Splinter: Practical Private Queries on Public Data

Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, Matei Zaharia[†]
MIT CSAIL, [†]Stanford InfoLab

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, a system that protects users' queries on public data and scales 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 up to an order of magnitude more efficient than prior systems based on Private Information Retrieval and garbled circuits. We develop protocols extending FSS to new types of queries, such as MAX and TOPK queries. We also provide an optimized implementation of FSS using AES-NI instructions and multicores. Splinter achieves end-to-end latencies below 1.6 seconds for realistic workloads including a Yelp clone, flight search, and map routing.

1 Introduction

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 her current location, political affiliation, sexual orientation, income, etc. [38, 39]. Web services can use this information maliciously and put users at risk to practices such as discriminatory pricing [26, 57, 61]. For example, online stores have charged users different prices based on location [29], and travel sites have also increased prices for certain frequently searched flights [58]. Even when the services are honest, server compromise and subpoenas can leak the sensitive user information on these services [31, 51, 52].

This paper presents Splinter, a system that protects users' queries on public datasets 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.

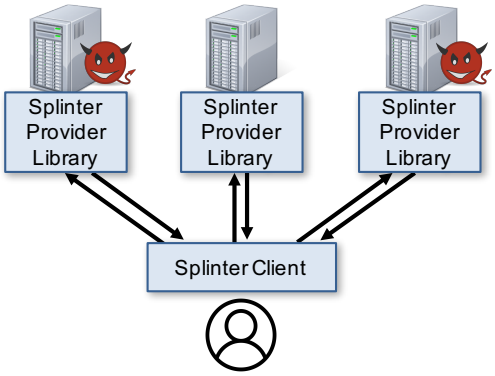

Figure 1: Splinter architecture. The Splinter client splits each user query into shares and sends them to multiple providers. It then combines their results to obtain the final answer. The user's query remains private as long as any one provider is honest.

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, 41, 53] require many round trips and high bandwidth for complex queries, while systems based on garbled circuits [8, 32, 64] 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, 21], 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 TOPK 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 [56] 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 over it: 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 1.6 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 similar to OpenStreetMap routing [46], an open-source maps service, while only charging users a few dollars per month.

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, e.g., TOPK and MAX.
- An optimized FSS implementation for modern CPUs.
- 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 [46] publishes publicly available map, point-of-interest, and traffic 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, 62].

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 use 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

The goal of Splinter is to hide sensitive parameters in a user's query. Specifically, 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 information represented by the questions marks, i.e., the source and destination in this example. The column names being selected and filtered are not hidden. Finally, Splinter also hides 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 information 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 user-chosen providers does not collude with others. Splinter also assumes these providers are *honest but curious*: a provider can observe the interactions between itself and the client, but Splinter does not protect against providers returning incorrect results or maliciously modifying the dataset.

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 implementation. In our implementation for multiple providers, the security of Paillier encryption [48] 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 then 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* $f_1, f_2, \ldots, f_k$ 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).

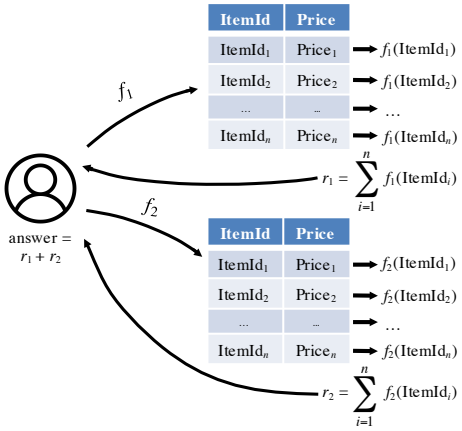

Figure 2: Overview of how FSS can be applied to database records on two providers to perform a COUNT query.

- They sum to the original function f. That is, for any input x, $\sum_{i=1}^{k} f_i(x) = f(x)$. We assume that all computations are done over $\mathbb{Z}_{2^m}$, where m is the number of bits in the output range.
- 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 [16], 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. [9]. 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), and n is the number of bits in the input domain.

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 $f_1, \ldots, f_n$ 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.

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.

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 the ItemId is equal to 5, and 1 time if it does not equal 5.

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.

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 6).

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 protocol 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 as shown in Section 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.

