Fast privacy-preserving edge-finding, SoNaR, and their uses in FinTracer pair problems

Michael Brand

November 23, 2023

Executive Summary

We describe a new algorithm for the FinTracer privacy-preserving algorithm suite, designed for the finding of edges in a directed graph. The new algorithm works in only a constant number of message passing rounds, making it by far more efficient than any previously-proposed solution to this problem. We demonstrate how the new algorithm can be leveraged to significantly improve classic FinTracer solutions to problems like path finding, pair finding, connectivity and bridging. For path finding, we use the classic FinTracer privacy model. For the remaining problems, we describe a new model, with slightly weaker privacy assumptions, which we refer to as the "semi-oblivious narrowing runs" (or SoNaR) model.

1 Introduction

The FinTracer algorithm is presently the core of the Purgles Platform solution for privacy-preserving intel investigation of financial transactions. It became so after superseding the earlier algorithm, Asterisk.

The difference between these algorithms has previously been explained in [5] as follows:

> "[W]hile both [FinTracer and Asterisk] ostensibly address the same problem, of examining which accounts in a target set B are connected to accounts in a source set A via edges of a directed graph G, Asterisk also provides the information of which specific element of A is connected to which specific elements of B.
>
> FinTracer, by omitting this extra information, was able to use much faster algorithms, but the price is that for those wishing to find the extra connectivity information there is no direct solution."

The problem of finding which $a \in A$ connects to which $b \in B$ is known as the *pair-finding problem*. It belongs to a family of problems that present a challenge to the FinTracer approach because the answer in each case is not a set of accounts. In normal FinTracer propagation, results are always sets of "accounts of interest". For example, the set of all members of B that can be reached from any member of A is such a set. However, in pair finding, we are looking for a set of (a, b) pairs, rather than a and b separately.

In the FinTracer system, computation runs over a distributed system, in which one node, designated the *coordinator node* is run by AUSTRAC, and the other nodes, known as *peer nodes* are operated by AUSTRAC's reporting entities (REs). In typical use, AUSTRAC collaborates with REs to decide on the details of the protocol to run. (We refer to such a protocol as a *query*.) The query then runs in a distributed fashion, and results are ultimately returned to AUSTRAC.

The directed graph, G, over which FinTracer runs, is typically a graph whose vertices, $V(G)$, are accounts managed by the peer-node-running REs. We will therefore refer to the vertices of G as *accounts*, and for each account, x, we denote by $\mathrm{RE}(x)$ the RE managing x.

Although the FinTracer algorithm suite includes queries in which the REs are privy to neither the query's specifics nor to the query's ultimate results [8, 9, 11], there is presently no intention to

deploy such "oblivious" algorithms in practice. The main line algorithms in the FinTracer suite are designed in such a way that all final answers revealed to AUSTRAC are also revealed to the REs.

Not every RE receives the full answer, however. Rather, the query results of FinTracer algorithms are typically expressed as a union of subsets, one subset for each RE, where the portion assigned to an RE is the portion of the result that pertains to accounts managed by that RE. The result fragment assigned to each RE contains, however, no information regarding accounts managed by other REs. In a main-line FinTracer algorithm, each RE only sees, in the end, their portion of the result, whereas AUSTRAC sees the entire result set.

We refer to the family of algorithms whose results cannot be partitioned in this way as *pair problems*. By definition, they pose a special challenge for the FinTracer approach.

In this paper, we address three problems that belong to this family:

Edge finding: Given a set A and a set B, determine which pairs (a, b) with $a \in A$ and $b \in B$ are edges in G.

Path finding: Given a set A and a set B, find all of the shortest paths (or all of the walks of length up to k) from any $a \in A$ to any $b \in B$.

Pair finding: As described above, given a set A and a set B, find all pairs (a, b) with $a \in A$ and $b \in B$ such that there exists a directed path on G from a to b.

Additionally, we present solutions to the following other problems, which, although not technically pair problems, we show can be tackled through the same tools that we develop for pair problems. These are:

Connectivity: How many elements from A have a path leading to each $b \in B$?

Bridging: For every vertex $v \in V(G)$, how many shortest paths from A to B travel through v? What is the probability of a random walker traversing these paths to go through v on its way from A to B?

These and similar are metrics that are useful in intel investigations because they can be used as proxies for an account-holder's relative importance in a particular criminal typology.

How the policy that all final query results need to be revealed to the REs gets translated to pair problems is a matter of definition. For the purpose of this document, however, we will say that if any information about the pair (a, b) is revealed to AUSTRAC, i.e. that the pair matches a particular condition, the fact that $(a, *)$ matches the condition will be shared with RE(a) but with the identity of b redacted, and the fact that $(*, b)$ matches the condition will be shared with RE(b) but with the identity of a redacted.

2 The problem setup

The FinTracer algorithm was originally described, as quoted here from [5], as an algorithm that works over a directed graph G. It gets as inputs two subsets of G's vertices, $A, B \subseteq V(G)$, as well as a natural k, and must determine which elements in B have a path on G leading to them from A that is of length at most k.

For any element $a \in V(G)$, the algorithm assumed that RE(a) is aware of the existence of a and all of its properties (such as membership in A and B), and for any two elements $a, b \in V(G)$, we assumed that RE(a) and RE(b) are both aware of whether $(a, b) \in E(G)$.

