Detecting non-comingled money laundering networks with FinTracer

Michael Brand

November 23, 2023

Executive Summary

We describe a privacy-preserving algorithm able to efficiently detect isolated networks inside larger graphs. We show that this algorithm can be adapted to detect money laundering networks that do not employ co-mingling by incorporating other indicators of money laundering activity. This algorithm is so far unique within the FinTracer family of privacy-preserving algorithms in that it requires no form of account-based qualifiers. This property allows it to be deployed as an initial typology-finding algorithm, because it requires no initial suspicion or even a narrowing down of accounts-of-interest.

1 Introduction

This algorithm is designed to detect a typology of the following description:

> We are interested in finding isolated portions of the financial system, composed of only a small number of accounts, such that all funds entering this small network are in cash. Funds leave the network via international transfer, the purchase of crypto-currency, or the purchase of assets.

This document describes an algorithm that, in addition to detecting this typology, can be used, more generally, to detect small connected components inside large graphs.

We will first describe the small connected components finder in its pure form, then show how it can be applied to the concrete problem setup.

Note 1.1. The algorithms described in this document are heavy, as compared with other algorithms in the FinTracer suite, but their purpose is to be able to detect networks of interest where the internal money flow may potentially be convoluted and where no individual account has any "suspicious" attributes.

This is in contrast to simple money laundering networks, where money flow is purely acyclic. If money flow can be assumed to be acyclic, standard, light-weight FinTracer-based algorithms will do the trick, based on the ability to define "placement", "layering" and "integration" accounts, and the use of the FinTracer algorithm to characterise money flows between these.

We will, in this document, not delve into algorithms for acyclic money-laundering typologies.

2 Finding small connected components

2.1 The problem definition

Consider an undirected graph, meaning a graph G where edges have no directionality: an edge (a, b) is equivalent to an edge (b, a). A connected component in such a graph is a set of vertices $S \subseteq V(G)$, which is a maximal set such that for every $x, y \in S$ there is a path connecting x to y.

Given G, we are interested in an algorithm that efficiently identifies in it all connected components up to size m (the size of a component being the number of vertices in it), where m is a parameter of the algorithm.

For a problem and a solution to be viable for our context, they must also satisfy the following.

1. The problem needs to be tackled as part of the FinTracer system [3]. This means that it needs to be run in a distributed environment where one computation node is managed by AUSTRAC and the other nodes are managed by AUSTRAC's reporting entities (REs). Also, it must be a privacy-preserving algorithm, adhering to the FinTracer privacy model. For this, we will assume that $V(G)$ is the set of accounts managed by the reporting entities.

2. It must be a FinTracer-compatible problem setup, in the sense that we will assume that $E(G)$ are edges that can be used in either FinTracer or FinTracer DARK [2, 6] propagation.

3. The algorithm must integrate with the wider FinTracer family of algorithms, in the sense that the output needs to be in the form of a FinTracer tag. A FinTracer tag is a mapping from vertices (accounts) to ElGamal ciphertexts, where a nonzero value indicates the accounts that are of ultimate interest. The ElGamal private key will be held by AUSTRAC, and the mapping will be distributed between the REs such that each RE only sees the portion of the map relevant to its own accounts.

We will, additionally, make the following tweaks to the problem, in order to present an efficient solution.

1. Unlike most algorithms in the FinTracer suite, this will be a probabilistic algorithm, allowing for errors of both kinds. We will aim for an algorithm that has an arbitrarily high probability of finding the accounts of a connected component if it has at most m accounts, while having an arbitrarily low probability of returning accounts that belong to connected components of size $2m$ or more. We introduce the parameters n and t, which will control these probabilities: n, which impacts the speed and communication volumes of the algorithm, will determine the margin of separation between the two hypotheses, and t ($t < n$), not impacting the speed or communication volumes of the algorithm, will determine how the different kinds of errors are balanced. We recommend to choose $n \geq \beta m$, for some constant β, when scaling m, and will assume this in our computation of the algorithm's errors. We also introduce a quality parameter w, a natural, that controls how many of the results returned will be marked as "invalid". (Regarding invalid results, see Section 2.3. For an algorithm that does not use w and omits all invalid results, see Section 2.6.)

