Splinter: Practical Private Queries on Public Data

by

Catherine Yun

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

Author .....
Department of Electrical Engineering and Computer Science
May 12, 2017

Certified by .....
Vinod Vaikuntanathan
Associate Professor, EECS
Thesis Supervisor

Accepted by .....
Dr. Christopher Terman
Chairman, Department Committee on Graduate Theses

Splinter: Practical Private Queries on Public Data

by

Catherine Yun

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Every day, millions of people rely on third party services like Google Maps to navigate from A to B. With existing technology, each query provides Google and their affiliates with a track record of where they've been and where they're going. In this thesis, I design, engineer, and implement a solution that offers absolute privacy when making routing queries, through the application of the Function Secret Sharing (FSS) cryptographic primitive.

I worked on a library in Golang that applied an optimized FSS protocol, and exposed an API to generate and evaluate different kinds of queries. I then built a system with servers that handle queries to the database, and clients that generate queries. I used DIMACS maps data and the Transit Node Routing (TNR) algorithm to obtain graph data hosted by the servers. Finally, I evaluated the performance of my system for practicality, and compared it to existing private map routing systems.

Thesis Supervisor: Vinod Vaikuntanathan
Title: Associate Professor, EECS

Acknowledgments

I'd like to thank Frank for being incredibly supportive throughout my Master's, and for providing priceless life advice along the way. Vinod, Matei, and Shafi have also been wonderful collaborators, and a pleasure to work with.

A special thanks to Eric for countless hours of system design, coding, and debugging help, as well as endless moral support and inspiration. I wouldn't be where I am without you.

A shoutout to the folks at the Digital Currency Initiative at the Media Lab, especially Jeremy and Neha, for helping me think through interesting applications of function secret sharing to Bitcoin privacy.

Many thanks to my family for always believing in me and putting up with my shenanigans, the MIT climbing team for keeping me sane, and my friends for making sure I didn't forget how to have fun.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivations for Private Querying

Many online services let users query large public datasets: some examples include restaurant sites, product catalogs, stock quotes, and searching for directions on maps. In these services, any user can query the data, and the datasets themselves are not sensitive. However, web services can infer a great deal of identifiable and sensitive user information from these queries, such as their current location, political affiliation, sexual orientation, income, etc. [35, 34]. This information can be used maliciously or put users at risk of being targeted by practices such as discriminatory pricing [49, 45, 23]. For example, online stores have charged users different prices based on location [25], and travel sites have also increased prices for certain frequently searched flights [46]. Even when the services are well-intentioned, server compromises and subpoenas can expose the sensitive user information these web services store [43, 27, 42].

We can eliminate the concerns that stem from revealing query information by building and using a system that can efficiently execute queries which are private to all intermediate parties, including the database itself.

1.2 Potential Applications of Private Querying

There are two main categories of applications that benefit greatly from private querying - applications where the queries reveal sensitive user data, such as locations or a user account numbers that are supposed to be secret, and applications where the queries reveal user behavior patterns.

In applications where the queries reveal sensitive user data, building clones of location-based services such as Yelp and Google Maps with private queries could be beneficial to people who don't want the services they use to know their location data, but still want to be able to use those services. In a similar vein, users of Bitcoin want their account identifiers and the transactions pertaining to them to be anonymous. However, when users fetch their balance from a Simplified Payment Verification (SPV) node or check the blockchain depth of a transaction, the node they query gains information about their account identifiers and the transactions they are interested in. This is another case where private queries could help mask sensitive information in queries, preserving the anonymity provided by Bitcoin.

In applications where the queries reveal user behavior patterns, building a flight querying service with private queries would allow users to do searches for the flights they are interested in, without worrying about travel sites taking advantage of that information to increase prices for those flights.

Chapter 2

Function Secret Sharing applied to private database queries

2.1 Overview of Function Secret Sharing

In this section, we give an overview of Function Secret Sharing (FSS), the main primitive used in Splinter [16], and show how to use it in simple queries.

2.1.1 Function Secret Sharing Primitive

Function Secret Sharing [10] lets a client divide a function f into *function shares* $f_1, f_2, \ldots, f_k$ so that multiple parties can help evaluate f without learning certain of its parameters. These shares have the following properties:

- They are close in size to a description of f.
- They can be evaluated quickly (similar in time to f).
- They sum to the original function f. That is, for any input x, $\sum_{i=1}^{k} f_i(x) = f(x)$.
- Given any $k-1$ shares f_i, an adversary cannot recover the parameters of f.

Although it is possible to perform FSS for arbitrary functions, practical FSS protocols only exist for *point* and *interval* functions. These take the following forms:

- Point functions f_a are defined as $f_a(x) = 1$ if $x = a$ or 0 otherwise.