More modern incarnations of the FinTracer algorithm, such as the one described in [12], work in more complicated environments. Instead of assuming that RE(a) and RE(b) are each aware of whether $(a, b) \in E(G)$, where G is the propagation graph, we introduce a new graph, G_2, being the *two-sided viewable* propagation graph, and each RE, R, has their own $E_R \subseteq E(G_2)$, composed

entirely of edges (a, b) such that either $R = \mathrm{RE}(a)$ or $R = \mathrm{RE}(b)$, which are the edges they are actually willing to use in paths. Thus, the true propagation graph, G, can now be described as

$$E(G) = \{(a, b) \in E(G_2) | (a, b) \in E_{\mathrm{RE}(a)} \cap E_{\mathrm{RE}(b)}\}.$$

In other words, each RE may apply for any edge $(a, b) \in E(G_2)$ additional pruning criteria, and it is no longer true that both $\mathrm{RE}(a)$ and $\mathrm{RE}(b)$ have full knowledge of whether $(a, b) \in E(G)$.

3 The privacy model

The concept of a FinTracer *narrowing run* is essentially a means to separate *intermediate results* from *final results*. A FinTracer final result is one that is revealed to AUSTRAC as well as (in partitioned form) to the REs.

However, in order to support modular query construction (i.e., the use of multiple base queries in order to define a more complicated, more case-specific query to be run), FinTracer can also produce intermediate results: query results that are stored obliviously, whose contents are not known to any party, but can be used in a follow-up query.

A typical example would be a query whose result is a FinTracer tag: this is information obliviously stored by the REs that is equivalent to the definition of a set $S \subseteq V(G)$. This set can then be used in subsequent queries, for example as the source account set or as the destination account set.

The key point here is that the information from intermediate results in narrowing runs is not revealed to any party. This is in contrast to simply having a sequence of queries that are follow-up queries to one another.

For example, FinTracer algorithms currently do not allow one to define as one's set of destination accounts, B, the set of all accounts whose overall cash flow is in the top quartile of any account in the entire market, because this is not information that individual REs possess. Instead, a FinTracer user can first use, say, unbounded binary search, in order to determine the 75% statistic for cash flow, s_{75}, and then to define B as those accounts whose cash flow is at least s_{75}. In this case, the answer to the first query, "How much is s_{75}?", must be made public to both AUSTRAC and the REs before it can be utilised. Such statistics could be commercially sensitive in some cases, however.[1]

We have therefore covered three models for continued querying:

Query cascade: Query results are visible to both AUSTRAC and the REs.

Narrowing run: Query results are revealed to neither AUSTRAC nor the REs.

Oblivious query: Results are visible to AUSTRAC but not to the REs (a mode we do not plan to make use of in actual FinTracer deployment).

This still leaves one more option, namely the option where results are visible to the REs but not to AUSTRAC. We refer to this option as the *semi-oblivious narrowing run* (or *SoNaR*) model.

The edge-finding algorithm introduced in this document, by itself, does not necessitate results to be made available to either side, so can be thought of as an algorithm fit for use in narrowing runs (i.e., able to produce intermediate results). We will, however, analyse potential uses for it in both the query cascade model and in SoNaR.

As we shall see, the ability to reveal intermediate results to REs is extremely powerful. It allows REs to run more complex algorithms and run deeper analysis, because the intermediate results allow them to focus on accounts of interest, rather than continue analysing, obliviously, the entire financial market.

[1] From a statistical point of view, it actually makes more sense to have each RE set B according to their own, individual s^R_{75}. Not only will this be more privacy preserving, it makes for better statistics because accounts at the different REs are unlikely to be a homogeneous population.

We leave as an open policy question in this document whether SoNaR is a reasonable model to use in practice.

A detailed discussion of the model, and possible mitigation strategies in order to avoid its pitfalls (including the risk of debanking) is presented in Section 6.4.

4 The edge-finding algorithm

All algorithms presented in this document are applications of variations of a single basic edge-finding algorithm.

We begin by describing this edge-finding algorithm in its purest form.

4.1 The problem definition

Recall the definition of a FinTracer propagation graph. We assume the existence of sets of accounts, V, where each account x is managed by an RE, $\mathrm{RE}(x)$, and is known to that RE. Define $V_R = \{x \in V | R = \mathrm{RE}(x)\}$.

We assume the existence of a directed graph G_2, with $V(G_2) = V$, satisfying the criterion of two-sided visibility, i.e. each of $\mathrm{RE}(x)$ and $\mathrm{RE}(y)$ must always know whether $(x, y) \in E(G_2)$. Next we assume that each RE, R, knows a set of edges, $E_R \subseteq E(G_2)$, such that if $(x, y) \in E_R$ then either $R = \mathrm{RE}(x)$ or $R = \mathrm{RE}(y)$ (and possibly both).

We define the FinTracer propagation graph G as the directed graph with $V(G) = V(G_2)$ and

$$E(G) = \{(a, b) \in E(G_2) | (a, b) \in E_{\mathrm{RE}(a)} \cap E_{\mathrm{RE}(b)}\}.$$

The overall purpose of the edge-finding algorithm is as follows. Given $A, B \subseteq V(G)$, determine for which pairs $(a, b) \subseteq A \times B$ is $(a, b) \in E(G)$, but without any RE discovering more information about G because of this, and without AUSTRAC discovering anything more about G (other than general statistics) except for the query's result.

When used as a final query (i.e., when results are meant to be reported to AUSTRAC), we expect this algorithm to apply policy constraints, such as, for example, limiting the sizes $|A|$ and $|B|$ over which it is willing to run. Without such limitations, AUSTRAC can simply query $A = B = V$ and get the entire information of all G, which would not be privacy preserving. (In a model where results are not reported to AUSTRAC, such limitations are not needed; only limits that will constrain the size of whatever final answer is reported to AUSTRAC are required.)

