Silent Compute
Technical Overview: Open Finance

Silence Laboratories
info@silencelaboratories.com

November 15, 2024

Abstract

This document serves as a companion to the whitepaper *Open Finance Revisited: Strengthening Data Governance with Cryptographic Privacy and Auditability*. Technical specifications of the Multiparty Computation (MPC) protocols referenced in that whitepaper were omitted in the interest of accessibility to non-expert readers. This document provides an overview of those protocols for interested expert readers, although complete protocol descriptions are deferred to the full version of the technical specification document.

1 Introduction

The *Open Finance Revisited* whitepaper [KKS+24] describes a system in which user data d is held in secret shared format amongst three parties, one of whom may be actively corrupt. Upon receiving the user's consent to operate on this data, the parties jointly run Multiparty Computation (MPC) protocols to compute $f(d)$, and expose only the output of this computation. The motivation of the system is to protect users and Financial Information Providers (FIPs) from data duplication by data fiduciaries (Financial Information Users, or FIUs) and to bind data usage to user consent. An important aspect of this system is how the parties are able to obtain secret shares of the user's data in a secure fashion. The running example considered in the Open Finance paper is that of the Account Aggregator (AA) framework [Sah], which delivers encrypted data to the system upon the user providing their consent. The workflow, without the MPC component, is recalled below in Figure 1.

As mentioned earlier, we envision the Financial Information User (FIU) to be decentralized by means of a three-party system, where sensitive data is never exposed in plaintext. Our foremost priority is to have this system be a drop-in replacement for the existing FIU role, i.e. it must be directly compatible with protocols in the AA framework, allowing the rest of the ecosystem to be agnostic to the FIU's decentralization. This principle is visualized in Figure 2.

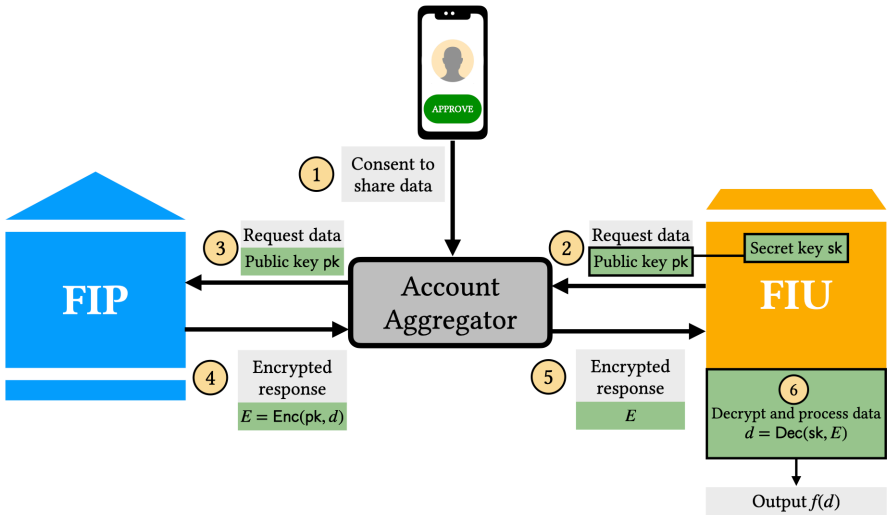

Figure 1: Existing workflow of the Account Aggregator framework, which facilitates the delivery of encrypted user data from a Financial Information Provider (FIP) to a data fiduciary, i.e. Financial Information User (FIU).