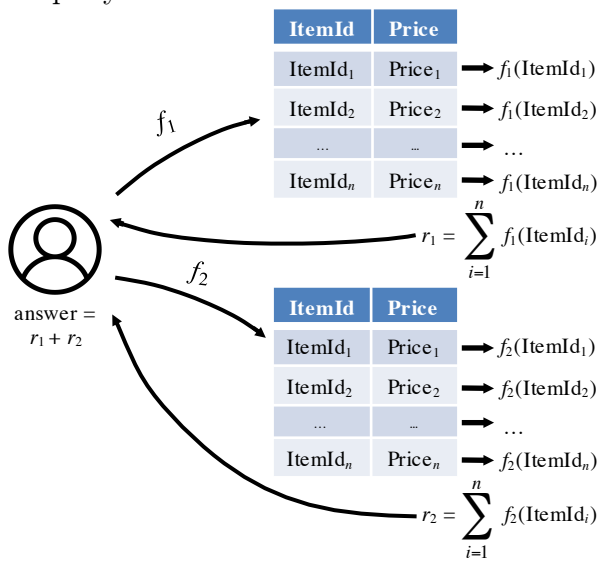
- Interval functions are defined as $f_{a,b}(x) = 1$ if $a \leq x \leq b$ or 0 otherwise.

In both cases, FSS keeps the parameters a and b private: an adversary can tell that it was given a share of a point or interval function, but cannot find a and b. In Splinter, we use the FSS scheme of Boyle et al. [10]. The functions f_a and $f_{a,b}$ operate over Z_{2^m} (integers mod 2^m) where m is the number of bits in the output range. For the rest of the paper, we will assume that all computations are done over Z_{2^m}. Under this scheme, the shares f_i for both functions require $O(\lambda n)$ bits to describe and $O(\lambda n)$ bit operations to evaluate for a security parameter λ (the size of cryptographic keys) where n is the number of bits in the input domain. This contrasts to $O(n)$ bits and operations to describe and evaluate the original functions.

2.1.2 Function Secret Sharing for Database Queries

We can use the additive nature of FSS shares to run private queries over an entire table in addition to a single data record. We illustrate here with two examples.

Figure 2-1: Overview of how FSS can be applied to database records on two providers to perform a COUNT query.



Example: COUNT query. Suppose that the user wants to run the following query on a table served by Splinter:

```
SELECT COUNT(*) FROM items WHERE ItemId = ?
```

Here, '`?`' denotes a parameter that the user would like to keep private; for example, suppose the user is searching for `ItemId = 5`, but does not want to reveal this value.

To run this query, the Splinter client defines a point function $f(x) = 1$ if $x = 5$ or 0 otherwise. It then divides this function into function shares $f_1, \ldots, f_n$ and distributes them to the providers, as shown in Figure 2-1. For simplicity, suppose that there are two providers, who receive shares f_1 and f_2.

Because these shares are additive, we know that $f_1(x) + f_2(x) = f(x)$ for every input x. Thus, each provider p can compute $f_p(\text{ItemId})$ for every ItemId in the database table, and send back $r_p = \sum_{i=1}^{n} f_p(\text{ItemId}_i)$ to the client. The client then computes $r_1 + r_2$, which is equal to $\sum_{i=1}^{n} f(\text{ItemId}_i)$, that is, the count of all matching records in the table.

ItemId	**Price**	f_1(ItemId)	f_2(ItemId)
5	8	10	-9
1	8	3	-3
5	9	10	-9

Figure 2-2: Simple example table with outputs for the FSS function shares f_1, f_2 applied to the ItemId column. The function is a point function that returns 1 if the input is 5, and 0 otherwise. All outputs are integers modulo 2^m for some m.

To make this more concrete, Figure 2-2 shows an example table and some sample outputs of the function shares, f_1 and f_2, applied to the ItemId column. There are a few important observations. First, to each provider, the outputs of their function share seem random. Consequently, the provider does not learn the original function f and the parameter "5". Second, because f evaluates to 1 on inputs of 5, $f_1(\text{ItemId}) + f_2(\text{ItemId}) = 1$ for rows 1 and 3. Similarly, $f_1(\text{ItemId}) + f_2(\text{ItemId}) = 0$ for row 2. Therefore, when summed across the providers, each row contributes 1 (if it matches) or 0 (if it does not match) to the final result. Finally, each provider aggregates the

outputs of their shares by summing them. In the example, one provider returns 23 to the client, and the other returns -21. The sum of these is the correct query output, 2.

This additivity of FSS enables Splinter to have *low communication costs* for aggregate queries, by aggregating data locally on each provider.

Example: SUM query. Suppose that instead of a COUNT, we wanted to run the following SUM query:

```
SELECT SUM(Price) FROM items WHERE ItemId=?
```

This query can be executed privately with a small extension to the COUNT scheme. As in COUNT, we define a point function f for our secret predicate, e.g., $f(x) = 1$ if $x = 5$ and 0 otherwise. We divide this function into shares f_1 and f_2. However, instead of computing $r_p = \sum_{i=1}^{n} f_p(\text{ItemId}_i)$, each provider p computes

$$r_p = \sum_{i=1}^{n} f_p(\text{ItemId}_i) \cdot \text{Price}_i$$

As before, $r_1 + r_2$ is the correct answer of the query, that is, $\sum_{i=1}^{n} f(\text{ItemId}_i) \cdot \text{Price}_i$. We add in each row's price, Price_i, 0 times if its ItemId is not 5, and 1 time if it is 5.

2.2 Implementation: Splinter

2.2.1 Splinter Architecture

There are two main principals in Splinter: the *user* and the *providers*. Each provider hosts a copy of the data. Providers can retrieve this data from a public repository or mirror site. For example, OpenStreetMap [38] hosts publicly available map, point-of-interest, and traffic data. Data owners can also charge providers to access this data. For a given user query, all the providers have to run it on the same view of the data. Maintaining data consistency from mirror sites is beyond the scope of this paper, but standard techniques can be used [50, 11].

