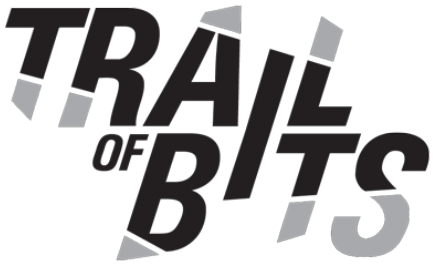

MobileCoin Fog

Security Assessment

January 29, 2021

Prepared For:
Sara Drakeley | *MobileCoin*
sara@mobilecoin.com

Prepared By:
Jim Miller | *Trail of Bits*
james.miller@trailofbits.com

Dominik Teiml | *Trail of Bits*
dominik.teiml@trailofbits.com

Sam Moelius | *Trail of Bits*
sam.moelius@trailofbits.com

Executive Summary

From January 19 to January 29, 2021, MobileCoin engaged Trail of Bits to review the security of its Fog protocol. Trail of Bits conducted this assessment over four person-weeks, with three engineers working from commit `1f518b5` of the `internal` repository and commit `3e2769f` of the `sgx-oblivious` repository.

We spent the beginning of the first week familiarizing ourselves with the MobileCoin Fog protocol. From there, we began manually reviewing the `sgx-oblivious` repository, focusing on the `sgx-oblivious-ram` crate and the `sgx-oblivious-map` crate. In addition to this review, we began fuzzing various functions in the ingest and view enclaves.

In the second week of the assessment, we completed our review of the `sgx-oblivious` repository, focusing on the `aligned-cmov` crate. We also performed an automated analysis to verify the constant-time properties of various functions in the `sgx-oblivious` repository, which we detail in Appendix D. In addition to this, we reviewed other parts of the system, including the key exchange in the ingest enclave, and we expanded our fuzzers.

Our review resulted in a total of eight findings. We identified one medium-severity issue related to non-constant-time behavior observed during our automated analysis. Two of these findings are classified as low severity, as they relate to an integer overflow and the protection of credentials. The remaining five findings are classified as informational severity. These findings do not represent exploitable vulnerabilities, but rather they detail defense-in-depth recommendations to further strengthen the system.

Overall, we found the code to be structured and documented clearly, despite the Fog protocol's complexity. Our initial automated analysis provided positive results for the `aligned-cmov-crate`. However, our manual review uncovered two informational-severity findings related to constant-time functions, and the automated analysis revealed a potential violation of these constant-time properties in practice.

Moving forward, we encourage MobileCoin to expand on our automated analysis detailed in Appendix D. Due to time constraints, we were able to cover only some of the oblivious functions in our analysis. In particular, various functions in `sgx-oblivious-map` would benefit from enhanced analysis. In addition, we recommend that an additional review be performed on the remaining areas of the system that we were unable to cover, such as certificate validation and iOS/Android SDKs.

Project Dashboard

Application Summary

Name	MobileCoin Fog
Version	`1f518b5` (`internal`), `3e2769f` (`sgx-oblivious`)
Type	Rust
Platform	Intel SGX

Engagement Summary

Dates	January 19–January 29, 2021
Method	Full knowledge
Consultants Engaged	3
Level of Effort	4 person-weeks

Vulnerability Summary

Total Medium-Severity Issues	1	■
Total Low-Severity Issues	2	■ ■
Total Informational-Severity Issues	5	■ ■ ■ ■ ■
Total	8	

Category Breakdown

Auditing and Logging	1	■
Cryptography	5	■ ■ ■ ■ ■
Data Exposure	1	■
Data Validation	1	■
Total	8	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Not applicable.**
Arithmetic	**Satisfactory.** Our fuzzer uncovered one arithmetic finding related to an overflow caused by a multiplication (TOB-MCF-003).
Assembly Use	**Strong.** The `aligned-cmov` crate is designed to integrate assembly to perform a conditional move in constant time. Both our manual and automated review of this crate did not result in any findings related to the violation of this constant-time property.
Centralization	**Strong.** Fog servers are meant to be run by third parties. We noticed nothing in the implementation that would prevent this.
Upgradeability	**Not considered.**
Function Composition	**Satisfactory.** The Fog protocol is a complex protocol involving complex data structures, such as oblivious RAM and an oblivious hashmap. These complex data structures introduce an inherent complexity to the system; however, the code is well documented and well structured, which somewhat remediates the complexity.
Front-Running	**Not considered.**
Key Management	**Satisfactory.** Our coverage related to key exchange and key management resulted in two findings (TOB-MCF-005, TOB-MCF-007).
Monitoring	**Not considered.**
Specification	**Satisfactory.** MobileCoin provided multiple documents that describe the Fog protocol and its various components, including oblivious RAM. Although informative, some of these documents were incomplete.
Testing and Verification	**Moderate.** The ingest, view, and ledger enclaves would benefit from tests that directly exercise their APIs, similar to the tests that now exist for the consensus enclave. Also, the ingest and view servers would benefit from tests that run them as distinct users (TOB-MCF-004).

Engagement Goals

The engagement was scoped to provide a security assessment of MobileCoin's `sgx-oblivious` and `internal` repositories.

Specifically, we sought to answer the following questions:

- Do access patterns to the oblivious map or oblivious RAM objects reveal any information?
- Does the implementation of the oblivious RAM conform with the Path ORAM protocol?
- Does the `aligned-cmov` crate safely use assembly?
- Do the oblivious functions in `sgx-oblivious-ram`, `sgx-oblivious-map`, and `aligned-cmov` operate in constant time?
- Are there any flaws or weaknesses in the key exchange design or implementation?
- Can we use fuzzers to identify any crashes or undesired behavior in the codebase?

Coverage

`sgx-oblivious`. We manually reviewed this repository to identify any discrepancies between the oblivious RAM implementation and Path ORAM. We also manually checked whether any functions that are intended to be oblivious contain any logic that could violate their constant-time properties. In addition to this, we supplemented this manual review with an automated analysis of the constant-time properties of several functions, which we detail in Appendix D.

