# Oblivious querying of FinTracer tag values

Michael Brand

November 23, 2023

**Executive Summary**

We describe a method for querying FinTracer tag values for an FIU-known, size-limited set of accounts that does not reveal private information to REs and does not reveal additional private information to the FIU.

## 1 Introduction

In the FinTracer algorithm, a central data type used is the *tag*, where a tag is a dictionary mapping from account identifiers (BSBs and account numbers) to semi-homomorphically encrypted values that are known as *tag values*. The tags are held by the reporting entities (REs), each RE's dictionary keys being accounts managed by that RE, and the semi-homomorphic encryption is such that its private key is held only by the FIU.

In the FinTracer algorithm itself, the only semantic information stored in tag values is a Boolean, encoded by whether the value is zero (signifying a "False") or a non-zero (signifying a "True"). In other applications, the full value may be meaningful.

Typical steps in the FinTracer protocol involve setting up tags, manipulating their values, and ultimately reading out the tag values of specific accounts in specific tags. Examples of these can be found in [1].

A key property of the FinTracer protocol is that the manipulation of tag values is *oblivious*, in the sense that the REs that hold the tag values and perform manipulation on them do not know which values they manipulate nor what the results are. Recently, in [2], a mechanism was described that allows the initial step, the setting up of tags, to also be performed in an oblivious way: even though it is the REs setting up the tags, only the FIU knows what the tag values are set up to be.

In this document, we add to our arsenal of FinTracer capabilities a new protocol, this one allowing oblivious *reading* of tag values. In other words, it allows the FIU to read tag values, as it normally would at the end of a FinTracer run, but without the REs gaining any information regarding which accounts' tag values have been read.

## 2 Privacy guarantees

As was the case in [2], the main challenge of the protocol is not the obliviousness it proffers (i.e., the fact that it hides information from the REs) but rather the fact that it maintains adequate information-hiding from the FIU itself. In order to enumerate the protocol's privacy guarantees, let us first introduce some terminology.

Let $\mathcal{B}$ be the set of all possible account IDs. This can be thought of, for example, as the set of all 15-digit strings, a space of size roughly $2^{50}$.

We assume that the FIU holds $B \subseteq \mathcal{B}$, a set of explicitly known account IDs regarding which the FIU wishes to determine their tag values for some tag $t$.

We assume that all accounts in $B$ exist and belong to a single RE. In other words, the protocol is a two-party protocol, with the FIU as one party and the chosen RE as the other. If the FIU

wishes to read tags from more than one RE, it can do so by invoking the bilateral protocol multiple times, each invocation with a separate RE.

Let $\tilde{B}$ be the set of accounts actually managed by the RE. Thus, if the FIU follows the protocol properly it will be true that $B \subseteq \tilde{B}$.

The algorithm begins with the FIU describing to the RE a set of accounts, $\hat{B} \subseteq \tilde{B}$, such that $B$ is a subset of $\hat{B}$ but such that the (necessarily implicit) description of $\hat{B}$ can safely be shared with the RE. Assuming $B \subseteq \tilde{B}$, such a set $\hat{B}$ can always be found, e.g. simply by using $\hat{B} = \tilde{B}$. In practice, the costs of the algorithm strongly depend on $|\hat{B}|$, so if smaller sets $\hat{B}$ can be safely chosen, that is preferred.

In any case, the algorithm does not protect the description of $\hat{B}$ from the REs. nor does it protect the approximate value of $|\hat{B}|$ from the FIU.

The algorithm does, however, confer the following safety guarantees.

1. The RE does not discover any information about the identity of accounts in $B$, other than $|B|$ and the fact that $B \subseteq \hat{B}$.

2. The REs *do*, necessarily, discover $|B|$, however, and can terminate the protocol if $|B|$ is deemed too large. If such termination occurs, the FIU should not learn any new information through the run of the algorithm. This condition limits the FIU's power in oblivious querying. Querying too many tag values is not allowed, whether the querying is oblivious or not.[1]

3. While the FIU does learn the tag value for accounts in $B$, it must not learn any information about tag values for any other account.

