MobileCoin Fog Whitepaper

Mobilecoin

April 2019

Note: This document has been updated as the terminology around Fog has changed.

1 Preface

A basic challenge in any "privacy coin" is that if, by design, it is hard to determine who owns any given coin, then it also becomes harder for a user to find their own coins. A user may need to do this if they get a new phone, for instance, because their phone was lost and stolen. If they still have their private keys they should still be able to do this.

In cryptocurrencies that are merely pseudonymous, a user can simply tell a server their pseudonym and ask for all parts of the blockchain connected to their pseudonym. The bandwidth of this is then only proportional to their transaction volume, and the query can be resolved by a standard database.

By contrast, in Cryptonote, the only mechanism to determine if a coin belongs to a user is to do "view key matching", wherein the user's private key is checked against a given Transaction Output (coin).

Whether you try to do this matching on the user's device or in a remote SGX enclave, it poses significant scalability problems. The root issue is that if for every transaction you have to do $O(n)$ elliptic curve operations where n is the number of users, you're going to struggle to handle significant transaction volume with a reasonable number of machines. If you try to push this computation onto the users' devices, then you force every user to download the entire blockchain and continually burn significant amounts of CPU doing these operations.

The goal of this document is to describe an alternative design, which we call *MobileCoin Fog*. This is a collection of servers that provides Mobilecoin users with an efficient alternative to Cryptonote view-key scanning.

Fog supports both getting only the most recent transactions of a user, and doing full wallet recovery.

The original audience for this document was Moxie Marlinspike circa 2019, to address concerns like the above. The concern is that users reinstall Signal regularly. If recovering their money when they do this is very expensive or time consuming then it will lead to a bad user experience.

Note that Signal does not at present support recovering chat history when you reinstall, precisely because it is difficult to support that without undermining privacy.

We will describe Fog in great detail – the goal of the document is to convince the reader that

1. it will meet our privacy goals
2. it is feasible to implement and scale, using existing techniques
3. it will not create intractable system administration problems to actually run it

2 Design goals

Our primary goals are:

- Efficiency: If there are n users, we want the total work that our servers do in handling a single transaction to be $O(1)$ rather than $O(n)$. If we think of transaction volume as growing as $O(n)$, we want the Fog services to scale like $O(n \log n)$ rather than $O(n^2)$. Particularly, we can't hope to do $O(n^2)$ elliptic curve operations per unit time.

- Privacy: The node operator has zero knowledge about the recipient or amount of any transaction, taking into account all traffic and databases they have access to, taking into account timing analysis, taking into account access pattern leakage, as long as SGX is secure.

- SGX break-in recovery: If a vulnerability is discovered against SGX and secrets are leaked from enclaves, the adversary only gets ephemeral keys. Ideally they learn only about "in-flight" transactions, not all past transactions. When SGX is fixed, the servers only have to apply the patch, and we have forward secrecy without e.g. the users all rotating their keys.

- Upgrade path: If a vulnerability is discovered in enclave code, it must be possible to patch it and redeploy. However, if the enclave contains secrets that are required for its operation, it isn't possible for the old enclave to trust the new version of the enclave and send it the secrets. Therefore, by design, all secrets in the enclave should be discardable. For example if there is a key that is periodically rotated, upgrading the enclave can coincide with key rotation.

- Durability of the service: If servers inevitably crash, machines fail, enclaves lose their secrets, we must not lose the ability to do full wallet recovery for the users. In the ideal scenario, there should be an encrypted dataset not inside of any enclave, which can be replicated and backed up, which we can use to recover a user's transactions, even if all the node operator keys and enclaves are lost.

Note that in a separate document (not included here), we analyzed the Cryptonote ledger with deleted inputs, and demonstrated that under certain assumptions, such a ledger meets a "zero knowledge" notion of privacy. We'll take that as a given in this document.

3 Overview

In this overview, we try to motivate two foundational design decisions. After these decisions are made, the remaining problems split up into a few smaller, and largely separate problems, which we will treat in separate sections.

3.1 Concept: Encrypted Fog hints

In Cryptonote view-key scanning, a user's private view key can be checked against a TXO, at a cost of $O(1)$ elliptic curve operations.

However, we consider it infeasible for the users to download the entire ledger and do this themselves. The next obvious alternative is that Fog can do view-key scanning on behalf of all the users against every block. But if transaction volume is proportional to number of users n, then this leads to $O(n^2)$ elliptic curve operations per unit time, since we do $O(n)$ work per TXO that goes by. Also, if SGX is compromised in this design, then the attacker gets everyones' private view keys, since you need the view keys to do view key scanning.

Ideally, we could circumvent this bottleneck with a different primitive – the users would attach some extra data to every TXO, and Fog would be able to look at this data and in $O(1)$ operations determine which user it is for. Then it would flag it somehow for long-term storage such that the user can find it later.

We propose that this different primitive is actually just any public key cryptosystem. Fog publishes a public key, and the corresponding private key exists only in SGX. When Alice sends Bob a transaction, she takes Bob's identity (or a piece of it) and encrypts it against the Fog service public key, and attaches it to the transaction. This is called the *encrypted Fog hint.*

Because the recipient of every transaction is getting encrypted for this private key and attached to every transaction in the ledger, it is very important to use good hygiene about rotating this Fog service key. Old keys must be deleted after the new one is rotated into place. If old keys are not deleted then temporary SGX compromise leads to revealing all recipients of all historical transactions.

There are a number of details here around how to accomplish key rotation and how do users discover the current public key of a Fog server. We think these problems have multiple reasonable solutions, and we'll defer discussion of these details to Section 6.

3.2 Concept: Fog Service Index

Given that we are going to use encrypted Fog hints, there is going to be at least one SGX enclave that contains the Fog service keys, consumes blocks, and decrypts the hints, and computes some results which it sends to some form of long-term storage so that the user can later access them. We call this enclave the *Ingest node* because it ingests the blockchain.

There are two possible places that the Fog Ingest node can send data for long-term storage:

1. The results are sent to a (cluster of) SGX enclaves and SGX is relied on to protect this sensitive data, which should not be accessible to the node operator.

2. The results are exported to a conventional database not in SGX, that is accessible to the node operator. Cryptography rather than SGX is used to secure it against the node operator, without making it harder for users to search it.

"Option 1" designs that we have considered generally struggle with Upgrade Path, SGX Compromise and Durability.

"Option 2" designs have an easier time with these goals, but require more work to see how the exported data can be both encrypted and efficiently searchable, for each user, without leaking information.

As a general principle, we are taking the point of view that secrets held in an enclave (or even, sealed to a particular enclave) should all be ephemeral – "data at rest" should be exportable outside the enclave. (That's not to say that "Option 1" is definitely not viable, but all currently proposed "Option 1"-type plans have struggled to create a viable Upgrade Path.)

This proposal is about "Option 2". The append-only database that the Ingest node writes is called the *Fog Service Index.*

The design of the Fog Service Index ensures that:

- A user who has access to the fog service index, and their private keys, can recover their transactions. They don't need anything else, particularly any secrets from any enclaves.

- The node operator doesn't learn anything from access to this index, it "contains zero knowledge" and it is okay for it to be publicly accessible.

The Fog Service Index thus has similar a cryptographic status to the blockchain itself, and can be handled and distributed in similar ways. The advantage is that it is efficiently searchable, and Bob can find his transactions within it without doing massive numbers of public key operations, or scanning the whole database.