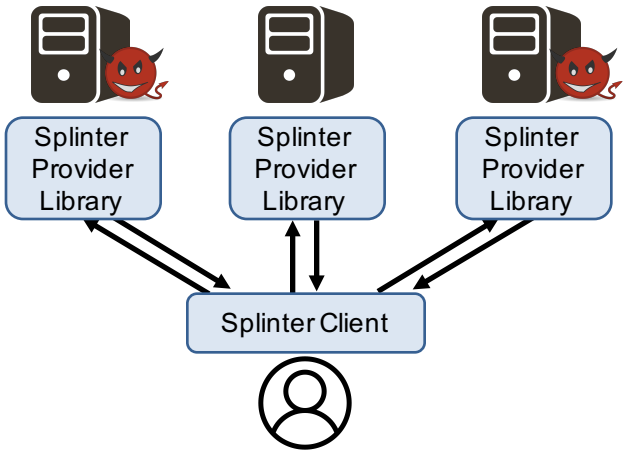


Figure 2-3: Splinter architecture. The Splinter client splits each user query into shares and sends them to multiple providers. It then combines their results to obtain the final answer. The user's query remains private as long as any one provider is honest.

As shown in Figure 2-3, to issue a query in Splinter, a user splits their query into *shares*, using the Splinter client, and submits each share to a different provider. The user can select any providers of their choice that host the dataset. The providers execute their shares to execute the user's query over the cleartext public data, using the Splinter provider library. As long as one provider is *honest* (does not collude with others), the user's sensitive information in the original query remains private. When the user receives the responses from the providers, the user combines the responses to obtain the final answer to their original query.

2.2.2 Splinter Query Model

Beyond the simple SUM and COUNT queries in the previous section, we have developed protocols to execute a large class of queries using FSS, including non-additive aggregates such as MAX and MIN, and queries that return multiple individual records instead of an aggregate. For all these queries, our protocols are efficient in both computation and communication. On a database of n records, all queries can be executed in $O(n \log n)$ time and $O(\log n)$ communication rounds, and most only require 1 or 2 communication rounds.

Figure 2-4 describes Splinter's supported queries using SQL syntax. Most op-

Query format:

SELECT $aggregate_1$, $aggregate_2$, ...
FROM *table*
WHERE *condition*
[GROUP BY $expr_1$, $expr_2$, ...]

aggregate:

- COUNT | SUM | AVG | STDEV (*expr*)
- MAX | MIN (*expr*)
- TOPK (*expr*, *k*, *sort_ expr*)
- HISTOGRAM (*expr*, *bins*)

condition:

- *expr* = *secret*
- $secret_1 \leq expr \leq secret_2$
- AND of '=' conditions and up to one interval
- OR of multiple disjoint conditions
  (e.g., `country="UK" OR country="USA"`)

expr: any public function of the fields in a table row
(e.g., `ItemId + 1` or `Price * Tax`)

Figure 2-4: Splinter query format. The TOPK aggregate returns the top k values of *expr* for matching rows in the query, sorting them by *sort_ expr*. In conditions, the parameters labeled *secret* are hidden from the providers.

erators are self-explanatory. The only exception is TOPK, which is used to return up to k individual records matching a predicate, sorting them by some expression *sort_expr*. This operator can be used to implement `SELECT...LIMIT` queries, but we show it as a single "aggregate" to simplify our exposition. To keep the number of matching records hidden from providers, the operator always pads its result to exactly k records.

Although Splinter does not support all of SQL, we found it expressive enough to support many real-world query services over public data. Finally, Splinter only natively supports fixed-width integer data types. However, such integers can also be used to encode strings and fixed-precision floating point numbers (e.g., SQL DECIMALs). We use them to represent other types of data in our sample applications.

2.2.3 Executing Splinter Queries

Given a query in Splinter's query format (Figure 2-4), the system executes it using the following steps:

1. The Splinter client builds function shares for the condition in the query.

2. The clients sends the query with all the secret parameters removed to each provider, along with that provider's share of the condition function.

3. If the query has a GROUP BY, each provider divides its data into groups using the grouping expressions; otherwise, it treats the whole table as one group.

4. For each group and each aggregate in the query, the provider runs an evaluation protocol that depends on the aggregate function and on properties of the condition. Some of the protocols require further communication with the client, in which case the provider batches its communication for all grouping keys together.

Chapter 3

Privacy-Preserving Map Routing

3.1 Overview

Implementing map routing in Splinter is challenging because the provider can only perform table lookups. Storing all possible shortest paths requires substantial amounts of storage. For example, storing all shortest paths for New York City requires 2^{36} rows! The user could download the map and find the shortest path locally, which is bandwidth heavy, and the user would also have to download updates constantly.

Extensive work has been done to optimize map routing [5]. One algorithm compatible with Splinter is transit node routing (TNR) [4, 7], which has been shown to work well in practice [6]. In TNR, the provider divides up a map into grids, which contain at least one transit node, i.e. a node that is part of a "fast" path. There is also a separate table that has the shortest paths between all pairs of transit nodes, which represent a smaller subset of the map. For a given source node src and target node trg, a client finds the shortest path between the two by first fetching the sets of closest transit nodes T_{src} and T_{trg}. Then, for each pair of nodes $t_{src} \in T_{src}$ and $t_{trg} \in T_{trg}$, the client finds the path connecting src to trg via t_{src} and t_{trg}. The shortest of these paths is the shortest path between src and trg.