2. FinTracer problems have always been bound by the number of propagation steps required to see the full typology. In our case, the equivalent parameter is d, an upper bound for the diameter of the connected components sought. If d can be meaningfully limited, the complexity of the algorithm can be presented as a function of d. If not, one can simply choose $d = m$, because a connected component of size m cannot have a diameter larger than m.

The diameter of a graph (or a subgraph, like a connected component) is the length of the shortest path between two vertices $a, b \in V(G)$, for the a and b that maximise this metric.

We will present a solution that works in $\mathrm{O}(dn)$ propagation steps and requires $\mathrm{O}(n)$ nonlinear operations. The algorithm is a Monte Carlo algorithm whose complexity is deterministic, but whose result is probabilistic.

The end result of the algorithm, provided as a FinTracer tag marking the accounts within the connected components of interest, is not revealed to any party, so can be used as an interim result, i.e. as input to later computations. One option of how to utilise the result, however, is to retrieve the anonymised topology of the result using the algorithms of [5], which is a SoNaR algorithm, revealing to the REs which accounts were marked.

Given the fact that the returned results will be disjoint small graphs, even if the overall number of returned accounts is large, there is no risk for de-anonymisation (other than with exponentially small probability).

Based on the revealed topologies, users can then perform extra filtering in the clear, such as by removing results whose size is above m or whose diameter is above d, or otherwise removing results whose patterns make them unlikely to be related to money laundering (ML). Once such filtering is done and the overall number of results is manageable, users can choose to reveal the account numbers within the found networks.

2.2 The algorithm

The algorithm is presented in Algorithm 1. It uses a constant, α, which we will set to $1/3$. Different settings of this constant can be used to optimise various metrics for the algorithm. In Appendix A we explain the metric that leads to the choice $\alpha = 1/3$. In terms of the overall performance of the algorithm, any choice of α in $(0, 1)$ is as good, up to constants.

We assume $t < (1 - \alpha)n$.

Algorithm 1 Finding small connected components

W is initialised as the all-zeroes tag.
S is initialised as the all-zeroes tag.
for $i \in 1, \ldots, n$ **do**
    T is initialised by setting each account, independently, to either 0 or nonzero, where 0 is chosen with probability $p = \alpha^{1/m}$.
    $T \leftarrow (G + I)^d T$.
    Normalise T. ▷ Nonlinear operation: make all tag values 0 or 1.
    $W \leftarrow W + ((G + I)T \setminus T)$.
    $S \leftarrow S + T$.
end for
$W \leftarrow (G + I)^{d+w} W$.
return $(S < t) \setminus W$.
▷ $S < t$ is a nonlinear operation: retain only the accounts where the value of S is less than t.

In algorithm listings in this document we use the following shorthand conventions.

- The statement "$T \leftarrow (G + I)^d T$" uses "G" to mean the adjacency matrix of the graph G. The matrix "I" is the identity matrix. Multiplying a tag by a matrix indicates a propagation step. Multiplying by "$G + I$" indicates that we sum the original value to the propagated value, because $(G + I)x = Gx + Ix = Gx + x$. Taking this to the power of d means that the operation is repeated d times. Hence, the statement in total means "Repeat d times 'Add to T the result of propagating T along G' ".

- In tag manipulation, linear operations "+" and "−" indicate that tag values associated with the same accounts are summed together / subtracted from each other. Nonlinear operations are indicated using set notation. So, for example, "$A \setminus B$" is a tag that is nonzero for every account where A was nonzero but B was zero. These are computed using methods explained in [1, **?**].

The complexity cited for the algorithm assumes that computing "$S < t$" is done using a single nonlinear operation. (It is assumed that t is small enough for AUSTRAC to prepare a hash table of all Ed25519 values from 0 to t.) If one wants to only pass to AUSTRAC sanitised values, this will have to be done by evaluating $S \cap (S - 1) \cap (S - 2) \cap \cdots \cap (S - t)$, which requires a communication amount proportional to t.

Note 2.1. Ed25519 representation in projective space, which is how tag values are normally stored in FinTracer, is not unique, and hence not conducive to hash table creation. The values will need

to be first converted to affine representation. (This, unfortunately, is a relatively heavy operation because it requires a division.)