With the Fog Service Index in hand, we will ultimately need to serve user requests to it *privately.* If the node operator learns which rows of the Fog Service Index a user cares about, they learn which blocks a given user received transactions from, since they know which rows of the Fog Service Index were emitted after processing which blocks.

Therefore, if users will not download the Fog Service Index themselves (which is as large as the blockchain), this design requires that a system is built to *obliviously* search the Fog Service Index on behalf of the users. Here *oblivious* means that no matter what queries the user makes, the memory access patterns are *the same* or *indistinguishable*, so the node operator learns nothing from this important side channel. Oblivious refers to a cryptographic standard of privacy, as opposed to mere obfuscation. We'll give a more precise definition of this in the next section.

We propose that a cluster of SGX machines called *Fog View Nodes* is used to serve this. Users connect to these machines and make requests over an encrypted channel, and Intel

Remote Attestation provides a chain of trust that the search will actually be performed obliviously without leaking the requests to the node operators. The merits of this design are:

- Easy SGX upgrade path: The View nodes only need to be trusted by the user, not Ingest or Consensus. They don't hold any long term secrets that would need to be moved.
- Durability: Even if all the enclaves got wiped out simultaneously and irrecoverably due to a datacenter failure, as long as the Fog Service Index is backed up durably, we will be able to restore easy access to everyone's historical Transaction Outputs.
- SGX compromise: Recovering from SGX compromise means patching SGX, bringing up new Ingest nodes and performing key rotation, and bringing up new View nodes, and patching the client to trust the new enclave versions. Users do not need to interact, or abandon their old keys – we can have forward secrecy.

The two major questions posed by the Fog Service Index idea, that we answer in this proposal, are:

- Exactly what crypto and structure should exist in the Fog Service Index? (Section 5)
- How exactly do we serve oblivious, private queries to a large dataset like the Fog Service Index? (Section 4).

It turns out that generally, solutions for obliviously accessing data don't care what the data looks like or what distribution it has, etc. So assuming we can solve the large, private data access problem, it's not going to impose restrictions on what kinds of crypto we can use to actually define the Fog Service Index contents.

Since this private data access problem is actually the most controversial part, and it is an important building block for the rest of the system, we will describe that now.

4 ORAM

For our purposes, an ORAM can be defined as: an abstract data-structure, like a binary tree or a hash table, which functions as a key-value store, and has the additional property that for any two sequences $(\sigma_1, \ldots, \sigma_m)$, $(\pi_1, \ldots, \pi_m)$ of keys and mixed reads or writes, the memory access patterns are indistinguishable.

The *trivial ORAM* is just an array of key-value pairs, and to read or write we make a *linear scan* across the entire array, using branchless operations to "pretend" to interact with every memory location. It is oblivious because each time, no matter what is requested, the access pattern is exactly the same.

If the ORAM has size n, the trivial ORAM takes computation time n. The general goal in ORAM is to try to do better than n, like, $\log n$ or so.

It is known that nontrivial ORAM must be probabilistic [5]. This means, the data structure has internal randomness not known to the adversary, and for any given sequence $\sigma_1, \ldots, \sigma_m$, there is a distribution of memory accesses rather than a fixed possible sequence. The goal is that these distributions are indistinguishable for any two usage patterns, under standard cryptographic assumptions.

ORAM goes well together with SGX, because

- SGX provides hardware assistance for actually encrypting the working memory, but does nothing about concealing the access patterns to the memory. ORAM fixes the access patterns, but does not directly do anything about actually encrypting the data. Together they do both.
- SGX provides the RDRAND source of randomness within the CPU, so we have access to randomness as part of the trusted hardware, that the adversary cannot control. This is needed for the access patterns to actually be unpredictable.

These observations were also made in the ZeroTrace paper [7], which implemented Path ORAM inside of SGX in 2017 and published performance numbers and C/C++ code.

4.1 Historical Background

Oblivious RAM [5] is a body of techniques that try to address the problem of "access pattern leakage" in cryptography. Initial papers anticipated that users would want to be able to store and retrieve data in "the cloud", but they point out that even if you encrypt the data, you reveal what parts of the data you are accessing when you access it, which may reveal important information.

The original definition is not the same as the definition we gave just now. In the first few papers, ORAM is described as trying to mock the idea of "block storage", backing it instead with the untrusted remote server. Usually these papers work just as well for an arbitrary key-value store. In either case, we want that the low-level memory access patterns on the server don't reveal what high-level key or address was requested from the ORAM dataset.

It is surprisingly tricky to come up with something significantly better than the trivial ORAM, that is actually secure in the desired sense.

As an example, another naive attempt is to use a hash-table with a cryptographic hash function and a private salt. The client chooses a private random number, then uses the server memory as the arena for a hash table. It hashes its items, then encrypts them and places them in the server at the specified address. The server is getting addresses that depend on these hashes, but in the random oracle model it is hard to take these hashes and learn anything about the requested item. It turns out that this strategy works, as long as no item is requested more than once. But once an item is requested twice, the hash will be the same – in different settings this can lead to many forms of doom. In some scenarios relevant to us, just knowing that Alice asked about the same item twice in part of the ledger will leak information about whether she received a payment or not.

The first interesting ORAM, with logarithmic overhead, and tolerant of any access pattern and repeated accesses of the same item, is Goldreich Ostrovsky '96 [5]. This design is simple and influential. Instead of a single hash table, there is a hierarchy of hash tables, each one 4 times larger than the previous. The first hash table is constant size. The design is based on the following ideas:

- Items are always added to the smallest table. Then we check the load factor of the table – if it exceeds a threshold, then we shuffle the entire table into the next larger hash table. This repeats as many times as necessary (but provably not very often).
- When we search for an item, we search for it in each hash table, starting with the smallest. Once we have found it, we must continue to make queries, at *random* positions in each of the remaining hash tables, irrespective of the hash.
- After searching for an item, if we found it, we must remove it. If we didn't actually want to remove it, it gets reinserted at the top.

It's important here that the shuffling is also *oblivious.*

Many subsequent papers tried to improve details such as the shuffling strategy, or to use cuckoo hashing instead of standard hashing.

More recently, there are a number of papers describing the drawbacks to these proposals and the difficulty of actually deploying something like this in practice, particularly Boneh, Mazieres, Popa in 2011 [2]:

- ORAM generally turns every read into a write. In a conventional replicated database like PostGRES this is doom – you need to have many readers but you can't have many writers, it will be impossible to synchronize them all.
- Early papers were only logarithmic in an amortized sense, but the worst case time was n and would happen regularly, and the forced reshuffles completely lock up the server.
- Because each query depends on many back-and-forth messages between client and server, there are some attack vectors like the malicious server responding to a request only partially and then dropping it, forcing the client to retry. In some schemes, like Goldreich Ostrovsky, when the client retries, the second request leaks information.

These concerns are all quite reasonable – there are a lot of pitfalls here associated to a "private distributed data structure" that these academic papers don't address.

It turns out that many of these difficulties can be sidestepped by using ORAM in a different configuration than initially envisioned. In more recent papers like Path ORAM, the setting of trusted hardware is treated as just as important as the cloud-storage model. It is very natural to put the "client-side" of the ORAM in protected memory in trusted hardware, and this has been one of the main reasons for interest in Path ORAM.