In our use case, all the paths between the pairs of transit nodes are pre-computed, so the user would perform a select query for each pair of $\{t_{src}, t_{trg}\}$. They would also perform a select query for the zone information for src and trg, so that they

can locally run a shortest path computation from *src* to *trg* through the transit node shortest paths.

3.2 Implementation

I implemented the map routing application in Golang, using the Golang crypto package for AES-NI hardware instructions for AES encryption. I implemented Splinter in Golang, not only because of familiarity and usability, but also because of its excellent cryptographic libraries.

The FSS library is about 1000 lines of code, and the applications on top of it are about 1000 lines of code. The implementation of the library, client, and server can be found at `https://github.com/cathieyun/libfss`.

3.2.1 The Server

I implemented a FSS server module that accepts queries for two-party FSS, though my implementation could easily be extended to multi-party FSS as well. In two-party FSS, there are two server instances that receive and process function secret shares.

The server expects input in the form of a GET request that includes the query type, FSS key, pseudorandom function (PRF) keys, and the number of bits to match on. It initializes an FSS server with the PRF keys and the number of bits to match on, by calling the FSS library. Then it parses the query using different techniques, depending on the query type:

1. COUNT: Integer matching

   This is the simplest query type. The server scans through the corresponding file line by line, getting the *key* (the integer to count) for each line. It evaluates the FSS point function over *key* using the FSS library functions, and adds that quantity to *answer*.

2. SELECT: Integer matching, with a small integer return value.

   The server scans through the corresponding file line by line, getting the *key* (the

integer to match) and *value* (the integer to return) for each line. It evaluates the FSS point function over *key* using the FSS library functions, multiplies the result by *value*, and adds that quantity to *answer*.

An example of this kind of query is the retrieval of what zone a node is in. The node id is an integer, and the zone id is an integer.

3. SELECT: Integer matching, with a large return value.
   The server scans through the corresponding file line by line, getting the *key* (the integer to match) and *value* (a large integer, or a non-integer) for each line. It evaluates the FSS point function over *key* using the FSS library functions, multiplies each byte value in the *value* byte array by that quantity, and adds the multiplied array to the *answer* array.

   An example of this kind of query is the retrieval of the zone data for a transit node zone. The *key* is the transit node id, which is an integer, and the *value* is the information for that zone, in the form of nodes and edges.

4. SELECT: Large input matching, with a large return value.
   The server scans through the corresponding file line by line, getting the *key* (multiple integers, or a non-integer) and *value* (a large integer, multiple integers, or a non-integer) for each line. It converts *key* into an integer using the sha256 hash function, evaluates the FSS point function over the hash output, multiplies each byte value in the *value* byte array by that quantity, and adds the multiplied array to the *answer* array.

   An example of this kind of query is the retrieval of the shortest path between two transit nodes. The *key* is two integers, the ordered transit node pair, that needs to be treated as one integer for the purpose of FSS evaluation. The *value* is the shortest path between the two transit nodes, which is a large return value.

After iterating through the entire table and applying an FSS evaluation to the *key* in

every line, the server now has *answer*, which will be either an integer or an integer array. The server will return *answer*, which represents its answer share, to the client.

3.2.2 The Client

The client takes a query over a database, and generates function secret shares to send to the server. The client does this by first determining the query type and the number of bits to match on. It initializes an FSS client with the number of bits to match on, and uses the FSS client to generate either two sets of point function keys, or two sets of interval keys.

Point function keys are used when the query is an equality query (such as a *select* or *count*). These keys represent two parts of a function that, when combined, evaluate to b when $key = a$, and evaluate to 0 otherwise. For the *count* query, $b = 1$ because the server actions should, when combined, increment *answer* by 1 every time $key = a$. For the *select* query, $b = 1$ because the server actions should, when combined, give us the corresponding *value* when $key = a$. We assume for *select* queries that only one *key* will match a. If this is not the case, and if the *key* always maps to the same *value*, one could simply issue a *count* query to figure out how many matches there are, and divide *answer* by that quantity.

Interval keys are used when the query is a comparison query (such as $>$ or $<$). These keys represent two parts of a function that, when combined, evaluate to b when $key < a$ or $key > a$ (depending on the comparison), and 0 otherwise. Interval queries are generally used with *count*, to figure out the number of values greater or less than a. Therefore, $b = 1$ because the server actions should, when combined, increment *answer* by 1 every time $key < a$ or $key > a$.

After the client generates two sets of keys, it sends one set to each server, along with the query type, the corresponding pseudorandom function (PRF) keys, and the number of bits to match on. It then waits for each server to send back an answer share. Depending on the query type, the answer share will either be an integer or integer array. The client adds the two answer shares together to determine the final answer, and returns the final answer.

3.2.3 Query Parsing

A query to the routing server is a request for directions from point src to point trg. The way this query is expressed as function secret shares depends on the proximity of src and trg.

If src and trg are in the same region, such as both within New York, then the client will use the regional routing protocol, which takes five FSS queries:

1. Fetching the zone id for src

   To figure out what zone src is in, we will construct a function of query type 2 (see Section 3.2.1), matching on the integer value of src, and run it over the *node data file*.

