class to support operations over a field Z/pZ (each value is computed modulo some value p).

Once the query matrices have been computed, the client li-

brary logs into Google Drive using the `google-drive-ruby` [9] library. This library wraps an HTTP API which can be used to retrieve data from and send data to a spreadsheet.

Once the *query sheet* is open, the request matrices are written into it. Request matrices are represented as a single row in the spreadsheet, where each row of the matrix is contained in a cell in the spreadsheet row. Matrix row values are represented as comma-separated base32 encoded strings. We split the matrix rows across multiple cells due to maximum length restrictions within a single cell. Once all query data has been written into the sheet, the client writes the request parameters to a request cell within the sheet. The values in this cell inform the server of the set of parameters the client used to create the query and by extension the parameters which should be used to encode the response. An example is shown in blue in Figure 3. Finally, the client inserts the *sentinel* value `Requested` is inserted into a special status cell within the sheet (red in Figure 3). This process corresponds to (a) in Figure 2.

Server response. Calculating the server response requires a number of sequential actions (b,c,d) in Figure 2. The first action is to notify the PIR code (contained within the 'PIR Library' script project) that a database request has been made and a response must be computed.

While Google Apps Scripts are written in Javascript, a fully functional programming language, they operate within a constrained execution environment. Most importantly there is an upper bound on the amount of time that a script is permitted to run, depending on the context in which it is executed. A custom function - called by writing `=callCustomFunction()` into a spreadsheet cell - can only execute for a few seconds. Scripts initiated from *triggers* are permitted to execute for five minutes. Since the response calculation requires more than several seconds, we use a *trigger* to begin response calculation.

In our implementation, a triggered function checks for the value `Requested` in the status cell each minute.[4] If found, then the status is changed to `Processing` and the first computation iteration is requested from the 'PIR Library' (*(c)* in Figure 2). Although the 'PIR Library' is associated with its own spreadsheet, its functions are capable of modifying the 'PIR DB' sheets when called from the 'PIR DB' interface.

The full response computation requires n iterations, one for each element in the database. If the database is large, n iterations may require more than the five minutes of execution time allowed for triggered scripts, so at the end of each iteration, the script compares the time taken by the last iteration to the time remaining in the execution window. If the window is smaller than twice the last iteration, the interface saves state in the query sheet and programatically schedules a new *trigger* for the next minute. This process is also contained in part (c) of Figure 2.

When the complete response (represented by a single $2 \times N$ element vector) is computed by the server, it is written into the query sheet and the *sentinel value* is adjusted to notify the client that computation is complete (Part (d) of Figure 2). The response vector is represented as comma-separated base32 encoded values.

[4]Triggers can be set to execute at specific times, but the lowest granularity is a minute. Additionally, execution occurs on a best effort basis and may not begin exactly at the requested time.

The server side implementation makes use of a native Javascript Big Integer Library [13] to enable matrix operations without loss of precision[5]. The server side implementation is also implemented as a Node.JS [10] module and supports extensible database and query `Readers`, which permit use outside of the confines of a Google Spreadsheet. We use this facility to compare the performance effects of the spreadsheet environment in Section 7.

Response extraction. After the client library submits the query to the server and requests computation of a response, it enters a sleep cycle during which it periodically checks if the computation is complete by querying the *sentinel* value (Part (e) of Figure 2). This periodic check increases the amount of time taken to retrieve a response, but is necessary given the stateless communication used in interacting with cloud services. The amount of time wasted while sleeping can be reduced by decreasing the length of the sleep, at the cost of increased communication overhead.

Once the response is available, the client retrieves it from the spreadsheet using the `google-drive-ruby` library and decodes the returned database element according to the protocol.

6. EVALUATION

In this section, we evaluate the correctness and efficiency of our Google Spreadsheet-based PIR platform. As a performance metric, we consider the *time-to-completion* of a query, measured from the time the query is submitted by the client to the time that a result is returned and processed.