Our proposal is that an ORAM can exist entirely as an in-memory data structure in an SGX enclave – both the client and server sides of it. This idea was described and implemented in the ZeroTrace paper from 2017 which also provided C/C++ code and benchmarks [7].

Importantly, this usage requires that both the "client" and "server" sides of the ORAM are both running in SGX, so the node operator can see access patterns of both, not just the server side code. It turns out that for real ORAM designs this isn't an issue, the client-side of the data structure is generally just hashing things or doing linear scans over blocks fetched from the server, and can already be done obliviously.

4.2 Tree-Based ORAM

A breakthrough in ORAM was the move to so-called "tree-based" ORAM. Tree-based ORAMs generally have logarithmic worst case access times, rather than merely amortized access times. They don't have costly reshuffles that cost $O(n)$ and lock up the whole database.

Tree-based ORAMs like SSS ORAM and Path ORAM have the following features [9]:

- The data is organized in a binary tree. Each node in the tree is called a "bucket" and can contain a fixed number of "blocks", for example, four blocks, of fixed size. The nodes of the tree are laid out contiguously in memory using standard techniques.
- An invariant of the data structure is that each data item is "mapped" to a leaf, and can only be located in a bucket on the path between the root and the leaf.
- When items are added to the tree they are added at the root bucket (similarly to Goldreich Ostrovsky), and mapped to a random leaf.
- After every operation, there is an operation that pushes items down from the root bucket towards the leaves. It does so only along one path in the tree. (This is different from Goldreich Ostrovsky where things are only pushed down to the lower hash tables as needed, and always an entire level at once.)
- In order to search for a particular item, we have to know what leaf it is mapped to. The data structure stores this mapping information in an auxiliary table. The nature of this table is different in different papers – in Path ORAM, this auxiliary table is another, smaller Path ORAM. In our case, the block size is expected to be something like 4096, the page size in Linux. With blocks this large, the recursion is expected to bottom out quickly in practice, like 3 or 4 rounds.
- Typically there is a small "stash" where items get placed if the buckets overflow, which gets flushed out afterwards.

The main ideas to see that a data structure like this is actually oblivious are

- Appeal to recursion, means searching for the block containing our item in the position map happens obliviously
- Searching along the path in the tree obliviously, leaks only the current position. But positions are chosen uniformly at random, so it is independent of the search key.
- Since the item is removed after it is found and reinserted, in any future query the position will be different and statistically independent

4.3 ORAM Cluster: Path ORAM

It's very important that the system that serves the Fog Service Index to users is *horizontally scalable.*

We propose that this dataset is partitioned, arbitrarily, across several worker nodes, with some replication of worker nodes to allow load balancing.

This is essentially the same cluster / partitioning strategy as was proposed in the Fri May 24 meeting, in the "Linear Scanning Proposal". The only major difference is that we propose to use an in-memory Path ORAM rather than in-memory Linear Scanning. This gives log-bounded worst-case response times for each worker node, so even if each partition is large, the worst case response time is fast compared to linear-scanning the whole partition.

Partitioning based on age of data item allows a strategy similar to cursoring. Today, Mobilecoin assumes that IP is deanonymizing, and can be linked back to the user's phone, so when the signal client queries the fog service, the node operator knows who they are. Cursoring means, when the user asks for new TXO's, they mention e.g. "don't search anything before block X" where X was the block time when the user last queried. Since the node operator knows who the user is by assumption, and also knows where they stopped searching last time they connected, cursoring reveals zero additional information, and it can dramatically improve the efficiency of the system.

However, we would like to ensure that the system won't fall over even if users don't use cursoring properly. Since we can't expect filtering to be 100% effective, we want to bound the worst case cost of any particular request that is handled by a worker node. There then should be enough workers so that the number of bad requests, times the cost of those requests, doesn't swamp the workers.

4.3.1 Path ORAM in DRAM

Let's estimate the cost of a request to a worker with 64GB of DRAM using Path ORAM.

Benchmark Assumptions:

- In Intel 9th Generation CPUs, the cost of a page fault in DRAM is ≈ 200 cycles, improved from ≈ 2000 cycles in earlier hardware
- On this hardware, the cost of an elliptic curve operation is ≈ 100 microseconds

This means that if in the course of interrogating the Path ORAM, we cause 1000 page faults, then servicing the lookup will take time similar to performing an elliptic curve operation. This means that if this response time already leads to DDOS, then we were already doomed, even without the database, due to the need to perform Intel Remote Attestation.

How many page faults do we actually expect to have?

If 64GB of RAM is divided into 4k blocks and arranged in a binary tree, this leads to ≈ 16 billion blocks and a tree of height of about 24. There are also the recursively smaller data structures, but these are progressively smaller. How much smaller depends on the block size that is used. In the position map, we need to store positions of blocks in the larger tree. If we use 8 bytes to represent a position, and use blocks of size 4k again in the position map, then we can store positions of 2^9 blocks in the main tree in a single block of the position map. This means there are fewer blocks in the position map, and the corresponding binary tree has height which would be 9 smaller. Since there might be overheads, lets assume its only a factor 2^8 smaller. If the first tree has height 24, the next one may be of height 16, and then 8. We have to follow two paths rather than just one in each tree in order to service a response, since we have to do the "cleanup" step afterwards that pushes items down in the tree obliviously. So, this suggests that we might have on the order of 100 page faults to service a response, leaving us significant headway until we hit 1000.

4.3.2 Path ORAM in an SSD

We can also run the numbers for what happens if the data is stored on a large SSD instead of in RAM. By turning on Swap, the Linux kernel can evict pages owned by SGX to disk, without a code change.

If we imagine we have a 16 TB SSD with an access time of 100 microseconds, divided into 4k sized blocks, this leads to a binary tree of height about 32. The height of the recursively smaller trees might be like 24, 16, 8. If we have to access two paths on each one, this leads to an estimate of 150 page faults, and a nominal total access time of ≈ 15 milliseconds.

There are many assumptions here and we will have to do real tests and measurements do predict realistic performance. However, these estimates also square with the published results in the ZeroTrace paper in 2017, which similarly implements Path ORAM in SGX [7]. and benchmarks the performance.

4.3.3 Projected Workloads

What sorts of loads do we actually expect on these servers?

We believe that the theoretical maximum speed for consensus is close to 1000 Tx/sec.

- Assuming 1000 Tx/sec avg continuous load, and 500 bytes per transaction with overheads, 500 kb/s leads to 300 gb / wk, of data generated
- Using a RAM strategy, this leads to ≈ 1.5 days of data in a 64GB RAM handling $\approx 10,000$ requests per second
- Using an SSD strategy, this leads to ≈ 1 year of data in a 16TB SSD handling ≈ 60 requests / second

Let's suppose that we model the Fog service user base as follows:

- ≈ 20 million daily active users
- ≈ 100 reinstalls per second

Let's assume that a reinstall for a daily active user means making on the order of 400 requests (for 12 months worth of transactions, 1 per day), this leads to 40,000 requests per second against the most recent year. In order to keep up with continuous request volume like this, we would need

- About 4 * 250 = 1000 RAM machines per 20 million users, or
- About 700 SSD machines per 20 million users

to service the most recent year.

Assuming we go with SSD's, this implies an (upper) estimate of 35,000 machines to support the requests of a billion users, for a year.

5 Fog Service Cryptography

Assuming that we can serve private queries to the Fog Service Index, in this section we try to think about what should the Ingest server actually be doing, and what is the structure of the Fog Service Index.