2. Fetching zone data for src

   To get the node data, edge data, and transit node data for the zone that src is in, we will construct a function of query type 3 (see Section 3.2.1), matching on the integer value of the zone ID returned in step 1, and run it over the *zone data file*.

3. Fetching the zone id for trg

   To figure out what zone trg is in, we will construct a function of query type 2 (see Section 3.2.1), matching on the integer value of trg, and run it over the *node data file*.

4. Fetching zone data for trg

   To get the node data, edge data, and transit node data for the zone that trg is in, we will construct a function of query type 3 (see Section 3.2.1), matching on the integer value of the zone ID returned in step 1, and run it over the *zone data file*.

5. Fetching the shortest path from T_{src} to T_{trg}

   To get the shortest path from T_{src} to T_{trg}, we will construct a function of query type 4 (see Section 3.2.1), matching on the combination of T_{src} and T_{trg}, and run it over the *shortest paths data file*.

If *src* and *trg* are not in the same region, such as in a cross-country query, then I use more advanced techniques as described in Section 3.3.

3.2.4 Data Generator

The data generator takes node and edge data from the DIMACS [13] data set and generates transit nodes using the Transit Node Routing (TNR) [4] algorithm, classifies nodes into zones, generates zone data, and generates shortest path data between transit nodes.

For example, for my inter-region query example, I used a real traffic map data set from the Center for Discrete Mathematics and Theoretical Computer Science [13] for New York City, which consisted of $264,346$ nodes and $733,846$ arcs. I modified the source code from the Transit Node Routing paper [4] to output transit nodes. As recommended in the paper, I generated 500 transit nodes, which was roughly the square root of the number of nodes in the data set.

I divided the map into zones based on the proximity to transit nodes. I then generated zone data, which includes transit node data, node data, and edge data, for each of those zones. I saved the mapping from zones to zone data as the *zone data file*. I also generated mappings from nodes to zones they belonged to, and saved that as the *node data file*.

I then calculated the shortest path between all transit node pairs. The transit node table for New York has about 250,000 rows. I saved the mapping from ordered transit node pairs to their shortest path as the *shortest paths data file*.

3.3 Optimization

Because the data set for the entire United States is significantly larger than that of New York [13], it also requires more transit nodes for the TNR algorithm to work. If we were to use the TNR approach [4], we would need 5 thousand transit nodes and a table of 25 million pair-wise shortest paths for the entire United States. Since the longest of those pair-wise shortest paths would be a path going across the country,

and the size of the *answer* array has to be at least as long as the number of bytes in the longest path, running an FSS query across this database would be impractical.

Instead, I implemented a second layer of TNR, by running the TNR algorithm twice - once for the data set for the entire United States, and again for the resulting transit nodes from the first iteration. Each node is associated with a *local* transit node and a *super* transit node. To get a route between a *src* and *trg* node in this case, the client would fetch the *local* and *super* transit nodes associated with *src* and *trg*, and then fetch the zone info for each of those transit nodes. The client would query for the shortest path data between the *super* transit nodes, and run a shortest path algorithm over the graph constructed from all of the zone and path data in order to find the shortest path from *src* to *trg*.

An analysis of the performance of regional routing and optimized cross-country routing will be presented in Section 4.1.

Chapter 4

Conclusion

4.1 Results

4.1.1 Regional Queries

	Purpose	Type	Rows	Key	Val	Time
1	src tn	2	264,346	20 B	20 B	1.25 sec
2	trg tn	2	264,346	20 B	20 B	1.25 sec
3	src zone	3	500	20 B	25 kB	0.120 sec
4	trg zone	3	500	20 B	25 kB	0.120 sec
5	T_{src} to T_{trg}	4	250,000	40 B	1.5 kB	2.2 sec

Figure 4-1: Performance metrics and lookup information for each FSS query issued for a regional routing query over the New York data set, which has 264,346 nodes and 733,846 edges.

Purpose: the purpose of the query, which is discussed in Section 4.2.3.

Type: query type, which is detailed in Section 4.2.1.

Rows: number of rows in the corresponding table; how the table was generated is detailed in Section 4.2.3.

Key: number of bytes in the key being matched on in the function secret share.

Val: number of bytes in the value being returned from the function secret share.

Time: amount of time the round-trip query took, on average.

Queries 1 and 2, and queries 3 and 4 can be issued in parallel. Therefore, the estimated total time for a regional query is approximately 3.5 seconds. The total bandwidth used is approximately 52 kB.

In comparison, the closest practical private querying implementation, built by Wu, et al. [51], uses Private Information Retrieval techniques and takes between 140 and 371 seconds to calculate a route within a major city. The bandwidth required is 8-11 MB. Splinter is two orders of magnitude faster, and requires two orders of magnitude less bandwidth to perform a shortest path routing query within a major city region.

We estimate this application's server-side computation cost on Amazon EC2, where the cost of a CPU-hour is about 5 cents [3]. Map queries cost about 2 cents to run a shortest path query for NYC. Although it's hard to predict real-world deployment, we believe that this routing application's low cost makes it economically feasible to launch.

Studies have shown that many consumers are willing to pay for services that protect their privacy [43, 44]. Well-known sites like OkCupid, Pandora, Youtube, and Slashdot allow users to pay a monthly fee to remove ads that collect their information, showing there is already a demographic willing to pay for privacy.