`internal`. We manually reviewed various components of this repository, including code related to key management and code that uses the oblivious hashmap from `sgx-oblivious`. This was a best-effort review to analyze as much of this repository as possible in the allotted time. In addition to this manual review, we fuzzed the ingest enclave's `ingest_txs` function, the view enclave's `add_records` and `query` functions, and the ledger enclave's `get_outputs`, `get_outputs_data`, `check_key_images`, and `check_key_images_data` functions.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

❑ **Clearly document that these `debug_assert` statements leak sensitive information and should never run in production.** TOB-MCF-001

❑ **Do not add one to the second result to preserve its uniform distribution.** Alternatively, instead of adding one, recompute a new hash value until a result different from the first result is achieved. TOB-MCF-002

❑ **Use `checked_mul` to ensure that the multiplication in `compute_mem_kb` does not overflow.** Gracefully handle the error when an overflow occurs. Doing so will help ensure that users observe predictable behavior from MobileCoin nodes. TOB-MCF-003

❑ **Two recommendations:**

- **Advise MobileCoin node operators to run the ingest and view servers using distinct operating system and database users.** Doing so will help to distinguish the actions performed by either server.
- **Implement tests to check for bugs that could arise from running the servers under distinct users.** This will help to ensure that such bugs do not occur in production. TOB-MCF-004

❑ **Use `zeroize` to protect all data structures containing sensitive data to reduce the risk of unintended exposure.** TOB-MCF-005

❑ **Consider implementing a greater-than comparison that avoids the possibility of being optimized into a branch, similar to how the `subtle` crate performs comparisons.** Alternatively, consider adjusting the comments surrounding these functions to reflect that they may not be constant time. TOB-MCF-006

❑ **Adjust the `prf` function to perform proper domain separation between these three values.** Consider using an encoding scheme such as RLP or ASN1 or the `merlin` crate, which provides natural domain separation. This will prevent collisions from occurring between different inputs, even as their lengths vary. TOB-MCF-007

❑ **Compile the `balanced-tree-index` crate in debug mode with debug assertions disabled (see TOB-MCF-001).** This non-constant-time bug was observed in release mode. To our knowledge, the bug does not occur when the code is compiled in debug mode. TOB-MCF-008

Long term

❑ **Rewrite these statements using constant-time operations.** TOB-MCF-001

❑ **Consider incorporating fuzzing into your continuous integration process.** Doing so could help to reveal similar bugs in the future. TOB-MCF-003

❑ **Consider ways to reduce the privileges of the ingest or view users.** This will help limit the damage that could be caused if either of those users were compromised. TOB-MCF-004

❑ **Consider incorporating one of the Rust bindings for `libsodium` into your codebase.** In addition to zeroing memory that is no longer in use, `libsodium` surrounds protected memory with guard pages and prevents it from being paged to the swap area. TOB-MCF-005

❑ **Consider performing a deeper analysis to determine whether using the > operator will be problematic in practice.** TOB-MCF-006

❑ **As future versions are implemented, use sufficient domain separation for all inputs.** TOB-MCF-007

❑ **Consider incorporating the instruction trace analysis described in Appendix D into your continuous integration process.** This non-constant-time bug was found using that analysis. Continuing to apply it could reveal similar bugs in the future. TOB-MCF-008

Findings Summary

#	Title	Type	Severity
1	Various debug_assert statements are not constant time	Cryptography	Informational
2	Handling of corner case in hash_query introduces bias	Cryptography	Informational
3	Multiplication overflow in compute_mem_kb	Data Validation	Low
4	Ingest and view servers should be run under distinct users	Auditing and Logging	Informational
5	zeroize is not used to protect HTTP basic authentication credentials	Data Exposure	Low
6	Call to vartime_write may not be oblivious for keys in the map	Cryptography	Informational
7	Insufficient domain separation in key exchange prf function	Cryptography	Informational
8	The common_ancestor_distance_of_peers function is not constant time when compiled in release mode	Cryptography	Medium

1. Various debug_assert statements are not constant time

Severity: Informational | Difficulty: High
Type: Cryptography | Finding ID: TOB-MCF-001
Target: `sgx-oblivious/sgx-oblivious-ram`, `sgx-oblivious/sgx-oblivious-map`

Description
To achieve the desired properties of Fog, MobileCoin uses oblivious RAM inside of Intel SGX. The protocol is intended to prevent memory accesses from being leaked to adversaries. To this end, MobileCoin implements the Path ORAM protocol in combination with other oblivious data structures and operations, such as constant-time comparisons and move operations.

In both the `sgx-oblivious-ram` and `sgx-oblivious-map` directories, several `debug_assert` statements are used to detect incorrect code behavior, presumably to aid in debugging. Inside some of these statements, non-constant-time operations are performed on sensitive objects. These `debug_assert` statements are enabled only by particular builds, and optimized builds will not execute these statements by default. However, if a MobileCoin user were to accidentally run a debug build in production, this would violate the privacy properties of the protocol.

Note that MobileCoin can disable these `debug_assert` statements and publish an enclave measurement. Then all devices running code with these `debug_assert` statements enabled should be detectable, as they would produce a different measurement.

Exploit Scenario
A MobileCoin user accidentally enables the `debug_assert` statements on code running in production. An attacker, Eve, is able to use timing information to violate the privacy properties that the ORAM protocol is intended to provide.

Recommendations
Short term, clearly document that these `debug_assert` statements leak sensitive information and should never run in production.

Long term, rewrite these statements using constant-time operations.

2. Handling of corner case in hash_query introduces bias

Severity: Informational
Type: Cryptography
Target: `sgx-oblivious/sgx-oblivious-map/src/lib.rs`

Difficulty: N/A
Finding ID: TOB-MCF-002

Description
The MobileCoin Fog protocol includes various oblivious data structures. One of these data structures is an oblivious hashmap. In particular, the `sgx-oblivious-map` crate implements a “power of two choices” hashmap on top of an oblivious RAM implementation. Their hashmap uses the `SipHash` hash function.