In order to simplify the presentation of the edge-finding algorithm, we will assume that there are two REs, R_1 and R_2, with $R_1 \neq R_2$, such that $A \subseteq V_{R_1}$ and $B \subseteq V_{R_2}$. In practice, to solve the problem for general A and B we partition each of A and B into their respective V_R components, and then solve for each pair of parts separately. This is a process that can be performed in parallel.

In order to deal with those pairs for which A and B are composed of accounts all managed by the same RE, i.e. when $R_1 = R_2$, no privacy-preserving protocol is required: the RE can simply report the set of edges directly to AUSTRAC.

Thus, the problem we have simplified edge finding to is the following: given two disjoint sets, V_1 and V_2, such that each V_i is known only to its respective R_i, and given two sets of pairs $E_1, E_2 \subseteq V_1 \times V_2$, again known to R_1 and R_2, respectively, return to AUSTRAC the value $E_1 \cap E_2$, without this adding any other information to any of the three parties involved in the computation.

Note 4.1. Readers familiar with the wider literature regarding privacy-preserving protocols will recognise this to be an incarnation of the "Private set intersection" (PSI) problem. (See, e.g., [4].) Indeed, our solution here can be used as a general private set intersection solution for our problem scenario. Other PSI problems have been tackled previously in past Alerting Project documents, such as [3]. The present edge-finding algorithm can be viewed as a progression.

4.2 The algorithm

The way to tackle edge finding is through a solution that matches edges based on a hash value, similar to what was done in the FinTracer DARK family of algorithms [10]. Where the difference lies, however, is in that whereas all FinTracer DARK algorithm are about propagating a FinTracer tag across edges that are visible to neither AUSTRAC nor the REs themselves, here the purpose is to reveal the edges to AUSTRAC.

Algorithm 1 shows how this can be done. This first presentation of the algorithm is a naive form of it.

Algorithm 1 Edge finding (naive version)

1: R_1 and R_2 coordinate between each other two keys, k_1 and k_2, for a keyed hash, *HASH*.
2: **for all** $(a, b) \in E_1$ **do**
3: R_1 sends to AUSTRAC the pair $(HASH_{k_1}((a, b)), \mathrm{ENC}_{HASH_{k_2}((a,b))}((a, b)))$.
4: ▷ Note that $HASH_{k_2}((a, b))$ is used as a symmetric encryption key.
5: **end for**
6: **for all** $(a, b) \in E_2$ **do**
7: R_2 sends to AUSTRAC the pair $(HASH_{k_1}((a, b)), HASH_{k_2}((a, b)))$.
8: **end for**
9: ▷ Sends from R_1 and R_2 *must* be in a non-revealing order.
10: **for all** (x, y) received from R_1 and (u, v) received from R_2 where $x = u$ **do**
11: AUSTRAC retrieves $(a, b) = \mathrm{DEC}_v(y)$.
12: ▷ DEC is the decryption function corresponding to ENC encryption.
13: **end for**

Some notes regarding this algorithm:

1. The fact that no party receives more information than they should is simply due to the cryptographic assumptions for the individual algorithms used.

2. Coordination of hash keys can be done in fancy ways, such as using Diffie-Hellman, but given that our communication channels are assumed to be secure, there is really no need for it. Even if one of R_1 and R_2 unilaterally decides on the keys, this is just as good, assuming their choice uses good random number generation. (If, on the contrary, R_1 and R_2 do not trust each other to not simply send the same key to all other REs as well, they can each allot a random string, and then use the xor of the two strings as a key.)

3. Use of symmetric encryption instead of a hash also works. It avoids the potential for hash collisions. On the other hand, it allows anyone who knows k_1 and sees the transmissions to retrieve all of E_1 and E_2. (When using a hash, the information does not remain cryptographically protected in such an eventuality, but it would at least require the attacker to enumerate over (a, b) combinations, which can be quite a heavy enumeration task.)

4. If the output of *HASH* is not immediately usable as a key for ENC, any transformation that preserves the full entropy required for a key of ENC can be performed on it to make it fit. (In FTIL, hash results cannot be directly used as encryption keys, but they can seed encryption keys, which performs the same job in this context.)

5. Computations at R_1 and R_2 can be done in parallel.

6. We did not specify what the "non-revealing order" for sending pairs should be, nor did we specify how AUSTRAC is to find those pairs with $x = u$. In an FTIL implementation, the easiest way would have been to send in a random order, and then to use a `hashmap` in order to match pairs. This would have taken advantage of any parallelism in the implementation. In a serial implementation, by contrast, a better solution would have been for the REs to simply sort the pairs by the first pair element, and then for AUSTRAC to use the zipper algorithm

to find the matching pairs. The reason to prefer this alternative implementation in a serial scenario is that it does not require the REs to generate random bits, and spares AUSTRAC from the need to allocate the extra memory required for the hash table. Regardless of the choice of implementation, however, the important point is that the order must critically be non-revealing: simply sending the pairs in the order that they are generated is not secure.

An important observation about the naive version of the algorithm is that even though R_1 and R_2 only coordinate between themselves two keys, in fact each time encryption is used it utilises a never-before-used key. In such a situation, one does not need to use any complicated encryption algorithm nor rely on the cryptographic guarantees of one. Instead, one can use the key as a One Time Pad, simply xor-ing it to the plaintext message. This makes no difference to the security or the complexity of the protocol, but speeds it up considerably (and obviates the need for implementing a symmetric encryption algorithm or adding it as a dependency).