One obstacle to this application's use is that many current data providers, such as Yelp and Google Maps, are based primarily on showing ads and mining user data. Nonetheless, there are already successful open databases containing most of the data in these services, such as OpenStreetMap [38], and basic data on locations does not change rapidly once collected.

4.1.2 United States Queries

	Purpose	Type	Rows	Key	Val	Time
1	src local, super tns	3	23,947,347	20 B	40 B	110 sec
2	trg local, super tns	3	23,947,347	20 B	40 B	110 sec
3	src local zone	3	5000	20 B	300 kB	0.220 sec
4	trg local zone	3	5000	20 B	300 kB	0.220 sec
5	src super zone	3	100	20 B	300 kB	0.09 sec
6	trg super zone	3	100	20 B	300 kB	0.09 sec
7	ST_{src} to ST_{trg}	4	10,000	40 B	100 kB	0.55 sec

Figure 4-2: Performance metrics and lookup information for each FSS query issued for a cross-country route request over the United States data set, which has 23,947,347 nodes and 58,333,344 edges.

Queries 1 and 2, and queries 3, 4, 5, and 6 can be issued in parallel. Therefore, the estimated total time for a cross-country query is approximately 110 seconds. The total bandwidth used is about 130 kB.

The bandwidth requirements are reasonable for a routing query, but the round trip times are not very practical. It is unlikely that a client would wait for almost two minutes for a response to a routing query, for the sake of privacy. We will discuss potential improvements in Section 4.2.1.

4.2 Improvements and Next Steps

4.2.1 Speeding up lookups over large tables

Queries 1 and 2 in Figure 4-2, which require iterating over about 24 million rows in the database of nodes, are by far the largest contributors to the run time of the United States queries. There are several potential ways to speed up this step, which I will discuss.

The client could find the local zone and super zone of their *src* and *trg* nodes without querying the server, if they have the latitude and longitude of their *src* and *trg* nodes and can locally run the code that groups nodes into local and super zones. This would take a trivial amount of processing on the client side, but it does assume that the client has prior knowledge of coordinate information about the *src* and *trg* nodes. If the client wanted to route between any nodes in the U.S. map, they would have to have a table of 24 million rows, with a mapping of nodes to their coordinates, stored locally. This would be costly to download and store, but it would be a one-time operation. This is a potential solution, but one goal of building this system is to limit the amount of data that clients have to download and store locally.

Alternatively, if the client is not concerned about revealing the regions that their *src* and *trg* nodes belong to, the system can be implemented such that queries 1 and 2 only access the tables corresponding to the region for each node. If each region is of a similar size to the New York region, queries 1 and 2 would take 1-2 seconds each (see Section 4.1.1).

4.2.2 Routing Visualization

A goal for this application is for it to be intuitively usable for people without coding experience. Having a visualization and turn-by-turn directions is crucial for this goal. One possibility for implementing this is by integrating the querying functionality of this application with the front-end of OpenStreetMap [38], an open-source map database and routing application.

4.2.3 FSS Server and Client Usability

The FSS server and client detailed in this paper are not restricted to just this map routing application - they can be used for any database and most queries. Therefore, I want to generalize the functionality of these modules, and make it easy to integrate them into other privacy-sensitive querying applications. Doing so would make it easier to build services that protect sensitive user information, as mentioned in Section 1.2. My research with Splinter and private maps querying has shown that private queries are practical to implement and run, and I hope that more services can adopt this querying model to allow for more protection of user data and behavior patterns online.

Chapter 5

Previous Work

5.1 Private Queries

Our research on private queries is related to work in Private Information Retrieval (PIR), garbled circuit systems, encrypted data systems, and Oblivious RAM (ORAM) systems. In this section, we will compare the private querying system we built - Splinter - to previous work in this space.

5.1.1 Private Information Retrieval

Our system, Splinter, is most closely related to systems that use Private Information Retrieval (PIR) [12] to query a database privately. In PIR, a user queries for the i^{th} record in the database, and the database does not learn the queried index i or result. Much work has been done on making PIR protocols more efficient [39, 37]. Work has also been done to extend PIR to return multiple records [21], but it is computationally expensive. Our work is most closely related to the system presented by Olumofin and Goldberg in [36], which implements a parametrized SQL-like query model similar to Splinter using PIR. However, because this system uses PIR, it has up to 10× more round trips and much higher response times for similar queries.

Popcorn [22] is a media delivery service that uses PIR to hide user consumption habits from the provider and content distributor. However, Popcorn is optimized for

streaming media databases, like Netflix, which have a small number (about 8000) of large records.

The systems above have a weaker security model: *all* the providers need to be honest. In Splinter, we only require *one* honest provider. Moreover, Splinter is more practical than these systems because it extends Function Secret Sharing (FSS) [10, 18], which lets it execute complex operations such as sums in one round trip instead of only extracting one data record at a time.

5.1.2 Garbled Circuits

Systems such as Embark [28], BlindBox [47], and private shortest path computation systems [51] use garbled circuits [9, 19] to perform private computation on a single untrusted server. Even with improvements in practicality [8], these techniques still have high computation and bandwidth costs for queries on large datasets because a new garbled circuit has to be generated for each query. (Reusable garbled circuits [20] are not yet practical.) For example, the recent map routing system by Wu et al. [51] has a 100× higher response time and 10× higher bandwidth cost than Splinter.