6.1 System Performance

To determine how our choice of parameters affects the performance of the system, we modify parameters on three axes independently: the size of the database elements as controlled by the parameter l_0, the dimensions N of the query matrix, and the number of elements in the database. To explore each of these dimensions, we fix the parameters at $N = 50$, $l_0 = 8$, $n = 10$ and adjust the parameter of interest. For each set of parameters, we retrieve an element ten times, providing a distribution over the results. Our figures show mean values over the ten runs with error bars signifying a standard distribution. In all experimental runs, we verify that the PIR algorithm returns the correct result.

Varying element size. Figure 4 shows that there is a performance cost associated with increasing the element size via parameter l_0. This parameter determines the size in bits of the chunks each database element is split into, and therefore the size of the scalars involved in multiplication and modulo operations on the server (cloud) side. With an element size of 400 bits, our office-suite-based PIR implementation incurs a response latency of approximately three minutes. While this represents a significant investment of time for one query, the qualities that our public cloud-based service provides - namely robustness, symmetric privacy, and unremarkable use patterns - may make this an attractive approach for those who are highly concerned with their privacy.

Since Javascript has no native support for large integers, we use a large precision integer library [13] that encodes

[5]Javascript represents all numbers as doubles, which are unable to represent integers larger than 22 bits exactly.

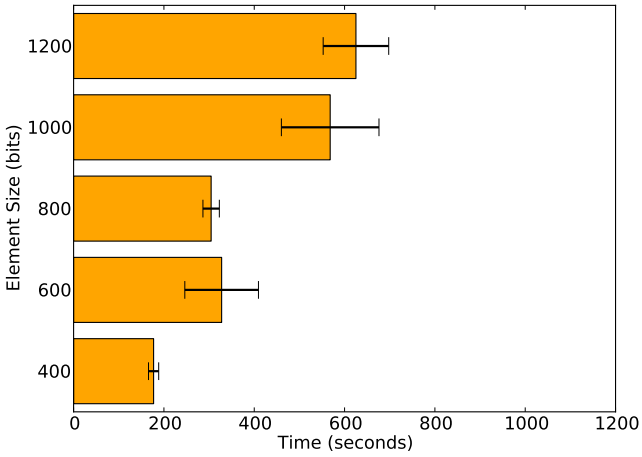

Figure 4: Varied database element size by adjusting l_0.

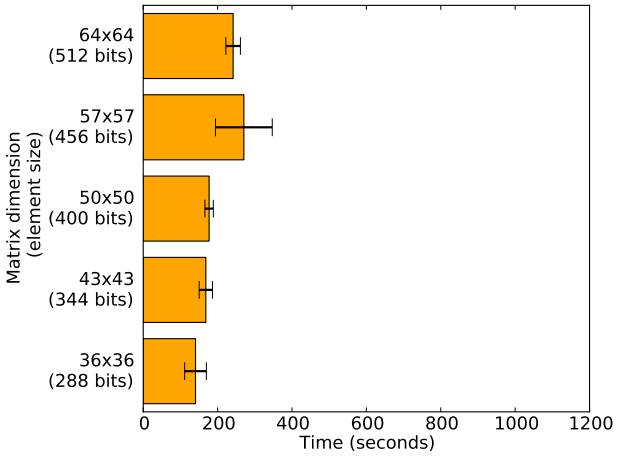

Figure 5: Varied PIR matrix size by adjusting N.

arbitrarily long integers using arrays of Javascript integers. This behavior is unlikely to have optimized support in the Javascript interpreter, so it is unsurprising that as this parameter grows, so does the time-to-completion of the PIR queries.

Trading off security and performance. Figure 5 shows the effect of changing the dimensions N of the matrix used to encode the query. Under the private information retrieval scheme used, the effort required to break the security of the scheme is approximately 2^{2N}, so adjusting this parameter provides a tradeoff between the security (privacy) offered by the PIR algorithm and the system's performance.