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 client 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 is designing efficient execution protocols for Splinter's complex conditions and aggregates (Step 4). Our contribution 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, and 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

To evaluate SUM, COUNT, AVG, STDEV and HISTOGRAM, Splinter sums one or more values for each row regardless of the condition function class. For SUM and COUNT, each provider sums the expression being aggregated or a 1 for each row and multiplies it by $f_i(\text{row})$, its share of the condition function, as in Section 3.2. Computing AVG(x) for an expression x, requires finding SUM(x) and COUNT(x), while computing STDEV(x) requires 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. Note that the binning expression is not private—only information about which rows pass the query's condition function.

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 $\sum \text{MAX}(e_0)\cdot f_i(e_1,\ldots,e_t)$ 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.

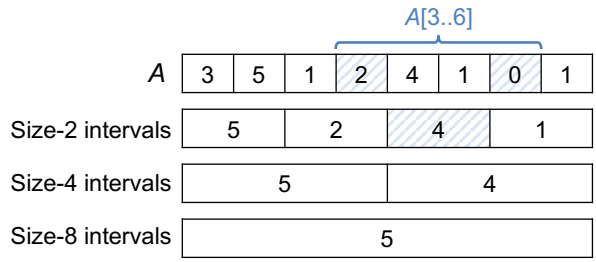

Figure 5: Data structure for querying MAX on intervals. We find the MAX on each power-of-2 aligned interval in the array, of which there are $O(n)$ total. Then, any interval query requires retrieving $O(\log n)$ of these values. For example, to find $MAX(A[3..6])$, we need two size-1 intervals and one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

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 of 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). This data structure is similar to a Fenwick tree [19].

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 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 $A[i..j]$.[1]

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

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

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,...,e_t) AND index ∈ [secret_1, secret_2],
```

where *index* represents the index of each row in A and the interval for it is kept private. 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 and works well if c has many conditions.

However, if c has a small number of OR clauses, an optimization is to run one query for each clause in parallel. The user then resolves the responses locally to find the answer to the original query. Although doing this optimization requires more bandwidth because the returned result size is larger, it avoids the $O(\log n)$ communication rounds and the $O(n \log n)$ computation.

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

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.