5.1.3 Encrypted Data Systems

Systems that compute on encrypted data, such as CryptDB [40], Mylar [41], SPORC [15], Depot [31], and SUNDR [29], all try to protect private data against a server compromise, which is a different problem than what Splinter tries to solve. CryptDB is most similar to Splinter because it allows for SQL-like queries over encrypted data. However, all these systems protect against a single, potentially compromised server where the user is storing data privately, but they do not hide data access patterns. In contrast, Splinter hides data access patterns but is only designed to operate on a public dataset that is hosted at multiple providers and hide the user's query parameters.

5.1.4 ORAM Systems

Splinter is also related to systems that use Oblivious RAM [48, 30]. ORAM allows a user to read and write data on an untrusted server without revealing her data access patterns to the server. However, ORAM cannot be easily applied into the Splinter setting. One main requirement of ORAM is that the user can only read data that she has written. In Splinter, the provider hosts a public dataset, not created by any specific user, and many users need to access the same dataset.

5.2 Map Routing Algorithms

5.2.1 Contraction Hierarchies

Using Contraction Hierarchies is a technique to speed up shortest-path routing by first precomputing a contracted graph [17]. This allows queries to be faster and to require less computation than other shortest-path routing methods. However, it still requires processing during the querying phase, which is not practical for our applications because any processing of one table entry would need to also be applied to every other table entry, to preserve query privacy.

5.2.2 Transit Node Routing

Transit Node Routing is another technique to speed up shortest-path routing, by first finding transit nodes and precomputing the shortest distances between them. Transit nodes are nodes with the property that every shortest path that is non-local (covers a large distance) will travel along the shortest path between the origin's transit node and the destination's transit node. The TRANSIT algorithm [24] implements Transit Node Routing, and can answer non-local shortest path queries with a speed-up of two orders of magnitude faster than the best previously reported query processing times. Most importantly for our purposes, TRANSIT finds shortest paths by combining information from a small number of look-ups in a table. This allows us to apply our techniques of Function Secret Sharing to the table look-ups that TRANSIT uses, in

order to implement efficient shortest path private queries.

5.3 Privacy-Preserving Map Routing

There have been numerous approaches toward private shortest path computation. Some approaches [14, 26] hide the client's location by using approximate locations, or by submitting dummy requests from other locations. Other works [32, 33, 52] use cryptographic protocols such as PIR to hide source and destination pairs, which provide location privacy but do not hide the server's routing information.

A practical implementation of privacy-preserving shortest path computations [51] uses Symmetric Private Information Retrieval (SPIR), Oblivious Transfer, and Yao's garbled circuits. It computes routes one hop at a time - that is, each query the user issues will return the next step along the shortest path to their destination, where a step roughly corresponds to an intersection between streets. This is arguably practical, as clients may want to view their whole route at once instead of loading it one step at a time.

Bibliography

[1] *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, October 2010.

[2] *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, Santa Clara, CA, March 2016.

[3] Amazon. Amazon EC2 Instance Pricing. `https://aws.amazon.com/ec2/pricing/`.

[4] Julian Arz, Dennis Luxen, and Peter Sanders. Transit node routing reconsidered. In *Experimental Algorithms*, pages 55–66. 2013.

[5] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. Route planning in transportation networks. *arXiv preprint arXiv:1504.05140*, 2015.

[6] Holger Bast, Stefan Funke, and Domagoj Matijevic. Ultrafast shortest-path queries via transit nodes. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, 74:175–192, 2009.

[7] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566–566, 2007.

[8] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, pages 478–492, San Francisco, CA, May 2013.

[9] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 784–796, Raleigh, NC, October 2012.

[10] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *Proceedings of the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 337–367. Sofia, Bulgaria, April 2015.

[11] Chi-Hung Chi, Choon-Keng Chua, and Weihong Song. A novel ownership scheme to maintain web content consistency. In *International Conference on Grid and Pervasive Computing*, pages 352–363. Springer, 2008.

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

[13] DIMACS. 9th DIMACS Implementation Challenge - Shortest Paths. `http://www.dis.uniroma1.it/challenge9/download.shtml`.

[14] Matt Duckham and Lars Kulik. A formal model of obfuscation and negotiation for location privacy. In *Proceedings of the 11th International Conference on Pervasive Computing*, pages 152–170, Munich, Germany, May 2005.

[15] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)* [1].

[16] Shafi Goldwasser Vinod Vaikuntanathan Frank Wang, Catherine Yun and Matei Zaharia. Splinter: Practical private queries on public data. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 299–313, Boston, MA, March 2017.

[17] Schultes Geisberger, Sanders and Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Proceedings of the 7th International Workshop on Experimental Algorithms*, pages 319–333, Cape Cod, MA, May 2008.

[18] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *Proceedings of the 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 640–658. Copenhagen, Denmark, May 2014.

[19] Shafi Goldwasser. Multi party computations: past and present. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PDC)*, pages 1–6, 1997.

[20] Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC)*, pages 555–564, Palo Alto, CA, June 2013.

[21] Jens Groth, Aggelos Kiayias, and Helger Lipmaa. Multi-query computationally-private information retrieval with constant communication rate. In *International Workshop on Public Key Cryptography*, pages 107–123. Springer, 2010.

[22] Trinabh Gupta, Natacha Crooks, Srinath TV Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with Popcorn. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)* [2], pages 91–107.