Adjusting this parameter also modifies the size of the database elements slightly - element size is bounded by $N \times l_0$ and we fix l_0 in this case. Unfortunately, we see little performance benefit to reducing computational complexity: element retrieval with $N = 36$ requires 140 seconds, 17% less time than the baseline parameters but with significantly reduced complexity.

Performance as a function of database size. We additionally evaluate time-to-completion as a function of the number of elements in the database. We observe from Figure 6 that the effect of increasing the database size is nearly linear. This is unsurprising – the private information retrieval algorithm requires applying the same operation to each element in the database. However, the high cost of retrieval (19.51 minutes for an element from a 50 element database) suggests that our PIR implementation is best suited for use with very small databases.

Given the above performance results, we discuss the feasibility and appropriateness of leveraging web-based office suites as PIR backends in greater detail in the next section.

6.2 Microbenchmarks

We find that a significant drawback of using Google Spreadsheet is that the Google Spreadsheet script engine results in poor performance: the baseline parameters, which permit a 10 element database with 400 bits per element, yield a retrieval time of 177 seconds.

We explore this performance penalty by executing the same queries (against the same database) using both Google

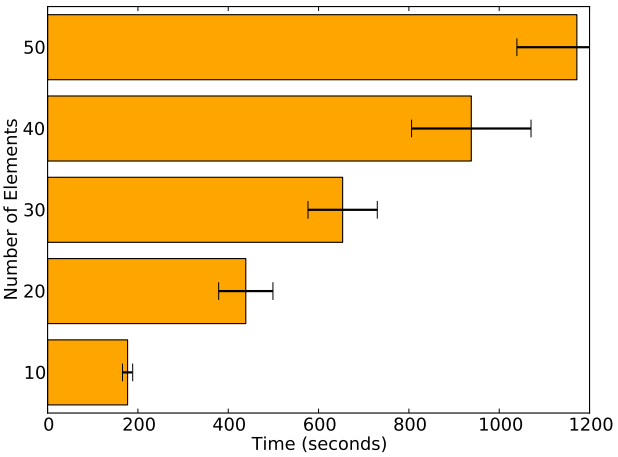

Figure 6: Varied number of database elements by adjusting n.

Spreadsheets as well as a locally maintained database backend. In both cases, the codebases used for both client and servers are the same; the server-side Javascript is executed as a Node.JS module.

There are several differences when operating locally: the database and query matrices are read from a file instead of from spreadsheet elements, transmission of the query is not required, and there is no need for a trigger cycle to request computation and announce its completion. We next consider how these may affect performance where appropriate.

Figure 7 shows the time required to perform several components of the PIR algorithm locally versus in a Google Spreadsheet. *Query generation* occurs locally in both cases and thus unsurprisingly shows no difference.

In contrast, *query transmission* incurs a significant latency for the cloud-based implementation. (No transmission is required for the local implementation.) Although our Google Spreadsheet-based implementation requires nearly 20 seconds, query transmission is only a small fraction (7%) of the total time required.

A significant fraction of the overall time-to-completion on the cloud implementation is due to the necessities of the trig-

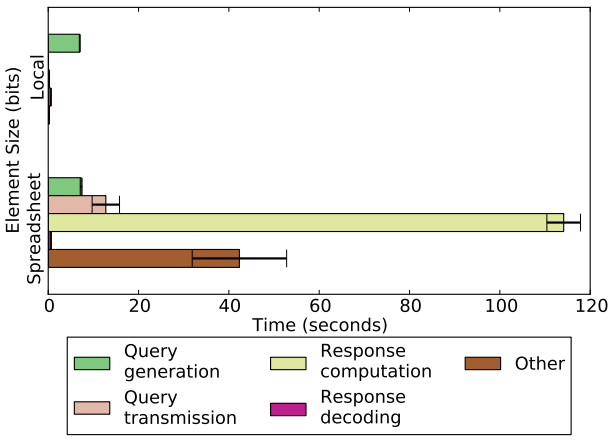

Figure 7: Comparison of time taken to complete portions of the PIR algorithm locally and on Google Apps.

ger cycle (to begin computation on the server side) and the sleep cycle (to check for a computed response). The *Other* bar in Figure 7 captures the time spent in these operations, and confirms our suspicions: nearly 24% of the time is spent waiting for an action to occur. This is an unfortunate but unavoidable result given the execution constraints imposed in the Google Spreadsheet environment. We remark that these costs could be significantly reduced, for example, if the spreadsheet were to include additional support for asynchronous event notification.