4. The FIU receives no protected information regarding accounts whose IDs are not in $B$, including whether such IDs correspond to accounts managed by the RE. (The FIU does receive the approximate value of $|\hat{B}|$ in the form of an upper bound.)

Additionally, through a separate part of the algorithm we refer to as the "honesty check", the algorithm guarantees to the RE that $B \subseteq \hat{B}$. This is important because it prevents the FIU from "fishing" for RE accounts; the algorithm has been designed so that the FIU only ever uses it to read the tag values of accounts that it knows in advance belong to the RE. Otherwise, it would have been able to use this protocol in order to determine whether particular accounts exist.

With the "honesty check" activated, any attempt by the FIU to query about nonexisting accounts will result in the following:

1. An alert will be raised on both sides,

2. The algorithm will be terminated, and

3. No further results will be returned from the algorithm to either party.

In [2] (Sections 5.2, 5.3 and 6) we discuss extensively the privacy implications of having/not having such an honesty check. This important discussion will not be reiterated here.

# 3 Mathematical tools

We begin describing the algorithm by presenting some of the basic underlying maths that we will use.

[1] In this algorithm the RE discovers the exact size of $B$, not just an upper bound. If the FIU wants to obscure the exact number of accounts being queried, it can do so by running this algorithm twice. The RE will learn the number of accounts queried in each instance, but not the amount of overlap between queries.

## 3.1 ElGamal

### 3.1.1 Encoding vs encrypting

The tag values used in FinTracer are encrypted using ElGamal over the Ed25519 additive group. This is a semi-homomorphic, public key encryption system. Using it involves at least three steps:

1. A plaintext message, $M$, needs to be *encoded* into an Ed25519 element, $m$. We denote this by $m = \text{Ed25519}(M)$.

2. The encoded message gets *encrypted* into a ciphertext, $c$, using a public key $(g, h)$. We denote this $c = \text{Enc}_{(g,h)}(m)$. It may also be encrypted using a private key, $x$, in which case we write $c = \text{Enc}_x(m)$. A private key $x$ matches a public key $(g, h)$ if $h = gx$. In this case, encrypting through either the private or public keys leads to equivalent results. The resulting ciphertext, $c$, is not an element of Ed25519, but rather a pair $(a, b)$ of such elements.

3. The encrypted message gets *decrypted* into an encoded message using a private key, $x$. We denote this $m' = \text{Dec}_x(c)$.

Importantly, the Ed25519 additive group does not allow "decoding". Essentially, the only way to effectively read the decrypted message, $m'$, is to test for equality against a known candidate message, $M$, e.g. by verifying that $m' = \text{Ed25519}(M)$.

This property of the Ed25519 additive group often leads to discounting the potential size of the plaintext space. In previous docucments we have stated that "[the plaintext] is limited largely by our ability to perform discrete log". In this document, however, we will be using the full range of the group's elements, utilising algorithms that do not rely on brute-force decoding. This being the case, we take the space of our "plaintext" to be $\mathbb{Z}/p\mathbb{Z}$, where $p$ is the size of the Ed25519 group, which is a prime number, approximately $2^{252}$ in size.

### 3.1.2 Homomorphic and module operations

ElGamal, being a semi-homomorphic encryption scheme, allows the user to sum encrypted elements. In other words, given two plaintexts, $M_1$ and $M_2$, encoded into $m_1$ and $m_2$, respectively, and encrypted using the same key into $c_1$ and $c_2$, respectively, it is possible to sum these together in all three settings: $M = M_1 + M_2$, $m = m_1 + m_2$ and $c = c_1 + c_2$, and the results will be equivalent in the sense that the decryption of $c$ is the encoding of $M$ is $m$.

Importantly, however, in $M = M_1 + M_2$ this is addition in $\mathbb{Z}/p\mathbb{Z}$, in $m = m_1 + m_2$ this is addition in Ed25519, and in $c = c_1 + c_2$ this is addition in the ciphertext group.

