Private Categorization

Jack Gurev, Aleksejs Popovs, Hannah Whisnant

May 15, 2019

Abstract

Budgeting apps are a useful tool for consumers, but also present a significant security concern as they require users to share records of their financial transactions with third parties so the Service Provider can categorize their expenses. One potential way to improve the security of budgeting apps offering expense categorization is through Private Set Intersection. In this paper, we examine the applicability of Private Set Intersection to the problem of expense categorization and implement a number of optimizations suggested by Chen, Laine, and Rindal [CLR17] in order to determine whether such a scheme would be practical. Our implementation found a similar runtime to that described in [CLR17] for unlabeled Private Set Intersection, and a significant slowdown for labeled Private Set Intersection. While we find that current performance of our protocol is not sufficient to provide a user of an expense categorization service with real-time results, it is sufficient for use in an asynchronous service with only a short delay between when the user submits their transactions for categorization and when the results are available, and future optimizations are likely to improve performance even further.

1 Motivating Problem: Expense Categorization

Budgeting apps are a useful tool that help customers track and manage their own expenses across many different categories. These apps solve a problem that many consumers face, where the record of each credit card transaction that appears on their account is an opaque string of characters that is not descriptive of what the expense recorded actually was. Budgeting apps seek to categorize an expense that appears on a transaction record more clearly—for example, "NYSTA REBILL ACH ACH TRAN," an obscure label used to describe highway tolls, may appear as "Travel," allowing a consumer to understand where their money is being spent and better manage their finances.

A wide variety of these apps are currently available to consumers, including popular programs like Intuit Mint and PocketGuard. In all cases, these apps require the user to agree to turn over financial data to the service provider in order to analyze spending information in a way that is useful. This agreement is often discouraged by banks and other financial institutions—TD Canada,

for example, states that they "do not endorse the use of aggregation services due to the potential risk to personal information" [Cha18]. Sharing personal financial information with a third-party service provides a potential avenue for customer's data to be breached by an attacker or misused by the service, and most consumers have no way to verify their information will be secure.

Consumers of these services, however, do care about security—a significant portion of the inquiries on Mint's and PocketGuard's Frequently Asked Questions sections are related to the security of their product [Min19] [Sta19]. Banking data is often very personal and sensitive for individuals, so we were motivated by the question: how can expense categorization in budgeting apps be made more secure?

In order to address this, we have proposed a solution based on Private Set Intersection (PSI), which would allow a user to provide a service with encrypted transaction data, and allow the service to return the labeled intersection of the user's set with their own without revealing any additional data to either party.

2 Literature Review

2.1 Origins of Private Set Intersection

Proposals for PSI have existed for several decades, beginning with a proposal in 1986 by C. Meadows that was fully described by Hoberman, Franklin, and Hogg. These proposals were secure and correct—however, they were significantly limited by their high computational costs, which increased dramatically with the sizes of the two sets being compared [CLR17].

Over the next twenty years, there was significant research into finding more and more efficient implementations of PSI that would be practically applicable on a large scale. Many researchers forcused on implementations of Oblivious Transfer, while others applied multi-party computation protocols like garbled circuits, the user of non-colluding servers, or implementations of protocols based on RSA accumulators [CLR17]. While these implementations demonstrated significant gains in computational efficiency, these continued to be linear to the size of each of the two sets being compared.

2.2 Unbalanced Private Set Intersection

In the case of expense categorization, it is overwhelmingly likely that the set belonging to the Service Provider (which will include the transaction label for every potential transaction any consumer can undertake) will be very significantly larger than the set belonging to the user (which includes only the transaction labels for transactions that that specific consumer has undertaken). This presents a similar set of challenges to a previously studied application: that of private contact discovery. A secure messaging service like WhatsApp or Signal might want to allow users to discover which of their existing contacts use their secure service, without allowing the service itself access to all of their contact

information. A user will have far fewer contacts than the app has users, leading to an unbalanced PSI problem as first proposed by Moxie Marlinspike [Mar14].

