Splinter: Practical Private Queries on Public Data

Frank Wang[1], Catherine Yun[1], Shafi Goldwasser[1], Vinod Vaikuntanathan[1], and Matei Zaharia[2]

[1]MIT CSAIL
[2]Stanford University

Abstract

Many online services let users query public datasets such as maps, flight prices, or restaurant reviews. Unfortunately, the queries to these services reveal highly sensitive information that can compromise users' privacy. This paper presents Splinter, the first system to protect users' queries on public data while scaling to realistic applications. A user splits her query into multiple parts and sends each part to a different provider that holds a copy of the data. As long as any one of the providers is honest and does not collude with the others, the providers cannot determine the query. Splinter uses and extends a new cryptographic primitive called Function Secret Sharing (FSS) that makes it significantly more efficient than prior systems based on Private Information Retrieval and garbled circuits. We develop protocols that extend FSS to new types of queries, such as maximum and top-K queries, as well as an optimized implementation of FSS using AES-NI instructions and multicores. Splinter achieves latencies below 1.2 seconds for realistic workloads including a Yelp clone, flight search, and map routing.

1 Introduction

Many online services let users query large public datasets: examples include restaurant sites, product catalogs, and searching for directions on maps. In these services, any user can query the data, and the datasets themselves are not sensitive. However, the queries to these services reveal a great deal of private information about users, including the user's current location, which products they would like to buy, and where they are travelling,. This information can be used maliciously or put users at risk to practices such as discriminatory pricing [25, 28, 53, 57]. For example, if many users search for a particular flight, travel sites can increase its price [54]. Even when the providers are honest, the sensitive query information they store puts users at risk of server compromise [30, 47, 48].

This paper presents Splinter, a system that protects users' queries on public data while achieving practical performance for many current web applications. In Splinter, the user divides each query into shares and sends them to different *providers*, which are services hosting a copy of the dataset (Figure 1). As long as any one of the providers is honest and does not collude with the others, the providers cannot discover sensitive information in the query. However, given responses from all the providers, the user can compute the answer to her query.

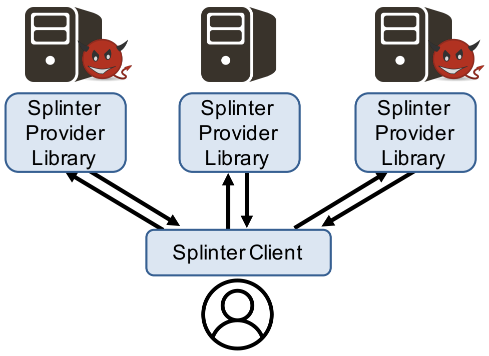
Splinter Provider Library
Splinter Provider Library
Splinter Provider Library
Splinter Client

Figure 1: 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.

Previous private query systems have generally not achieved practical performance because they use expensive cryptographic primitives and protocols. For example, systems based on Private Information Retrieval (PIR) [11, 39, 49] require many round trips and high bandwidth for complex queries, while systems based on garbled circuits [8, 31, 59] have a high computational cost. These approaches are especially costly for mobile clients on high-latency networks.

Instead, Splinter uses and extends a recent cryptographic primitive called Function Secret Sharing (FSS) [9, 20], which makes it up to an order of magnitude faster than prior systems. FSS allows the client to split certain functions into shares that keep parameters of the function hidden unless all the providers collude. With judicious use of FSS, many queries can be answered at low CPU and bandwidth cost in only a single network round trip.

Splinter makes two contributions over previous work on FSS. First, prior work has only demonstrated efficient FSS protocols for point and interval functions with additive aggregates such as sums [9]. We present protocols that support a more complex set of non-additive aggregates such as max/min and top K at low computational and communication cost. Together, these protocols let Splinter support a subset of SQL that can capture many popular online applications.

Second, we develop an optimized implementation of

FSS for modern hardware that leverages AES-NI [52] instructions and multicore CPUs. For example, using the one-way compression functions that utilize modern AES instruction sets, our implementation is 2.5× faster per core than a naïve implementation of FSS. Together, these optimizations let Splinter query datasets with millions of records at sub-second latency on a single server.

We evaluate Splinter by implementing three applications: a restaurant review site similar to Yelp, airline ticket search, and map routing. For all of our applications, Splinter can execute queries in less than 1.2 seconds, at a cost of less than 0.02¢ in server resources on Amazon EC2. Splinter's low cost means that providers could profitably run a Splinter-based service such as OpenStreetMap routing while only charging users a few dollars per month.

Finally, Splinter does have some limitations. First, FSS, like PIR, requires scanning the whole input dataset on every query, to prevent providers from figuring out which records have been accessed. Second, Splinter does not support some SQL features, such as private join conditions. Despite these limitations, we show that Splinter is practical on large real-world datasets, such as maps, and can support many of today's online applications. Because human-created datasets are unlikely to grow faster than hardware capabilities in the future, we believe Splinter's techniques will only become more practical over time.

In summary, our contributions are:

- Splinter, a private query system for public datasets that achieves significantly lower CPU and communication costs than previous systems.
- New protocols that extend FSS to complex queries with non-additive aggregates such as top-K and max.
- An optimized FSS codebase for modern hardware.
- An evaluation of Splinter on realistic applications.

2 Splinter Architecture

Splinter aims to protect sensitive information in users' queries from providers. This section provides an overview of Splinter's architecture, security goals and threat model.

2.1 Splinter Overview

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 [42] 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 [10, 58].

As shown in Figure 1, to issue a query in Splinter, a user splits her query into *shares*, using the Splinter client, and submits each share to a different provider. The user can select any providers of her 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, she combines them to obtain the final answer to her original query.

2.2 Security Goals

Splinter lets users run *parametrized queries*, where both the parameters and query results are hidden from providers. For example, consider the following query, which finds the 10 cheapest flights between a source and destination:

```
SELECT TOP 10 flightid FROM flights
WHERE source = ? AND dest = ?
ORDER BY price
```

Splinter hides the missing information represented by questions marks, i.e., the source and destination in this example. The column names being selected and filtered are not hidden. Finally, the providers also do not learn the query's results—otherwise, these might be used to infer the source and destination. Splinter supports a subset of the SQL language, which we describe in Section 4.

The easiest way to achieve this property would be for users to download the whole database and run the queries locally. However, this requires substantial bandwidth and computation for the user. Moreover, many datasets change constantly, e.g., to include traffic informations or new product reviews. It would be impractical for the user to continuously download these updates. Therefore, our performance objective is to minimize computation and communication costs. For a database of n records, Splinter only requires $O(n \log n)$ computation at the providers and $O(\log n)$ communication (Section 5).