```
SELECT TOPK(e, k, e_sort) FROM data
GROUP BY e_1,...,e_t
```

This gives an intermediate table 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.

This protocol needs $O(\log n)$ communication rounds and $O(n \log n)$ computation if there are many clauses, but like the protocol for MAX, if the number of clauses in c is small, the user can issue parallel queries for each clause to reduce the communication rounds and 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 com-

putation 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 with millions of records, which covers many common public datasets, as shown in Section 8. Finally, the main operations used to evaluate Splinter queries at providers, namely sorting and sums, are highly parallelizable, letting Splinter take advantage of parallel hardware.

6 Optimized FSS Implementation

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 and also discuss how to select the best multi-party FSS scheme for a given query.

6.1 One-Way Compression Functions

The two-party FSS protocol [9] is efficient because of its use of one-way functions. A common class of one-way functions is pseudorandom generators (PRGs) [33], and in practice, AES is the most commonly used PRG because of hardware accelerations, i.e. the AES-NI [56] instruction. Generally, using AES as a PRG is straightforward (use AES in counter mode). However, the use of PRGs in FSS is not only atypical, but it also represents a large portion of the computation cost in the protocol. The FSS protocol requires many instantiations of a PRG with different initial seed values, especially in the two-party protocol [9]. Initializing multiple PRGs with different seed values is very computationally expensive because AES cipher initialization is *much slower* than performing an AES evaluation on an input. Therefore, the challenge in Splinter is to find an efficient PRG for FSS.

Our solution is to use *one-way compression functions*. One way compression functions are commonly used as a primitive in hash functions, like SHA, and are built using a block cipher like AES. In particular, Splinter uses the Matyas-Meyer-Oseas one-way compression function [37] because this function utilizes a *fixed key* cipher. As a result, the Splinter protocol initializes the cipher only 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 with 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. Security is preserved because a function that is a concatenation of one-way functions is still a one-way function.

With this one-way compression function, Splinter initializes the cipher, E_k, at the beginning of the query and reuses it for the rest of the query, avoiding expensive AES initialization operations in the FSS protocol. For each record, the Splinter protocol needs to perform only n XORs and n AES evaluations using the AES-NI instruction, where n is the input domain size of the record. In Section 8.3, we show that Splinter's use of one-way compression functions results in a 2.5× speedup over using AES directly as a PRG.

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], [14]) that offer different tradeoffs between bandwidth and CPU usage. Both still only require that 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 responds with all the records that match for a particular user-provided condition, and the client performs the aggregation locally. The size of the result is the largest number of records for a distinct condition, which is usually smaller than the database size. Other queries remain unaffected by this limitation. Applications should use this scheme by default because it provides the fastest response times on low-latency networks. However, for SUM and COUNT queries, an application should be careful using this scheme in settings that are bandwidth-sensitive. Similarly, an application should avoid using this scheme for queries that involve many providers.

Multi-Party FSS with Paillier: In [14], only a point function for FSS is provided, but we modified the scheme to handle interval functions. This scheme has the same additive properties as the two-party FSS protocol 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 uses the Paillier [48] cryptosystem instead of one-way functions. However, it is useful for SUM and COUNT queries in bandwidth-sensitive settings like queries over cellular network, and it is also

beneficial in settings where the user uses many providers.

7 Implementation

We implemented Splinter in C++, using OpenSSL 1.0.2e [45] and the AES-NI hardware instructions for AES encryption. We used GMP [20] for large integers and OpenMP [44] 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 [14] using 2048 bit Paillier encryption [48]. Our FSS library implementation can be found at `https://github.com/frankw2/libfss`.

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. We also compare Splinter to previous private systems, and estimate hosting costs. Our providers ran on 64-core Amazon EC2 `x1` servers with Intel Xeon E5-2666 Haswell processors and 1.9 TB of RAM. The client was a 2 GHz Intel Core i7 machine with 8 GB 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 end-to-end latencies below 1.6 sec for queries in these applications on realistic data.
- Splinter's protocols use up to 10× fewer round trips than prior systems and have lower response 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 [65]. The original dataset contains information for local businesses in 10 cities, but we duplicate the dataset 4 times so that it would approximately represent local businesses in 40 cities. We use 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 quantize the locations of each restaurant into overlapping hexagons of different radii (e.g., 1, 2 and 5 miles), following the scheme from [40]. We precompute which hexagons each restaurant is in and expose 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" OR category="Indian"
    OR category="Greek" OR category="Thai"
    OR category="Japanese"
    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 from a subset of categories. This requires more communication than other queries because it performs a MAX with many disjoint conditions, as described in Section 5.2.2. Although most queries will probably not have this many disjoint conditions, we test this query to show that Splinter's protocol for this case is also practical.

Flight search: We implement a flight search service similar to Kayak [30], using a public flight dataset [17]. 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: We implement a private map routing service, using real traffic map data from [15] for New York City. However, implementing map routing in Splinter is difficult because the providers can perform only a re-

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 restaurant in category subset (Q3)	Two-party Multi-party	11	11	~244 KB ~880 KB	~0.7 KB ~396 KB	1.3 s 1.6 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	Size (MB)	Cardinality
Yelp [65]	225,000	23	900 categories
Flights [17]	6,100,000	225	5000 flights
NYC Map [15]	260,000 nodes 733,000 edges	300	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.