A potential solution to this problem of unbalanced PSI was proposed by Chen, Laine, and Rindal in 2014. Rather than relying on Oblivious Transfer or multi-party computation, this solution takes advantage of a number of developments in fully homomorphic encryption. When optimized, this solution is still linear to the size of the smaller set, but it is logarithmic to the size of the larger set, allowing significant gains in performance in cases like expense categorization where one set is significantly larger than the other [CLR17]

2.3 Fully Homomorphic Encryption

Fully homomorphic encryption (FHE) is not a new concept, first described in 1978 by Rivest, Adleman, Dertouzos. However, no implementation of FHE was constructed until 2009, by Craig Gentry. The development of FHE closely mirrored the development of PSI—while early implementations were too costly to be effective for practical implementation, a number of researchers have offered continuing improvements in recent years that bring FHE closer and closer to applicable [CLR17].

Currently, a number of powerful tools are available to those who wish to leverage FHE for secure communication and computation. With the primary emphasis of this project on the implementation and optimization of PSI, we have chosen to use Microsoft SEAL, a preexisting open-source homomorphic encryption library [SEA19].

2.4 Application to Expense Categorization

In order to apply PSI to expense categorization, we have attempted to implement the basic protocols and optimizations in unbalanced PSI as described by Chen, Laine, and Rindal and compare the performance of the implementation to that described in their paper. We do not have access to the datasets that commonly used budgeting apps use to categorize expenses, as these apps maintain the privacy of their data in order to gain a competitive advantage. However, we present a proof of concept that demonstrates that Chen, Lain, and Rindal's optimized and labeled implementation of PSI, using Microsoft SEAL to provide FHE, would be practically applicable to the problem of expense categorization.

3 Formal Problem: Private Set Intersection

Expense categorization can be thought of as, for some set of categories C, for every $c \in C$ informing the user which of their transactions are in category c. We can imagine the Service Provider has a large list of merchants for each of these categories, and so for each such category must compute the intersection of the user's list of transactions and the list of merchants for that category. In this simplest case of a single category of merchants, we arrive at the problem

of Private Set Intersection, or PSI, which for our purposes we will model as a Service Provider SP with a set S, of size $N_S := |S|$ along with a user who has a smaller set U of size N_U. A private set intersection algorithm should satisfy the following three guarantees:

1. The user should learn the intersection $U \cap S$.
2. The user should not learn anything else about the elements of S.
3. The Service Provider should not learn anything about the elements of U.

In [CLR17], they describe a basic algorithm for PSI using homomorphic encryption, and then a series of improvements to the basic algorithm using several techniques, which decrease this communication complexity as well as the circuit depth of the required operations on encrypted data. This circuit depth is the most important factor in the runtime of the algorithm.

We will describe the basic algorithm and its improvements, and then our own implementation and testing.

4 Naive Private Set Intersection Using Homomorphic Encryption

The User chooses a public/secret key pair $(\mathbf{pk}, \mathbf{sk})$ for a Fully Homomorphic Encryption scheme. They send $\mathbf{pk}$ to the Service Provider, as well as ciphertexts $(c_1, \ldots, c_{N_U}$ for each element of U.

For each i, the Service Provider chooses a random non-zero plaintext r_i, and homomorphically computes

$$d_i := r_i \prod_{s \in S} (c_i - s)$$

They then return these ciphertexts $(d_1, \ldots, d_{N_U}$ to the user. The user decrypts each ciphertext, and outputs the set of all i for which d_i decrypts to 0.

We can verify that this scheme satisfies all of the security guarantees for PSI: the product $r_i \prod_{s \in S} (c_i - S)$ is 0 if $c_i \in S$ and a random plaintext otherwise, so the user correctly learns the intersection and nothing else about the Service Provider's set S. Meanwhile, by the security guarantees of FHE, the Service Provider does not learn anything about U from the encryption of its elements.

5 Optimization of Private Set Intersection

The Naive PSI algorithm requires $O(N_U N_S)$ homomorphic operations to be performed by the Service Provider. In addition, a circuit depth of $O(\log(N_S))$ is required. We will now go through techniques which reduce this to $O(N_U \log N_S)$ homomorphic operations, as well as decrease the circuit depth.

5.1 Batching

Batching is a method of decreasing the amount of communication required between the User and Service Provider, as well as the computation time, by grouping the input items together.