Accordingly, we use Algorithm 2, instead of the naive Algorithm 1, as our full edge-finding protocol.

Algorithm 2 Edge finding (full version)

R_1 and R_2 coordinate between each other two keys, k_1 and k_2, for a keyed hash, *HASH*.
for all $(a, b) \in E_1$ **do**
    R_1 sends to AUSTRAC the pair $(HASH_{k_1}((a, b)), (a, b) \oplus HASH_{k_2}((a, b)))$.
end for
for all $(a, b) \in E_2$ **do**
    R_2 sends to AUSTRAC the pair $(HASH_{k_1}((a, b)), HASH_{k_2}((a, b)))$.
end for
▷ Sends from R_1 and R_2 *must* be in a non-revealing order.
for all (x, y) received from R_1 and (u, v) received from R_2 where $x = u$ **do**
    AUSTRAC retrieves $(a, b) = v \oplus y$.
end for

Notably, when implementing this version in FTIL, no key seeding is required.

5 Edge finding as a final query

We begin by considering edge finding as a tool for solving some classic Alerting problems, using standard FinTracer assumptions: at the end of querying, we would like AUSTRAC to learn the answer to the query, each RE to learn the part of the answer that pertains to them, and no party to learn anything beyond this.

5.1 Finding shortest paths

Let A and B be sets of accounts in $V(G)$. The problem of finding all shortest paths of length up to k connecting any $a \in A$ with any $b \in B$ is a classic FinTracer problem. An improved algorithm was presented in [7].

Here, we show a faster algorithm, running in only $O(k)$ communication rounds, that addresses this problem.

To do this, first recall the algorithm for in-between vertices presented in [13]. This algorithm, in effectively only 2 runs of the basic FinTracer algorithm, retrieves $k + 1$ tags, $M[0], \ldots, M[k]$, where each $M[i]$ signifies the set of accounts that can appear on the i'th step of a shortest path of length at most k from any $a \in A$ to any $b \in B$.

The basic algorithm of [13] does this in an oblivious way, with neither AUSTRAC nor the REs retrieving the value of any $M[i]$. However, for our present purposes we will reveal this information to both sides. Note that because we have assumed that the shortest paths query is a final query, whose answer gets revealed to both sides, and because the $M[i]$ information is a subset of the shortest paths information, there is no issue in revealing it now.

The remaining question is, for each i, which elements in $M[i]$ have an edge connecting them to which elements of $M[i+1]$. The solution for this in [7] was not only heavy, but involved a trade-off between speed and information reveal: one could either solve this problem for each i separately, or could find all edges among the set $M = \bigcup_i M[i]$, but in this latter case the result set may include many edges that are not part of any shortest path (such as, for example, backwards edges).

The new edge-finding algorithm allows us to find all edges along shortest paths from A to B in only a constant number of communication iterations, irrespective of k, without revealing any extraneous edges. Algorithm 3 demonstrates how this can be done.

Algorithm 3 Shortest paths

1: Run the "In-between accounts" algorithm of [13].
2: Let $M[0], \ldots, M[k]$ be the results, i.e. the set of accounts on shortest paths of length up to k between A and B, partitioned such that each $M[i]$ is the subset of accounts that are reached on the shortest path's i'th step.
3: Reveal all $M[i]$ to both AUSTRAC and the REs.
4: For every account x, let $s(x)$ be the value such that $x \in M[s(x)]$.
5: R_1 and R_2 coordinate between each other two keys, k_1 and k_2, for a keyed hash, *HASH*.
6: **for all** $(a, b) \in E_1$ **do**
7: $\quad R_1$ sends to AUSTRAC the pair $(HASH_{k_1}((a, b, d(a))), (a, b) \oplus HASH_{k_2}((a, b)))$.
8: **end for**
9: **for all** $(a, b) \in E_2$ **do**
10: $\quad R_2$ sends to AUSTRAC the pair $(HASH_{k_1}((a, b, d(b) - 1)), HASH_{k_2}((a, b)))$.
11: **end for**
12: ▷ Sends from R_1 and R_2 *must* be in a non-revealing order.
13: **for all** (x, y) received from R_1 and (u, v) received from R_2 where $x = u$ **do**
14: $\quad$ AUSTRAC retrieves $(a, b) = v \oplus y$.
15: **end for**

At the end of this algorithm, AUSTRAC, by retrieving all (a, b) pairs, knows exactly those edges that compose the shortest paths. The REs, meanwhile, discover the information of the $M[i]$, which is a subset of this information.

AUSTRAC can now add to the REs' knowledge any additional details required in order to satisfy whatever information parity policies have been agreed on. This could be no additional knowledge, or it could be the number of incoming and outgoing edges connecting to any account, as two examples.

5.2 Finding k-paths

Suppose we wish to find all walks of length up to k that lead from A to B, whether or not they are shortest.

We will adapt Algorithm 3 to this new scenario, essentially by switching the use of the "In-between accounts" finder in the algorithm's first line with an algorithm that returns all accounts on such up-to-k length walks. Algorithm "k-path accounts" of [13] returns these accounts, but we will not use it because it returns the result in the form of a partition that is not conducive to our needs. We would like to place each account in an $M'[i]$ based on what step of such a walk the account may appear on, but because accounts may appear in different steps on different walks, a single account may appear in several $M'[i]$. Thus, our desired $M'[i]$ are no longer a partition. Instead, the appearance of an account x in $M'[i]$ will simply indicate that there is a walk of length up to i connecting some $a \in A$ with it, and there is a walk of length up to $k - i$ connecting it to some $b \in B$.