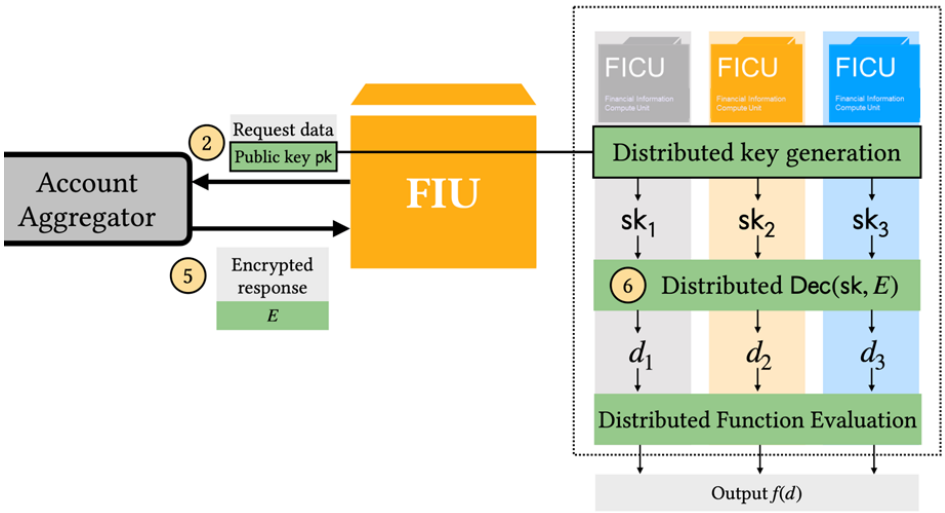

Figure 2: Drop-in decentralization of the FIU. The rest of the AA ecosystem is agnostic to the FIU's new configuration.

2 High Level Structure

With the context having been established, we proceed to describing the MPC protocols involved. The workflow can be divided into three phases:

1. **Fetching the Data**. The first step is to establish a mechanism to import the encrypted data delivered by the AA into the 3PC system. This entails generating the FIU's public key pk in a distributed fashion, and decrypting the AA's encrypted response $E = \mathsf{Enc}(\mathsf{pk}, d)$ directly into secret shared format—without exposing d or the FIU's secret key sk to any one party.

2. **Parsing Data into a Query-friendly Format**. Once the data d has been decrypted into secret shares, it is in principle feasible to run queries upon it immediately. However, note that d is typically a string that must be tokenized before queries can be run; for instance in the AA specification, d is an XML file. While this operation may be straightforward in plaintext, performing this operation in MPC is non-trivial. Empirically, we have observed this step to be most expensive in the workflow, but fortunately this is done only once when loading the data into the 3PC system. Therefore, we treat it as a separate phase.

3. **Executing Queries**. Once data is decrypted and tokenized, we discuss optimized primitive operations that can be composed to construct reponses to standard queries on it in the Open Finance setting.

First, we establish the secret sharing format: given that the setting is that of three parties with tolerance to one active corruption, we make use of Replicated Secret Sharing [CDI05]. This secret sharing methodology is remarkably versatile in that it does not require any particular structure of the underlying ring/field, which suits our setting. The computations relevant to this setting are diverse in their native representations, as they include Boolean and arithmetic circuits, as well as elliptic curve group operations. We therefore choose the most appropriate representation for any given operation, and describe how we convert between representations when relevant.

There are several places in which we invoke general MPC as a subroutine, which we implement in the common paradigm of running a lightweight "secure with additive errors" protocol first [GIP+14, GIP15], and then verifying the result before it is ready to be used. We employ the techniques of Furukawa et al. [FLNW17] in the Boolean case, and Lindell and Nof [LN17] in the arithmetic case for this purpose. While more theoretically efficient protocols have been developed, these are conceptually simpler (enabling easier implementation and validation), and efficient enough in for our use cases.

We now proceed to describe the tools that comprise each phase.

3 Threshold Decryption

We first visualize the ideal functionality that we wish to enable in this step, in Figure 3.