In short, Ed25519 is isomorphic to the additive group of $\mathbb{Z}/p\mathbb{Z}$. Moreover, because $\mathbb{Z}/p\mathbb{Z}$ is a field, it also defines a multiplication function. We can extend the isomorphism of $\mathbb{Z}/p\mathbb{Z}$ to Ed25519, to also define a new product function on Ed25519, making Ed25519 into a field. This product is not feasible to compute over the Ed25519 elements directly, but we can compute it for products of the form $a \cdot x$, where $x$ is an Ed25519 member and $a$ is a $\mathbb{Z}/p\mathbb{Z}$ member. The way to do this is to perform multiplication by repeated addition: $x + \cdots + x$, repeated $a$ times (where $a$ is treated as an integer in the canonical way, by identifying it with the unique integer in $0, \ldots, p-1$ that is in its $\mathbb{Z}/p\mathbb{Z}$ residue class).

**Note 3.1.** A better algorithm is to use repeated doubling, in which case the complexity of computing $a \cdot x$ becomes $\text{O}(\log a)$ additions. The result is the same in either case.

The product, in this case, is derived as a member of Ed25519. It is similarly possible to compute $a \cdot x$ where $a$ is a $\mathbb{Z}/p\mathbb{Z}$ member and $x$ is a ciphertext. In this case, the result will be a ciphertext.

Mathematically speaking, one can treat both Ed25519 and the ciphertext space as $\mathbb{Z}/p\mathbb{Z}$-modules. This allows us to compute on them not only products, but any linear function whose coefficients are plaintexts.

### 3.1.3 Security functions

In addition to the mathematical operations described in the previous section, we will use two convenience functions for our ciphertexts.

1. The function "refresh($x$)" takes a ciphertext $x$ and adds to it a newly encrypted zero. The result is a new ciphertext that has the same decryption as the original $x$, but is uniformly distributed among all ciphertexts leading to the same decryption. Effectively, this ensures that the new ciphertext is one that has never been used.

2. The function "sanitise($x$)" multiplies $x$ by a random, uniformly-chosen nonzero element of $\mathbb{Z}/p\mathbb{Z}$. The effect on the plaintext value of $x$ is that if it was zero the result also has the value zero, whereas if it is nonzero the result is uniformly distributed among all nonzero numbers. This function is useful for erasing all information in $\mathrm{Dec}(x)$ other than whether it was originally zero or not.

## 3.2 Polynomial calculation

Consider a polynomial, $C$, of some degree $\mathrm{Deg}(C) = k$. We will use $C[i]$ to denote its $i$'th coefficient, taking $C[i]$ to be zero for $i > k$.

Mathematically, this polynomial can be thought of as $\sum_{i=0}^{k} C[i]X^i$ over a formal variable $X$, but in computing terms we will simply store $C$ as a vector of coefficients. If $C$ is known to be monic, i.e. $C[k]$ is known to equal 1, it will not be necessary to store or transmit $C[k]$, and the polynomial will be fully defined by the $k$ coefficients $C[0], \ldots, C[k-1]$.

There are many operations commonly done on polynomials, and a surprising number of them can be used when the coefficients are ciphertexts. Here are the ones we will use in this document.

### 3.2.1 Evaluation

Given a plaintext value $x$, we can evaluate $C(x)$, i.e. $C$ at $X = x$. This is possible because the computation of $x^i$ can be performed in plaintext, and the remaining computation is linear.

### 3.2.2 Summation

Encrypted polynomials can be summed. Computing $C = C_1 + C_2$ is done by $\forall i, C[i] = C_1[i] + C_2[i]$.

### 3.2.3 Multiplication by a scalar

An encrypted polynomial $C$ can be multiplied by a plaintext scalar $a$. The result, $D$, has coefficients $D[i] = aC[i]$.

### 3.2.4 Multiplication by a plaintext polynomial

An encrypted polynomial $C$ can be multiplied by a plaintext polynomial $P$. The result, $D$, has coefficients $D[i] = \sum_{j=0}^{i} P[i-j]C[j]$, each of which is a linear function in the ciphertext.

### 3.2.5 Division and Residual modulo a plaintext polynomial

Let $C$ be an encrypted polynomial and $P$ be a plaintext polynomial which we will assume, without loss of generality, to be monic. Consider how one would compute the quotient polynomial $D$ and the remainder polynomial $R$ when dividing $C$ by $P$. Algorithm 1 shows how these can be computed using standard long division.