The end-to-end algorithm is given in Algorithm 4.

A key difference between this algorithm and the algorithm presented in Section 5.1 is that we have switched "$HASH_{k_1}((a, b))$" from Algorithm 3 and replaced it by "$HASH_{k_1}((a, b, j))$". This follows from the fact that the $M'[i]$ are not a partition.

Algorithm 4 Up to k length paths

$A[0] \leftarrow A$
$B[0] \leftarrow B$
for $i = 1, \ldots, k$ **do**
    $A[i] \leftarrow (G + I)A[i-1]$
    $B[i] \leftarrow (G^{-1} + I)B[i-1]$
end for
for $i = 0, \ldots, k$ **do**
    $M'[i] \leftarrow A[i] \cap B[k-i]$
end for
Reveal all $M'[i]$ to both AUSTRAC and the REs.
R_1 and R_2 coordinate between each other two keys, k_1 and k_2, for a keyed hash, *HASH*.
for $i = 0, \ldots, k-1$ **do**
    for all $a \in M'[i]$ **do**
        for all $b : (a, b) \in E_1$ **do**
            R_1 sends to AUSTRAC the pair $(HASH_{k_1}((a, b, i+1)), (a, b, i+1) \oplus HASH_{k_2}((a, b, i+1)))$.
        end for
    end for
end for
for $j = 1, \ldots, k$ **do**
    for all $b \in M'[j]$ **do**
        for all $a : (a, b) \in E_2$ **do**
            R_2 sends to AUSTRAC the pair $(HASH_{k_1}((a, b, j)), HASH_{k_2}((a, b, j)))$.
        end for
    end for
end for
▷ Sends from R_1 and R_2 *must* be in a non-revealing order.
for all (x, y) received from R_1 and (u, v) received from R_2 where $x = u$ **do**
    AUSTRAC retrieves $(a, b, t) = v \oplus y$.
        ▷ This indicates that the (a, b) directed edge can appear as the t'th edge on a path of length up to k.
end for

In total, there are three types of uses of (a, b) in Algorithm 3 that may or may not be supplemented by additional details—appearing, in total, as (a, b, t)—in variants of this algorithm. The general rule regarding what information belongs where is as follows.

In $\boldsymbol{HASH}_{k_1}$: The k_1 hash determines which information pairs can be matched together. In finding walks of length up-to-k, it is not enough to just match based on (a, b) alone, because not every walk from A to B that goes through (a, b) is of length at most k. The addition of j to the matching criteria ensures that going through the (a, b) edge at the j'th step will still allow a walker traversing the graph to reach all the way from A to B without increasing the total length of the walk beyond k.

In $\boldsymbol{HASH}_{k_2}$: The k_2 hash is used as a One Time Pad. It is therefore critical to the security of the system that it can never be reused. Utilising all of k_2, a, b and t together as inputs to the computation that generates the one time pad ensures its uniqueness. Without it, if the same directed edge can appear at different positions on a walk of length at most k from A to B, this would trigger a repetition of our "one time pad", rendering it insecure.

In the plaintext under the $\boldsymbol{HASH}_{k_2}$ pad: This is the information that is ultimately revealed to AUSTRAC. In the case of discovering walks of length up-to-k, simply knowing that the edge (a, b) exists in one such walk is not enough to easily and reliably determine which walks of length up-to-k go through it.

We leave to the reader the exercise of using the same methods to determine the paths of length k exactly. In solving this exercise, however, readers should take particular care to determine, in each of the three cases described above, what extra information, if any, should be encoded in addition to the identity of the edge itself.

6 Edge finding as an intermediate query

As stated in [13],

> "In terms of their usability, protocols that can be used to compute intermediate results are superior to those that can only compute final results."

In the previous section, we explored how to use edge finding as a final result. Here, we discuss its potential as an intermediate query.

The edge-finding algorithm itself can be made into an "intermediate query" in a straightforward manner. Algorithm 5 shows how this can be done.

Algorithm 5 Edge finding as an intermediate query

R_1 and R_2 coordinate between each other two keys, k_1 and k_2, for a keyed hash, *HASH*.
Each R_i generates a private/public key pair, $(K_i^{\text{priv}}, K_i^{\text{pub}})$.
for all $(a, b) \in E_1$ **do**
    R_1 sends to AUSTRAC the pair $(HASH_{k_1}((a, b)), (\text{ENC}_{K_1^{\text{pub}}}(a), \text{ENC}_{K_2^{\text{pub}}}(b)) \oplus HASH_{k_2}((a, b)))$.
end for
for all $(a, b) \in E_2$ **do**
    R_2 sends to AUSTRAC the pair $(HASH_{k_1}((a, b)), HASH_{k_2}((a, b)))$.
end for
▷ Sends from R_1 and R_2 *must* be in a non-revealing order.
for all (x, y) received from R_1 and (u, v) received from R_2 where $x = u$ **do**
    AUSTRAC retrieves $(\text{ENC}_{K_1^{\text{pub}}}(a), \text{ENC}_{K_2^{\text{pub}}}(b)) = v \oplus y$.
end for

Here, we assume that the public/private key pair is for a public key encryption scheme that can be properly decrypted. This can be RSA, it can also be a semi-homomorphic encryption

scheme, like multiplicative ElGamal, but it cannot be additive ElGamal, because that would not allow decryption.