Recall that the Ingest server consumes blocks, attempts to decrypt the fog hints using the private key to determine the recipient, Bob. If it succeeds, it needs to create a record that goes into the Index that Bob can eventually find.

Recall that in Cryptonote, a user has the following keys:

- "private spend" a
- "private view" b
- "public spend" A
- "public view" B

We propose that the "encrypted fog hint", which Alice attaches to the transaction for Bob, is simply an encryption of Bob's B value, using the Fog Service's current public key.

We propose that record that the Ingest node emits, which becomes a row in the "transaction table" in the Fog Server Index, contains:

- A random search key, which is predictable by Bob, but no one else
- An encryption of the TXO, against the public key B, using a public-key cryptosystem like ECIES.

Where do these search keys come from? The simplest thing that could work is to use a PRNG, where the seed to the PRNG is the result of a key exchange between Bob and the Ingest Server.

- The ingest server enclave generates a private nonce n (using RDRAND)
- The shared secret is e.g. HKDF of nB. This is used to seed a PRNG which is stored in memory in the ingest server, in a table $B \to PRNG$.

- The ingest server computes the corresponding curve point N and publishes a record $B \to N$, to the "seed table", an additional table within the Fog Service Index
- The Ingest server securely deletes n inside the enclave.

Under the same assumptions that Cryptonote is secure, and the ECIES cryptosystem is secure, this usage of key exchange is also secure – given access to the seed table, an adversary has zero knowledge about what Bob's PRNG seed is. If the PRNG is cryptographically secure, then given access to arbitrarily many of the random outputs of the PRNG, the adversary cannot predict future or past outputs of the PRNG.

With this PRNG in place, the ingest server can now handle any transactions that come in for Bob by encrypting them against B, finding his PRNG and pulling a random number from it and emitting this output and the ciphertext as a new row for the "transaction table" in the Fog Service Index.

5.1 When do the PRNG's get created

We propose that the node operator should explicitly tell the Ingest node about each user B that it should support, and at that time, it creates a PRNG and emits a seed row for this user. Because the seed row is emitted in response to this plaintext request to support Bob, the B part leaks no new information, and the N part is a nonce. This lets the node operator predict how much memory the node needs to support its pool of users.

5.2 Handling Unknown Users

What should happen when a TXO comes in and its fog hint does not decrypt properly, or decrypts to a curve point not corresponding to any of our users?

If in this case we output nothing to the Fog Server Index, then the node operator has a simple test they can use to determine if TXOs are for a specific user – they spin up a new fog server replicated to have the current Fog server key, tell it to support only this particular user, and then watch for when it emits new rows.

Therefore, the observable behavior of the ingest node must be indistinguishable whether or not the TXO was for a supported user. We must output a totally random Fog Server Index row, with the same distribution as if it had been authentically constructed.

5.3 Upgrade Path

Because we now have additional state associated to Ingest enclaves, not only the Fog Service private key, it raises the question of how the upgrade path works.

The main observation is that all of these PRNG's can be safely discarded – none of this state actually has to be moved to the new version of the enclave.

In the next version of the Ingest node, key rotation will happen for the Fog Service Private Key, and the old one will be deleted, and similarly, new PRNG's will be created for all of the users that this Ingest node supports, and new seed rows emitted for all of them as well. Users will be able to reconstruct their PRNGs for any version of the enclave and make appropriate requests to the view nodes later.

Naively this means that a user that has k seed rows has to make k times as many requests to the view node to search for new transactions, which means that performance would degrade over time. By putting metadata in the seed table about which PRNG's are still active, the users can stop making requests against PRNG's that have been abandoned after the first time they get a missed request for that PRNG.

The seed table will grow over time as more upgrades happen, but it will always be much smaller than the transaction table.

Finally, note that the Upgrade here doesn't require a simultaneous upgrade to Consensus or View node enclaves, because none of these nodes directly attest to each other. The user needs to trust that the Ingest node is actually operating in SGX as specified, so the Signal client should refuse to encrypt fog hints for a fog server public key unless there is a chain of trust proving that that public key came from an Ingest node enclave of the expected version. But this only means that upgrading the Ingest node requires patching Signal and not anything else.

5.4 Data structures & Scaling

Because the keys to the PRNG table are sensitive (they identify the recipients of transactions), this data structure must be oblivious. If it isn't, the malicious node operator can get nontrivial information about the recipients – for example, if it is a simple hash table, Eve can send a penny to Bob and write down the address that was looked up in the hash table, then later if Alice sends a transaction, and that same address (for Bob's PRNG) is accessed, then Eve knows that Alice must have sent money to Bob.

We'll consider two different possibilities here for oblivious data structures, and try to make estimates when the block time is 5s and the peak load is 1000 transactions / sec. The main question is whether the fog server node will be able to keep up, and how many users it can handle.

5.4.1 Workload

There are two parts to the fog service workload – decrypting the fog hints, which requires elliptic curve operations and is CPU bound, and finding and advancing the PRNGs which is memory bound. In particular, if we use a PRNG based on AES as the state transition function, then advancing the PRNG state is extremely cheap on our hardware, because Intel SGX is based on hardware-accelerated AES.

If it takes us 100 microseconds ms for a single core to do an elliptic curve operation, then at 10,000 tx / sec we have fully loaded a single core, just to decrypt the fog hints. Since we only expect 1000 tx/sec with our consensus algorithm, this means that we spend only 10% of our 5 second budget to handle the block to do the decryptions, using only a single core.

A good implementation would probably take advantage of multithreading, since the fog hints could all be decrypted in parallel. In a machine with 10 cores doing this, we'll spend about 1% of our budget doing the decryptions.

5.4.2 Path ORAM on an SSD

Suppose that we use Path ORAM to store the map from users to PRNGs, and we store it in swap on a 1TB SSD, which as before has an access time of 100 microseconds. In 1TB we can store at least 10 billion PRNGs and user pubkeys, with overheads, enough for all the users we would plausibly have. This leads to a binary tree height of about 28, and we estimate that it takes about 128 faults to do a single lookup. This leads to 12.8 milliseconds to find a single PRNG, and the cost of obliviously advancing it will be much less than that. This means a single machine can only actually do about 80 Tx / sec this way.

To make up the difference, we can share the load across 12 machines instead. Each one can be configured to only decrypt and handle every 12'th Tx in the block. They would each share the fog server private key used to decrypt the hints, via node-to-node attestation, and the node operator would configure them to make sure that the whole block is handled. They don't otherwise have to communicate, and their outputs are pooled into a unified Fog Server Index.

These 12 machines could handle load up to 1000 tx/sec and over 10 billion users, so they could handle all expected load for the foreseeable future, with the caveat that the SSD's might have to be replaced regularly.

The main drawback of this kind of work-sharing is that these machines have completely separate PRNG's and a user now has 12 times as many seed rows. This means that they also make 12 times as many requests to the View node cluster. Additionally, they use 12 times as much bandwidth on their cell plan, to accomplish the same thing as if they only had one seed and one PRNG, so this kind of work-sharing is not free. Nevertheless, Mobilecoin transactions are not that big, close to 400 bytes at time of writing, and a user who does 1 daily transaction, but must send 12 seeds and obliviously recover 12 possible TXOs from the fog server per day, is still only doing about 5kb / day of traffic.