An alternative would be for REs to report T to AUSTRAC, under a random permutation, after every round, and for this permutation to be the same in each round. In this variant, AUSTRAC, not using any hash tables, will be able to count how many of the T results for each account were positive, and will be able to report the value of $S < t$ back to the REs (for them to undo the permutation). This will give AUSTRAC, as collateral, the distribution of the number of accounts in a d-neighbourhood around each account, which may be commercially interesting information, but is not privacy-invasive.

2.3 Filtering out invalid results

Algorithm 1 promises high probability of correctness for each of its results. However, the probability of an incorrect result is still positive. Moreover, results are returned on a per-account basis, not on a per-connected-component basis, and in some cases a few accounts in an otherwise correctly-returned connected component may be incorrectly labelled.

If results are non-uniform across a connected component, we say that the results returned for those accounts, as well as for the connected component itself, are *invalid*.

The algorithm guarantees that no correctly returned components will be invalid (and more generally, no components with diameter at most d will be invalid). However, it may still be the case that a connected component with diameter greater than d will only return a subset of its accounts, and that subset, if taken on its own, may appear to the user as a component with diameter at most d.

If the purpose is to examine the topology of the results in order to make later decisions on a per-component basis, it is therefore important that the user be able to filter out invalid results in post processing. For this, we present Algorithm 2, which returns a tag with invalid accounts. Not all invalid accounts are guaranteed to be returned, but the algorithm does guarantee that in every invalid component returned by Algorithm 1, at least one account will be marked invalid by Algorithm 2, while valid components will never have any of their accounts marked as invalid.

The algorithm uses O(1) propagation steps and O(1) nonlinear computations.

Algorithm 2 Filtering out invalid results

1: Let R be the result returned by Algorithm 1.
2: **return** $((G+I)((G+I)R \setminus R)) \cap R$

To see the correctness of Algorithm 2, consider each connected component individually. If R includes every account in a component (or none of them), $(G+I)R$ will simply equal R, so $(G+I)R \setminus R$ will be the empty set and no accounts in the component will be marked invalid. If, on the other hand, R includes some accounts in the component but not others, there must exist in G an edge, (a, b), such that $a \in R$ but $b \notin R$. In this case, $(G+I)R \setminus R$ will include b, and so the final result (using the fact that G is undirected), will include a.

2.4 Proof of complexity

The complexity of Algorithm 1 is straightforward: we go through n iterations, where each iteration involves d propagation steps and O(1) nonlinear computations. This adds up to dn propagation steps and $O(n)$ nonlinear computations. We then perform $d + O(1)$ more propagation steps and O(1) more nonlinear computations, for the claimed totals.

2.5 Parameter choice and proof of correctness

To demonstrate the correctness of Algorithm 1, let us first consider the simpler (and heavier) variant presented in Algorithm 3.

Algorithm 3 Finding small connected components (Naive version)

S is initialised as the all-zeroes tag.
for $i \in 1, \ldots, n$ **do**
    T is initialised by setting each account, independently, to either 0 or nonzero, where 0 is chosen with probability $p = \alpha^{1/m}$.
    repeat
        $T \leftarrow (G + I)T$.
    until $(G + I)T = T$
    Normalise T. ▷ Make all tag values 0 or 1.
    $S \leftarrow S + T$.
end for
return $S < t$.

In Algorithm 3 there is no use of W or of d. The algorithm propagates the initial T value until no further account is added, meaning that for every connected component either all its accounts or none of its accounts are ultimately tagged.

We claim that the probability for an account, a, to be tagged is a direct function of the number of accounts in its connected component. This probability is then estimated by measuring in how many of the n attempts the account was tagged. If this is less than t, we return that the number of accounts in the component was at most m.

Let us compute this probability. If there are m' accounts in the connected component, the probability of all to be untagged is $p^{m'} = \alpha^{m'/m}$. Account a will be untagged at the end of propagation if and only if this happens. Hence, the probability for a to be untagged is a direct function of m', as desired.

See Appendix A for why α was chosen to be $1/3$. To properly pick t given this choice, consider the following.

Let $p_0 = \alpha^{m'/m}$.

We wish to differentiate the case $m' \leq m$ (where $p_0 = \alpha^{m'/m} \geq 1/3$) from the case $m' \geq 2m$ (where $p_0 = \alpha^{m'/m} \leq 1/9$). After n iterations, the distribution of the number of cases where a's tag is zero is binomial (which, for a large n is approximately normal), with expectation np_0 and standard deviation $\sqrt{np_0(1-p_0)}$.