The “power of two choices” hashmap uses two hash functions to more efficiently handle collisions. Even with two different hash functions, an edge case arises when two different functions produce the same output. The MobileCoin implementation handles this edge case by adding one to the second result to make them different (see figure 2.1)

```
        // The rest of our code as written is believed to be correct even if
        // result1 sometimes equals result2. However, this follows a bunch of
        // corner-case paths in this data structure, and in the path oram data
        // structure during the access2 call.
        //
        // As a "defense in depth" against bugs, it is simpler to just
        // eliminate this corner case, and it is essentially free to do so,
        // if we test for the corner case and fix it, by on-the-spot redefining
        // the hash function whenever this happens.
        //
        // Note: Nothing interacts with hash1 or hash2 other than via this
        // function, so this is the canonical definition of these hashes.
        // TODO: Put this code in a separate module to ensure that this continues
        // to be the case, and that hash_query is the only way to get the hashes.
        let result2_plus_one = result2.wrapping_add(1) & (self.num_buckets - 1);
        result2.cmov(result1.ct_eq(&result2), &result2_plus_one);
```

Figure 2.1: *sgx-oblivious/sgx-oblivious-map/src/lib.rs#L127-142*

However, by adding one to the result, a bias is introduced to the output of this function. In particular, the distribution of the second result is no longer uniform, but it is correlated with the first result.

Recommendations
Short term, do not add one to the second result to preserve its uniform distribution. Alternatively, instead of adding one, recompute a new hash value until a result different from the first result is achieved.

3. Multiplication overflow in compute_mem_kb

Severity: Low
Difficulty: High
Type: Data Validation
Finding ID: TOB-MCF-003
Target: `src/fog/ocall_oram_storage/untrusted/src/lib.rs`

Description

The `compute_mem_kb` function is used by `UntrustedAllocation::new` to determine the size of a memory allocation. A multiplication in `compute_mem_kb` can overflow. In a debug build, this causes a panic. In a release build, this could cause the allocation to be too small.

The vulnerable code appears in figure 3.1. If either `count` or the sum of `data_item_size` and `meta_item_size` is too large, the multiplication will overflow. A partial stack trace leading to the vulnerable code appears in figure 3.2.

```
/// Helper which computes the total memory in kb allocated for count, data_item_size,
meta_item_size
fn compute_mem_kb(count: usize, data_item_size: usize, meta_item_size: usize) -> u64 {
    let num_bytes = (count * (data_item_size + meta_item_size)) as u64;
    // Divide by 1024 and round up, to compute num_bytes in kb
    (num_bytes + 1023) / 1024
}

impl UntrustedAllocation {
    /// Create a new untrusted allocation for given count and item sizes, on the heap
    ///
    /// Data and meta item sizes must be divisible by 8, consistent with the contract
    /// described in the edl file
    pub fn new(count: usize, data_item_size: usize, meta_item_size: usize) -> Self {
        let mem_kb = compute_mem_kb(count, data_item_size, meta_item_size);
```

Figure 3.1: src/fog/ocall_oram_storage/untrusted/src/lib.rs#L69-L82

```
• fog_ocall_oram_storage_untrusted::compute_mem_kb
            at src/fog/ocall_oram_storage/untrusted/src/lib.rs:71
• fog_ocall_oram_storage_untrusted::UntrustedAllocation::new
            at src/fog/ocall_oram_storage/untrusted/src/lib.rs:82
• allocate_oram_storage
            at src/fog/ocall_oram_storage/untrusted/src/lib.rs:160
• fog_ocall_oram_storage_trusted::OcallORAMStorage<DataSize,MetaSize>::new
            at src/fog/ocall_oram_storage/trusted/src/lib.rs:162
• <fog_ocall_oram_storage_trusted::OcallORAMStorageCreator as
  sgx_oblivious_traits::creators::ORAMStorageCreator<DataSize,MetaSize>>::create
            at src/fog/ocall_oram_storage/trusted/src/lib.rs:472
• sgx_oblivious_ram::path_oram::PathORAM<ValueSize,Z,StorageType,RngType>::new
            at sgx-oblivious-ram/src/path_oram/mod.rs:139
• <sgx_oblivious_ram::PathORAM4096Z4Creator<R,SC> as
  sgx_oblivious_traits::creators::ORAMCreator<typenum::uint::UInt<typenum::uint::UInt<typ
  enum::uint::UInt<typenum::uint::UInt<typenum::uint::UInt<typenum::uint::UInt<typenum::u
  int::UInt<typenum::uint::UInt<typenum::uint::UInt<typenum::uint::UInt<typenum::uint::UI
  nt<typenum::uint::UTerm,typenum::bit::B1>,typenum::bit::B0>,typenum::bit::B0>,typenum::
  bit::B0>,typenum::bit::B0>,typenum::bit::B0>,typenum::bit::B0>,typenum::bit::B0>,typenu
  m::bit::B0>,typenum::bit::B0>,typenum::bit::B0>,R>>::create
            at sgx-oblivious-ram/src/lib.rs:90
```

```
- sgx_oblivious_map::TwoChoiceHashTable<KeySize,ValueSize,BlockSize,RngType,O>::new
          at sgx-oblivious-map/src/lib.rs:95
- <sgx_oblivious_map::TwoChoiceHashTableCreator<BlockSize,RngType,OC> as
  sgx_oblivious_traits::creators::OMapCreator<KeySize,ValueSize,RngType>>::create
          at sgx-oblivious-map/src/lib.rs:566
- fog_view_enclave_impl::e_tx_out_store::ETxOutStore<OSC>::new
          at src/fog/view/enclave/impl/src/e_tx_out_store.rs:75
```

Figure 3.2: Partial stack trace leading to the multiplication overflow

Exploit Scenario
Alice runs a MobileCoin node. She unintentionally configures it in a way that causes the multiplication to overflow. Alice's node is subject to memory exhaustion, which can be difficult to diagnose and may cause unintended behavior. Alice wastes time and effort trying to understand why.

Recommendations
Short term, use `checked_mul` to ensure that the multiplication in `compute_mem_kb` does not overflow. Gracefully handle the error when an overflow occurs. Doing so will help ensure that users observe predictable behavior from MobileCoin nodes.

Long term, consider incorporating fuzzing into your continuous integration process. Doing so could help to reveal similar bugs in the future.