We could imagine trying to share the PRNG states across several machines, so there is an ORAM cluster that stores PRNG states and several fog server worker nodes that recover the PRNGs from this cluster and then put them back, so that Bob still only has one seed even though we are sharing the work across several machines. This would require networking overhead which swamps everything else, and it doesn't same possible that such a proposal leads to fewer than thousands of machines to deal with this bottleneck. If the number of seeds / required bandwidth for daily active users to do full wallet recovery becomes too high, a better direction may be simply to get machines with faster SSD's, since that allows each machine to handle a larger transaction volume without having more seeds or sharing the memory.

5.4.3 Path ORAM in DRAM

By using a faster, smaller memory, we could support a smaller set of users, using fewer machines, because each machine can handle a larger transaction volume when the memory is faster.

Suppose that we use Path ORAM to store the map from users to PRNGs, and store it in 64 GB of RAM, which we assume on recent CPUs takes 200 cycles or about 200 nanoseconds to access. In this RAM we could plausibly store RNGs for somewhat less than 1 billion users (its 64 bytes for the pubkey and PRNG together, but there will be various data structure overheads as well).

Assuming as before about 100 page faults to service a single lookup in the data structure, that puts us at about 20 microseconds ms / lookup. So we can find about 50,000 PRNGs obliviously in 1 second. So, a single such machine should easily be able to keep up with 1000 Tx/sec volume, and store RNGs for almost a billion users.

When we exceed the number of users that can be supported by one machine this way, we could partition the users arbitrarily and then assign different machines to support the different parts, using the same fog service key. (A.k.a sharding). Because as described in section 5.2 each shard outputs a row no matter what, even when the user isn't in its shard, the node operator doesn't learn which shards had users in any particular block. The main drawback of this is that if there are 10 shards, this results in an Fog Service Index that is 10 times as large, and is therefore more expensive to search.

An alternative strategy is to replicate all of this infrastructure for each shard – each shard has a different FQDN, e.g. "fog.1.signal.org", a different fog server public key, a different Fog Service Index, and a different set of view nodes. Everyone knows what shard every user is in, because the FQDN is part of their public information (this is how the users figure out what fog server public key to use for the given user). Each user only needs to search the Fog Service Index for their shard, and "accounts.1.signal.org" and "accounts.2.signal.org" are totally independent.

5.4.4 Linear Scanning in DRAM

For completeness, we consider what happens if we use the trivial ORAM here and do a linear scan over the set of all PRNGs, for each block.

Assuming that there are 64 bytes per user (32 bytes for their key, 32 bytes for the RNG), each 4k page has 64 users. On older hardware a common estimate is 2000 cycles to fault in a page of DRAM – on 9th generation hardware it's projected to be closer to 100 nanoseconds.

If we are attempting to handle a block with $k \leq 5000$ TXOs, we would have k decrypted fog hints. To recover the corresponding RNG states, we would have a buffer for each one. We would scan across the PRNG table, and for each PRNG, we would obliviously compare the hint with the search key, and then obliviously copy it to this buffer in the case of a match. If each key and RNG state is 32 bytes, then for each combination it's at least 4 operations worth of XORs, 4 operations worth of binary AND, and 4 operations worth of CMOV. Since $k \geq 64$, we need to scan across several pages here. If this is memory bound, we estimate that we spend at least 100 nanoseconds here, times $k/64$. Then we do this 64 times to handle the current page of users. If $k = 5000$, this leads to about 500 microseconds to handle 64 users, so in a second, perhaps we can handle 128,000 users. This represents 8MB of RAM at 64 bytes a user, so we aren't going to be able to make full use of our RAM in this configuration.

Then we would advance the PRNG's obliviously as appropriate. Since it's possible that a single user received all of the transactions in a block, we would have to pretend to advance each of the k recovered PRNG's potentially k times. If we use a PRNG based on the AES-NI instruction set, it might take us about 100 cycles to compute one step of a hash. Even if $k = 5000$ we have to do this 25 million times, it is still significantly less than a second.

If we have 5 seconds to spend scanning the users, then we can have 512,000 users on a machine this way, and still do the block in less than 5 seconds. We could set up many fog servers, with different public keys etc., and make sure only to assign 512,000 users to each one. But then we need to replicate the view nodes for each fog service that we create this way. An advantage is that not as many users are making requests against those view nodes, since each user is only making requests against the nodes for their assigned fog service. The main problem with this is that as the number of users scale up, we would need thousands of parallel fog servers to serve all the users. This puts a lot of pressure on the view-node side of the equation. Either you can serve a large set to 512,000 users with a small number of machines, or you end up with a very large number of view nodes overall.

An alternative configuration is something like, choose $k = 64$, and have 64 machines each handling only the 64'th TXO in the block. Working in parallel, they could support 4096 transactions in a block, and all the temporary RNG buffers fit on a page now. If we can scan a page of users 64 times faster now, then each ingest node can possibly visit 8 million users' PRNG's in a second. This leads to 64 machines being assigned to maintain PRNGs for 8 million users and match their fog hints against the PRNGs. But the machines cannot sync these PRNGs – they are different, and the user has 64 times as many seeds, and makes 64 times as many requests to the view nodes. And since we can only deal with 500MB of RAM in a second, and the block time is 5 seconds, we are still only actually using a few gigabytes of the RAM on these machines. This leaves us with fewer parallel fog servers in the end, but the view nodes are serving many times more people, and those people are making more requests, so ultimately this doesn't seem favorable.

5.4.5 Conclusion

The main challenge in scaling the Ingest node is that the design require per-user state, which then has to be accessed privately. It makes sense to consider whether this is really a design requirement.

We think that it is, because, if we don't want Bob's TXOs to be linkable, then we need to be able to output a predictable sequence of different numbers for Bob's first TXO, his second TXO, his third TXO, etc. This means at a minimum, the Ingest node needs to be in a different state after the first time Bob gets a TXO, the second time, the third time, etc. At a minimum, this suggests there must be a counter per-user, and an RNG state isn't that much larger than this, especially if we also need to identify users by their public key.

This application seems to benefit significantly from using a nontrivial ORAM, for a few reasons

- When doing linear scanning, we can only page in memory so fast – we get bottlenecked on access time and can't even make use of a full 64 GB of RAM before the block is over.
- When doing linear scanning, to avoid timing attacks we must actually spend the same amount of time on every user, and it leads to us actually becoming CPU bound. When using a nontrivial ORAM we only need to spend the same amount of time for each entry that we actually loaded, and the only CPU-bounded part is decrypting the fog hints.

Linear scanning may be adequate for a small number of users, but as the number of users and transaction volume scales up towards 1 billion users and 1000 tx/sec, a Tree-based ORAM like Path ORAM is expected to be several orders of magnitudes more efficient in terms of hardware requirements, whether using RAM or SSD's.

Moreover, it seems that for quite a long time after launch, the Ingest node load could plausibly be handled by just a single machine with commodity hardware, even up to 1000 tx/sec and hundreds of millions of users, when using Path ORAM or similar.

6 Fog Server Key Rotation and Discovery

There are a few questions about how does a fog server stand up, announce its key, and how do users find it.

- We propose that fog servers are identified by an FQDN, the "name" of the fog service, and a given name resolves to different public keys over time as the keys are rotated
- The FQDN of a user e.g. Bob is part of his public address. When Alice wants to send Bob money, and Bob is using an fog service, Alice must resolve Bob's fog service FQDN correctly. If she doesn't, Bob is assumed not to be able to find the money she sent him. This is similar to if Alice sent money to the wrong address – it's like sending cash in an envelope and isn't recoverable.