2.3 Threat Model

Splinter keeps the parameters in the user's query hidden as long as at least one of the providers that the user chose does not collude with others. In other words, the providers can only learn the query parameters if they all collude. Splinter does not protect against providers returning incorrect results; a misbehaving provider can result in the wrong answer for the user's query. However, these incorrect results do not violate the user's privacy.

We assume that the user communicates with each provider through a secure channel (e.g., using SSL), and that the user's Splinter client is uncompromised. Our cryptographic assumptions are standard. We only assume the existence of one-way functions in our two-provider model. In our implementation for multiple providers, Pallier encryption [44] is also assumed.

3 Function Secret Sharing

In this section, we give an overview of Function Secret Sharing (FSS), the main primitive used in Splinter, and show how to use it in simple queries. Sections 4 and 5 describe Splinter's full query model and our new techniques for more complex queries.

3.1 Overview of Function Secret Sharing

Function Secret Sharing [9] lets a client divide a function f into *function shares* $f_1, f_2, \dots, f_k$ 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 [15], 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 $a \le x \le b$ 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. [9], where the functions f_a and $f_{a,b}$ operate over $\mathbb{Z}_{2^m}$ (integers mod 2^m). Under this scheme, the shares f_i for both functions require $O(ml)$ bits to describe and $O(ml)$ bit operations to evaluate for a security parameter l (the size of cryptographic keys). This contrasts to $O(m)$ bits and operations to describe and evaluate the original functions.

3.2 Using FSS 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.

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 $f_1, \dots, f_n$ and distributes them to the providers, as shown in Figure 2. 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.

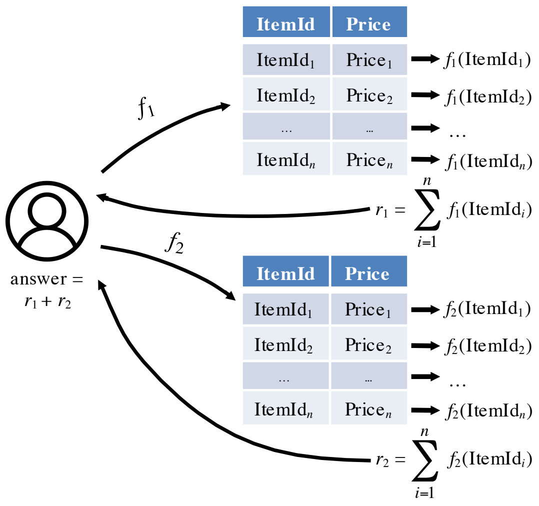

Figure 2: Overview of how FSS can be applied to database records on two providers to perform a COUNT query.

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

Figure 3: 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 3 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. Another key benefit is that FSS, unlike PIR, can efficiently match *multiple* records in one scan of the database.

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. Note that as before, all computations are over $\mathbb{Z}_{2^m}$.

4 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 6 on page 7).

Figure 4 describes Splinter's supported queries using SQL syntax. Most operators 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. We examined various websites, including Yelp, Hotels.com, and Kayak, and found we can support most of their search features (§8.1).

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.

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 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.

5 Executing Splinter Queries

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

1. The Splinter client builds function shares for the condition in the query, as we shall describe in Section 5.1.
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. We describe these protocols in Section 5.2. Some of the protocols require further communication with the client, in which case the provider batches its communication for all grouping keys together.

The main challenge in developing Splinter was designing efficient execution protocols for step 4 for Splinter's complex conditions and aggregates. Our contribution here is multiple protocols that can execute non-additive aggregates with low computation and communication costs.

One key insight that pervades our design is that *the best strategy to compute each aggregate depends on properties of the condition function*. For example, if we know that the condition can only match one value of the expression it takes as input, we can simply compute the aggregate's result for *all* distinct values of the expression in the data, then use a point function to return just one of these results to the client. On the other hand, if the condition can match multiple values, we need a different strategy that can combine results across the matching values. To reason about these properties, we define three *condition classes* that we then use in aggregate evaluation.

5.1 Condition Types and Classes

For any condition c, the Splinter client defines a function f_c that evaluates to 1 on rows where c is true and 0 otherwise, and divides f_c into shares for each provider. Given a condition c, let $E_c = (e_1, \ldots, e_t)$ be the list of expressions referenced in c (the *expr* parameters in its clauses). Because the best strategy for evaluating aggregates depends on c, we divide conditions into three classes:

- *Single-value conditions.* These are conditions that can only be true on one combination of the values of $(e_1, \ldots, e_t)$. For example, conditions consisting of an AND of '=' clauses are single-value.
- *Interval conditions.* These are conditions where the input expressions $e_1, \ldots, e_t$ can be ordered such that c is true on an interval of the range of values $e_1||e_2||\ldots||e_t$ (where $||$ denotes string concatenation).
- *Disjoint conditions,* i.e., all other conditions.

The condition types described in our query model (Figure 4) can all be converted into sharable functions, and categorized into these classes, as follows:

Equality-only conditions. Conditions of the form $e_1 = secret_1$ AND ... AND $e_t = secret_t$ can be executed as a single point function on the binary string $e_1||\ldots||e_t$. This is simply a point function that can be shared using existing FSS schemes [9]. These conditions are also single-value.

Interval and equality. Conditions of the form $e_1 = secret_1$ AND ... AND $e_{t-1} = secret_{t-1}$ AND $secret_t \leq e_t \leq secret_{t+1}$ can be executed as a single interval function on the binary string $e_1||\ldots||e_t$. This is again supported by existing FSS schemes [9], and is an interval condition.

Disjoint OR. Suppose that $c_1, \ldots, c_t$ are *disjoint* conditions that can be represented using functions $f_{c_1}, \ldots, f_{c_t}$. Then $c = c_1$ OR... OR c_t is captured by $f_c = f_{c_1} + \cdots + f_{c_t}$. We share this function across providers by simply giving them shares of the underlying functions f_{c_i}. In the general case, however, c is a disjoint condition where we cannot say much about which inputs give 0 or 1.