4. Ingest and view servers should be run under distinct users

Severity: Informational
Type: Auditing and Logging
Target: `src/fog/ingest/server`, `src/fog/view/server`

Difficulty: High
Finding ID: TOB-MCF-004

Description
The ingest and view servers should be run using distinct operating system and database users. Doing so will help distinguish the servers' actions, which could be useful (e.g., if either server were compromised).

Currently, the documentation does not address this topic directly. The documentation suggests using the developer's account for testing purposes (e.g., figures 4.1–4.3). Clearly, using the developer's account is acceptable for most tests. However, some tests should check for bugs that could arise from running the servers under distinct users.

> 8. Fog services that require connecting to the database need the `DATABASE_URL` environment variable set:
>
> ```
> export DATABASE_URL=postgres://$USER@localhost/fog_test
> ```

Figure 4.1: Documentation for connecting the ingest and view servers to the Postgres database

```
    // Open databases.
    let recovery_db = SqlRecoveryDb::new_from_url(
        &std::env::var("DATABASE_URL").expect("DATABASE_URL environment variable
missing"),
    )
    .expect("Failed connecting to database");
```

Figure 4.2: src/fog/ingest/server/src/bin/main.rs#L74-L78

```
    let recovery_db = SqlRecoveryDb::new_from_url(
        &std::env::var("DATABASE_URL").expect("DATABASE_URL environment variable
missing"),
    )
    .expect("Failed connecting to database");
```

Figure 4.3: src/fog/view/server/src/bin/main.rs#L22-L25

Exploit Scenario
Alice runs ingest and view servers under a single user. Eve gains remote code execution in the ingest server. Because the servers run under the same user, Eve's actions cannot be distinguished from legitimate ones. Moreover, the subsequent cleanup requires greater effort.

Recommendations
Short term, take the following actions:

- Advise MobileCoin node operators to run the ingest and view servers using distinct operating system and database users. Doing so will help to distinguish the actions performed by either server.
- Implement tests to check for bugs that could arise from running the servers under distinct users. This will help to ensure that such bugs do not occur in production.

Long term, consider ways to reduce the privileges of the ingest or view users. This will help limit the damage that could be caused if either of those users were compromised.

References

- NIST: Principle of Least Privilege
- Dear PostgreSQL: Where are my logs?

5. zeroize is not used to protect HTTP basic authentication credentials

Severity: Low | Difficulty: High
Type: Data Exposure | Finding ID: TOB-MCF-005
Target: `public/util/uri/src/uri.rs`, `public/util/grpc/src/auth/mod.rs`

Description
Sensitive data can remain in memory even after the memory has been freed. Failing to scrub such data from memory can result in its unintended exposure.

While Rust's ownership model guarantees that bindings go out of scope at the end of execution, allocations made may still persist in memory. Therefore, code should explicitly clear memory allocations if they contain sensitive data. Note that such a process must also ensure that the clearing code does not get optimized away, as compilers tend to remove such instructions due to dead store elimination optimizations.

The `zeroize` crate provides a solution to these problems by zeroing memory when it is dropped. Currently, `zeroize` is used to protect certain data structures, such as `AccountKey` (figure 5.1). It should also be used to protect HTTP basic authentication credentials and the URIs from which they are derived (figures 5.2 and 5.3).

```
/// Complete AccountKey, containing the pair of secret keys, which can be used
/// for spending, and optionally some fog-related info,
/// can be used for spending. This should only ever be present in client code.
#[derive(Clone, Message, Zeroize)]
#[zeroize(drop)]
pub struct AccountKey {
    /// Private key 'a' used for view-key matching.
    #[prost(message, required, tag = "1")]
    view_private_key: RistrettoPrivate,
```

Figure 5.1: public/account-keys/src/account_keys.rs#L174-L182

```
/// Standard username/password credentials.
pub struct BasicCredentials {
    username: String,
    password: String,
}
```

Figure 5.2: public/util/grpc/src/auth/mod.rs#L80-L84

```
#[derive(Clone, Debug, Eq, PartialEq, Ord, PartialOrd, Hash)]
pub struct Uri<Scheme: UriScheme> {
    /// The original Url object used to construct this object.
    url: Url,

    /// Hostname.
    host: String,

    /// Consensus port.
    port: u16,
```

```
    /// Whether to use TLS when connecting.
    use_tls: bool,

    /// Optional username.
    username: String,

    /// Optional password.
    password: String,

    /// The uri scheme
    _scheme: PhantomData<fn() -> Scheme>,
}
```

Figure 5.3: public/util/uri/src/uri.rs#L25-L47

Exploit Scenario
Eve utilizes an unrelated memory access exploit that allows her to read an ingest client's memory. Eve uses the bug to steal Alice's credentials and connect to the ingest server as though she were Alice.

Recommendations
Short term, use `zeroize` to protect all data structures containing sensitive data to reduce the risk of unintended exposure.

Long term, consider incorporating one of the Rust bindings for `libsodium` into your codebase. In addition to zeroing memory that is no longer in use, `libsodium` surrounds protected memory with guard pages and prevents it from being paged to the swap area.

6. Call to vartime_write may not be oblivious for keys in the map

Severity: Informational
Type: Cryptography
Target: `sgx-oblivious/sgx-oblivious-map/src/lib.rs`

Difficulty: High
Finding ID: TOB-MCF-006

Description
The MobileCoin Fog protocol includes various oblivious data structures. One of these data structures is an oblivious hashmap. In particular, the `sgx-oblivious-map` crate implements a “power of two choices'' hashmap on top of an oblivious RAM implementation.

The implementation includes the `vartime_write` function, which writes to the oblivious map at a particular position. As seen in the comments surrounding this function, this function is not strongly constant time. However, the comments do enumerate various security properties about the function. One of these properties claims that when both keys are in the map, the call to this function will occur in constant time (relative to the input key).