By choosing, for example, $t = n/5$, the probability for an error of either kind (under the normal approximation) becomes $\phi\left(-\frac{\sqrt{2n}}{5}\right)$.

For negative a values, $\phi(a)$ can be bounded by

$$\phi(a) = \int_{-\infty}^{a} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \leq \int_{-\infty}^{a} \frac{1}{\sqrt{2\pi}} e^{-\frac{xa}{2}} = -\frac{\sqrt{2}}{a\sqrt{\pi}} e^{-\frac{a^2}{2}},$$

so

$$\phi\left(-\frac{\sqrt{2n}}{5}\right) \leq \frac{5}{\sqrt{\pi}} \cdot \frac{1}{\sqrt{n}} e^{-\frac{n}{25}},$$

making the error probability, at worst, exponentially decreasing with n.

Computing with the real ϕ values, at $n = 53$ the probability drops under 2% (two sigmas is at $n = 50$), and at $n = 100$ it is already at roughly 0.2%.

Consider, now, how Algorithm 1 differs from Algorithm 3.

First off, Algorithm 1 merely propagates tags d times in each iteration, rather than to an unlimited amount. If propagating within a connected component of diameter at most d, these two are equivalent, so the result of Algorithm 1 will in this case be the same, and will never be marked as invalid.

Consider, therefore, a connected component that is of diameter greater than d.

To analyse this case, let us introduce another simplified version of Algorithm 1, this being namely Algorithm 4.

Algorithm 4 Finding small connected components (Second naive version)

W is initialised as the all-zeroes tag.
S is initialised as the all-zeroes tag.
for $i \in 1, \ldots, n$ **do**
    T is initialised by setting each account, independently, to either 0 or nonzero, where 0 is chosen with probability $p = \alpha^{1/m}$.
    $T \leftarrow (G + I)^d T$.
    Normalise T. ▷ Make all tag values 0 or 1.
    $W \leftarrow W + ((G + I)T \setminus T)$.
    $S \leftarrow S + T$.
end for
repeat
    $W \leftarrow (G + I)W$.
until $W = (G + I)W$
return $(S < t) \setminus W$.

In Algorithm 4, if a component is non-homogeneous at the end of any of the n iterations, this is marked in W, and any such component is excluded from the final result. This differs from the original algorithm only in that the final propagation of W is not limited.

Consider, now, how this algorithm behaves. If at the end of every iteration the entire component is homogeneous, the algorithm behaves exactly like Algorithm 3, so accurately measures (in a probabilistic sense) the size of the component. If it is ever non-homogeneous, the entire component is disqualified because its diameter is greater than d.

The only additional error source is that a component may have less than m vertices, but has a diameter greater than d and despite this remains homogeneous at the end of each iteration. This is not a dramatic error, because

1. It is a false-positive for which the entire component is reported, which can easily be filtered by the user in post-process, and

2. It does not violate privacy preservation, because the component is small, so cannot be de-anonymised.

We will, nevertheless, show that the probability for such an error is low if there exists any vertex a for which there are many vertices b in the component whose distance to a is greater than d. (By assumption, at least one such (a, b) pair exists.)

Consider such an (a, b) pair, and consider all iterations where a's value for tag T was zero at the end of the iteration. By assumption, there are at least $n - t > \alpha n$ of them (which for our choice of α equals $n/3$), or else the component would have been disqualified as too large. Each such case requires the entire d-neighbourhood of a to start out as all-zeroes, but is independent of all other choices. In particular, it is independent of the initial choice of tag value T at b. With probability $1 - p$, this is therefore nonnegative, causing the component to be disqualified. For the component not to be disqualified, b must have an initial tag value of 0 at each of these rounds, this occurring with probability $p^{n-t} < \alpha^{\alpha\beta}$, a constant that can be set by our choice of β. Let $\gamma = \alpha^{\alpha\beta}$.

If for a given a there are k vertices of distance greater than d from it within a single component, the probability for the component not to be disqualified by any of them is at most γ^k, so drops at least exponentially with k.

For our choice of α, even $\beta = 2$ is enough to bring γ below $1/2$.