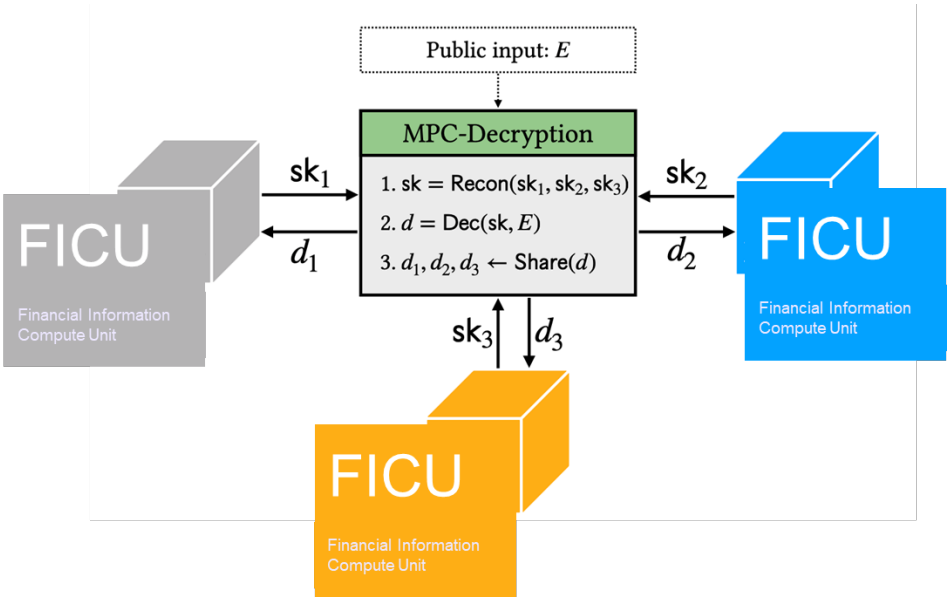

Figure 3: Multiparty Decryption.

The AA ecosystem, like most internet infrastructure, is not designed to integrate directly with MPC. The structure of the ciphertext E delivered by AA to the FIU is one such point of friction for decentralization. The encryption scheme follows a standard recipe:

- FIU's public key pk is a point on an elliptic curve, specifically Curve25519.
- The ciphertext E consists of a nonce $R = r \cdot G$, which is used to derive a shared value as per Diffie-Hellman: $K_{\mathsf{DH}} = (r \cdot \mathsf{sk}) \cdot G$.
- The above value is then hashed to derive a shared encryption key, $k = \mathsf{HKDF}(K_{\mathsf{DH}})$
- Data d is encrypted using a symmetric key cipher—AES256 in GCM mode for the AA setting—keyed by k as derived above.

One of the difficulties in designing an MPC protocol to process ciphertexts of this form lies in the use of cryptographic tools whose native modes of computation differ from each other. In particular, K_{DH} is very efficient to compute when operating with a secret sharing defined as per the elliptic curve group addition law, whereas HKDF and AES256 are best represented as Boolean circuits.

3.1 Generating pk

The first step towards decentralizing the FIU's decryption phase is to generate $\mathsf{pk} = \mathsf{sk} \cdot G$ in a distributed fashion, while keeping the secret key sk secret shared. This is quite straightforward, as has been done in a number of threshold signing and decryption protocols for elliptic curve based primitives. Roughly, parties jointly sample sk as a random secret shared $\mathbb{Z}_q$ value (where q is the order of the elliptic curve) and expose $\mathsf{pk} = \mathsf{sk} \cdot G$ by broadcasting their respective shares in the exponent, and performing reconstruction in the exponent.

3.2 Deriving Shares of K_{DH}

It is straightforward to derive shares of K_{DH} as per the group addition law, but the challenge lies in then converting these shares for the next phase of the computation, which is over a Boolean circuit.

Shares of K_{DH} in EC form. Given R and secret shares of sk, parties can simply exponentiate R by their respective shares of sk, in order to derive shares of K_{DH}.

Shares of K_{DH} in Boolean form. Borrowing ideas from the Oblivious TLS protocol of Abram et al. [ADST21], our approach is to translate shares of K_{DH} to a standard arithmetic sharing in the base field $\mathbb{Z}_p$ of the elliptic curve, and then convert these arithmetic shares to Boolean shares. As we operate in the 3-party 1-corruption setting (unlike Abram et al. who assume a dishonest majority), we are able to adapt the share conversion techniques of Mohassel and Rindal [MR18] for this purpose. In particular, we express the group addition law as a circuit in $\mathbb{Z}_p$, and use an arithmetic 3PC over $\mathbb{Z}_p$ to derive shares of K_{DH} as per the $\mathbb{Z}_p$ addition law applied to the input EC shares of K_{DH}. This circuit is quite compact, and involves only a handful of multiplications; the only slightly nontrivial arithmetic operation is field inversion, which can be performed efficiently via the Bar-Ilan Beaver trick [BIB89]. Subsequently, we tweak the arithmetic to Boolean conversion protocol of Mohassel and Rindal [MR18] to work over integers modulo a prime (rather than 2^ℓ) to convert the $\mathbb{Z}_p$ sharing of K_{DH} into a Boolean sharing.

3.3 Evaluating HKDF and AES256-GCM

Once Boolean shares of K_{DH} are available, we use the Boolean 3PC of Furukawa et al. [FLNW17] to derive shares of $k = \mathsf{HKDF}(K_{\mathsf{DH}})$. As this is a relatively deep Boolean circuit, we may consider using a Garbled Circuit based protocol to perform this evaluation rather than a secret sharing based approach, based on empirically observed performance in the future.