Specifically the user divides U into vectors of length n, encrypts these vectors, and sends the N_U/n ciphertexts $\mathbf{c}_i$ to the Service Provider.

For each i, the Service Provider then samples a vector $\mathbf{r}_i = (r_{i1}, \ldots, r_{in})$ of uniformly random non-zero elements, computes

$$\mathbf{d}_i = \mathbf{r}_i \prod_{s \in S} (c_i - s)$$

and sends them back to the user.

Because the Service Provider can operate on n items simultaneously, we have an n-fold improvement in both communication and computation.

5.2 Hashing

Another improvement comes from hashing. Intuitively, instead of comparing every pair (u, s) to compute the intersection, first hash all of the elements of U and S into buckets B. Then we only have to check whether elements of U and S which hash to the same bucket are equal.

More carefully, suppose first that we use a single random hash function H into N_U buckets B. The user hashes each element of U, encrypts these hashes into ciphertexts c_i, then sends $\{(c_i, b_i)\}$ to the Service Provider.

The maximum load will on expectation be $O(\log N_U)$.

As is, the second security assumption will be broken: the service provider will learn how many elements of U hash to each bucket. To fix this, the user must fill all buckets up to the maximum load, which requires on expectation $O(N_U \log N_U)$ ciphertexts to be sent by the user.

To decrease the maximum load, we can use Cuckoo hashing. The user chooses $h > 1$ random hash functions into $(1 + \epsilon)N_U$ buckets B. When an element of N_U hashes into a bucket which is already full, they switch the element it is colliding with to its next hash function.

Cuckoo hashing requires the Service Provider to hash all of their elements using all h of the hash functions, but it still provides a large speedup by decreasing the number of multiplications which need to be done homomorphically.

5.3 Windowing

The third optimization from Chen, Laine, and Rindal that we have implemented is windowing. When the Service Provider computes the value of d_i for a given value c_i, we can see that we must compute this value as:

$$\mathbf{r}_i \prod_{s \in S} (c_i - s) = rc^{N_s} + ra_{N_s - 1}c^{N_s - 1} + \ldots + ra_0$$

However, if the user, whose computational requirements are relatively small, instead sends the Service provider additional powers of c, the Service Provider has many fewer necessary computations. The computations user sends take the form $c^{i \cdot 2^j}$, where, given a window size of l bits, $0 \leq i \leq 2^l - 1$ and $1 \leq j \leq \lfloor log_2(N_s)/l \rfloor$.

5.3.1 Effect of Windowing on Computation

Windowing results in gains in computational efficiency. Without windowing, the Service Provider must compute at most the product rc^{N_s}, which requires a circuit of depth $\lceil log_2(N_s + 1) \rceil$. However, with windowing, the Service Provider needs to compute only the product of $\lfloor log_2(N_s)/l + 1) \rfloor$ terms at most, which is equivalent to a circuit depth of $\lceil log_2(\lfloor log_2(N_s)/l + 1) \rfloor) \rceil$ The true circuit depth depending on the number of encryptions that the user sends.

5.3.2 Effect of Windowing on Communication

Unlike batching and hashing, windowing increases the costs of communication. While the communication from Service Provider to user remains unchanged, in order to send powers of c to the Service Provider, the user's communication to that Service Provider must increase by a factor of $(2^l - 1) \cdot (\lfloor log_2(N_x)/l \rfloor + 1)$. This is equivalent to the total number of encrypted values of $c^{i \cdot 2^j}$.

5.4 Partitioning

The fourth optimization from Chen, Laine, and Rindal that we have implemented is partitioning. Partitioning occurs when the Service Provider divides its set into α subsets of roughly equal size. This partitioning should be uniformly random in order to preserve security.

5.4.1 Effect of Partitioning of Computation

When combined with windowing, partitioning has the further effect of reducing the number of powers of c that either the user of the Service Provider must compute. Given a partition of size k, this is a reduction from c^{N_s} to $c^{N_s/k}$. These computations can then be reused for each of the k-sized subsets.