Note that if a and b are a pair of vertices whose distance d_0 is the diameter of the component then a has at least one vertex at distance x from it, for every x between d and d_0, so the probability for a component not to be disqualified also drops exponentially with its diameter.

As stated, Algorithm 4 is different to Algorithm 1 only to the extent that in Algorithm 1 the tag W is propagated only $d + w$ times.

Consider, therefore, that around each a for which $S < t$ is true, but to which Algorithm 4 propagates W causing it to be removed from the answer set, there are vertices b in the same component for which W is nonzero.

Algorithm 1 propagates $d + w$ steps, meaning that for any account a that has no account b in the same component that is of distance greater than $d + w$ from it, the algorithm will return the same result as Algorithm 4, whereas for any a that does, its probability of a appearing in the final result, even on the assumption that its d-neighbourhood has less than m vertices, is at most γ^k, where k is the number of vertices at distance between $d + 1$ and $d + w$ from a. This not only decreases exponentially with the number of these vertices, but also with w.

2.6 An Atlantic City algorithm

It is possible to omit the possibility of invalid results altogether, by using Algorithm 5.

Algorithm 5 Finding small connected components (Atlantic City algorithm)

W is initialised as the all-zeroes tag.
S is initialised as the all-zeroes tag.
for $i \in 1, \ldots, n$ **do**
    T is initialised by setting each account, independently, to either 0 or nonzero, where 0 is chosen with probability $p = \alpha^{1/m}$.
    $T \leftarrow (G + I)^d T$.
    Normalise T. ▷ Make all tag values 0 or 1.
    $W \leftarrow W + ((G + I)T \setminus T)$.
    $S \leftarrow S + T$.
end for
$S \leftarrow S < t$.
$s_1 \leftarrow 0$.
$c \leftarrow d$.
repeat
    $W \leftarrow ((G + I)^c W) \cap S$.
    $s_0 \leftarrow s_1$.
    $s_1 \leftarrow |W|$. ▷ Number of accounts in W. This is a nonlinear operation.
    $c \leftarrow 2c$.
until $s_0 = s_1$
return $S \setminus W$.

This is essentially an identical algorithm to Algorithm 1, except that it propagates W as many times as is necessary, rather than a predetermined amount.

In order to keep the number of steps in which we propagate W small, we make sure only to propagate inside $S < t$, i.e. within those accounts that otherwise would have appeared in the final result.

Note that whenever there is an edge, $(a, b) \in E(G)$, such that a is in $S < t$ but b not, this indicates that at least in one iteration the tag value of T was zero in a and nonzero in b. This ensures that W is initially nonzero at a. More generally, any connected component of the $S < t$ subgraph in G has at least one account that is nonzero in W. For this reason, we can propagate W only within $S < t$, without this affecting the results (proving that Algorithm 5 cannot return any invalid components). Within $S < t$, we have already shown that the diameter of connected components is at worst exponentially distributed, making the number of propagation steps necessary for a given a be $O(1)$ in expectation, and over all of G at worst $O(\log |V(G)|)$.

A final trick used in Algorithm 5 is that the halting condition on the final loop is only checked each time after doubling the propagation step count. This has no impact on the complexity of the number of steps taken (because it increases the step count at most by a factor of 4), but makes sure that the number of nonlinear operations in this step is only $O(\log \log |V(G)|)$.

3 Finding non-comingled ML networks

Connected components are merely mathematical objects. In order to use the algorithms presented in an intel context, consider money laundering (ML) networks that use no co-mingling with legitimate accounts. These are networks of accounts, not connected to any other account, such that the only influx of funds into the network is via cash deposits, and the only out-flow of funds is through international transfers, the purchase of crypto-currencies or the purchase of assets.

Thus, we want to find small connected components with these additional properties. Given that we have algorithms for finding small connected components, the question becomes how to impose the remaining restrictions.

Omitting networks that have, e.g., international deposits into them can be done by setting the initial value of W, instead of to the empty tag, to include all accounts into which funds have been deposited other than in cash or through domestic transactions.[1] This ensures that any component containing such accounts that otherwise would have passed our criteria regarding component size and diameter will have a nonzero W within the component. In a component meeting our diameter criterion, after the first d steps of propagating W, this nonzero value will have been propagated to the entire component, ensuring that any such component will not appear in the final result.