The `vartime_write` function takes as input a key and value and subsequently calls the `vartime_write_extended` function with these inputs. However, the `vartime_write_extended` function uses a generic comparison operator (rather than using a specialized comparison, like `ct_eq`) that could violate this constant-time property. As shown in figure 6.1, the `block1_empty_count` and `block2_empty_count`, which are derived from the input key (called `query` here) are compared using the > operator. Unlike the `ct_eq` used from the `subtle` crate, this > operator could be optimized into a branch that would violate this constant-time property.

```
self.oram.access2(hashes[0], hashes[1], |block1, block2| {
            // Note: These calls don't need to be constant-time to meet the
requirement,
            // but we already had the code that way, and it may serve as
"defense-in-depth".
            // If they show up in profiling, then we can make variable time versions
            let (block1_found, block1_empty_count) =
Self::count_before_insert(query, block1);
            let (block2_found, block2_empty_count) =
Self::count_before_insert(query, block2);
            debug_assert!(
                !bool::from(block1_found & block2_found) || hashes[0] == hashes[1],
                "key should not be found twice, unless hashes[0] == hashes[1]!"
            );

            let found = block1_found | block2_found;
            result_code.cmov(found, &OMAP_FOUND);

            // Scope for "condition" variable
            {
                // condition is false when side-effects are disallowed, OR
                // if we found the item and we aren't allowed to overwrite
                let condition = allow_sideeffects_and_eviction & (allow_overwrite |
!found);
                // write_to_block1 is true when we should prefer to write to block1
over block2
                // watch the case that hashes[0] == hashes[1] !
```

```
                // in that case we prefer to modify block2
                // because that is what will become the final value of the block
                // So, if block2_found, we should be false, even if block1_found.
                // And if not found in either place and block1_empty_count ==
block2_empty_count,
                // prefer block2.
                let write_to_block1 = !block2_found
                    & (block1_found
                        | Choice::from((block1_empty_count > block2_empty_count) as
u8));

                Self::insert_to_block(condition & write_to_block1, query,
new_value, block1);
                Self::insert_to_block(condition & !write_to_block1, query,
new_value, block2);
            }
```

Figure 6.1: sgx-oblivious/sgx-oblivious-map/src/lib.rs#L354-386

Note that this comparison with the > operator could still violate this constant-time property regardless of whether the two keys are in the map, which contradicts the comments surrounding the `vartime_write_extended` function. In addition, this could affect the `access_and_insert` function, which makes a similar security property claim and subsequently calls this `vartime_write` function.

Exploit Scenario
A developer uses an oblivious map under the assumption that both the `vartime_write` and `access_and_insert` functions have the property of being constant time when both keys are in the map. The security of their overall protocol relies on these security properties. An attacker, Eve, notices that these properties do not hold in practice and is able to violate the security of the protocol.

Recommendations
Short term, consider implementing a greater-than comparison that avoids the possibility of being optimized into a branch, similar to how the `subtle` crate performs comparisons. Alternatively, consider adjusting the comments surrounding these functions to reflect that they may not be constant time.

Long term, consider performing a deeper analysis to determine whether using the > operator will be problematic in practice.

7. Insufficient domain separation in key exchange prf function

Severity: Informational | Difficulty: N/A
Type: Cryptography | Finding ID: TOB-MCF-007
Target: `internal/src/fog/kex_rng/src/versioned/kexrng20201124.rs`

Description
The MobileCoin Fog protocol includes a protocol for deterministically generating secret keys shared between a user and the ingest server. Specifically, a Diffie-Hellman style key exchange is performed between the client and server to derive a shared secret. This shared secret is then used as input, along with a version number and a counter value, into a hash function to deterministically generate multiple secrets shared between both parties.

In the current implementation of this key exchange protocol, the `prf` function uses the `Blake2b` hash function to derive these secrets. As shown in figure 7.1, the secrets are computed by passing in byte strings of the version (currently fixed at "`20201124`"), `secret`, and `counter`.

```
fn prf(secret: &GenericArray<u8, U32>, counter: &u64) -> Output {
        let mut hasher = Blake2b::new();
        hasher.update(b"20201124");
        hasher.update(secret.as_slice());
        hasher.update(counter.to_le_bytes());
        let result = hasher.finalize();
        let (output, _): (Output, _) = result.split();
        output
    }
```

Figure 7.1: `internal/src/fog/kex_rng/src/versioned/kexrng20201124.rs#L22-30`

As shown above, there is no domain separation between these byte strings that are input into `Blake2b`. This can become problematic if the individual byte strings can vary in length, as this could cause a collision to occur even when the inputs actually differ. In future versions of this `prf`, it is possible that the byte string for the version could vary in length. In addition to this, if the type of the `secret` or the `counter` changed, this would result in a differently sized byte string.

Recommendations
Short term, adjust the `prf` function to perform proper domain separation between these three values. Consider using an encoding scheme such as RLP or ASN1 or the `merlin` crate, which provides natural domain separation. This will prevent collisions from occurring between different inputs, even as their lengths vary.

Long term, as future versions are implemented, use sufficient domain separation for all inputs.

8. The common_ancestor_distance_of_peers function is not constant time when compiled in release mode

Severity: Medium
Type: Cryptography
Target: `sgx-oblivious/balanced-tree-index/src/lib.rs`

Difficulty: High
Finding ID: TOB-MCF-008

Description
When compiled in release mode, the `common_ancestor_distance_of_peers` function is not constant time. This fact could be abused to leak sensitive information from a Path ORAM implementation that uses the function.

The `common_ancestor_distance_of_peers` function appears in figure 8.1. Code produced by compiling it in release mode appears in figure 8.2. Note the presence of the "`je`" (jump if equal) instruction, which appears to short-circuit the case in which `self` and `other` are equal.

```
            fn common_ancestor_distance_of_peers(&self, other: &Self) -> u32 {
                debug_assert!(self.height() == other.height());
                const DIGITS: u32 = <$uint>::MAX.leading_ones();
                // Wrapping sub is used to avoid panics
                // Note: We assume that leading_zeroes is compiling down to ctlz
                // and is constant time.
                DIGITS.wrapping_sub((self ^ other).leading_zeros())
            }
```