5.2 Aggregate Evaluation

5.2.1 Sum-Based Aggregates

SUM, COUNT, AVG, STDEV and HISTOGRAM can be evaluated by summing one or more values for each row regardless of the class of the condition function. For SUM and COUNT, each provider sums the expression being aggregated or a 1 for each row and multiplies it by f_i(row), its share of the condition function, as in Section 3.2. AVG(x) for an expression x can be computed by finding SUM(x) and COUNT(x), while STDEV(x) can be computed by finding these values and SUM(x^2). Finally, computing a HISTOGRAM into bin boundaries provided by the user simply requires tracking one count per bin, and adding each row's result to the count for its bin.[1]

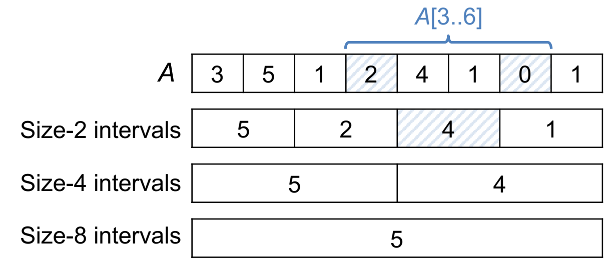

Figure 5: Data structure for querying MAX on intervals. We find the MAX on each power-of-2 aligned interval in the array, of which there are $O(n)$ total. Then, any interval query requires retrieving $O(\log n)$ of these values. For example, to find $MAX(A[3..6])$, we need two size-1 intervals and one size-2.

5.2.2 MAX and MIN

Suppose we are given a query to find MAX(e_0) WHERE $c(e_1, \ldots, e_t)$, for expressions $e_0, \ldots, e_t$. The best evaluation strategy depends on the class of the condition c.

Single-value conditions. If c is only true for one combination of the values $e_1, \ldots, e_t$, each provider starts by evaluating the query

```
SELECT MAX(e0) FROM data GROUP BY e1,...,et
```

This query gives an *intermediate table* with the tuples $(e_1, \ldots, e_t)$ as keys and MAX(e_0) as values. Next, each provider computes $\sum$MAX$(e_0) \cdot f_i(e_1, \ldots, e_t)$ across the rows of the intermediate table, where f_i is its share of the condition function. This sum will add a 0 for each non-matching row and MAX(e_0) for the matching row, thus returning the right value. Note that if the original table had n rows, the intermediate table can be built in $O(n)$ time and space using a hash table.

Interval conditions. Suppose that c is true if and only if $e_1||\ldots||e_t$ is in an interval $[a,b]$, where a and b are secret parameters. As in the single-value case, the providers can build a data structure that helps them evaluate the query without knowing a and b.

In this case, each provider builds an array A entries (k,v), where the keys are all values of $e_1||\ldots||e_t$ in lexicographic order, and the values are MAX(e_0) for each key. It then computes MAX($A[i..j]$) for all *power-of-2 aligned* intervals of the array A (Figure 5).[2]

Query evaluation then proceeds in two rounds. First, Splinter counts how many keys in A are less than a and how many are less than b: the client sends the providers shares of the interval functions $k \in [0, a-1]$ and $k \in [0, b-1]$, and the providers apply these to all keys k and

[1]Note that the binning expression is not private—only information about which rows pass the query's condition function is.

[2] This data structure is similar to a Fenwick tree [18].

return their results. This lets the client find indices i and j in A such that all the keys $k \in [a,b]$ are in $A[i..j]$.

Second, the client sends each provider shares of new point functions that select up to two intervals of size 1, up to two intervals of size 2, etc out of the power-of-2 sized intervals that the providers computed MAXes on, so as to cover exactly $A[i..j]$. Note that any integer interval can be covered using at most 2 intervals of each power of 2. The providers evaluate these functions to return the MAXes for the selected intervals, and the client combines these $O(\log n)$ MAXes to find the overall MAX on $A[i..j]$.[3]

For a table of size n, this protocol requires $O(n \log n)$ time at each provider (to sort the data to build A, and then to answer $O(\log n)$ point function queries). It also only requires two communication rounds, and $O(\log n)$ communication bandwidth. The same protocol can be used for other associative aggregates, such as products.

Disjoint conditions. If we must find MAX(e_0) WHERE $c(e_1, \ldots, e_t)$ but know nothing about c, Splinter builds an array A of all rows in the dataset sorted by e_0. Finding MAX(e_0) WHERE c is then equivalent to finding the largest index i in A such that $c(A[i])$ is true. To do this, Splinter uses binary search. The client repeatedly sends private queries of the form

```
SELECT COUNT(*) FROM $A$
WHERE $c(e_1, \ldots, e_t)$ AND $index \in [secret_1, secret_2]$,
```

where *index* represents the index of each row in A and the interval for it is kept private. (We shall discuss how to do this.) By searching for secret intervals in decreasing power-of-2 sizes, the client can find the largest index i such that $c(A[i])$ is true. For example, if we had an array A of size 8 with largest matching element at $i = 5$, the client would probe $A[0..3]$, $A[4..7]$, $A[4..5]$, $A[6..7]$ and finally $A[4]$ to find that 5 is the largest matching index.

Normally, ANDing the new condition $index \in [secret_1, secret_2]$ with c would cause problems, because the resulting conditions might no longer be in Splinter's supported condition format (ANDs with at most one interval and ORs of disjoint clauses). Fortunately, because the intervals in our condition are always power-of-2 aligned, it can also be written as an equality on the first k bits of *index*. For example, supposing that *index* is a 3-bit value, the condition $index \in [4,5]$ can be written as $index_{0,1} =$ "10", where $index_{0,1}$ is the first two bits of *index*. This lets us AND the condition into all clauses of c.

Once the client has found the largest matching index i, it runs one more query with a point function to select the row with $index = i$. The whole protocol requires $O(\log n)$ communication rounds and $O(n \log n)$ computation.

[3] To hide which sizes of intervals were actually required, the client should always request 2 intervals of each size and ignore unneeded ones.

5.2.3 TOPK

Our protocols for evaluating TOPK are similar to those for MAX and MIN. Suppose we are given a query to find TOPK(e, k, e_{sort}) WHERE $c(e_1, \ldots, e_t)$. The evaluation strategy depends on the class of the condition c.

Single-value conditions. If c is only true for one combination of $e_1, \ldots, e_t$, each provider starts by evaluating

```
SELECT TOPK($e$, $k$, $e_{\text{sort}}$) FROM data
GROUP BY $e_1, \ldots, e_t$
```

This gives an intermediate with the tuples $(e_1, \ldots, e_t)$ as keys and TOPK($\cdot$) for each group as values, from which we can select the single row matching c as in MAX.

Interval conditions. Here, the providers build the same auxiliary array A as in MAX, storing the TOPK for each key instead. They then compute the TOPKs for power-of-2 aligned intervals in this array. The client finds the interval $A[i..j]$ it needs to query, extracts the top k values for power-of-2 intervals covering it, and finds the overall top k. As in MAX, this protocol requires 2 rounds and $O(\log n)$ communication bandwidth.