Dealing with outgoing funds is more complex. Let us begin with the simplest example, namely of funds leaving the network by international transfers.

If the international transfer is directly from the account, this is simple to handle, and, indeed, nothing at all needs to be done in order to account for it.

The problem is that international transfers often happen through remitter accounts. Similarly, crypto-currency is often bought through digital exchanges and assets are often purchased through trust accounts. If we apply our algorithm in its raw form on networks performing any of the above, chances are that we will not find the network. This is because remitter accounts, exchange accounts and trust accounts are not part of the isolated network we are looking for, and will, in fact, likely connect to a great many other accounts.

We can avoid this if we are able, through some characterisation (which can be either explicit or implicit), to create a tag that identifies all such accounts that need to be ignored.

In defining G, we will use a one-sided-viewable criterion that will ignore monetary flow towards any such accounts. The accounts themselves will be added to the initial W, so that any accounts receiving funds from them will be filtered out, together with their entire components. (Such money flow indicates non-cash money influx.)

4 Conclusions

We have shown an algorithm (with both Monte Carlo and Atlantic City variants) that is able to detect ML networks not employing co-mingling. If money out-flow from these networks is done through remitter accounts, exchange accounts or other intermediary accounts, the algorithm requires the identity of such accounts to be provided, for whitelisting.

The algorithm, in all its variants, is heavier than other FinTracer algorithms: it relies on hundreds of FinTracer propagation steps and tens of nonlinear computations. However, this is probably still reasonable to do, given that this is not an algorithm that needs to be re-run many times. It requires no pre-suspicion or the identification of particular target accounts. Instead, using only general parameters like the size (and possibly the diameter) of networks being sought, it is able to run over the entire Australian economy and return all small networks funded only through cash.

These results can be used as input for later down-stream processing, but can also be returned in an anonymised form to the AUSTRAC user (at the price of revealing the set of accounts in the result to the REs, i.e. in SoNaR mode [5]), after which the AUSTRAC user can consider the structure of each such network in the clear before deciding which networks merit de-anonymisation.

[1] Accounts receiving domestic transactions from accounts not held by participating banks should also be placed in W because we have no way of determining where these funds originated.

A Choosing α

In Section 2.2 we introduced α and defined it to be 1/3.

Other choices are also possible, and can be used to optimise various criteria.

We chose 1/3 because it can be presented as the optimal solution to one such criterion, namely an asymptotically-optimal separation between our hypotheses at equal error rates.

Specifically, given that we want to differentiate between components with (up to) m elements and components with (at least) $2m$ elements based on a per-element statistic, $s = |S|$ (the number of times T is nonzero for a given account), and given that s is distributed roughly normally, we choose α so as to minimise, for an asymptotically-large m, the worst probability for any error type. This happens at a t value for which the probability to make type-one and type-two errors is the same. Under our normal approximation, this error rate is a function of the number of standard deviations between the expectation of the distribution and the chosen threshold value.

If a component has exactly m elements, the expectation of s is $E_0 = \alpha n$, and the standard deviation is $S_0 = \sqrt{\alpha(1-\alpha)n}$.

If a component has exactly $2m$ elements, the expectation of s is $E_1 = \alpha^2 n$, and the standard deviation is $S_1 = \sqrt{\alpha^2(1-\alpha^2)n}$.

Consider a t value for which the distance from the expectation to t, as measured in units of standard deviation, is the same in both hypotheses:

$$\frac{E_0 - t}{S_0} = \frac{t - E_1}{S_1} = \Delta.$$

Then

$$(S_0 + S_1)\Delta = S_0\Delta + S_1\Delta = (E_0 - t) + (t - E_1) = E_0 - E_1,$$

so

$$\Delta = \frac{E_0 - E_1}{S_0 + S_1}.$$

Taking Δ as a function of α, this function has a unique maximum at $\alpha = 1/3$, where $\Delta = \frac{\sqrt{2n}}{5}$ and $t = n/5$ (which is the t value we use in Section 2.5).

References

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

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

[3] 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`)

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

[5] Brand, M. (2022), Fast privacy-preserving edge-finding, SoNaR, and their uses in FinTracer pair problems. (Available as: `edge_finding.pdf`)

[6] Brand, M. (2022), DARK Lite: A fast DARK implementation. (Available as: `darklite.pdf`)