stricted set of operations. The challenge is to find a shortest path algorithm compatible with Splinter. Fortunately, 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 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 paths 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, but as shown in Figure 7, the two-party version requires much less bandwidth.

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

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

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

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. 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. [41], 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. As shown in Section 5.2.2 and Section 5.2.3, the exception is for MIN/MAX and TOPK queries with many disjoint conditions where Splinter's communication is similar; if there are a small number of disjoint conditions, Splinter will be faster than previous systems because the user can issue parallel queries.

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 [41] by up to an order of magnitude. Restaurant Q3 uses a disjoint MAX, so the communication is similar.

We see a similar performance difference from the results in [41]'s evaluation, which reports response times of of 2-18 seconds for queries with several million records, compared to 50 ms to 1.6 seconds in Splinter. Moreover, the experiments in [41] do not use a real network, despite having a large number of round trips, so their response times would be even longer on high-latency networks. Finally, the system in [41] 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 [41]	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. [41] and Splinter.[2]

For maps, a recent system by Wu et al. [64] 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. 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 evaluations 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 functions, the time to generate the function share and evaluate it is proportional to $2^{n/2}$ where n is the number of bits in

[2] 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.

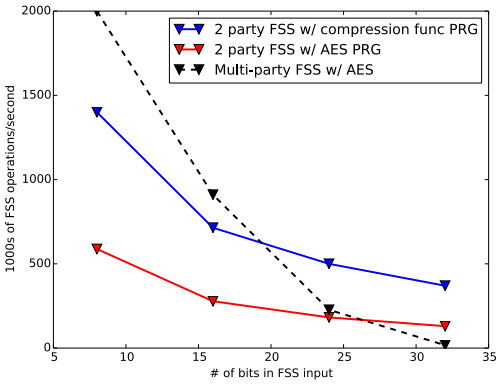

Figure 12: Per-core throughput of various FSS protocols. The graph shows the number of FSS operations that can be performed, i.e. database records processed, per second for various input sizes, on one core.

Time to generate function shares

# of bits	Query Gen in Boyle et al [9]	Query Gen in Riposte [14]
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 [14].

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 less often compared to two-party FSS. 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, a provider cannot aggregate locally for SUM and COUNT queries.

For the scheme from [14], 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 [14] 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 this lower performance is useful for SUM and COUNT operations on bandwidth-constrained clients.

8.4 Hosting Costs

We estimate Splinter's server-side computation cost on Amazon EC2, where the cost of a CPU-hour is about 5 cents [1]. We found that most of our queries cost less than 0.002¢. Map queries are a bit more costly, about 0.02¢ to run a shortest path query for NYC, because the amount of computation required is higher.

9 Discussion and Limitations

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 [28, 55]. In fact, users might not use certain services because of privacy concerns [52, 54]. 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 our most expensive query, map routing, costing less than 0.02¢ in AWS resources. 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 trust model, where only one provider needs to be honest, also makes it easy for new providers to join the market, increasing users' privacy. Whether such a business model would work in practice is beyond the scope of this paper.

One obstacle to Splinter's use is that many current data providers, such as Yelp and Google Maps, generate revenue primarily by showing ads and mining user data. Nonetheless, there are already successful open databases containing most of the data in these services, such as OpenStreetMap [46], 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 privacy regulations in HIPAA [27].

Unsupported queries: As shown in Section 4, Splinter supports only a subset of SQL. Splinter does not support partial text matching or image matching, which are common in types of applications that might use Splinter. Moreover, Splinter cannot support private joins, i.e. Splinter can only support joining with another table if the join condition is public. Despite these limitations, our study in Section 8.1 shows Splinter can support many application search interfaces.

Number of providers: One limitation of Splinter is that a Splinter-based service has to be deployed on at least two providers. However, previous PIR systems described in Section 10 also require at least two providers. Unlike those systems, Splinter requires only *one* honest provider whereas those systems require *all* providers be honest. Moreover, current multi-party FSS schemes do not scale well past three providers, but we believe that further research will improve its efficiency.

Full table scans: FSS, like PIR, requires scanning the whole input dataset on every Splinter query, to prevent providers from figuring out which records have been accessed. Despite this limitation, we have shown that Splinter is practical on large real-world datasets, such as maps.

Splinter needs to scan the whole table only for conditions that contain sensitive parameters. 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 to be 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 conditions only to records in this table. In a similar manner, users querying geographic data may be willing to reveal their location at the country or state level but would like to keep their location inside the state or country private.

Maintaining consistent data views: Splinter requires that each provider executes a given user query on the same copy of the data. Much research in distributed systems has focused on ensuring databases consistency across multiple providers [13, 43, 63]. Using the appropriate consistency techniques is dependent on the application and an active area of research. Applying those techniques in Splinter is beyond the scope of this paper.

10 Related Work

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 the result. Much work has been done on improving PIR protocols [42, 47]. Work has also been done to extend PIR to return multiple records [24], but it is computationally expensive. Our work is most closely related to the system in [41], 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.

Popcorn [25] 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. Splinter only requires *one* honest provider, and it is more practical because it extends Function Secret Sharing (FSS) [9, 21], 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 [32], BlindBox [59], and private shortest path computation systems [64] use garbled circuits [7, 22] 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. (Reusable garbled circuits [23] are not yet practical.) For example, the recent map routing system by Wu et al. [64] uses garbled circuits and has $100\times$ higher response time and $10\times$ higher bandwidth cost than Splinter.