Figure 8.1: balanced-tree-index/src/lib.rs#L133-L140

```
000000000009fa90
<_ZN54_$LT$u64$u20$as$u20$balanced_tree_index..TreeIndex$GT$33common_ancestor_dista
nce_of_peers17ha3a59b7c5876fc16E>:
   9fa90:       48 8b 06                mov    (%rsi),%rax
   9fa93:       48 33 07                xor    (%rdi),%rax
   9fa96:       74 10                   je     9faa8
<_ZN54_$LT$u64$u20$as$u20$balanced_tree_index..TreeIndex$GT$33common_ancestor_dista
nce_of_peers17ha3a59b7c5876fc16E+0x18>
   9fa98:       48 0f bd c8             bsr    %rax,%rcx
   9fa9c:       48 83 f1 3f             xor    $0x3f,%rcx
   9faa0:       b8 40 00 00 00          mov    $0x40,%eax
   9faa5:       29 c8                   sub    %ecx,%eax
   9faa7:       c3                      retq
   9faa8:       b9 40 00 00 00          mov    $0x40,%ecx
   9faad:       b8 40 00 00 00          mov    $0x40,%eax
   9fab2:       29 c8                   sub    %ecx,%eax
   9fab4:       c3                      retq
   9fab5:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
   9fabc:       00 00 00
   9fabf:       90                      nop
```

Figure 8.2: Code produced from figure 8.1 by compiling in release mode

A call chain leading to `common_ancestor_distance_of_peers` is as follows:

```
ct_insert → lowest_legal_index → lowest_legal_index_impl →
      common_ancestor_height → common_ancestor_distance_of_peers
```

Note that `ct_insert` is expected to be constant time. If an attacker could cause `common_ancestor_distance_of_peers` to be executed repeatedly, he could learn something about `ct_insert`'s arguments.

Exploit Scenario
Eve observes access patterns made on the contents of the oblivious RAM. Since the `common_ancestor_distance_of_peers` function is not constant time, Eve is able to learn sensitive information from these access patterns.

Recommendations
Short term, compile the `balanced-tree-index` crate in debug mode with debug assertions disabled (see TOB-MCF-001). This bug was observed in release mode. To our knowledge, the bug does not occur when the code is compiled in debug mode.

Long term, consider incorporating the instruction trace analysis described in Appendix D into your continuous integration process. This bug was found using that analysis. Continuing to apply it could reveal similar bugs in the future.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing a system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or the order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices or Defense in Depth.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is relatively small or is not a risk the customer has indicated is important.
Medium	Individual users' information is at risk; exploitation could pose reputational, legal, or moderate financial risks to the client.

High	The issue could affect numerous users and have serious reputational, legal, or financial implications for the client.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is commonly exploited; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of a complex system.
High	An attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components
Arithmetic	Related to the proper use of mathematical operations and semantics
Assembly Use	Related to the use of inline assembly
Centralization	Related to the existence of a single point of failure
Upgradeability	Related to contract upgradeability
Function Composition	Related to separation of the logic into functions with clear purposes
Front-Running	Related to resilience against front-running
Key Management	Related to the existence of proper procedures for key generation, distribution, and access
Monitoring	Related to the use of events and monitoring procedures
Specification	Related to the expected codebase documentation
Testing and Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.)

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed, and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Non-Security-Related Findings

This appendix contains findings that do not have immediate or obvious security implications.

- **The ingest enclave's `attempt_ingest_txs` function does not use its `self` argument.** Consider removing the argument and making `attempt_ingest_txs` an associated function (i.e., `SgxIngestEnclave::attempt_ingest_txs` instead of `self.attempt_ingest_txs`).

- **Panic messages in `src/fog/ocall_oram_storage/untrusted/src/lib.rs` could be improved.** In two locations, there is a `panic!` invocation of the following general form:

```
panic!(
    "Could not allocate memory for data segment: {}",
    count * data_item_size
)
```

  The message would reveal more information if it were rewritten as follows:

```
panic!(
    "Could not allocate memory for data segment: {} * {}",
    count, data_item_size
)
```

D. Instruction Trace Analysis

To help check the constant-time behavior of certain functions within MobileCoin Fog, Trail of Bits used a modified version of QEMU to obtain instruction traces for those functions. This appendix details our methodology, presents our results, and discusses ways in which the analysis could be improved.

Methodology

We used emulation and instrumentation to obtain instruction traces for runs of an x86_64 executable. We wrote a tool to extract from those traces the portions specific to the functions of interest to us. We tried to verify that the set of traces specific to any one function was a singleton by hashing each resulting trace and comparing the hashes.

We made two types of modifications to QEMU:

- Stock QEMU provides the option to dump assembly code for a translation block (TB) the first time it is seen.[1] We disabled one conditional within QEMU so it would dump assembly code for a TB every time it is seen.
- We eliminated some of QEMU's log output to make it more concise and easier to parse.

With our modifications in place, we could obtain a complete instruction trace for an x86_64 executable with a command like the following:

```
qemu-x86_64 -d in_asm,nochain executable arguments
```

Our tool to extract function-specific traces takes as input a list of pairs of the form (*address*, *path-prefix*):

- Each *address* is the start of a function of interest.
- Each *path-prefix* determines where traces for the associated function should be written.

[1] In QEMU, translation blocks (TBs) are constructed as follows. Suppose that PC is the current program counter and that I_{PC} is the instruction at PC. If execution of I_{PC} would always be followed by execution of the next instruction in memory, then I_{PC} is included in the current TB, and the process continues with PC equal to the address of the next instruction in memory. If, on the other hand, execution of I_{PC} could be followed by execution of some instruction other than the next instruction in memory, then the current TB ends with I_{PC}, and, for each instruction that could be executed following I_{PC}, a new TB is constructed with PC equal to the address of that instruction. A branch would be an example of the latter kind of instruction. To our knowledge, each QEMU translation block is a "basic block" in the standard compiler sense.

The tool monitors an incoming stream of assembly instructions. When an instruction matches an *address* of interest, the following occurs. First, the address of the previous instruction is recorded; call this value PREV-PC. Second, an associated counter is incremented, and a file is opened at the following:

path-prefix '-' counter

Instructions are streamed out to that file until an instruction is seen with an address between PREV-PC + 2 and PREV-PC + 6. That is because when targeting the x86_64, function calls are typically compiled into `callq` ("call quick") instructions. A `callq` instruction can take as few as two bytes (e.g., `callq *%rax`) or as many as six bytes (e.g., `callq *0xc97e9(%rip)`). Thus, if the instruction at PREV-PC was a `callq` instruction corresponding to a function call, the function will return to an address between PREV-PC + 2 and PREV-PC + 6. So once an instruction with an address in that range is observed, we can assume the function has returned.[2]

While instructions are being streamed out to a file, monitoring for *address*es of interest is disabled. In other words, if `foo` and `bar` are functions of interest, and `foo` calls `bar`, then those calls to `bar` from within `foo` will not generate new instruction traces for `bar`. Rather, those calls will be reflected in `foo`'s instruction traces.