Decryption as per the GCM mode of operation for AES is highly parallelized; each block of AES has no dependency on the others, and therefore all blocks can be processed in parallel. When decrypting ciphertexts that are over a kilobyte in size, for most standard regimes of network latency, round complexity is unlikely to be the bottleneck. Therefore, secret sharing based Boolean 3PC is clearly

the most efficient paradigm for this task, and once more we employ the protocol of Furukawa et al. to complete decryption and derive standard Boolean secret shares of the data.

4 Secure Parsing

Typically, decrypted data will be represented using a text format, such as JSON or XML. These formats are structured, and we can assume that the field names are pre-determined. Nevertheless, this representation is not easy for a secure computation to work with directly. The main challenge is that the location of relevant fields is variable, dependent on fields values. Consider the following simple JSON string:

```
[{"name":"Alice", "age":21}, {"name":"Bob", "age":23}]
```

In this case, the position in the string at which the field for Alice's age is located (in this case the 24^{th} character) depends on the length of the name "Alice". Likewise, the position in the string at which the entry for "Bob" begins depends both on the length of Alice's name, and the order of magnitude of Alice's age. Therefore, revealing the locations at which field values are located would leak information about the field names themselves.

We would prefer to have data in an array, in which each field is padded to a maximum length. For instance:

Name	Age
`Alice---`	`21-`
`Bob-----`	`23-`

The main task needed to achieve this can be expressed by a functionality we refer to as SplitAndPad. This takes a secret-shared string that is delimited by a certain character (e.g. "{" or ",") and splits the string into an array of secret-shared segments, divided by the delimiter, each padded to a certain length. For instance this could be used in the above JSON string to split separate records (using delimiter "{"), to separate fields within a record (using delimiter ",") or to separate field names from values (using delimiter ":"). The protocol should take as input the maximum length of each delimiter-divided segment, and should reveal nothing except the number of delimiters/segments.

Surprisingly, our solution has a number of rounds which does not depend on the number of segments. We do this by making heavy use of conversions from Boolean to arithmetic share representations, local computation of cumulative sums and constant-round secure shuffles. First, we obtain a mapping from i to the index of the i^{th} delimiter. This is achieved by evaluating the equality circuit to check whether an item is a delimiter, converting this secret-shared Boolean value to an arithmetic representation and then computing the cumulative sum, which gives for each delimiter the number of delimiters which have occurred up to that point. Second, using this and cumulative sum techniques, we obtain for each non-delimiter character secret-shared values i and j such that the previous delimiter was the i^{th} delimiter and occurred j locations prior. This means that the given non-delimiter character should occur in the i^{th} segment, at position j

in that segment. Third, we create whitespace padding characters. We calculate the (secret-shared) length of each unpadded segment from the difference of the indexes of adjacent delimiters, which in turn allows us to calculate the amount of padding we need. Each needed padding character can then be assigned a segment and a location within that segment. Finally, we shuffle all non-delimiter characters and needed padding characters, reveal the segment and position to which each belongs (while keeping the character itself secret-shared) and relocate each secret-shared character accordingly.

5 Primitives for Query Execution

We assume that records are stored in a table, with different secret-shared fields. The number of records is potentially very large. In order to attain good performance, we therefore ensure that the round complexity of the MPC protocol does not depend on the number of records.

5.1 Subset Count

This calculates the number of records for which one of the fields satisfies a property that can be represented using a comparison. For instance "compute the number of orders that were in the category **Books**" or "compute the number of customers who were born before 1959".

The protocol first performs the comparison using a Boolean circuit. It then converts the output (1 if the statement holds, 0 if it doesn't) into an *arithmetic* sharing. This is important, because it allows the challenging aspect of combining data from all the records to occur without any communication. By converting to an arithmetic sharing, the sum of the predicates can be obtained using only *local operations*. Finally, these shares are sent to the user who can reconstruct the result.

Note that the conversions from Boolean to arithmetic sharings can be executed in parallel. Therefore the round complexity does not depend on the number of records.

5.2 Subset Sum

This assumes two different fields, one of which is a key used for filtering, and another of which is a value over which a sum is computed. For instance, the following queries are of this type: "compute the total spending of orders in the category **Books**" or "compute the total electricity consumed today by customers who joined within the past 5 months".