This algorithm is not data-parallel, and so is not best suited for, e.g., GPU implementation, but it proves that the values of both $D$ and $R$ are linear in $C$. Thus, $D$ and $R$ can be computed on encrypted $C$ dividend polynomials. The dependence on the divisor, $P$, is, however, not linear.

**Algorithm 1** Long division: $C/P$

```
▷ P is assumed to be monic.
C' ← C.
for i from Deg(C) downto Deg(P) do
    D[i − Deg(P)] ← C'[i]
    for j ∈ [1, Deg(P)] do
        C'[i − j] ← C'[i − j] − C'[i]P[Deg(P) − j]
    end for
end for
for i ∈ [0, Deg(P) − 1] do
    R[i] ← C'[i]
end for
Return the quotient D and remainder R.
```

In this document, we will only be interested in calculating $R$. The degree of $R$ is at most $\mathrm{Deg}(P)-1$ and each of its coefficients can be computed as a linear combination of the $\mathrm{Deg}(C)+1$ coefficients of $C$. For an efficient implementation, we recommend to first compute, in plaintext, all coefficients for all $\mathrm{Deg}(P)-1$ linear combinations, and then to perform all approximately $\mathrm{Deg}(P)\times\mathrm{Deg}(C)$ products and summations in parallel.

#### 3.2.6 Random polynomial

Though not an operation on encrypted polynomials, it is nevertheless important to point out that given a nonnegative integer $k$ we can generate a random polynomial of maximum degree $k$, uniformly among all such polynomials, simply by allotting each of its $k+1$ coefficients from a uniform distribution, independently.

We will typically use this to generate random plaintext polynomials. Such an operation will be denoted by $R \leftarrow \mathrm{UPoly}(k)$. We will refer to polynomials taken from this distribution as *uniformly-distributed $k$-polynomials*.

# 4 The algorithm

## 4.1 Preliminaries

As mentioned, one item that the algorithm does not keep private is the approximate size of $\hat{B}$. In fact, this is communicated from the RE to the FIU explicitly, in the form of a parameter, $S$, that serves as an upper bound for $|\hat{B}|$, and that determines the overall communication and computation costs of the algorithm.

We use the method described in Section 3 of [2] in order to choose $S$ so as to ensure that communicating the approximate size does not in itself reveal protected private information.

Throughout the algorithm, we will treat $\hat{B}$ as having size $S$ exactly, doing so by appending the appropriate number of random elements to it. This is important to do, as it avoids potential non-data attacks that may try to determine the true size of $\hat{B}$ by measuring the time it takes for the RE to compute replies, rather than by the stated $S$. (The values of $t$ assigned to the fictitious added elements can be arbitrary.)

For the purpose of this algorithm, account identifiers are mapped into $\mathbb{Z}/p\mathbb{Z}$, so sets like $\hat{B}$ are subsets of the plaintext space. Random values used in this document, such as those appended to $\hat{B}$, are uniformly-chosen elements in this group. Note that the size of the plaintext space is roughly $2^{252}$, which is much larger than $2^{50}$ which is the entire potential size of account identifiers (which, in turn, is much larger than the approximately $2^{27}$ actual existing accounts). Because of this, it can be safely assumed than no collisions will occur between true account identifiers and random values. This is a property we will make repeated use of, throughout the algorithm.

## 4.2 The core algorithm

Let $t$ be a tag for which the FIU wants to read the values of $t[b]$, for all $b \in B$. We assume that the FIU knows that $B$ is a subset of $\tilde{B}$, the set of accounts managed by the RE, and also that $B \subseteq \hat{B}$, where $\hat{B}$ is defined by a description to the RE.

The core algorithm for oblivious querying proceeds as follows.

**Note 4.1.** Throughout both the core algorithm and the honesty check, assume that any communicated encrypted value is refreshed before it is transmitted.