Disjoint conditions. Finding TOPK for disjoint conditions is different from MAX because we need to return multiple records instead of just the largest record in the table that matches c. This protocol proceeds as follows:

1. The providers sort the whole table by e_{sort} to create an auxiliary array A.
2. The client uses binary search to find indices i and j in A such that the top k items matching c are in $A[i..j]$. This is done the same way as in MAX, but searching for the largest indices where the count of later items matching c is 0 and k.
3. The client uses a sampling technique (Appendix A) to extract the k records from $A[i..j]$ that match c. Intuitively, although we do not know which rows these are, we build a result table of $> k$ values initialized to 0, and add the FSS share for each row of the data to one row in the result table, chosen by a hash. This scheme extracts all matching records with high probability.

Like the protocol for MAX, this protocol needs $O(\log n)$ communication rounds and $O(n \log n)$ computation.

5.3 Complexity

Figure 6 summarizes the complexity of Splinter's query evaluation protocols based on the aggregates and condition classes used. We note that in all cases, the computation time is $O(n \log n)$ and the communication costs are much smaller than the size of the database. This makes Splinter practical even for databases of with millions of records, which covers many common public datasets, as we show in our Evaluation. Finally, the main operations

Aggregate	**Condition**	**Time**	**Rounds**	**Bandwidth**
Sum-based	any	$O(n)$	1	$O(1)$
MAX/MIN	1-value	$O(n)$	1	$O(1)$
MAX/MIN	interval	$O(n\log n)$	2	$O(\log n)$
MAX/MIN	disjoint	$O(n\log n)$	$O(\log n)$	$O(\log n)$
TOPK	1-value	$O(n)$	1	$O(1)$
TOPK	interval	$O(n\log n)$	2	$O(\log n)$
TOPK	disjoint	$O(n\log n)$	$O(\log n)$	$O(\log n)$

Figure 6: Complexity of Splinter's query evaluation protocols for a database of size n. For bandwidth, we report the multiplier over the query's normal result size.

used to evaluate Splinter queries at providers, namely sorting and sums, are highly parallelizable, letting Splinter take advantage of parallel hardware.

6 Optimizing FSS

Apart from introducing new protocols to evaluate complex queries using FSS, Splinter includes an FSS implementation optimized for modern hardware. In this section, we describe our implementation, which uses one-way compression functions to take advantage of AES-NI instructions. We also discuss how to select the best multi-party FSS protocol for a given application.

6.1 One-way Compression Functions

As described in Section 3.1, the main reason for the efficiency of the two-party FSS protocol [9] is its use of one-way functions. Pseudorandom generators (PRGs) [32] are a common class of one-way functions. A PRG is a function initialized with a short random seed value and provides a long stream of bits that are indistinguishable from random. Hardware support for AES [52] has made its use as a PRG particularly attractive. Generally, using AES as a PRG is straightforward (use AES in counter mode), but using AES as a PRG for FSS is not optimal. FSS requires many instantiations of a PRG with different initial seed values, especially in the two-party protocol [9]. This use of AES results in many expensive AES cipher initializations.

Cipher initialization is much slower than processing long inputs, but typically, these cipher initializations can be done in parallel. However, in the case of FSS, we cannot do this because the initial seed for a PRG *depends on the output* of the previous PRG. Recall that PRG outputs are pseudorandom, so we cannot predict the output of the previous PRG. This means that FSS requires sequentially initializing many AES cipher blocks with different keys. The challenge in Splinter is to find a suitable PRG for FSS that can avoid these expensive cipher initializations.

Our solution is to use *one-way compression functions*. As shown in Section 8.3, this provides a 2.5× speedup over using AES as a PRG. They are commonly used as a primitive in hash functions, like SHA, and are built using a block cipher like AES. In particular, we use the Matyas-Meyer-Oseas one-way compression function [36]. This compression function only needs to be initialized *once per query*. More precisely, the Matyas-Meyer-Oseas one-way compression function is defined as:

$$F(x) = E_k(x) \oplus x$$

where x is the input, i.e. PRG seed value, and E is a block cipher keyed using a fixed key k.

The output of a one-way compression function is a fixed number of bits, but we can use multiple one-way compression functions with different keys and concatenate the outputs to obtain more bits. This is still secure because the concatenation of one-way functions is still a one-way function.

In FSS, our input for the one-way compression function would be the seed of the PRG. Instead of having to re-key AES each time, we use a fixed key one-way compression function as the PRG and perform a small number of AES encryptions and XOR operations while having similar security to AES in counter mode.

6.2 Selecting the Correct Multi-Party FSS Protocol

There is one efficient protocol for two-party FSS, but for multi-party (more than 2 parties) FSS, there are two different schemes [9] and [13] that offer different tradeoffs between bandwidth and CPU usage. Both still require that only one provider is honest and does not collude with the remaining providers. In this section, we will provide an overview of the two schemes and discuss their tradeoffs and applicability to different types of applications.

Multi-Party FSS with one-way functions: In [9], the authors present a multi-party protocol based on only one-way functions, which provides good performance. However, there are two limitations. First, the function share size is proportional to the number of parties. Second, the output of the evaluated function share is only additive mod 2 (xor homomorphic), which means that the provider cannot add values locally. This limitation affects queries where there are multiple matches for a condition that requires aggregation, i.e. COUNT and SUM queries. To solve this, the provider can return all the records that match for a particular condition to the user, and can perform the aggregation locally. The size of the result is the largest number of records for a distinct predicate value, which is usually smaller than the database size. Other queries remain unaffected by this limitation. Therefore, this scheme should be used in an application that has a small number SUM or COUNT queries or is not bandwidth-sensitive. This scheme also be used if the users use a small number of providers and want faster query response times.

Multi-Party FSS with Paillier: In [13], only a point function for FSS is provided, but we modified it to handle interval functions. This scheme has the same additive properties as the two-party FSS function in [9], and does not suffer from the limitations of the scheme described above. In fact, the size of the function shares is *independent* of the number of parties. However, this scheme is slower because it requires using the Paillier [44] cryptosystem instead of one way functions and AES, but it is useful for applications where a user might want to query many providers, perform primarily SUM or COUNT queries, or has limited bandwidth.

7 Implementation

We implemented Splinter in C++, using OpenSSL 1.0.2e [41] for AES encryption. We also exposed the AES-NI hardware instructions used in OpenSSL directly to call them with less overhead. We used GMP [19] for large integers and OpenMP for multithreading. Our optimized FSS library is about 2000 lines of code, and the applications on top of it are about 2000 lines of code. There is around 1500 lines of test code to issue the queries. For comparison, we also implement the multi-party FSS scheme in [13] using 2048 bit Paillier encryption [44].

8 Evaluation

In our evaluation, we aim to answer one main question: can Splinter be used practically for real applications? To answer this question, we built and evaluated clones of three applications on Splinter: restaurant reviews, flight data, and map routing, using real datasets. Our providers ran on 64-core Amazon EC2 `x1` servers with Intel Xeon E5-2666 Haswell processors and 1.9 TB of RAM. Our client's network latency to the providers was 14 ms.