If *n* is the number of times that the function associated with some *path-prefix* was called, then when the tool finishes, one will have a set of files named the following:

path-prefix`-0`
path-prefix`-1`
path-prefix`-2`
...
path-prefix-n

One can verify that the files are all the same by verifying that they all have the same hash.

Results

We recorded instruction traces for all tests in the `aligned` and `aligned-cmov` crates and for the `exercise_path_oram_z4_8192` test in the `sgx-oblivious-ram` crate with `testing::exercise_oram`'s `num_rounds` parameter reduced from 20,000 to 20. Furthermore, we built the test executables in two different configurations: in debug mode with debug assertions disabled (see TOB-MCF-001) and in release mode. Finally, we

[2] One could contrive an executable for which this assumption does not hold (e.g., using recursion or tail calls). But the executables we are testing do not involve such trickery.

compiled statically (i.e., using MUSL libc) to avoid any uncertainty that could arise from using shared objects.

We used the following specific commands to build the test executables:

- `RUSTFLAGS='-Cinline-threshold=0 -Cdebug-assertions=off' \`
  `cargo test --no-run --target x86_64-unknown-linux-musl`

- `RUSTFLAGS='-Cinline-threshold=0' \`
  `cargo test --no-run --target x86_64-unknown-linux-musl --release`

We ran the test executables using the modified version of QEMU described above. Furthermore, we ran all tests under a single thread. For example, for the `sgx-oblivious-ram` crate compiled in debug mode, we used the following specific command:

```
qemu-x86_64 -d in_asm,nochain sgx_oblivious_ram-3663d31830098817 \
    --test-threads=1
```

For each crate `X`, we collected instruction traces for any function whose mangled name included `X`, but did not include any of the following substrings:

- `6access`
- `6create`
- `4main`
- `3new`
- `4test`
- `7testing`

Our main reason for excluding such functions was so that instruction traces would be generated for their callees individually. Also, functions like “create” and “new” tended to have large instruction traces, which wasted disk space.

Our results for the `aligned` (debug mode), `aligned-cmov` (debug mode), `sgx-oblivious-ram` (debug mode), and `sgx-oblivious-ram` (release mode) appear in tables D.1–D.4, respectively. Note that no instruction traces were collected for `aligned` or `aligned-cmov` in release mode. We suspect this is because the compiler inlined all of the functions of interest, though further investigation is required.

Each table shows the name of a function, the total number of instruction traces collected for that function, the number of distinct traces (i.e., how many would be left after deduping), the number of instructions in the shortest trace, and the number of instructions in the longest trace.

So for functions that are expected to be constant time, the number of distinct traces should be one.

Furthermore, as a matter of correctness, wherever the number of distinct traces is one, the lengths of the shortest and longest traces should be equal.

Highlighted in yellow are functions that were not expected to be constant time, but also that did not appear to be.

There was one example of a function that was expected to be constant-time, but that did not appear to be. That example, `BranchCheckout::ct_insert`, is highlighted in red. The issue is discussed further in TOB-MCF-008. The instruction traces that led to the finding appear in figure D.1.

Function	Number of traces	Number of distinct traces	Length of shortest trace	Length of longest trace
`<aligned::Aligned<A,generic_array::GenericArray<u8,N>> as aligned::AsNeSlice>::as_ne_u16_slice::h4ca17d4d3234c723`	2	1	795	795
`<aligned::Aligned<A,generic_array::GenericArray<u8,N>> as aligned::AsNeSlice>::as_ne_u32_slice::h5d6f669af528a78f`	3	1	783	783
`<aligned::Aligned<A,generic_array::GenericArray<u8,N>> as aligned::AsNeSlice>::as_ne_u64_slice::h9129ee30d15d1e1a`	2	1	795	795
`<aligned::Aligned<A,generic_array::GenericArray<u8,N>> as aligned::AsNeSlice>::as_mut_ne_u16_slice::h45169ebd24b54fbe`	1	1	591	591
`<aligned::Aligned<A,generic_array::GenericArray<u8,N>> as aligned::AsNeSlice>::as_mut_ne_u64_slice::he1758cf5a39ec1f0`	1	1	591	591
`<aligned::Aligned<A,generic_array::GenericArray<u8,N>> as aligned::AsAlignedChunks<A2,M>>::as_aligned_chunks::h8ce24f8d60287d9f`	1	1	36	36
`<aligned::Aligned<A,generic_array::GenericArray<u8,N>> as aligned::AsAlignedChunks<A2,M>>::as_aligned_chunks::h918048e29826e5f6`	1	1	36	36
`<aligned::Aligned<A,generic_array::GenericArray<u8,N>> as aligned::AsAlignedChunks<A2,M>>::as_aligned_chunks::ha9f12d247ef1142a`	1	1	36	36
`<aligned::Aligned<A,generic_array::GenericArray<u8,N>> as aligned::AsAlignedChunks<A2,M>>::as_aligned_chunks::hab408180645c074e`	1	1	36	36
`<aligned::Aligned<A,generic_array::GenericArray<u8,N>> as aligned::AsAlignedChunks<A2,M>>::as_aligned_chunks::hc2f02b8d6726e612`	1	1	36	36
`<aligned::Aligned<A,generic_array::GenericArray<u8,N>> as`	1	1	36	36