1. The FIU sends to the RE $k = |B|$ and the description of $\hat{B}$.
2. The RE verifies that $k$ is not above a predetermined threshold, and halts if it is.
3. The FIU computes the monic polynomial $\prod_{b \in B}(X - b)$, encrypts its non-trivial coefficients and sends these, as an encrypted polynomial $C$, to the RE.
4. The RE verifies that it received exactly $k$ coefficients and halts if not.
5. The RE sets $C[k]$ to be (an encrypted) 1, to make sure the polynomial is monic and full degree.
6. The RE determines $\hat{B}$, computes $S$, and returns the value of $S$ to the FIU. It also appends $\hat{B}$ with random elements to size $S$.
7. If an honesty check is required (See Section 4.3), it is performed now.
8. For each $b \in \hat{B}$, the RE now computes the polynomial's value at $b$. Let the result be $C(b)$.
9. For each $b \in \hat{B}$, let $(x, y) = (b + C_1(b), t[b] + C_2(b))$, where $C_1$ and $C_2$ are two separate sanitisations of the $C(\cdot)$ values. The RE computes these and sends to the FIU the $S$ $(x, y)$ pairs under a random order permutation.
10. After decryption, the $(x, y)$ pairs are Ed25519 members. The FIU can now map all member $b$ of $B$ to Ed25519 in order to find those $(x, y)$ pairs where $x = \text{Ed25519}(b)$. This provides the FIU with the value of $t[b]$.[2]

## 4.3 Honesty check

In a proper execution of this protocol, the FIU is supposed to only query for the tag values of accounts $b$ for which it knows in advance that $b \in \hat{B}$. The following sub-protocol allows the RE to verify that no $b \in B$ violates this condition.

Mathematically, the RE now has the encrypted coefficients of a polynomial $C$ of degree $k$, and it wishes to make sure that all $k$ of its roots are in $\hat{B}$.

1. If $k > S$, the RE halts and alerts both sides.[3]
2. The RE computes the polynomial $P_{\hat{B}} = \prod_{b \in \hat{B}}(X - b)$. This is a monic polynomial of degree $S$ that the RE can compute in plaintext.
3. The RE generates two random polynomials, $R_1 \leftarrow \text{UPoly}(k - 1)$ and $R_2 \leftarrow \text{UPoly}(S - k)$, as well as a uniform random $\mathbb{Z}/p\mathbb{Z}$ scalar, $s$.
4. The RE computes $C' \leftarrow sP_{\hat{B}} + R_2C + R_1$ (which it can only do in encrypted form) and transmits it to the FIU.

[2] Note that while $t[b]$ is provided to the FIU as an Ed25519 member, not as an integer, this is no different to how the FIU would have received the information had the query not been oblivious.

[3] For the protocol to not be revealing, one should set $S \gg k$. That, however, is the FIU's responsibility in defining $\hat{B}$. The protocol does not enforce this.

5. Let $P_B$ be the (plaintext) polynomial $\prod_{b \in B}(X - b)$, which the FIU has previously computed. The FIU now computes $R$ as the remainder in $C'$ modulo $P_B$ and returns it to the RE in encrypted form.

6. The RE now computes $V \leftarrow R - R_1$.

7. If the FIU attempts to query values outside of $\hat{B}$, $V$ will be nonzero. Otherwise it will be zero. We can now use the same method as was used in the honesty check of [2] to make sure this syndrome is zero, as expected. If it is not, the algorithm halts and alerts are sent to both sides.

**Note 4.2.** It is, in fact, not necessary to check all coefficients of $V$. Testing whether $V[k-1] = 0$ is sufficient.

**Note 4.3.** When computing $R$, it is tempting to believe that $C'$ modulo $P_B$ can be computed in plaintext. This is not true, however, because while the FIU can decrypt $C'$, it cannot decode its coefficients. At best, they will be Ed25519 group elements. Calculating $R$ can be done either over an encrypted $C'$ or over its Ed25519-encoded version, with only the final result then returned in encrypted form. To determine which variant is best in a given situation, note that calculations in Ed25519 are typically twice as fast as calculations over encrypted values, but on the other hand decrypting the $S$ elements of $C'$ introduces a significant time overhead because decryption is slow. The tradeoff can therefore favour either variant.