Overall, our experiments show the following:

- Splinter can support realistic applications including the search features of Yelp and flight search sites, and data structures required for map routing.
- Splinter achieves latency below 2 seconds for queries in these applications on realistic datasets.
- Splinter's protocols use up to 10× fewer round trips than prior systems and have lower responses times.

8.1 Case Studies

Here, we discuss the three application clones we built on Splinter. Figure 7 summarizes our results, and Figure 8 describes the sizes and characteristics of our three datasets. Finally, we also reviewed the search features available in real websites to study how many Splinter supports.

Restaurant review site: We implement a restaurant review site using the Yelp academic dataset [60]. The original dataset contained information for local businesses in 10 cities, but we duplicated the dataset 4 times so that it would approximately represent local businesses in 40 cities. We used the following columns in the data to perform many of the queries expressible on Yelp: name, stars, review count, category, neighborhood and location.

For location-based queries, e.g., restaurants within 5 miles of a user's current location, multiple interval conditions on the longitude and latitude would typically be used. To run these queries faster, we quantized the locations of each restaurant into overlapping hexagons of different radii (e.g., 1, 2 and 5 miles), following the scheme from [38]. We precompute which hexagons each restaurant is in and exposed these as additional columns in the data (e.g., `hex1mi` and `hex2mi`). This allows the location queries to use '=' predicates instead of intervals.

For this dataset, we present results for the following three queries:

```
Q1: SELECT COUNT(*) WHERE category="Thai"

Q2: SELECT TOP 10 restaurant
    WHERE category="Mexican" AND
    (hex2mi=1 OR hex2mi=2 OR hex2mi=3)
    ORDER BY stars

Q3: SELECT restaurant, MAX(stars)
    WHERE category="Mexican" OR
    category="Chinese" GROUP BY category
```

Q1 is a count on the number of Thai restaurants. Q2 returns the top 10 Mexican restaurants within a 2 mile radius of a user-specified location by querying three hexagons. We assume that the provider caches the intermediate table for the Top 10 query as described in Section 5.2.3 because it is a common query. Finally, Q3 returns the best rated restaurant that Mexican or Chinese. This requires more communication than other queries because it performs a MAX with a disjoint condition (§5.2.2).

Flight search: We implement a flight search service similar to Kayak [29], using the a public flight dataset [16]. The columns are flight number, origin, destination, month, delay, and price. To find a flight, we search by origin-destination pairs. We present results for two queries:

```
Q1: SELECT AVG(price) WHERE month=3
   AND origin=1 AND dest=2

Q2: SELECT TOP 10 flight_no
   WHERE origin=1 and dest=2 ORDER BY price
```

Q1 shows the average price for a flight during a certain month. Q2 returns the top 10 cheapest flights for a given source and destination, which we encode as integers. Since this is a common query, the results in Figure 7 assume a cached Top 10 intermediate table.

Map routing: 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

Dataset	Query Desc.	FSS Scheme	Input Bits	Round Trips	Query Size	Response Size	Response Time
Restaurant	COUNT of Thai restaurants (Q1)	Two-party Multi-party	11	1	~2.75 KB ~10 KB	~0.03 KB ~18 KB	57 ms 52 ms
Restaurant	Top 10 Mexican restaurants near user (Q2)	Two-party Multi-party	22	1	~16.5 KB ~1.9 MB	~7 KB ~0.21 KB	150 ms 542 ms
Restaurant	Best rated Mexican or Chinese restaurant (Q3)	Two-party Multi-party	11	11	~61 KB ~220 KB	~0.7 KB ~396 KB	0.7 s 1.0 s
Flights	AVG monthly price for a certain flight route (Q1)	Two-party Multi-party	17	1	~8.5 KB ~160 KB	~0.06 KB ~300 KB	1.0 s 1.2 s
Flights	Top 10 cheapest flights for a route (Q2)	Two-party Multi-party	13	1	~3.25 KB ~20 KB	~0.3 KB ~0.13 KB	30 ms 39 ms
Maps	Routing query on NYC map	Two-party Multi-party	Grid: 14 Transit Node: 22	2	~12.5 KB ~720 KB	~31 KB ~1.1 KB	1.2 s 1.0 s

Figure 7: Performance of various queries in our case study applications on Splinter. Response times include 14 ms network latency per network round trip. All subqueries are issued in parallel unless they depend on a previous subquery. Query and response sizes are measured per provider. For the multi-party FSS scheme, we run 3 parties. Input bits represent the number of bits in the input domain for FSS, i.e., the maximum size of a column value.

Dataset	# of rows	Cardinality
Yelp [60]	225,000	900 categories
Flights [16]	6,100,000	5000 origin-destination pairs
NYC Map [14]	260,000 nodes 733,000 edges	1333 transit nodes

Figure 8: Datasets used in the evaluation. The cardinality of queried columns affects the input bit size in our FSS queries.

FSS scheme	Grid	Transit Node	Total
Two Party	0.35 s	0.85 s	1.2 s
Multi-party	0.15 s	0.85 s	1.0 s

Figure 9: Grid, transit node, and total query times for NYC map. A user issues 2 grid queries and one transit node query. The two grid queries are issued together in one message, so there are a total of 2 network round trips.

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 [3] One algorithm compatible with Splinter is transit node routing (TNR) [2, 5], which has been shown to work well in practice [4]. In TNR, the provider divides up a map into grids, which contain at least one transit node, i.e. a transit 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. To execute a shortest path query for a given source and destination, the user can use FSS to download the paths in her source and destination grid. She locally finds the shortest path to the source transit node and destination transit node. Finally, she queries the provider for the shortest path between the two transit nodes.

We build this routing application, using a real traffic map data set from [14] for New York City. We used the source code from [2] and identified the 1333 transit nodes. We divided the map into 5000 grids, and calculated the shortest path for all transit node pairs. The grid table has 5000 rows representing the edges and nodes in a grid, and the transit node table has about 800,000 rows representing the number of shortest path for all transit node pairs.

Figure 7 shows the total response time for a routing query between a source and destination in NYC. Figure 9 shows the breakdown of time spent on querying the grid and transit node table. One observation is that the multi-party version is slightly faster than the two party version because it is faster at processing the grid query as shown in Figure 9. The two-party version of FSS requires using GMP operations, which is slower than integer operations used in the multi-party version.

Communication costs: Figure 7 shows the total bandwidth of a query request and response for the various case study queries. The sum of those two values represents total bandwidth between the provider and user.