Here are three possible plans:

1. Alice simply contacts Bob's FQDN directly, and asks for the current public key. A expiry date will be attached. Attestation is used so that Alice can trust that this key actually came from an ingest node.

   The main drawback here is that traffic analysis reveals that Alice contacted Bob's fog service before sending him money, which leaks information about who Bob is. To fix this, there can be a centralized, oblivious lookup service, of some kind.

2. A service exists which serves queries to the map from FQDN to public keys. This map lives in SGX. The node operator provisions this service with a list of FQDN's and root certs used to authenticate a third party attempting to change the key. To publish a new key associated to an FQDN, both Intel Remote Attestation must be used to prove that the new public key once came from an Ingest node, and a measurement of that enclave recorded as well so that users can know what version of the ingest node (so that they can decide whether to trust the key.) This allows that even if different service providers use slightly different ingest enclaves, they can still use the same version of this infrastructure, and clients will be able to interact on the network as long as they mutually trust each others ingest enclaves. The key is also signed with a cert chain leading to the root cert for this FQDN, to prevent a malicious person from triggering key rotation to arbitrary incorrect keys and causing lost blocks.

   The main drawback here is that potentially, an adversary could try to block or filter out requests to perform key rotation for a given FQDN, to prevent this key rotation. (This might be because they managed to steal a key and want to force users to send traffic that they can read.) One way to mitigate this is to have expiry dates on the keys – an adversary gets nothing from this, except that users of an particular fog service can't send money.

3. Key rotation actually happens on the mobilecoin blockchain. This is the same as (2), but requests to perform key rotation are processed by consensus nodes as part of a block. A difficulty here is that for this to work, each consensus node should have the same authorization key associated to each FQDN. They could use the consensus algorithm to vote for changes to this list, so that in order for an fog service to stand up, a quorum of node operator must agree to vote for an FQDN / root-cert pair.

We prefer solution (2) to solution (3) because (3) adds significant additional complexity, and it's not clear that there's a compelling advantage to (3) over (2). (3) distributes the trust across node operators, and makes it harder for someone to filter out the "publish key rotation messages" since they could potentially look indistinguishable to a "send transaction" message. But (2) could also be done in a way that distributes trust across several people. When expiry dates on keys are used, it's not clear that blocking key rotation messages gets an adversary anything more than just blocking user access to the key fog service does.

An additional question is, who can stand up fog services, and is there an automated process for this.

1. We suggest that to stand up an fog service against an FQDN, a LetsEncrypt certificate of control of that FQDN could be required as part of an automated process.

2. Possibly, they could also have to pay some money, to help rate-limit this.

3. In solution (2), where the service is standing alone from the blockchain and everything else, it could serve these requests in an ORAM in RAM or on an SSD. This would allow the service to sustain even a massive Sybil attack, since we only expect a handful of legitimate fog services.

4. The database mapping FQDN $\rightarrow$ Pubkey could be written out to disk and loaded from disk reliably, as long as we include all the associated proofs that validated the entries – the Letsencrypt certs, the Attestation reports, the signatures, etc. This would allow us to create read-only slaves that serve the data.

5. Alternately, instead of a single writer with many read-slaves, we could actually use Stellar Consensus Protocol and have a blockchain containing all key rotation records, so that messages to publish a new key can be published at any node.

6. There are likely already existing blockchain projects that want to use block-chain to solve key fog problems – we could use one of them and add an SGX-based oblivious lookup service that serves requests to its blockchain. As long as this service is able to also include proofs of Intel remote attestation with the certs then it seems likely to work.

Since when key rotation happens, lots of users will make requests to this service, it should be possible to scale it for reads from the beginning. But at least at the beginning, the simplest thing seems to be to have a manually-curated whitelist of fog servers, and to not worry about scaling to handle many writers, since key rotation is unlikely to happen often.

We think that there should be a centralized service to prevent balkanization, if there are several Mobilecoin clients and even, several Mobilecoin Ingest servers. If Alice is not able to discover Bob's fog server public key, then Alice cannot send Bob money, so there should be a common interface so that Alice and Bob can transact even if they don't use the same client. If each client is building its own fog service, and each has their own key fog platform, then each ingest server would have to publish its keys in many places and not just one. It should be possible to build a single reasonable API for key publishing and key fog, especially since it is important for privacy that client's don't get tricked into using a key controlled by an adversary.

One possible advantage of putting the fog server key rotation in the consensus nodes is that, it can avoid race conditions around key rotation. When Alice submits a transaction for Bob, she can essentially attach an assertion "and the key for FQDN foo is bar". The consensus enclave can check this assertion – if it is wrong, it can reject the transaction and tell Alice that her key for foo is stale, and tell her the new key, so that she can rebuild the transaction. If the assertion is correct, then the transaction is accepted.

7 Operationalizing Fog Service Key Rotation

When running a service, failure of individual machines is inevitable, and the service should plan for and adopt practices that mitigate the impact of this.

When an fog service operator wants to rotate the fog service key, the first thing they must do is spin up a new ingest node enclave, or ask an existing one, to choose a new random private key. An API can be exposed that tells the service operator what the new public key is. The service operator can then sign this key, and give the cert chain to the enclave.

At this point, they could attempt to publish the key, and a third party would have both Attestation (proving that the service operator doesn't know the private key) and the cert chain (proving that the service operator really did authorize this rotation).

However, they still have a single point of failure. Suppose this new public key goes live, and then the machine fails. It's possible that after it creates the key, it could write it to disk, sealed against the MRENCLAVE value, so that the node operator can't read it but a future incarnation of the enclave could.

This doesn't help if

- The machine has a hardware failure and is irrecoverable
- The machine fails not permanently, but for an extended period of time – in the meantime an enormous backlog is building up of blocks that we have not processed, and no other machine can create an enclave with the same MRENCLAVE value so we can't get this key back until this particular piece of hardware is working again.

In general, sealing has limited use in SGX besides protecting against transient power failure.

Instead, we propose that

- Ingest nodes are permitted to use node-to-node attestation and communicate fog service private keys
- This is always done to replicate a key, ideally with several machines in several data centers, before it goes live and users start encrypting against it

This node-to-node attestation is useful for more than just redundancy. We described earlier in the section 5.4.2 that when using Path ORAM with an SSD, in order to keep up with 1000 Tx/sec and overcome a memory bottleneck, we may need to have 12 machines working in parallel. This node-to-node attestation can be used by the node operator to make sure that they have the same key.

Note that we *do not* suggest that the PRNG's associated to users should be replicatable by node-to-node attestation. If this is permitted, it allows for off-line attacks, where the node operator replicates a node, gives Bob hundreds of transactions, and writes down the PRNG output values. Then for the next several hundred transactions of Bob's, the node operator can predict the PRNG values. Because the PRNG seeds come from RDRAND which is part of the trusted computing platform, the node operator has no way to control the PRNG seeds given to users as long as we don't give them a means to replicate those seeds [1].

When describing the upgrade path for Ingest nodes, we explained that both the fog service private key, and the PRNG states, are discardable. However, for a normal key rotation operation, it is unnecessary to rebuild PRNG states. We suggest that e.g. an fog service operator would have two groups of 12 ingest nodes, the A group and the B group. If A group is live, then B group will support the next round of key rotation. A node in B group will be asked to generate a new key – then the other B group nodes will use node-to-node attestation to get the key. We can validate that these keys match. Then we will provide a cert for the public key and ask B group to publish it, but also ask users not to use it until block $N + 1$, where block N is the block height at which the key in use by A group will expire. Once we pass block N, B group will start consuming blocks and writing them out. When we are sure that A group has fully processed block N then we can take them offline, or, prepare them for the next round of key rotation.