The idea of this algorithm is that the final result delivered to AUSTRAC is no longer (a, b), but rather $(\mathrm{ENC}_{K_1^{\mathrm{pub}}}(a), \mathrm{ENC}_{K_2^{\mathrm{pub}}}(b))$. In other words, each item has been individually encrypted, and AUSTRAC is only informed of what the encrypted value is. (Recall that AUSTRAC also knows that the match was between R_1 and R_2, and so is aware of who holds the decryption key for each item.)

In this way, the edges discovered are held obliviously by AUSTRAC, and neither AUSTRAC nor any of the REs gains direct information about the identity of their endpoint vertices. However, AUSTRAC does see the topology of the graph created by the revealed edges.

For completion, we add that in the first iteration of the Purgles Platform it is likely that no public-key encryption mechanism will be available to us. In this case, it is possible to use Algorithm 6, which substitutes the original public-key mechanism with symmetric encryption. (Considering that *HASH* itself can be implemented in terms of a symmetric encryption algorithm, all algorithms in this document can be powered using only a single cryptographic primitive, this being namely a block cipher.)

Algorithm 6 Edge finding as an intermediate query (using symmetric encryption)

R_1 and R_2 coordinate between each other two keys, k_1 and k_2, for a keyed hash, *HASH*.
Each R_i generates a private key, K_i^{priv} for a symmetric encryption scheme.
for all $(a, b) \in E_1$ **do**
    R_1 sends to AUSTRAC the pair $(HASH_{k_1}((a, b)), (\mathrm{ENC}_{K_1^{\mathrm{priv}}}(a), 0) \oplus HASH_{k_2}((a, b)))$.
end for
for all $(a, b) \in E_2$ **do**
    R_2 sends to AUSTRAC the pair $(HASH_{k_1}((a, b)), (0, \mathrm{ENC}_{K_2^{\mathrm{priv}}}(b)) \oplus HASH_{k_2}((a, b)))$.
end for
▷ Sends from R_1 and R_2 *must* be in a non-revealing order.
for all (x, y) received from R_1 and (u, v) received from R_2 where $x = u$ **do**
    AUSTRAC retrieves $(\mathrm{ENC}_{K_1^{\mathrm{priv}}}(a), \mathrm{ENC}_{K_2^{\mathrm{priv}}}(b)) = v \oplus y$.
        ▷ The storage format for pairs is assumed to be such that $(z, 0) \oplus (0, w) = (z, w)$.
end for

The privacy mechanism used in both algorithms above is a form of *anonymisation* (similar to tokenisation) which is often used to keep the privacy of data. Elsewhere in the Alerting Project we have deemed such an approach not strong enough because of the risk of de-anonymisation. Specifically, any source of rich enough "Big Data" runs a high risk of such anonymisation measures being reversible [1], and even just network topology data, as can be derived from the anonymised edges themselves, without any additional data overlaid on them, has already been shown [2] to allow such de-anonymisation.

And yet, we believe that in this specific context, our use of anonymisation without further measures does provide adequate privacy preservation.

To explain, let us consider the wider privacy model and how we intend to make use of it.

6.1 Use of SoNaR

The main problem with privacy-preserving algorithms is that they are heavy. The reasons that they are so much heavier than non-privacy-preserving algorithms can be divided into two core reasons:

1. They rely on oblivious computation, which requires heavy cryptographic operations and considerable amounts of communication in order to hide the underlying data from parties, and

2. Because parties cannot be told what they are looking for and what they've already found, processing cannot focus on where processing power is needed, and must be applied equally over the entire domain of possibilities.

Together, these constraints lead privacy-preserving protocols to require much processing spread over large domains, which, in total, sums up to quite heavy algorithms.

With the FinTracer algorithm, the Alerting Project developed an algorithm that is, in relative terms, very light in the amount of per-account computation and per-account communication it requires. which is why it is possible to run it scaled up to every account across the entire Australian economy. However, that is by no means uniformly true across the board in every other algorithm in the FinTracer algorithm suite as well.

The deeper into analysis one goes, undoubtedly heavier and heavier algorithms will be required.

To overcome this problem, we propose to use the SoNaR privacy model.

The idea here is that much like AUSTRAC may define for FinTracer a source set A and a destination set B by means of an implicit description, creating by this a situation where the REs' familiarity with the set of accounts of interest is strictly greater than AUSTRAC's, so will be the case with intermediate results: as AUSTRAC runs deeper analysis algorithms on the population set, the results of these analyses effectively exclude some accounts as "no longer accounts of interest". In the SoNaR model, AUSTRAC has the option to reveal to the REs (but not to itself) the identity of the accounts that have thus been ruled out (and, equivalently, those that have not).

The end result is that the REs can now have a much narrower set of accounts and transactions to consider. AUSTRAC will therefore be able to run heavier algorithms on this narrower set. This process can be repeated as necessary, allowing iteratively heavier processing on iteratively smaller target sets.

6.2 Pairs, connectivity and bridging in SoNaR

Edge finding is an example of an algorithm that is likely too heavy to scale up to the entire Australian economy. The reason for this is that it requires communication in an amount that scales proportionally to the number of edges in G, rather than to the number of vertices. The propagation graph G is likely to have orders of magnitude more edges than vertices.

In Section 5, we were able to use the edge-finding algorithm because we first determined the sets of accounts of interest ($M[i]$ or $M'[i]$, depending on the case) and then revealed the identity of these accounts to the REs. This made it possible to process subsequently only this narrowed set of accounts. The reason it was possible for us to perform this narrowing is that the identity of the accounts of interest was not just an intermediate step, but also part of the final query result which the system would have been required to report to the REs in any case.