There are two main observations. First, both the query and response sizes are *much smaller* than the size of the database. Second, for non-aggregate queries, the multi-party protocol has a smaller response size compared to the two-party protocol but the query size is much larger than the two-party protocol, leading to higher overall communication. For aggregate queries, in Section 6.2, we mention that the faster multi-party FSS scheme is only xor homomorphic, so it outputs all the matches for a specific predicate. The user has to perform the aggregation locally, leading to a larger response size than the two-party protocol. Overall, the multi-party protocols have higher

Website	Search Feature	Splinter Primitive
Yelp	Booking Method, Cities, Distance Price Best Match, Top Rated, Most Reviews Free text search	Equality Range Sorting —
Hotels.com	Destination, Room type, Amenities Check in/out, Price, Ratings Stars, Distance, Ratings, Price Name contains	Equality Range Sorting —
Kayak	From/To, Cabin, Passengers, Stops Date, Flight time, Layover time, Price Include nearby	Equality Range —
Google Maps	From/To, Transit type, Route options	Equality

Figure 10: Example website search features and their equivalent Splinter query class.

bandwidth compared to the two-party protocols despite some differences in response size.

Coverage of supported queries: We also manually characterized the applicability of Splinter to several widely used online services by studying how many of the search fields on these services' interfaces Splinter can support. Figure 10 shows the results. As shown in the figure, most services use equality and range predicates: for example, the Hotels.com user interface includes checkboxes for selecting categories, neighborhoods, stars, etc, a range fields for price, and one free-text search field that Splinter does not support. In general, all features except free-text search could be supported by Splinter. For free-text search, simple keywords that map to a category (e.g., "grocery store") could also be supported.

8.2 Comparison to Other Private Query Systems

To the best of our knowledge, the most recent private query system that can perform a similar class of queries as Splinter is that of Olumofin et al. [39], which uses multi-party PIR. Olumofin et al. creates an m-ary ($m = 4$) B+ index tree for the dataset and uses PIR to search through it to return various results. As a result, their queries require $O(\log_m n)$ round trips, where n is the number of records. In Splinter, the number of rounds trips does not depend on the size of the database for most queries. The exception is for MIN/MAX and Top-K queries with disjoint conditions (§5.2.2,§5.2.3) where Splinter's communication is similar.

Figure 11 shows the round trips required in Olumofin et al.'s system and in Splinter for the queries in our case studies. Splinter improves over [39] by up to an order of magnitude. Restaurant Q3 uses a disjoint MAX, so the communication is similar.

A similar difference in performance can be seen from the results in [39]'s evaluation, which reports response times of of 2-18 seconds for queries with several million records, compared to 1.2 seconds in Splinter. Moreover, the experiments in [39] do not use a real network, despite having a large number of round trips. Therefore, their response times would be even longer on high-latency networks. Finally, the system in [39] has weaker security guarantees: it requires *all* the providers to be honest, whereas Splinter only requires that *one* provider is honest.

Splinter Query	RTs in [39]	RTs in Splinter
Restaurant Q1	10	1
Restaurant Q2	6	1
Restaurant Q3	6	11
Flights Q1	13	1
Flights Q2	8	1
Map Routing	19	2

Figure 11: For our queries, we show the round trips required for the system of Olumofin et al. [39] and Splinter.[4]

For maps, a recent system by Wu et al. [59] used garbled circuits for map routing. They achieve response times of 140-784 seconds for their maps with Los Angeles as their largest map, and require 8-16 MB of total bandwidth. Splinter has a response time of 1.2 seconds on a larger map (NYC), which is 100× lower, and with a total bandwidth of 45-725 KB, which is 10× lower.

8.3 FSS Microbenchmarks

Cryptographic operations are the main cost in Splinter. We present microbenchmarks to show these costs of various parts of the FSS protocol, tradeoffs between various FSS protocols, and the throughput of FSS. The microbenchmarks also show why the response times in Figure 7 are different between the two-party and multi-party FSS cases. All of these experiments are done on one core to show the per-core throughput of the FSS protocol.

Two-party FSS: For two-party FSS, generating a function share takes less than 1 ms for all record sizes. The speed of FSS evaluation is proportional to the size of the input domain, i.e. number of bits per record. We can perform around 700,000 FSS evaluation operations per second on 24-bit records, i.e. process around 700,000 distinct 24-bit records, using one-way compression functions. Figure 12 shows the per-core throughput of our implementation for different FSS schemes, i.e. number of unique database records that can be processed per second. It also shows that using one-way compression functions as described in Section 6, we obtain a 2.5× speedup over using AES as a PRG.

Multi-party FSS: As shown in Figure 12, for the multi-party FSS scheme from [9] that only uses one-way func-

[4] The number of round trips in Restaurant Q3 is $O(\log n)$ in both Splinter and Olumofin et al., but the absolute number is higher in Splinter because we use a binary search whereas Olumofin et al. use a 4-ary tree. Splinter could also use a 4-ary search to achieve the same number of round trips, but we have not yet implemented this.

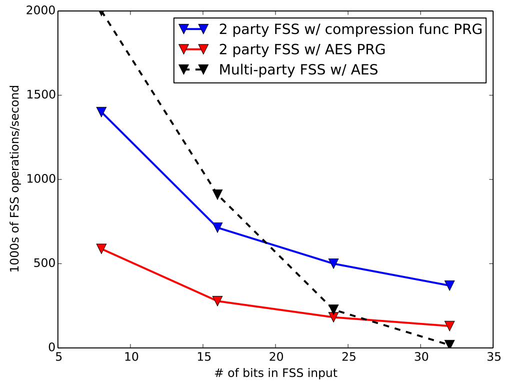

Figure 12: Per-core throughput of various FSS protocols. The graph shows the number of FSS operations that can be performed, i.e. database records processed, per second for various input sizes, on one core.

Time to generate function shares

# of bits	Query Gen in Boyle et al [9]	Query Gen in Riposte [13]
8	< 1 ms	0.06 s
16	< 1 ms	1 s
24	44 ms	16 s
32	166 ms	265 s

Figure 13: Query generation times for multi-party FSS schemes using one-way functions [9] and using Paillier [13].

tions, the time to generate the function share and evaluate it is proportional to $2^{n/2}$ where n is the number of bits in the input domain. The size of the share scales with $2^{n/2}$ rather than just n in the two-party case. An important observation is that using one-way compression functions instead of AES does not make a significant difference for multi-party FSS because the PRG is called much less and with bigger input domains compared to two-party FSS. Moreover, for small input domains (< 20 bits), the multi-party version of FSS is faster than the 2-party version, but as stated in Section 6.2, the output of the FSS evaluation is only additive mod 2 instead of modulus a larger integer group, which means a provider cannot aggregate locally for SUM and COUNT queries.