Figure 7 shows that the lions share of the difference occurs during the process of actually computing the response. The magnitude of the difference in response computation time is surprising, with a 400-fold increase when operating on Google Spreadsheets. This is somewhat surprising given that both cases share the same code.

One potential cause may be that Google splits Apps Scripts into functional chunks and maps them to compute servers which have spare cycles using a map-reduce framework. Additionally, since triggers are usually background tasks and may be queued with low priority, this could contribute to the system's slowness. Since the architectural details of Google Apps' backend is not currently publicly disclosed, we leave a more fine-grain exploration of Google App's computation strategy as a future research direction.

7. DISCUSSION AND POTENTIAL ENHANCEMENTS

Our evaluation demonstrates the feasibility of implementing PIR databases using only existing online office suites. Although inefficient, our implementation enables any client that can connect to Google's web applications to privately retrieve cells from a Spreadsheet document. In this section, we describe extensions to our proof-of-concept implementation that provide additional privacy properties (Section 7.1), add support for multiple simultaneous clients (Section 7.2), enhance error reporting (Section 7.3), and suggest use cases that leverage the limited search capability of PIR (Section 7.4).

7.1 Support for Symmetric PIR

As discussed in Section 3, web-based office software inherently performs trivial private information retrieval because the client browser downloads all data before presenting it to the viewer. To make cloud-based PIR interesting, we consider its ability to support a *symmetric* private information retrieval scheme in which clients are additionally precluded from viewing all the data in the spreadsheet.

Google Spreadsheet allows the *database account* (the Google account that owns the spreadsheet) to *hide* sheets within the spreadsheet. This prevents others users with whom the spreadsheet has been shared from reading it. Currently, such users can learn of the sheet's existence through Google's interface.

At first blush, this would appear to enable symmetric PIR since the user cannot directly access the data. While developing our system, we discovered however that hidden values can be accessed by specifying spreadsheet formulas. That is, a user who has read-only access to a spreadsheet containing a hidden sheet can access the hidden data by constructing a new spreadsheet, and then creating references to cells on the shared spreadsheet's hidden sheet. In our implementation, the query client has access to the query sheet, and could potentially use that access to reference specific cells in the hidden sheet.

To thwart this method of information leakage and enable symmetric PIR, we encrypt all database elements (cells) in the spreadsheet using AES in *output feedback (OFB)* mode with a randomly generated key. We store the private key in the *spreadsheet script properties* for the PIR library spreadsheet. This key-value store that Google provides for spreadsheet scripts is accessible to the database account (i.e., the user who owns the spreadsheet containing the data) but not to the querier. The PIR library decrypts elements on the fly in the process of encoding them into the response, but without storing the plaintext anywhere visible to the querier.

In summary, we observe a potential weakness in Google's access control enforcement mechanism and note that the querier can exploit that weakness to discover the values of cells in the spreadsheet. To provide symmetric PIR and prevent the client from accessing data cells beyond the one requested in the PIR query, we store the data in encrypted form and secure the key material using spreadsheet script properties, over which access controls are properly enforced.

To provide *useful* symmetric PIR, the PIR scheme should be coupled with access control and rate-limiting mechanisms. Rate-limiting and access control are essential since otherwise a user can simply make repeated queries to enumerate all of the elements in the database. Although we do not implement these features in our proof-of-concept prototype, we briefly outline such a construction: As discussed above, Google provides its own access control mechanism for Google documents. Allowing only specified Google accounts to access the spreadsheet can be accomplished by granting those users write-access to the query sheet and read-access to the database sheet, and denying all other accesses. Rate-limiting can be straightforwardly achieved by amending our scripts to store access records with timestamps in a "log" sheet within the spreadsheet document, and verifying that rate-limits have not been exceeded by counting the number of records belonging to the user within some threshold timespan. Triggers could then be added to periodically prune unnecessary entries from the log.