5.4.2 Effect of Partitioning on Communication

Like windowing, partitioning increases the complexity of communication. In this case, the communication from the Service Provider back to the user increases by a factor of k, since the Service Provider must return whether or not each encrypted c intersects with each of the k-sized subsets.

5.5 Labeled Private Set Intersection

An additional challenge of the expense categorization problem is that its solution requires not only the indication that the user's transaction label exists in the Service Provider's set, but additionally the return of information regarding what category label the Service Provider has assigned to that transaction label (i.e., what category the expense falls into). The solution to this problem lies in labeled PSI.

Labeled PSI was proposed by Chen, Huang, Laine, and Rindal in 2018 [CHLR18]. We have described and implemented labeled PSI is compatible with PSI as described in [CLR17], setting aside further optimizations that are possible given fundamental changes to the PSI implementation.

5.5.1 Labeled Private Set Intersection Protocol

We construct a scheme where the Service Provider returns the pair $(0, l_i)$ where l_i is the label for a given element if the user's element exists in the Service Provider's set, and returns a pair of random elements of the field otherwise. First we create a polynomial H such that $H(c_i) = l_i$ for all i. The Service Provider returns the pair $(d_i, r \cdot d_i + H(c_i))$ to the user. d_i is calculated identically in labeled PSI as it is in unlabeled PSI. In the case that the value is in the Service Provider's set, homomorphic encryption allows the user to view the label if the decryption of $d_i = 0$, and the multiplication by a uniformly randomly chosen non-zero element r obfuscates the Service Provider's labels in cases where the user's element does not exist in the Service Provider's set.

5.5.2 Computational Complexity

Labeled PSI introduces some amount of additional computational complexity. Much of this occurs offline, as the Service Provider must calculate the appropriate polynomial H. Additionally, during the categorization process there is some amount of additional computational complexity, as the Service Provider must homomorphically evaluate the polynomial it has interpolated. According to [], this difference is equivalent to $O(\frac{N_S N_U}{m^2} \cdot l/\sigma)$, where m is the number of bins determined in hashing, l is the label length, and σ is the element length.

5.5.3 Complexity of Communication

Labeled PSI also increases the complexity of communication, as the Service Provider must return both the label and the value d_i. This results in additional complexity of $O(N_U \cdot l)$, where l is the label length.

6 Networking

While most of the optimizations of PSI that we have implemented as a proof of concept for expense categorization are intended to optimize the computational

efficiency of the protocol, communication costs are also a key component of evaluating PSI for usability. As such, we have designed up a very basic network protocol and implemented a server and a client in C++ in order to ensure that communication between user and Service Provider will not be prohibitively expensive. Although the use of a single client is not necessarily reflective of the real network environment over which a potential budgeting app would be run, our mock network application serves as a further proof of concept for an app that would apply PSI for expense categorization.

7 Implementation results

We have implemented the [CLR17] PSI algorithm, along with the [CHLR18] trick for Labeled PSI and our networked protocol for Private Categorization, in a C++ application, using the Microsoft SEAL library [SEA19] for an implementation of a leveled fully homomorphic encryption scheme. The implementation is available under a free software license at `https://github.com/popoffka/6857-private-categorization`. In the rest of this section, we evaluate the performance of our implementation and the practicality of using it to provide a Private Categorization-based service online.

7.1 Performance comparison with [CLR17]

In [CLR17], the authors report on the performance of their own implementation, which uses an older version of the SEAL library and is not publicly available. We have compared the performance of our implementation against the reported numbers in [CLR17], and the results of the comparison are summarized in Table 1. In the same table, we also demonstrate the performance of our Labeled PSI implementation, in order to analyze the cost of the [CHLR18] trick. All times are averages of 10 runs.

Three limitations apply to this comparison:

- our implementation is single-threaded, so we also only list single-thread performance results for [CLR17],
- our implementation does not separate the sender's computations into a pre-processing and online phase, so we list total sender time for both implementations, and
- the results in [CLR17] were obtained on a powerful server machine ("two 18-core Intel Xeon CPU E5-2699 v3 @ 2.3 GHz and 256GB of RAM"), whereas our results were obtained on a commodity laptop (dual-core Intel Core i3-6100U @ 2.3 GHz and 8GB of RAM).