Throughout this, A and B do not have to toss their PRNG states – they can keep using across key rotation windows. The PRNG's only need to be discarded when SGX itself, or these enclaves, are patched and must be upgraded. This limits the growth of the seed table in the Fog Service Index – each user only ends up with a seed for each time that we upgrade the ingest enclave, and each node that we run in parallel, rather than with every time that we rotate keys. This also allows us to rotate keys very quickly and regularly, which is good.

7.1 Mitigating a total failure

Suppose that despite our best efforts, every node in A group goes offline irrecoverably, and we have completely lost the fog service key for the current blocks. Given a block time of 5 seconds, we will miss one block for every 5 seconds that goes by before we are able to put a new key into place.

There are a few things that we can do to try to mitigate this:

- Have a C group which is waiting in the wings, with seeds already ready to go, and ready to rotate a new key into place in case A group fails. If A group failed and couldn't be recovered quickly, we can announce that the C seed is live. If we are relying on this mechanism, then the "expiry time" of keys should be short, possibly shorter than the key-rotation window. When the expiry time is much shorter than the key-rotation window, like, 100 blocks lets say, it functions like a "heartbeat" from the ingest server. Having short expiry times means that there may be periods of times where Alice cannot send Bob a transaction because Bob's fog server key expired and a new key was not rotated into place yet – but this may be better than Bob being unable to recover his transactions
- If A group fails and we miss a handful of blocks, we can write down the numbers of those blocks and add this note to the Fog Service Index. As a final backstop, we can report these blocks to users so that they can fallback to Cryptonote view key scanning for these blocks, against the primary Mobilecoin ledger.

There are two ways that this backstop could work

1. Users download this handful of blocks and do view-key scanning locally against them
2. We provide an SGX-server that takes their view keys and then does the view key scanning for them, for these blocks only, and gives them any transactions that are theirs

Option 2 is less bandwidth for the users, since each user only gets the transactions that they need. However, in this proposal we advocate option 1.

This is because in the option 2 scenario, it's possible that a malicious adversary could exploit a zero-day vulnerability against SGX as follows:

- Eve discovers a new side-channel attack against SGX
- Eve hacks into (roots) the machine that would do this view-key scanning on behalf of users, and installs malware that can conduct the side-channel attack
- Eve attempts to take down all the live Fog Service ingest nodes. These nodes aren't normally exposed to the public so it doesn't seem that they can be DDOS'ed, but Eve achieves this by hacking them and rooting them, by physical access at a datacenter, by intimidating cloud service providers, etc.
- The service operator's pager goes off and they discover that they have lost some blocks. They attempt to rotate in a new key, and add a note that they lost a few blocks, and activate the machinery to do view key scanning.
- The signal clients, the next time they talk to the fog server, discover that the server missed some blocks, and they are asked to provide their view key for the view key scanning, which they do.
- Eve's malware is now able to steal every Signal user's view keys.
- Unless someone detected the intrusion, the users don't know that they should move to new identities, so they continue to use the same identities forever, and Eve tracks all their activity on the network forever. This compromise persists even if SGX is later patched.

Basically, Option 2, an SGX server which handles all the users' view keys, creates a honeypot which can be exploited in the event of an SGX compromise. It's true that the ingest node itself is also a tempting target if SGX is compromised, but the difference is that that situation has forward secrecy – once the Ingest node is patched and redeployed, the old key isn't used anymore, and the old PRNG's aren't used anymore. So even without users actively rotating their keys, the adversary doesn't get useful information going into the future.

8 Life-cycle of a Signal Transaction

In this section we try to walk through a Mobilecoin transaction from Alice to Bob in Signal. We pay particular attention to timing analysis and traffic analysis. We will also consider how balance checks work for Bob.

8.1 Alice Submits

1. Alice gets Bob's public information, from a public server. This includes his Cryptonote A, B pubkeys, and his fog service FQDN "foo".
2. Alice checks if she has a current (not expired) key for "foo". If not, she gets an updated key from an SGX mediated central database – this doesn't reveal anything except that she is about to submit a transaction
3. Alice builds the transaction and submits it to consensus. The consensus node gives her a short-term tracking number.
4. The consensus node validates it and nominates it as part of the next block, per SCP.
5. When the block closes, the consensus node writes an encrypted note against this short-term tracking number. The note either explains why the TXO was rejected, or states that it is part of the block, and encloses a proof of this, which may be a Merkle proof.
6. Alice waits 5 seconds and asks for the note associated to her short-term tracking number has a note, recovering it if so, trying again later if not.
7. If the transaction was accepted, Alice sends the Merkle proof to Bob via Signal.

Some comments:

- How Alice finds Bob's credentials is out of scope for this proposal, but it could be an oblivious lookup of some kind
- Traffic analysis will always reveal that Alice is submitting a transaction. Talking to a Fog key server, or talking to the same consensus node again to get the result note, doesn't reveal anything new.
- The encrypted note from consensus can be encrypted with a symmetric key, which Alice generates and gives to the server at the time that she submits her transaction. This won't take up substantially more memory than the pending transaction.
- Alice sending the Merkle proof to Bob via Signal means that Bob does not always need to talk to the Fog server to find his transactions. If he did, then traffic analysis would reveal that he probably just received a transaction. It's possible that Signal may drop his message, and he'll have to recover the transaction in the Fog service. But hopefully, typically, this won't happen, which reduces the load on the fog service. We here leaning on one of Signal's privacy properties – the node operator doesn't know who Alice is sending messages to. Or at least, the degree of privacy for the transaction is similar to the degree of privacy in sending a Signal message.

8.1.1 Block closes

After the block closes, it is released from Consensus to the ingest server.

- Untrusted code passes the block to the ingest enclave.
- The ingest enclave reads the TXOs from this block, and decrypts the fog hints in parallel. The decrypted fog hints are the putative public view keys of users.
- For each one, it attempts to find a PRNG associated to the decrypted fog hint, obliviously. It then draws a number from this PRNG (to be the search key), and view-key encrypts the TXO (to be the payload). These will be output as a record for the transaction table in the Fog Server Index.
- If it does not find a PRNG for this user, it will nevertheless generate a random looking row for the Fog Server Index. (This will be implemented branchlessly, so the actual code will be something like, before performing the oblivious lookup, we choose a random fake PRNG state. Then we perform the oblivious lookup, and if it succeeds, we obliviously copy it over the fake PRNG state using a constant time conditional move. Then we evaluate the PRNG transition function and the encryption as before, so that the code paths are the same whether or not we actually found a user.)
- The PRNG's are obliviously returned to the ORAM data structure.
- The new rows for the Fog Server Index are output to untrusted.

When the new rows are added to the Fog Server Index, they are relayed to all the view nodes (e.g. using a Message Bus of some kind) that are serving the most recent partition of the Fog Server Index. Untrusted then passes these updates to the view node enclaves, which adds them to their in-memory ORAM's.