At times, however, we are not actively interested in retrieving an entire network of paths (which may still be too large to be retrieved in full, from a social license standpoint). Instead, our purpose is to find either properties of this network or details within this network that are of interest.

The way to do this is by following the process described in Algorithm 7.

An example of an analysis whose purpose is to determine properties of the graph is the computation of general statistics about the graph, such as regarding the path lengths involved, or regarding the number of A and B accounts each node connects to.

An example of an analysis whose purpose is to find accounts of interest within the remaining graph is bridging: the AUSTRAC researcher can use their favourite network analysis tool in order to find which accounts are more central to the network (or which accounts are likely to be owned by more prominent people in a criminal organisation). This is possible because the network structure is retrieved by AUSTRAC in the clear; only the identities of the individual accounts composing the structure are encrypted. Once the accounts of interest are found, AUSTRAC can ask the REs to decrypt the account number information for only those accounts of interest.

If AUSTRAC is interested to solve the pair-finding problem (i.e., the problem of which $a \in A$ connects to which $b \in B$), or, alternatively, if it wants to solve the connectivity problem (i.e., determine for each $a \in A$ how many $b \in B$ it connects to, and vice versa), one way to do this is

Algorithm 7 SoNaR process (in the edge-finding context)

1: Identify V, the accounts over which edges need to be determined (e.g., using the In-between accounts algorithm of [13]).
2: Reveal these accounts to the REs.
3: Run the intermediate edge-finding algorithm (Algorithm 5) only on those $(a, b) \in E_1$ where $a \in V$ and only on those $(a, b) \in E_2$ where $b \in V$.
4: ▷ Note that this reduction in E_1 and E_2 can be worked out by R_1 and R_2, respectively, in an independent way, each leveraging only their knowledge of their own portion of V.
5: Analyse the resulting anonymised sub-graph.
6: If the purpose is to determine properties of the graph, these can be found on the anonymised sub-graph.
7: If the purpose is to find accounts of interest in the graph, the encrypted names of these can be discovered, and AUSTRAC can then ask the relevant REs to decrypt them.

for the REs to simply decrypt the values of every $a \in A$ and every $b \in B$, while keeping the rest of the graph anonymised. However, we believe that mostly AUSTRAC's information needs will be smaller than this, and there will be more privacy-preserving approaches than direct decryption to satisfy them. For example, AUSTRAC may not actually need to know how many $b \in B$ each $a \in A$ connects to. It may be enough for AUSTRAC to only get a list of those $a \in A$ that connect to at least 10 of the $b \in B$ (or any other threshold number). AUSTRAC can, in such a case, only provide to the REs the anonymised account numbers for those accounts that satisfy the retrieval criteria, and the REs can supply AUSTRAC with the decrypted account lists.

Neither in this example nor in other "accounts of interest" examples is it necessary for the REs to tell AUSTRAC, when decrypting account identifiers, which ciphertext values correspond to which accounts of interest. It is enough for AUSTRAC to provide a request list and for the REs to provide, in return, the corresponding decrypted list, but given in a random order so as not to allow determining the network position of any account.

These are all, of course, just examples of how the SoNaR model can be used in practice. Our main takeaway is that it is a flexible usage model that can be employed to provide AUSTRAC with whatever information it needs, while utilising any of multiple possible techniques to ensure that communication of extraneous, unnecessary information is avoided or minimised.

6.3 The effectiveness of anonymisation

Now that the use model of SoNaR has been clarified, let us return to the problem that anonymisation is in general not a good-enough solution for privacy preservation, not in network analysis, and not in Big Data in general.

The reasons we believe anonymisation is effective and non-reversible in our specific usage scenario are the following.

1. First and foremost, this anonymisation is only of a small subgraph of interest, not the entire economy.

2. Even within this population, we do not show every connecting edge but only those relevant to our needs. (In mathematical terms: our subgraph is not an induced subgraph.) This reduces significantly the amount of information that can be derived from it regarding the overall network.

3. The edges and accounts carry no information other than their topology. (They carry no identifiers that can be used to connect with other information.) Thus, the information is not only not "big", but also not rich.

4. Whenever a new query is asked and the edge-finding algorithm needs to be run, the results will use a brand new encryption key, and therefore will hinder attempts to collate results from multiple queries. This is a form of *forward secrecy*.

At the moment, we do not plan to obfuscate the resulting mini-graphs by use of any form of differential privacy noise. This is because we are not aware of any such means that can effectively add differential privacy without reducing, perhaps dramatically, the usefulness of the returned results.

6.4 SoNaR and RE-held information: a discussion

This document is the first to introduce the SoNaR model, i.e. the model in which REs receive information about intermediate results which AUSTRAC remains not privy to. To conclude, in this section we discuss the aspects that need to be considered in deciding whether or not SoNaR should be a feasible model for the Purgles Platform.

1. REs are provided with information about accounts that are not directly "accounts of interest" but rather are "crime adjacent" in some way: they are chosen because they have matched partial typologies, and in the case described in this section are certainly placed between account sets of interest.

2. SoNaR, by telling an RE that one of their accounts matches a particular partial typology, might divulge for this RE information regarding other accounts, potentially ones belonging to other REs. (Consider, for example, an account that only transacts with one other account, and yet appears on a walk from A to B.)

If required, the knowledge given to REs in SoNaR can be diluted without this affecting AUSTRAC in any qualitative way.

An example of how to do this is as follows.