Nevertheless, we observe that the performance of our implementation is on par with that of [CLR17]. We believe that the difference in benchmarking hardware is mostly insignificant because we are only comparing single-thread performance, while the machines primarily differ on the number of threads they can

N_S	N_U	n	α	l	Implementation	Sender	Rec. enc.	Rec. dec.
2^{16}	5535	8192	8	3	[CLR17]	1.3	0.3	0.1
					Ours	1.1	0.1	0.1
					Ours, labeled	2.0	0.1	0.1
	11041	16384	8	2	[CLR17]	2.5	0.3	0.3
					Ours	3.5	0.1	0.1
					Ours, labeled	5.2	0.1	0.2
2^{20}	5535	8192	64	2	[CLR17]	7.8	0.2	1.0
					Ours	7.4	0.1	0.2
					Ours, labeled	21.7	0.1	0.4
	11041	16384	32	3	[CLR17]	10.9	0.7	1.3
					Ours	13.5	0.3	0.4
					Ours, labeled	32.3	0.3	0.8
2^{24}	5535	8192	256	1	[CLR17]	100.1	0.2	4.9
					Ours	103.4	2.1	0.8
					Ours, labeled	682.9	2.1	1.5
	11041	16384	128	2	[CLR17]	109.8	0.5	5.1
					Ours	158.4	2.2	1.5
					Ours, labeled	758.2	2.2	3.0

Table 1: Running time in seconds for multiple implementations of PSI and Labeled PSI. "Rec. enc." denotes the time spent by the user encrypting their inputs, "Sender" denotes the time spent by the service provider computing on the encrypted inputs, and "Rec. dec." denotes the time spend by the user decrypting the results of the computation. $\sigma = 32, h = 3$. Optimization parameters are taken from [CLR17].

run (and also on memory, but the limited amount of memory available on our testing machine is sufficient for the computations tested). We expect that parallelizing our implementation and separating the sender's pre-processing phase should be relatively straightforward, and that we would again see performance similar to that of [CLR17]. Therefore, we believe that our results accurately represent the real-world performance of the [CLR17] scheme.

As for the [CHLR18] trick for Labeled PSI, its effects on the runtime of the receiver are completely predictable: no effect on encryption times and double the decryption times, since the encryption process is exactly the same and the decryption process involves decrypting exactly twice as many ciphertexts. The sender runtime is affected in a more complicated way, and the slowdown ranges from 1.5x up to 6.6x for the parameters tested. We believe that this is because the polynomial interpolation procedure involved in Labeled PSI is slower than the one involved in regular PSI, so cases where the sender's runtime is dominated by polynomial interpolations are affected more than those where it is dominated by homomorphic multiplications. We conjecture that increasing l should lead to better performance for Labeled PSI in these cases, by decreasing the degrees of the polynomials being interpolated.

N_S	N_U	n	α	l	Sender	Rec. enc.	Rec. dec.
2^{20}	100	8192	256	2	22.5	0.1	1.5
2^{21}		8192	512	2	138.8	0.1	3.1
2^{22}		8192	1024	1	172.7	0.1	6.3
2^{23}		8192	2048	1	227.9	0.1	12.3

Table 2: Running time in seconds for our Labeled PSI implementation. $\sigma = 46, h = 3$.

7.2 Application to real-world categorization problems

We are now going to consider an application of our implementation of the Labeled PSI protocol to our real-world motivating problem, that of categorizing credit card transactions by expense type.

According to [RC13], there are on the order of 2^{23} credit card merchants in the United States. By the birthday paradox, this means that, if the parties are going to use hashing to obtain short binary strings corresponding to the merchant identifier strings found on credit card statements, the output length of the hash must be at least 46 bits to keep the number of collisions low.

Therefore, we are considering an instance of Labeled PSI with item length $\sigma \geq 46$. [CLR17] warn that the encryption parameters have to be substantially increased to accommodate longer items, leading to "a large negative impact on performance." We are now going to quantify this impact.