**Note 4.4.** Note that the FIU receives the value of $S$ even if the honesty check fails.

# 5 Security analysis

There are three types of potential security issues this algorithm needs to address.

1. The RE must not discover any new information other than $k$ and the definition of $\hat{B}$.

2. An honest-but-curious the FIU must not discover privacy-protected information on non-target accounts.

3. An FIU departing from the protocol must not discover privacy-protected information on non-target accounts.

We will consider these in turn.

## 5.1 Leakage to the RE

In both the core algorithm and the honesty check, all the FIU transmits to the RE other than $k$ and the description of $\hat{B}$ are fresh encrypted values in an amount that is a function of $k$ and of the value $S$ (which, in turn, is a value determined by the RE itself in earlier stages). Thus, no information leakage to the RE is possible unless there is a weakness in the encryption algorithm.

## 5.2 Leakage to an honest FIU

In the core algorithm, the FIU receives from the RE the value of $S$. In [2] it is discussed how the particular choice of $S$ avoids revealing protected information.

Other than this, in the core algorithm the FIU only receives a set of $S$ pairs of the form $(x, y) = (b + C_1(b), t[b] + C_2(b))$. By construction, for $b \in B$ (and only for these), the value of $C(b)$ will be zero, making the FIU receive for such $b$ the desired $(b, t[b])$ pairs. The value of all other pairs is random, and therefore contains no protected information.

In the honesty check, the FIU receives from the RE the polynomial $C' = sP_{\hat{B}} + R_2C + R_1$. Even assuming that the FIU can fully decode this polynomial (which it cannot; it can only decrypt

its coefficients), this polynomial contains no new information: it is a uniformly-distributed $S$-polynomial, whose value is independent of any protected information.

To prove this, consider that the $i$'th coefficient of $R_2C$ is random, independent and uniformly distributed for $i \in [k, S]$, whereas the $i$'th coefficient of $R_1$ is random, independent and uniformly distributed for $i < k$. Their sum is therefore a uniformly-distributed $S$-polynomial. This is a property that is preserved when adding further (independent) components such as $sP_{\hat{B}}$.

Additionally, we can prove that when the parties are honest the value of $V$ will be zero. The reason for this is that if the FIU is honest then $P_{\hat{B}}$ is a multiple of $P_B$. The value of $C'$ is therefore of the form $RP_B + R_1$, where by construction $\mathrm{Deg}(P_B) > \mathrm{Deg}(R_1)$. The FIU calculates $C'$ modulo $P_B$, which returns $R_1$, and the RE subtracts $R_1$ to reach zero.

### 5.3 Leakage to an FIU who departs from the protocol

Throughout the algorithm, the RE sends to the FIU only the following: $S$, $C'$, and the $(x, y)$ pairs. The value of $S$ reveals information to the FIU, but we have designed $S$ to protect private information under an $(\epsilon, \delta)$-differential privacy assumption. We have already shown that $C'$ contains no information. Finally, by design, the $(x, y)$ pairs are only meaningful for $b$ where $C(b) = 0$. By design, $C$ is a polynomial of degree $k$, so the FIU cannot glean the values of more than $k$ $(x, y)$ pairs.

It remains to be seen that if the FIU attempts sending to the RE a polynomial $C$ that has zeroes outside $\hat{B}$ (a fishing attempt), this will be caught by the honesty check.

To see this, consider that $C'$, which equals $sP_{\hat{B}} + R_2C + R_1$, has the same distribution as $R \cdot G + R_1$, where $R$ is a uniformly-distributed $(S - \mathrm{Deg}(G))$-polynomial and $G$ is the unique monic polynomial that is the GCD of $P_{\hat{B}}$ and $C$.

If $C$ contains zeroes outside $P_{\hat{B}}$, $G$ will be of degree lower than $k = \mathrm{Deg}(C)$. As such, without any knowledge of $R$, retrieving $R_1$ from $C'$ and $G$ can only be done up to a residue class. Any polynomial $P \cdot G + R_1$ where $P$ is of maximum degree $k - 1 - \mathrm{Deg}(G)$ is an equally likely candidate.