The protocol to achieve this is similar to that of Subset Sum above. The initial step is the same: a Boolean circuit is evaluated to filter records. However, rather than storing a "1" for records which match the filter, the protocol stores the desired value field (e.g. spending or electricity consumption), and stores a

"0" for any record which does not match the filter. Summing over these will give the subset sum.

5.3 Subset Average

Like Subset Sum above, but compute the average over the elements that match the filter. For instance "compute the average income of graduates who graduated between 2004 and 2007" or "compute the average age of customers who joined in the past month".

To achieve this, we first evaluate a subset sum and then evaluate a subset count. The average is computed as the sum divided by the count. This, in turn, needs a protocol for fixed-point division, which we can implement using standard a long division algorithm evaluated over Boolean circuits.

5.4 Sort

This is useful for finding the records that have the highest, or lowest, values for some particular field. For instance "return the 10 most expensive purchases by the given customer" or "return the 50 customers who have joined most recently". Unlike previous queries, the round complexity of this protocol does depend on the number of records, n. The (expected) round complexity is $\Theta(\log(n))$ and the total number of comparisons required is $\Theta(n \log(n))$. This performance is significantly better than a standard Oblivious sort which either requires $\Theta(n \log^2(n))$ comparisons [Bat68] with small constants or $\Theta(n \log(n))$ comparisons [AKS83] with enormous constants. This improvement is possible because we can first make use of a secure shuffle which randomly permutes the elements. Following this, we can use a *non-oblivious* sort, such as QuickSort, which reveals the relative ordering of indices in the permuted array. Since the array was first permuted, this reveals nothing about the relative ordering in the original array.

We make use of the honest-majority 3-party shuffle protocol of Laur et al. [LWZ11]. This securely shuffles items by shuffling the data 3 times, once by each pair of parties using a permutation unknown to the third party. We extend the protocol of Laur et al. to the malicious-security setting by using information-theoretic Message Authentication Codes (MACs).

5.5 Most Common Item

This is similar to the sort protocol above, but first we need to group by count. For instance, it could answer the query: "given an array of transactions, calculate on which day the most transactions occurred". We adopt the approach of the Vogue protocol [JKK+22]. First, we sort records according to the item field. Storing the index of the first record of each type, we sort again, placing all first occurrences at the beginning. The number of occurrences of a given item is equal to the the index of its first occurrence subtracted from the index of the first occurrence of the next item. Sorting by number of occurrences gives the desired result.

References

[ADST21] Damiano Abram, Ivan Damgård, Peter Scholl, and Sven Trieflinger. Oblivious TLS via multi-party computation. In Kenneth G. Paterson, editor, *CT-RSA 2021*, volume 12704 of *LNCS*, pages 51–74. Springer, Heidelberg, May 2021.

[AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *15th ACM STOC*, pages 1–9. ACM Press, April 1983.

[Bat68] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[BIB89] Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In Piotr Rudnicki, editor, *8th ACM PODC*, pages 201–209. ACM, August 1989.

[CDI05] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 342–362. Springer, Heidelberg, February 2005.

[FLNW17] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 225–255. Springer, Heidelberg, April / May 2017.

[GIP+14] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In David B. Shmoys, editor, *46th ACM STOC*, pages 495–504. ACM Press, May / June 2014.

[GIP15] Daniel Genkin, Yuval Ishai, and Antigoni Polychroniadou. Efficient multi-party computation: From passive to active security via secure SIMD circuits. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 721–741. Springer, Heidelberg, August 2015.

[JKK+22] Pranav Jangir, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Vogue: Faster computation of private heavy hitters. Cryptology ePrint Archive, Report 2022/1561, 2022. `https://eprint.iacr.org/2022/1561`.

[KKS+24] Yashvanth Kondi, Kush Kanwar, Siddharth Shetty, Anurag Arjun, and Jay Prakash. Open finance revisited: Strengthening data governance with cryptographic privacy and auditability. https://hackmd.io/p7uDvlbRQfaRZrHXpLSaMg, 2024.

[LN17] Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 259–276. ACM Press, October / November 2017.

[LWZ11] Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-efficient oblivious database manipulation. Cryptology ePrint Archive, Report 2011/429, 2011. https://eprint.iacr.org/2011/429.

[MR18] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 35–52. ACM Press, October 2018.

[Sah] Sahamati. Account aggregators. https://sahamati.org.in/account-aggregators/.