For the scheme from [13], which uses Paillier encryption, generating a function share is slower compared to [9] because it requires many exponentiations over a large integer group and depends on the number of record bits. Figure 13 shows a summary of the query generation times for both schemes. However, the evaluation of the function share is independent of the size of the input domain. The output of the function share in [13] returns a group element that is additive in a large integer group, but in order to have this property, the performance is lower compared to [9]. We can perform 250 FSS evaluations a second, but for most datasets as shown in Section 8.1 and as described in Section 9, we only have to perform FSS evaluations on the distinct values in a column, and we can cache the output for future use. As a result, this lower performance is practical for datasets that are more bandwidth-sensitive and have a small number of distinct values.

8.4 Hosting Costs

We roughly estimate the server-side computation cost of using Splinter. We evaluated our queries on one core, using a compute-optimized Amazon EC2 instance, where the cost of a CPU-hour is about 5 cents [1]. We found that most of our queries cost less than 0.002¢. For the map queries, the amount of computation required is higher, so it would cost about 0.02¢ to run a shortest path query for New York City. A service could charge a user $5 a month for 1000 queries. Costs for Amazon AWS and other cloud services have been decreasing as a result of innovation and competition [37], and further improvements to the FSS protocol will reduce computation time. Therefore, we expect our costs per query to decrease over time. Although there are other costs associated with running a web service, our results show that Splinter's server-side computation costs are very reasonable.

9 Discussion

9.1 Economic Feasibility

Although it is hard to predict real-world deployment, we believe that Splinter's low cost makes it economically feasible for several types of applications. Studies have shown that many consumers are willing to pay for services that protect their privacy [27, 51], and users do not use certain services because of privacy concerns [48, 50]. 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. As shown in Section 8.4, the cost of running queries on Splinter is low, with the most expensive query in our case studies, map routing, costing less than 0.02¢ in AWS resources per query. At this cost, providers could offer Splinter-based map routing for a subscription fee of $1 per month, assuming each user makes 100 map queries per day. Splinter's anytrust model, where only one provider needs to be honest, also makes it easy for new providers to join the market and let users increase their privacy.

One obstacle to Splinter'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 [42], and basic data on locations does not change rapidly once collected. Moreover, the availability of techniques like Splinter might make it easier to introduce regulation about privacy in certain settings, similar to current regulation on encryption in HIPAA [26].

9.2 Extensions to Splinter

To support more workloads, Splinter's query model and evaluation algorithms can be extended in several ways.

Joins: Splinter can support joining data against another table if the join condition is public: providers can run the join before filtering the data using the private condition.

Avoiding full table scans: One limitation of Splinter is that it must scan the whole dataset on each query to prevent providers from learning which records were accessed. This is true only for conditions containing sensitive parameters, however. For example, consider the query:

```
SELECT flight from table WHERE src=SFO
AND dst=LGA AND delay < 20
```

If the user does not consider the delay of 20 in this query private, Splinter could send it in the clear. The providers can then create an intermediate table with only flights where the delay < 20 and apply the private condition only to records in this table. In a similar manner, users querying geographic data may be willing to narrow the set of data they search to the country or state level, while keeping their location inside the state or country private.

10 Related Work

To our knowledge, Splinter is the first system for private queries on public data that can achieve practical performance on realistic applications. Splinter is related to work in Private Information Retrieval (PIR), garbled circuit systems, encrypted data systems, and Oblivious RAM (ORAM) systems. Splinter achieves higher performance than these systems through its mapping of database queries to the Function Secret Sharing (FSS) primitive.

PIR Systems: 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 improving PIR protocols [40, 43]. Work has also been done to extend PIR to return multiple records [23], but it is computationally expensive. Our work is most closely related to the system in [39], which implements a parametrized SQL-like query model similar to Splinter using PIR. However, because this system uses PIR, it has up to $10\times$ more round trips and much higher response times for similar queries. In addition, the system in [39] has a weaker security model: *all* the providers need to be honest. In Splinter, we only require *one* honest provider.

Popcorn [24] 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. Like [39], Popcorn also requires that all providers be honest.

Splinter is more practical than these systems because it extends Function Secret Sharing (FSS) [9, 20], which lets it execute complex operations such as sums in one round trip instead of only extracting one data record at a time.

Garbled Circuits: Systems such as Embark [31], BlindBox [55], and private shortest path computation systems [59] use garbled circuits [7, 21] to perform private computation on a single untrusted server. Even with improvements in practicality [6], 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. For example, the recent map routing system by Wu et al. [59] has $100\times$ higher response time and $10\times$ higher bandwidth cost than Splinter. Reusable garbled circuits [22] are not yet practical.

Encrypted Data Systems: Systems that compute on encrypted data, such as CryptDB [45], Mylar [46], SPORC [17], Depot [35], and SUNDR [33], 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.

ORAM Systems: Splinter is also related to systems that use Oblivious RAM [34, 56]. 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.

11 Conclusion

Splinter is a new private query system that protects sensitive parameters in SQL-like queries. Splinter uses and extends a recent cryptography primitive, Function Secret Sharing (FSS), allowing it to achieve better performance than previous systems. In Splinter, we develop protocols to execute complex queries with low computation and bandwidth, making it the first practical system for many realistic applications. As a proof of concept, we built three sample applications using Splinter—a Yelp clone, map routing, and flight search–and showed that Splinter has low response times from 50 ms to 1.2 seconds.

References

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

[2] J. Arz, D. Luxen, and P. Sanders. Transit node routing reconsidered. In *Experimental Algorithms*, pages 55–66. Springer, 2013.

[3] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. *arXiv preprint arXiv:1504.05140*, 2015.

[4] H. Bast, S. Funke, and D. Matijevic. Ultrafast shortest-path queries via transit nodes. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, 74:175–192, 2009.

[5] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566–566, 2007.

[6] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 478–492. IEEE, 2013.

[7] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796. ACM, 2012.

[8] A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 257–266. ACM, 2008.

[9] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. In *Advances in Cryptology-EUROCRYPT 2015*, pages 337–367. Springer, 2015.

[10] C.-H. Chi, C.-K. Chua, and W. Song. A novel ownership scheme to maintain web content consistency. In *International Conference on Grid and Pervasive Computing*, pages 352–363. Springer, 2008.

[11] B. Chor, N. Gilboa, and M. Naor. *Private information retrieval by keywords*. Citeseer, 1997.

[12] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[13] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, San Jose, CA, May 2015.

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

[15] Y. Dodis, S. Halevi, R. D. Rothblum, and D. Wichs. Spooky encryption and its applications. 2016.

[16] Enigma. Arrival data for non-stop domestic flights by major air carriers for 2012. https://app.enigma.io/table/us.gov.dot.rita.trans-stats.on-time-performance.2012.