8.2 Bob recovers his transactions from the fog service

when Bob does full wallet recovery, he makes two rounds of queries to the Fog Service view nodes

1. Bob first must obtain his seed rows $B \to N$, which were emitted when his PRNG's were seeded. He also recovers any metadata regarding over what periods of time these seed rows were active. This lookup can be oblivious – the seed table is much smaller than the ledger etc.
2. Bob uses his private key b to deduce the seeds of his PRNGs. He then makes a second round of queries, sending the outputs of the PRNGs to the View node over an attested channel. Each time Bob makes a response, the view node either sends back an encrypted TXO, or a miss, but in case of a miss, responds with the same number of bytes.

3. Since Bob doesn't know in advance how many TXOs he actually got against any given PRNG, he can use an exponential strategy, i.e. request twice as many each time until you start missing or getting throttled.

4. These requests can all be made in parallel, so with an exponential strategy Bob finds n transactions in $\log n$ total time, with $O(n)$ bandwidth.

What does the node operator learn from this interaction?

- The node operator knows that Bob connected, if IP is deanonymizing
- The node operator learns that Bob received about $O(n)$ TXOs in total, judging by bandwidth

However, the node operator doesn't learn who sent him these TXOs, or what blocks these TXOs are from – because of Oblivious lookup, they don't know which parts of the fog service index he accessed, just that he accessed a certain number of items from it.

What if Bob uses the service repeatedly?

If Bob conducts full wallet recovery once and exchanges 20 transactions worth of bandwidth, then a month later, exchanges 30 transactions worth of bandwidth, there are two possibilities

1. he was offline and missed signal messages, and he received 30 transactions in the last month

2. he got a new phone and needed to install, but he only received 10 transactions in the last month

If the wallet recovery service is efficient in the sense that it only sends Bob the transactions he cares about rather than the whole blockchain, then it seems that leaking rough usage numbers over bandwidth is somewhat inevitable. One way that Bob can try to mitigate this is by periodically paying himself to add noise to these numbers.

8.2.1 Key Image Checks

In addition to getting his TXOs (his credits), Bob must all find his debits to find his current balance.

In Cryptonote, the "burned key image" is published as part of the public ledger after a transaction is validated. It is a nonce and can't be used to deduce anything about the sender or recipient, but only one possible key image is consistent with any given TXO. This key image is used by the network to prevent double-spends.

In Cryptonote, Bob can calculate, using his private keys, the key image corresponding to any one of his TXOs. If he is downloading the entire ledger, then he knows when a TXO is no longer spendable because its key image has already gone by.

In mobilecoin, Bob does not want to download the entire ledger, or even, the set of all key images, which grows indefinitely, and is roughly ledger-sized.

In order to perform balance checks, Bob takes his known TXO set and calculates the key image privately. Then, he sends these to remote server, which can tell him which of them have been spent.

If an adversary knows what TXOs Bob is checking, they know whenever he spends money – this key image will later appear in the clear in the ledger, after Bob submits the corresponding transaction, and Bob will stop asking about it during subsequent balance checks.

More importantly, when Bob starts asking about new key images, they know that he has new TXOs, and so they know that he got paid. This is somewhat different from when Bob contacts the fog service and asks for TXOs – in that case, the server sends him the same number of encrypted bytes in response whether he got paid or didn't.

This also means that if Bob is regularly checking key images in this way, an adversary learns the numbers of transactions that he received in each of these time periods.

8.2.2 Oblivious Key Image Lookup Service

In order to allow Bob to do key image checks privately, an oblivious service can serve the set of key images. Assuming that it's possible to serve the Fog Service Index this way, the same can be done with key images.

Key images are only 32 bytes, so this dataset is expected to be significantly smaller than the Fog Service Index, which expects several hundred bytes per TXO.

This service would work equally for any user of an fog service, so it need not be duplicated even if there are several fog services.

By making key image lookups oblivious, an adversary only learns about the number of key images Bob asks about, which is what they learn from watching bandwidth. Bob can easily pad his requests up with dummy requests to make sure if he asks about e.g. fewer than eight key images, it looks similar. Similarly, as he gets paid, he can pay his TXOs to himself in order to pool them into one transaction. If he usually has only one or two unspent TXOs, then all of his key image requests have less than eight, so they look the same after this padding.

9 Conclusion

We think that this proposal achieves the design goals as stated:

- Path ORAM, in principle, scales as roughly $O(\log^2 n)$ worst-case to access a dataset of size n
- In practice, we expect it to provide something like four orders of magnitude improvement over linear scanning, in terms the number of machines just for the ingest nodes.

- For the view nodes, we expect it to allow a machine to service a request in on the order of one or ten milliseconds, worst-case, unlike linear scanning. This will make it much harder to DDOS the network, and allow us to plausibly scale up to a billion users with tens or hundreds of thousands of machines.
- We think this design of the fog server is as private as it can plausibly be when SGX is working. The node operator learns only that a particular IP connected and downloaded a certain number of encrypted bytes. We think that a user can use this to provide highly private balance checks and wallet recovery, with the caveat that, it's better for them to avoid a design where they always check their balance in response to being paid, because this leaks the fact that they got paid in traffic analysis. It seems better for Bob usually to rely on Alice to send him a proof that he got paid, and only when this message is dropped, or periodically, does Bob check with the Fog Service.
- When SGX is compromised, we think this design effectively provides forward secrecy, and allows a straightforward upgrade for each of the various enclaves. Bugs discovered in ingest or view nodes do not require a hard fork of consensus.
- The service is resilient against machines failing – the enclaves are generally, by design, discardable, and the fog service index, which is simple data and not sealed to an enclave, is the only thing that cannot be lost.

The main drawbacks of the proposal are that it still requires "a lot" of machines: tens of thousands are projected, to support billions of users, and the main bottleneck is the oblivious lookup. This proposal is several orders of magnitude better than linear scanning, and has a pretty complete story of privacy and upgrade path. A strength of the proposal is that if a system like this is deployed, there is a clear path to incrementally improve the performance of oblivious lookup, and hence the view nodes, and take old view nodes down, without forking consensus or invalidating history.

Note: In practice, MobileCoin Fog supports millions of queries on a daily basis in 2022 from world-wide users of Signal's payments beta feature, using only two Fog view nodes.

References

[1] Fritz Alder, Arseny Kurnikov, Andrew Paverd, and N. Asokan. Migrating SGX enclaves with persistent state. *CoRR*, abs/1803.11021, 2018.

[2] Dan Boneh, David Mazières, and Raluca A. Popa. Remote oblivious storage: Making oblivious ram practical. 2011.

[3] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Secure processors part I: background, taxonomy for secure enclaves and intel SGX architecture. *Foundations and Trends in Electronic Design Automation*, 11(1-2):1–248, 2017.

[4] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Secure processors part II: intel SGX security analysis and MIT sanctum architecture. *Foundations and Trends in Electronic Design Automation*, 11(3):249–361, 2017.

[5] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[6] Jacob R. Lorch, Bryan Parno, James W. Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: ensuring private access to large-scale data in the data center. In Keith A. Smith and Yuanyuan Zhou, editors, *Proceedings of the 11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013*, pages 199–214. USENIX, 2013.

[7] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. Zerotrace : Oblivious memory primitives from intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[8] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious distributed cloud data store. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[9] Emil Stefanov, Marten van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. *J. ACM*, 65(4):18:1–18:26, 2018.

[10] Nicolas van Saberhagen. Cryptonote v2.0. Whitepaper, 2013. `https://cryptonote.org/whitepaper.pdf`.