7.2 Supporting Concurrency

The stateless communication paradigms supported by Google Drive, combined with the shared nature of Google Spreadsheets increase the challenge of supporting multiple concurrent clients. Where traditional server implementations devote an entire server thread or process to communication with a client, private information retrieval requests to our Google Spreadsheets implementation communicate with a shared interface (the spreadsheet itself), and have the potential to be interleaved with other concurrent requests.

While this presents a potential challenge, certain aspects of this communication paradigm help lead to a potential solution. In particular, the fact that each client has the same view of ongoing queries means that clients can perform 'traffic control' to ensure that they do not interrupt or overwrite ongoing requests. We suggest that a production implementation would contain a *clearing house* sheet in which requests could be queued until sufficient room in data query sheets became available.

7.3 Enhancing Error Feedback

As described in Section 5.3, our implementation uses a time-based polling mechanism to obtain the computation response. A lack of a response indicates either that the computation is still ongoing or that an error has occurred (e.g., due to timeouts during data processing). Google Apps Script provides no immediate[6] mechanism for distinguishing between the two cases, making it difficult for the client to decide whether to abort and reissue the query or to continue to wait for the results. As a result, the lack of feedback translates directly into a usability concern; users may be forced to interrupt queries they believe have failed and incur the costs of processing multiple times.

We foresee two approaches that would alleviate this issue. First, the server side implementation could return partial results directly to the client as they became available at the cost of increased client-server communication for coordination purposes. Since the maximum expected execution time for each partial result is relatively small, these are unlikely to fail due to limits placed on execution time. The private information retrieval scheme proposed by Melchor and Gaborit conveniently supports returning partial results as they become available. A second approach would make use of a second monitoring script which could abort the PIR process if insufficient progress is maintained.

Of these approaches, the former is both simpler because it does not require any ability to watch or halt the execution of another script and more efficient because the client can resume calculation from an intermediate checkpoint if failures occur. As such we suggest this is the best approach for alleviating the general lack of feedback provided.

7.4 Leveraging Index-based Search

One frequent major drawback of private information retrieval schemes, including the Melchor-Gaborit scheme that we use in our implementation, is that private information retrieval requests are made by the database element number, but not by more complex attributes. Essentially, the search capability provided by PIR is limited to index-based queries, which precludes the support for other richer search semantics such as by the (in)equality of attributes. From an information theoretic point of view, this is understandable, especially for symmetric PIR, considering that the client would have to hide a more complex search criteria (which entails more information for its encoding) within the query, which is considerably difficult without non-trivial assumptions on the application scenario.

While the search capability supported by our cloud-based symmetric private information retrieval scheme limits the scope of applications for which it is practical to a set smaller than that expected of a general purpose database, this limited capability is not specific to our implementation.

In fact, we suggest that despite this limitation, there are plausible use cases for such a private information retrieval implementation. Consider a case where clients are interested in requesting a *random* element from the database but the database has an interest in limiting the release of information. This situation is not as implausible as it might sound: imagine external auditors responsible for examining financial values sampled from a company's accounting books. The auditors may wish to retrieve values at random without the company being able to know which values they are examining. On the other hand, the company may wish to maintain the secrecy of their accounting data *in aggregate* and so desire that the auditors only receive the values they request. If the accounting database is hosted within a cloud service by a third party, then a PIR implementation such as ours could be an effective enabling tool.