[23] Aniko Hannak, Gary Soeller, David Lazer, Alan Mislove, and Christo Wilson. Measuring price discrimination and steering on e-commerce web sites. In *Proceedings of the Conference on Internet Measurement Conference (IMC)*, pages 305–318, 2014.

[24] Stefan Funke Holger Bast and Domagoj Matijevic. TRANSIT: Ultrafast shortest-path queries with linear-time preprocessing. In *Proceedings of the 9th Center for Discrete Mathematics and Theoretical Computer Science*, Piscataway, NJ, September 2006.

[25] Jeremy Singer-Vine Jennifer Valentino-Devries and Ashkan Soltani. Websites Vary Prices, Deals Based on Users' Information, December 24 2012. Wall Street Journal.

[26] Hong Va Leong Ken C. K. Lee, Wang-Chien Lee and Baihua Zheng. Navigational path privacy protection. In *Proceedings of the 18th International Conference on Information and Knowledge Management*, pages 691–700, Hong Kong, China, November 2009.

[27] Jason Kincaid. Another Security Hole Found on Yelp, Facebook Data Once Again Put at Risk, May 11 2010. `http://techcrunch.com/2010/05/11/another-security-hole-found-on-yelp-facebook-data-once-again-put-at-risk/`.

[28] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely outsourcing middleboxes to the cloud. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)* [2], pages 255–273.

[29] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–106, San Francisco, CA, December 2004.

[30] Jacob R Lorch, Bryan Parno, James Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 199–213, San Jose, CA, February 2013.

[31] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)* [1].

[32] Kyriakos Mouratidis. Strong location privacy: A case study on shortest path queries. In *Proceedings of the 29th IEEE International Conference on Data Engineering*, pages 136–143, Brisbane, Australia, April 2013.

[33] Kyriakos Mouratidis and Man Lung Yiu. Shortest path computation with no information leakage. In *Proceedings of the 38thInternational Conference on Very Large Databases*, pages 692–703, Istanbul, Turkey, August 2012.

[34] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, pages 111–125, Oakland, CA, May 2008.

[35] Arvind Narayanan and Vitaly Shmatikov. Myths and fallacies of personally identifiable information. *Communications of the ACM*, 53(6):24–26, 2010.

[36] Femi Olumofin and Ian Goldberg. Privacy-preserving queries over relational databases. In *Proceedings of the 10th Privacy Enhancing Technologies Symposium*, pages 75–92, Berlin, Germany, July 2010.

[37] Femi Olumofin and Ian Goldberg. Revisiting the computational practicality of private information retrieval. In *Financial Cryptography and Data Security*, pages 158–172. 2011.

[38] OpenStreetMap. OpenStreetMap. `https://www.openstreetmap.org/`.

[39] Rafail Ostrovsky and William E Skeith III. A survey of single-database private information retrieval: Techniques and applications. In *Public Key Cryptography (PKC)*, pages 393–411. 2007.

[40] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 85–100, Cascais, Portugal, October 2011.

[41] Raluca Ada Popa, Emily Stark, Jonas Helfer, Steven Valdez, Nickolai Zeldovich, M. Frans Kaashoek, and Hari Balakrishnan. Building web applications on top of encrypted data using Mylar. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 157–172, Seattle, WA, April 2014.

[42] Fahmida Y. Rashid. Twitter Breached, Attackers Stole 250,000 User Data, February 2 2013. `http://securitywatch.pcmag.com/none/307708-twitter-breached-attackers-stole-250-000-user-data`.

[43] Ramprasad Ravichandran, Michael Benisch, Patrick Gage Kelley, and Norman M Sadeh. Capturing social networking privacy preferences. In *Proceedings of the 9th Privacy Enhancing Technologies Symposium*, pages 1–18, Seattle, WA, August 2009.

[44] Patrick F Riley. The Tolls of Privacy: An underestimated roadblock for electronic toll collection usage. *Computer Law & Security Review*, 24(6):521–528, 2008.

[45] Felix Salmon. Why the Internet is Perfect for Price Discrimination, September 3 2013. `http://blogs.reuters.com/felix-salmon/2013/09/03/why-the-internet-is-perfect-for-price-discrimination/`.

[46] Rick Seaney. Do Cookies Really Raise Airfares?, April 30 2013. `http://www.usatoday.com/story/travel/columnist/seaney/2013/04/30/airfare-expert-do-cookies-really-raise-airfares/2121981/`.

[47] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM SIGCOMM*, pages 213–226, London, United Kingdom, August 2015.

[48] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, November 2013.

[49] Adam Tanner. Different customers, Different prices, Thanks to big data, March 21 2014. `http://www.forbes.com/sites/adamtanner/2014/03/26/different-customers-different-prices-thanks-to-big-data/`.

[50] Renu Tewari, Thirumale Niranjan, and Srikanth Ramamurthy. WCDP: A protocol for web cache consistency. In *Proceedings of the 7th Web Caching Workshop*. Citeseer, 2002.

[51] David J Wu, Joe Zimmerman, Jérémy Planul, and John C Mitchell. Privacy-preserving shortest path computation. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2016.

[52] Loren Schwiebert Yong Xi and Weisong Shi. Privacy preserving shortest path routing with an application to navigation. In *Pervasive and Mobile Computing*, pages 13:142–149. 2014.