Assuming that a user uses the categorization service once per month, after receiving their statement from their bank, we expect the number of transactions that they need to categorize to be on the order of 100, based on personal experience. However, notice that the exact size of the receiver's set N_U does not directly affect the performance of the protocol, but rather the number of buckets that the set is cuckoo hashed into does. Because of batching, there is no reason to hash into fewer buckets than n, the number of slots in a single ciphertext, so, as long as $n \geq 8192$, we should expect to see similar performance for values of N_U of up to at least 4096.

It is plausible that a version of the Pareto principle (80/20 rule) applies to credit card merchants, that is, that roughly 20% of the merchants generate around 80% of all transactions. Then a transaction categorizing service doesn't need to know about every single merchant to be useful to most users most of the time. Therefore we consider different possible sizes of the server's set from 2^{20} up to 2^{23}.

In Table 2, we have summarized the performance of our implementation of Labeled PSI for $\sigma = 46, N_U = 100$ and various values of N_S from 2^{20} to 2^{23}, using optimization parameters that we have found to yield the best performance.

As we can see, the performance is not sufficient for a real-time application, as checking the user's 100 transactions against a list of even just 2^{21} merchants (around a quarter of the total number of merchants) requires more than two minutes of computation on the sender's side. However, the protocol could still plausibly be used in an asynchronous service, where the user uploads their

encrypted transactions to the service and later receives a push notification when the processed results are available. Parallelization could also be used to decrease wait times somewhat.

Given that each user would only submit their transactions once per month, the per-user processing cost is not prohibitive—228 seconds of CPU time per month is equivalent to serving about 150–380 average web requests per day[1], which is a manageable number of requests for a dynamic application to receive from a user.

8 Conclusion

In this paper, we have considered one possible practical application of Private Set Intersection protocols, and, after analyzing a particular implementation, found that the performance, while not real-time, is nevertheless sufficient to enable practical use. We believe that PSI-based protocols are a powerful building block for services that respect the users' privacy while not requiring service providers to make their datasets public, and we would like to see them adopted by commercial service providers.

We have only considered instances of Private Categorization where categorization is performed by exact matching against a labeled list of known items. In many applications, it might be useful to also allow fuzzy matching, substring matching, or other matching methods. More research is required to determine whether modifications of PSI-based protocols or another secure multiparty computation protocol can be used to provide Private Categorization with more sophisticated matching with performance suitable for real-world applications.

References

[Cha18] Karoun Chahinian. Are budgeting apps like Mint safe to use? *MoneySense*, June 2018. `https://www.moneysense.ca/save/budgeting/mint-budgeting-app-safe/`.

[CHLR18] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from Fully Homomorphic Encryption with Malicious Security. Cryptology ePrint Archive, Report 2018/787, 2018. `https://eprint.iacr.org/2018/787`.

[CLR17] Hao Chen, Kim Laine, and Peter Rindal. Fast Private Set Intersection from Homomorphic Encryption. Cryptology ePrint Archive, Report 2017/299, 2017. `https://eprint.iacr.org/2017/299`.

[1] Assuming it takes 20 to 50 milliseconds to serve a web request. Google recommends that servers take at most 200 ms to serve a request [Goo19], so we assume that 10%–25% of that is a reasonable conservative average.

[Goo19] Google. PageSpeed Insights: Improve Server Response Time. February 2019. https://developers.google.com/speed/docs/insights/Server.

[Mar14] Moxie Marlinspike. The difficulty of private contact discovery. *Whisper Systems*, January 2014. https://whispersystems.org/blog/contact-discovery/.

[Min19] Intuit Mint. How to protect your Mint account. *MintHelp*, 2019. https://help.mint.com/Mint-Account-Management/888963021/How-to-protect-your-Mint-account.htm.

[RC13] Brian Roemmele and David Charlot. How many payment card merchants are in the US? *Quora*, January 2013. https://www.quora.com/How-many-payment-card-merchants-are-in-the-US.

[SEA19] Microsoft SEAL (release 3.2). https://github.com/Microsoft/SEAL, February 2019. Microsoft Research, Redmond, WA.

[Sta19] PocketGuard Stan. Is it secure to link my accounts? *PocketGuard*, April 2019. https://help.pocketguard.com/hc/en-us/articles/360002167020-Is-it-secure-to-link-my-accounts-.