8. CONCLUSION

In this paper we present a novel private information retrieval (PIR) implementation that exploits a web-based office suite as the database backend. We suggest that there are several major advantages to this scheme, including the reliability, robustness and rigorous access control provided by non-specialized, state-of-the-art server technology, and the fact that platform (web-based office suite) is already widely recognized and for free. We identify the technical challenges for deploying a PIR service on public cloud service, and present a prototype implementation based on Google Spreadsheet that addresses these challenges. Our implementation permits clients to request database elements by index using the PIR scheme proposed by Melchor and Gaborit. While there remain a number of challenges, including low performance, our experimental evaluation validates the feasibility of deploying PIR on public cloud.

This is the first paper of which we are aware that demonstrates PIR schemes that can operate within commodity cloud environments. In a mature form, private information retrieval systems such as this could be used to support a variety of private data lookups.

[6]Digests of errors are sent to the script owner daily.

References

[1] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally Private Information Retrieval with Polylogarithmic Communication. In *Advances in Cryptology (EUROCRYPT)*, 1999.

[2] Yan-Cheng Chang. Single Database Private Information Retrieval with Logarithmic Communication. In *Information Security and Privacy*, 2004.

[3] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private Information Retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[4] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium (USENIX)*, 2004.

[5] Evin Levey. Apps Script Gallery for Google. http://googledocs.blogspot.com/2010/03/apps-script-gallery-for-google.html, 2013.

[6] Gabriel Ghinita, Panos Kalnis, Ali Khoshgozaran, Cyrus Shahabi, and Kian-Lee Tan. Private Queries in Location Based Services: Anonymizers are Not Necessary. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2008.

[7] Thomas A. Grossman. Spreadsheet Engineering: A Research Framework. In *European Spreadsheet Risks Interest Group Symposium*, 2002. Available at http://arxiv.org/pdf/0711.0538.pdf.

[8] Ryan Henry, Femi Olumofin, and Ian Goldberg. Practical PIR for Electronic Commerce. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[9] Hiroshi Ichikawa. google-drive-ruby. https://github.com/gimite/google-drive-ruby, 2013.

[10] Joyent, Inc. NodeJS. http://nodejs.org/, 2013.

[11] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not Needed: Single Database, Computationally-Private Information Retrieval. In *Foundations of Computer Science (FOCS)*.

[12] Eyal Kushilevitz and Rafail Ostrovsky. One-way Trapdoor Permutations Are Sufficient for Non-Trivial Single-Server Private Information Retrieval. In *Advances in Cryptology (EUROCRYPT)*, 2000.

[13] Leemon Baird. Javascript Big Integer Library v5.5. http://www.leemon.com/crypto/BigInt.js, 2013.

[14] Philipp Lenssen. *Google Apps Hacks*. O'Reilly Media, 2008.

[15] Laura Beth Lincoln. *Symmetric Private Information Retrieval*. PhD thesis, Rochester Institute of Technology, 2006.

[16] Yanbin Lu and Gene Tsudik. Towards Plugging Privacy Leaks in the Domain Name System. In *Peer-to-Peer Computing (P2P)*, 2010.

[17] Carlos Aquilar Melchor and Philippe Gaborit. A Lattice-based Computationally-efficient Private Information Retrieval Protocol. In *Western European Workshop on Research in Cryptology*, 2007.

[18] Carlos Aquilar Melchor and Philippe Gaborit. A Fast Private Information Retrieval Protocol. In *International Symposium on Information Theory (ISIT)*, 2008.

[19] Prateek Mittal, Femi Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. PIR-Tor: Scalable Anonymous Communication using Private Information Retrieval. In *USENIX Security Symposium (USENIX)*, 2011.

[20] Femi Olumofin and Ian Goldberg. Revisiting the Computational Practicality of Private Information Retrieval. In *Financial Cryptography and Data Security (FC)*, 2012.

[21] Radu Sion. On the Computational Practicality of Private Information Retrieval. In *Network and Distributed Systems Security Symposium (NDSS)*, 2007.

[22] Paul F. Syverson, David M. Goldschlag, and Michael G. Reed. Anonymous Connections and Onion Routing. In *IEEE Symposium on Security and Privacy (Oakland)*, 1997.