[17] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[18] P. M. Fenwick. A new data structure for cumulative frequency tables. *Softw. Pract. Exper.*, 24(3):327–336, Mar. 1994.

[19] F. S. Foundation. GNU Multi Precision Arithmetic Library. https://gmplib.org/.

[20] N. Gilboa and Y. Ishai. Distributed point functions and their applications. In *Advances in Cryptology–EUROCRYPT 2014*, pages 640–658. Springer, 2014.

[21] S. Goldwasser. Multi party computations: past and present. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 1–6. ACM, 1997.

[22] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 555–564. ACM, 2013.

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

[24] T. Gupta, N. Crooks, S. T. Setty, L. Alvisi, and M. Walfish. Scalable and private media consumption with popcorn. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 91–107, 2016.

[25] A. Hannak, G. Soeller, D. Lazer, A. Mislove, and C. Wilson. Measuring price discrimination and steering on e-commerce web sites. In *Proceedings of the 2014 conference on internet measurement conference*, pages 305–318. ACM, 2014.

[26] Health insurance portability and accountability act. https://en.wikipedia.org/wiki/Health_Insurance_Portability_and_Accountability_Act.

[27] D. Indiviglio. Most Internet Users Willing to Pay for Privacy, December 22 2010. http://www.theatlantic.com/business/archive/2010/12/most-internet-users-willing-to-pay-for-privacy/68443/.

[28] J. S.-V. Jennifer Valentino-Devries and A. Soltani. Websites Vary Prices, Deals Based on Users' Information, December 24 2012. Wall Street Journal.

[29] Kayak. Kayak. https://www.kayak.com.

[30] J. 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/.

[31] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu. Embark: securely outsourcing middleboxes to the cloud. In *Proceedings of the 13th Usenix Conference on Net-*

worked Systems Design and Implementation, pages 255–273. USENIX Association, 2016.

[32] L. A. Levin. One way functions and pseudorandom generators. *Combinatorica*, 7(4):357–363, 1987.

[33] J. Li, M. Krohn, D. Mazières, and D. 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, Dec. 2004.

[34] J. R. Lorch, B. Parno, J. Mickens, M. Raykova, and J. Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 199–213, 2013.

[35] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[36] S. M. Matyas, C. H. Meyer, and J. Oseas. Generating strong one-way functions with cryptographic algorithm. *IBM Technical Disclosure Bulletin*, 27(10A):5658–5659, 1985.

[37] Y. Mor. Analyzing AWS EC2 Price Drops over the Past 5 Years, November 15 2015. `https://www.cloudyn.com/blog/analyzing-aws-ec2-price-drops-over-the-past-5-years/`.

[38] A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, and D. Boneh. Location privacy via private proximity testing. In *NDSS*, 2011.

[39] F. Olumofin and I. Goldberg. Privacy-preserving queries over relational databases. In *Privacy enhancing technologies*, pages 75–92. Springer, 2010.

[40] F. Olumofin and I. Goldberg. Revisiting the computational practicality of private information retrieval. In *Financial Cryptography and Data Security*, pages 158–172. Springer, 2011.

[41] OpenSSL. OpenSSL. `https://openssl.org`.

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

[43] R. Ostrovsky and W. E. Skeith III. A survey of single-database private information retrieval: Techniques and applications. In *Public Key Cryptography–PKC 2007*, pages 393–411. Springer, 2007.

[44] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in cryptology-EUROCRYPT'99*, pages 223–238. Springer, 1999.

[45] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100, Cascais, Portugal, Oct. 2011.

[46] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. 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, Apr. 2014.

[47] F. 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`.

[48] R. Ravichandran, M. Benisch, P. G. Kelley, and N. M. Sadeh. Capturing social networking privacy preferences. In *Privacy Enhancing Technologies*, pages 1–18. Springer, 2009.

[49] J. Reardon, J. Pound, and I. Goldberg. Relational-complete private information retrieval. *University of Waterloo, Tech. Rep. CACR*, 34:2007, 2007.

[50] P. F. Riley. The tolls of privacy: An underestimated roadblock for electronic toll collection usage. *Computer Law & Security Review*, 24(6):521–528, 2008.

[51] R. J. Rosen. Study: Consumers Will Pay $5 for an App That Respects Their Privacy, December 26 2013. `http://www.theatlantic.com/technology/archive/2013/12/study-consumers-will-pay-5-for-an-app-that-respects-their-privacy/282663/`.

[52] J. Rott. Intel advanced encryption standard instructions (aes-ni). Technical report, Technical report, Intel, 2010.

[53] F. Salmon. Why the internet is perfoect for price discrimination, September 3 2013. `http://blogs.reuters.com/felix-salmon/2013/09/03/why-the-internet-is-perfect-for-price-discrimination/`.

[54] R. 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/`.

[55] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 213–226. ACM, 2015.

[56] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Nov. 2013.

[57] A. 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/`.

[58] R. Tewari, T. Niranjan, and S. Ramamurthy. Wcdp: A protocol for web cache consistency. In *Proc. 7th Web Caching Workshop*. Citeseer, 2002.

[59] D. J. Wu, J. Zimmerman, J. Planul, and J. C. Mitchell. Privacy-preserving shortest path computation. In *NDSS*, 2016.

[60] Yelp. Yelp Academic Dataset. `https://www.yelp.com/dataset_challenge/dataset`.

A Extracting Disjoint Records with FSS

This appendix describes our sampling-based technique for returning multiple records using FSS, used in TOPK queries with disjoint conditions (Section 5.2.3). Given a table T of records and a condition c that matches up to k records, we wish to return those records to the client with high probability without revealing c.

To solve this problem, the providers each create a result table R of size $l > k$, containing (value, count) columns all initialized to 0. They then iterate through the records and choose a result row to update for each record based on a hash function h of its index i. For each record r, each provider adds $1 \cdot f_c(r)$ to $R[h(i)]$.count and $r \cdot f_c(r)$ to $R[h(i)]$.value, where f_c is its share of the condition c. The client then adds up the R tables from all the providers to build up a single table, which contains a value and count for all indices that a record matching c hashed into.

Given this information, the client can tell how many records hashed into each index: entries with count=1 have only one record, which can be read from the entry's value. Unfortunately, entries with higher counts hold multiple records that were added together in the value field. To recover these entries, the client can run the same process multiple times in parallel with different hash functions h.

In general, for any given value of r and k, the probability of a given record colliding with another under each hash function is a constant (e.g., it is less than $1/3$ for $r = 3k$). Repeating this process with more hash functions causes the probability to fall exponentially. Thus, for any k, we can return all the distinct results with high probability using only $O(\log k)$ hash functions and hence only $O(\log k)$ extra communication bandwidth.