Not only will an FIU who departs from the protocol not be able to properly reconstruct $R_1$, it will have no information (maximum entropy) in any attempt to guess $R_1[k - 1]$.

## 6 Computational considerations

### 6.1 Complexity

In the core algorithm, the FIU sends to the RE $k$ ciphertexts and the RE sends to the FIU $2S$ ciphertexts, with all other communication being negligible. In terms of communication complexity, this is best possible.

In the honesty check, the same communication amounts are used, albeit in the opposite directions.

In terms of computational complexity, the heaviest portions of the algorithm involve $\mathrm{O}(kS)$ plain-by-cipher products. Though we cannot reduce this complexity, the next section discusses how to compute the products themselves more efficiently.

### 6.2 Product computation

Consider a situation where one wishes to multiply a $\mathbb{Z}/p\mathbb{Z}$ plaintext value, $a$, by a value $x$ that is either encoded or encrypted.

The brute-force method for doing so is by repeated summation, which we've shown works in $\mathrm{O}(\log a)$ if implemented as a sequential doubling process.

In the honesty check, the calculation of $C'$ modulo $P_B$ involves roughly $kS$ such computations, which, unfortunately, cannot be sped up by much. The best we can do regarding this calculation is

1. To implement it using a GPU-parallelisable algorithm, and not by long division, and

2. To compute it using products between plaintexts and encoded values, not between plaintexts and encrypted values and by this save a factor of 2 on performance.

Both these possibilities have been discussed in detail in previous sections.

For other heavy operations used throughout the algorithm, there is, however, a better solution.

Specifically, for both the polynomial evaluation used in the core algorithm and for the product $R_2C$ used in the honesty check, both of which require a total of $\mathrm{O}(Sk)$ products, we can speed up multiplication by taking into account that only $k$ distinct encrypted values are being used.

To utilise this fact, we perform preliminary work that allows us to speed up later multiplications.

Suppose $x$ is an encrypted or encoded value for which we wish to compute many products of the form $ax$.

Let $t$ be a parameter to be optimised later and let $w = \lceil \log_t p \rceil$, where $p$, as usual, is the size of the Ed25519 group.

It takes $tw$ summations to calculate $bx$ for all values $b$ lower than $p$ of the form $rt^i$, where $r < t$. These $bx$ values can be precalculated and stored.

When computing $ax$, one merely computes the base $t$ representation of $a$, e.g. $a = a_i t^i + \cdots a_0$ (easiest to do when $t$ is a power of 2), after which computing $ax$ can be done using $w$ summations: $ax = \sum_i a_i t^i x$.

In the actual algorithm, we have $k$ such frequently-used $x$ values, each of which is multiplied by $S$ plaintext values. For each such $x$, we require $tw$ summations for the precalculation and $Sw$ summations to compute all products. We choose $t$ so as to minimise $tw + Sw$.

The optimum $t$ is at $S = (t \log t) - t$. This places $t$ at approximately $S/\log S$. For the maximal $S$ of $2^{25}$, the value of the optimal $t$ is 2447350, leading to $tw$, the number of ciphertexts that need to be precalculated and stored to be under 30 million (for a total storage size of roughly 7GB) and the total number of summations required to be $tw + Sw \approx 432$ million (amounting to around 156 CPU seconds, or 1.9 GPU seconds).

# 7 Vulnerabilities

As discussed in [2], the following are attacks the algorithm does not protect against:

1. The RE may not use its real $t$ in responses. This does not cause data leakage, but theorectically allows an RE to mislead the FIU.

2. Both the RE and the FIU can use the existence of an honesty check (if it is used) in order to derive information covertly by means of a "Guess Correctly or Get Detected" strategy. (If an honesty check is not used, the RE cannot illicitly derive information, but the FIU would be able to fish for RE accounts without being detected.)

# References


[1] Brand, M. (2022), *The Complete, Authorised and Definitive FinTracer Compendium.* (Available as: `FT_combined.pdf`)

[2] Brand, M. (2020), Oblivious setting of source accounts in FinTracer. (Available as: `oblivious_a.pdf`)