Consider Algorithm 7. The knowledge revealed to the REs in this algorithm is V, the output of an algorithm such as the In-between accounts algorithm. This algorithm, however, produces V by computing the intersection of two account sets. So far, we have assumed that computing such intersections is done as per the process described in [13] and [6], which is the best solution we have for intersection-finding, and is resilient to result tampering by AUSTRAC.

It is possible, however, to use here an inferior algorithm, such as the one recapped in [6, Section 1]. This alternative algorithm has the ability to generate a "noisy" intersection result.

The algorithm for this is as described in Algorithm 8. It computes the intersection of some u and some v, with additional noise. AUSTRAC can only meaningfully control the amount of noise, however, not which accounts are impacted by it.

Algorithm 8 Noisy intersection

1: REs generate individual random permutations σ.
2: REs send $\tilde{u} = \sigma u$ and $\tilde{v} = \sigma v$ to AUSTRAC (with any required differential privacy noise).
3: AUSTRAC computes $\tilde{w} = \tilde{u} \cap \tilde{v}$.
4: ▷ This is the result without any noise.
5: AUSTRAC computes $|\tilde{w}|$ and marks another $\alpha|\tilde{w}|$ accounts as "True" in $\tilde{w}$.
6: ▷ The constant α determines the noise amount and is a policy parameter.
7: AUSTRAC returns the modified $\tilde{w}$ to the REs.
8: The REs compute the final answer as $w = \sigma^{-1}\tilde{w}$.

Note 6.1. The intersection operation is not necessary for the addition of noise. The standard algorithm, implemented in the FTIL script `NormaliseTag` of [12, Section 4.10.1] can easily be adapted to inject noise. We provide Algorithm 8 to show that noise addition can be integrated into the existing algorithms without this requiring any additional communication rounds.

Note, however, that when adding noise during an AND operation there will be slightly different requirements regarding differential noise. In [12] it was calculated what amount of noise is required to hide the exact size of a set, and this was added twice: once to the set of accounts that had a "True" value and once for the set of accounts that had a "False" value. In Algorithm 8, AUSTRAC

is privy to two bits of information for each account, hence each account can be categorised as belonging to one of four sets. Each of these sets ("00", "01", "10" and "11") will require its own addition of noise, but the amount of noise for each set will be the same as in the original calculation.

The point of Algorithm 8 is that REs now receiving the intermediate result V can no longer tell whether any account in it actually matches a partial typology or whether it was simply randomly added. With probability $\frac{\alpha}{\alpha+1}$, this is a completely innocent bystander account.

Consider, for example, choosing $\alpha = 1$. Any account in V now has only a 50% chance of matching the partial typology.

At the same time, adding this noise to V in Algorithm 7 does not provide AUSTRAC with any additional information. The reason for this is that, when subsequently running Algorithm 5, the probability of any half-edge connecting to a randomly-added account to connect with another half-edge (connecting to another randomly-added account, or to one of the accounts that would have originally been in V) is very slim.

For this reason, AUSTRAC will not be able to match edge signatures, and will gain no additional knowledge at all from the extra noise. The resulting edge set will, with high probability, stay exactly the same as it was without noise.

In terms of a quantitative change, the only difference is that in running Algorithm 5 AUSTRAC will have more edge signatures to sift through, by a factor of $1 + \alpha$. Whether AUSTRAC matches edge pairs using a hash or using the zipper algorithm, this requires more work by a factor of only $1 + \alpha$, and additional memory by the same factor. The REs, whether sorting by the data or by a random permutation, will require similar amounts of extra work in order to sort $1 + \alpha$ more elements.

We present as an open question whether the SoNaR model (either with or without the noisy intersection methods) meets the privacy and policy requirements of the RE and FIU participants.

References

[1] Mielikäinen, T. (2004). Privacy problems with anonymized transaction databases. In *International Conference on Discovery Science*, 219–229. Springer, Berlin, Heidelberg.

[2] Biryukov, A., & Tikhomirov, S. (2019). Deanonymization and linkability of cryptocurrency transactions based on network analysis. In *2019 IEEE European symposium on security and privacy (EuroS&P)*, 172–184. IEEE.

[3] Brand, M. (2020), Fast, Lightweight, Secure Record Set Intersection Without Homomorphic Encryption. (Available as: `fast_intersection.pdf`)

[4] Chen, J., Huang, Z., Laine, K. & Rindal, P. (2018) Labeled PSI from fully homomorphic encryption with malicious security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 1223–1237. Available at https://eprint.iacr.org/2018/787.pdf.

[5] Brand, M. (2020), Efficient connection-finding with FinTracer, AUSTRAC internal. (Available as: `pair_finding.pdf.pdf`)

[6] Brand, M. (2020), Improved differential privacy for FinTracer Boolean queries. (Available as: `boolean_queries.pdf`)

[7] Brand, M. (2020), Massively parallel FinTracer path finding. (Available as: `path_finding.pdf`)

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

[9] Brand, M. (2021), Oblivious querying of FinTracer tag values. (Available as: `oblivious_b.pdf`)

[10] Brand, M. (2021), Extreme DARK: A more secure DARK variant. (Available as: `extreme.pdf`)

[11] Brand, M. (2021), Oblivious (and non-oblivious) discovery of sparse matches. (Available as: `oblivious_c.pdf`)

[12] Brand, M. (2022), *The Complete, Authorised and Definitive FinTracer Compendium (Volume 1, Part 1, 3rd Revision [a.k.a. "Purgles" edition]): A work in progress.* (Available as: `FT_combined.pdf`)

[13] Brand, M. (2022), Some incremental progress in FinTracer technology. (Available as: `small_updates.pdf`)