Encrypted data systems: Systems that compute on encrypted data, such as CryptDB [49], Mylar [50], SPORC [18], Depot [36], and SUNDR [34], 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 and a user's query parameters but is only designed to operate on a public dataset that is hosted at multiple providers.

ORAM systems: Splinter is also related to systems that use Oblivious RAM [35, 60]. 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 while scaling to realistic applications. Splinter uses and extends a recent cryptography primitive, Function Secret Sharing (FSS), allowing it to achieve up to an order of magnitude better performance compared to previous private query systems. We develop protocols to execute complex queries with low computation and bandwidth. As a proof of concept, we have evaluated Splinter with three sample applications—a Yelp clone, map routing, and flight search—and showed that Splinter has low response times from 50 ms to 1.6 seconds with low hosting costs.

Acknowledgements

We thank our anonymous reviewers and our shepherd Tom Anderson for their useful feedback. We would also like to thank James Mickens, Tej Chajed, Jon Gjengset, David Lazar, Malte Schwarzkopf, Amy Ousterhout, Shoumik Palkar, and Peter Bailis for their comments. This work was partially supported by an NSF Graduate Research Fellowship (Grant No. 2013135952), NSF awards CNS-1053143, CNS-1413920, CNS-1350619, CNS-1414119, Alfred P. Sloan Research Fellowship, Microsoft Faculty Fellowship, an Analog Devices grant, SIMONS Investigator award Agreement Dated 6-5-12, and VMware.

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. 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 *Proceedings of the 34th IEEE Symposium on Security and Privacy*, pages 478–492, San Francisco, CA, May 2013.

[7] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 784–796, Raleigh, NC, Oct. 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 (CCS)*, pages 257–266, Alexandria, VA, Oct. 2008.

[9] E. Boyle, N. Gilboa, and Y. 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, Apr. 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] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[14] 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.

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

[16] Y. Dodis, S. Halevi, R. Rothblum, and D. Wichs. Spooky encryption and its applications. In *Proceedings of the 36th Annual International Cryptology Conference (CRYPTO)*, pages 93–122, Santa Barbara, CA, Aug. 2016.

[17] Enigma. Arrival Data for Non-Stop Domestic Flights by Major Air Carriers for 2012. `https://app.enigma.io/table/us.gov.dot.rita.transstats.on-time-performance.2012`.

[18] 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.

[19] P. M. Fenwick. A new data structure for cumulative frequency tables. *Software – Practice and Experience*, 24(3):327–336, Mar. 1994.

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

[21] N. Gilboa and Y. 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.

[22] S. 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.

[23] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. 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.

[24] 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.

[25] T. Gupta, N. Crooks, S. T. Setty, L. Alvisi, and M. Walfish. Scalable and private media consumption with Popcorn. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 91–107, Santa Clara, CA, Mar. 2016.

[26] 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 Conference on Internet Measurement Conference (IMC)*, pages 305–318, 2014.

[27] Health Insurance Portability and Accountability Act. `https://en.wikipedia.org/wiki/Health_Insurance_Portability_and_Accountability_Act`.

[28] 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/`.

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

[30] Kayak. Kayak. `https://www.kayak.com`.

[31] 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/`.

[32] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu. Embark: Securely outsourcing middleboxes to the cloud. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 255–273, Santa Clara, CA, Mar. 2016.

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

[34] 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.

[35] 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 *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 199–213, San Jose, CA, Feb. 2013.

[36] 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.

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

[38] A. Narayanan and V. 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.

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

[40] A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, and D. Boneh. Location privacy via private proximity testing. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2011.

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

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

[43] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 305–319, Philadelphia, PA, June 2014.

[44] OpenMP. OpenMP. `http://www.openmp.org/`.

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

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

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

[48] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the 18th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 223–238, Prague, Czech Republic, May 1999.

[49] 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.

[50] 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.

[51] 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`.

[52] R. Ravichandran, M. Benisch, P. G. Kelley, and N. M. Sadeh. Capturing social networking privacy preferences. In *Proceedings of the 9th Privacy Enhancing Technologies Symposium*, pages 1–18, Seattle, WA, Aug. 2009.

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

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

[55] 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/.

[56] J. Rott. Intel advanced encryption standard instructions (AES-NI). Technical report, Technical report, Intel, 2010.

[57] F. 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/.

[58] 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/.

[59] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM SIGCOMM*, pages 213–226, London, United Kingdom, Aug. 2015.

[60] 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.

[61] 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/.

[62] R. Tewari, T. Niranjan, and S. Ramamurthy. WCDP: A protocol for web cache consistency. In *Proceedings of the 7th Web Caching Workshop*. Citeseer, 2002.

[63] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013.

[64] D. J. Wu, J. Zimmerman, J. Planul, and J. C. Mitchell. Privacy-preserving shortest path computation. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2016.

[65] 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.