aligned::AsAlignedChunks<A2,M>>::as_aligned_chunks::hd5d7ca9706d5b2a4				
<aligned::Aligned<A,[T]> as core::ops::index::Index<core::ops::range::RangeTo<usize>>>::index::h1f82fffbf551a987	1	1	149	149
<aligned::Aligned<A,[T]> as core::ops::index::Index<core::ops::range::RangeTo<usize>>>::index::h38d4cd35469a814f	1	1	149	149
<aligned::Aligned<A,[T]> as core::ops::index::Index<core::ops::range::RangeTo<usize>>>::index::h605e4cb0ab6f0f6c	1	1	149	149
<aligned::Aligned<A,[T]> as core::ops::index::Index<core::ops::range::RangeTo<usize>>>::index::h6f9240c9e2e327b4	1	1	149	149
<&aligned::Aligned<A,generic_array::GenericArray<u8,N>> as generic_array::sequence::Split<u8,K>>::split::h1083d42841a1f219	1	1	94	94
<&aligned::Aligned<A,generic_array::GenericArray<u8,N>> as generic_array::sequence::Split<u8,K>>::split::h1416c8ae9ccc61cb	1	1	94	94
<&aligned::Aligned<A,generic_array::GenericArray<u8,N>> as generic_array::sequence::Split<u8,K>>::split::h5d39967c6d5825dc	1	1	94	94
<&aligned::Aligned<A,generic_array::GenericArray<u8,N>> as generic_array::sequence::Split<u8,K>>::split::h9eb5ea4298880712	1	1	94	94
<aligned::Aligned<A,T> as core::default::Default>::default::h62e7c3f49fc46e80	1	1	15745	15745
<aligned::Aligned<A,T> as core::default::Default>::default::h7f644f7f026aca67	1	1	23321	23321
<aligned::Aligned<A,T> as core::default::Default>::default::h9380e0dc43bc747e	2	1	8055	8055
<aligned::Aligned<A,T> as core::default::Default>::default::hb186df1690052170	1	1	15745	15745

<aligned::Aligned<A,T> as core::default::Default>::default::hb8061970d5cf4261	1	1	46083	46083
<aligned::Aligned<A,T> as core::default::Default>::default::he8189944fc64491f	1	1	30931	30931
<aligned::Aligned<A,T> as core::default::Default>::default::hfe88daedcfe12140	1	1	23321	23321
<aligned::Aligned<A,T> as core::ops::deref::Deref>::deref::h489721e7b0b5fd24	2	1	5	5
<aligned::Aligned<A,T> as core::ops::deref::Deref>::deref::h7378b9fe55496d7f	2	1	5	5
<aligned::Aligned<A,T> as core::ops::deref::Deref>::deref::h88f44093f7c1f61d	2	1	5	5
<aligned::Aligned<A,T> as core::ops::deref::Deref>::deref::h8f727caafd5ec82b	1	1	7	7
<aligned::Aligned<A,T> as core::ops::deref::Deref>::deref::h8f7e822d2dde0b0a	2	1	5	5
<aligned::Aligned<A,T> as core::ops::deref::Deref>::deref::h9391186fb4725b68	1	1	7	7
<aligned::Aligned<A,T> as core::ops::deref::Deref>::deref::hbae6865f9b9194dc	2	1	5	5
<aligned::Aligned<A,T> as core::ops::deref::Deref>::deref::hbb4918d6e4ac311c	2	1	7	7
<aligned::Aligned<A,T> as core::ops::deref::Deref>::deref::hef2e62d5a845f432	2	1	5	5
<aligned::Aligned<A,T> as core::ops::deref::Deref>::deref::hf8573932fa930b23	1	1	7	7
<aligned::Aligned<A,T> as core::ops::deref::DerefMut>::deref_mut::h8ef414491e366531	1	1	5	5
aligned::Aligned::h23e2a954885d3e3b	3	1	20	20
aligned::Aligned::h274eb056ab5b230e	2	1	20	20

`aligned::Aligned::h6513936f7affe494`	2	1	20	20
`aligned::Aligned::h6f4a475334e9bef3`	2	1	12	12
`aligned::Aligned::h917a57e38b5bef7b`	2	1	11	11
`aligned::Aligned::he8e7adfebd128b58`	1	1	16	16
`aligned::Aligned::hf4a90ee2072d2f62`	2	1	20	20

Table D.1: Result for the `aligned` crate compiled in debug mode

Function	Number of traces	Number of distinct traces	Length of shortest trace	Length of longest trace
`<aligned::Aligned<aligned::A8,generic_array::GenericArray<u8,N>> as aligned_cmov::CMov>::cmov::h08c1fba89551f8a1`	8	1	135	135
`<aligned::Aligned<aligned::A8,generic_array::GenericArray<u8,N>> as aligned_cmov::CMov>::cmov::h70033abc37cf1616`	8	1	80	80
`<aligned::Aligned<aligned::A8,generic_array::GenericArray<u8,N>> as aligned_cmov::CMov>::cmov::h794eb1e9fdb57512`	8	1	115	115
`<aligned::Aligned<aligned::A8,generic_array::GenericArray<u8,N>> as aligned_cmov::CMov>::cmov::hcd6cda01b671c68f`	8	1	120	120
`<aligned::Aligned<aligned::A64,generic_array::GenericArray<u8,N>> as aligned_cmov::CMov>::cmov::h21d53ec6a7d5a38d`	8	1	275	275
`<aligned::Aligned<aligned::A64,generic_array::GenericArray<u8,N>> as aligned_cmov::CMov>::cmov::h2d7330d705aebbb2`	8	1	82	82
`<aligned::Aligned<aligned::A64,generic_array::GenericArray<u8,N>> as aligned_cmov::CMov>::cmov::h31b862a518a1c37c`	8	1	155	155
`<aligned::Aligned<aligned::A64,generic_array::GenericArray<u8,N>> as aligned_cmov::CMov>::cmov::h511cec995998b5cc`	8	1	115	115
`<aligned::Aligned<aligned::A64,generic_array::GenericArray<u8,N>> as aligned_cmov::CMov>::cmov::h5f66b5a5bb8a3996`	8	1	355	355
`<aligned::Aligned<aligned::A64,generic_array::GenericArray<u8,N>> as aligned_cmov::CMov>::cmov::hc216287842b3716b`	8	1	120	120
`<aligned::Aligned<aligned::A64,generic_array::GenericArray<u8,N>> as`	8	1	135	135

`aligned_cmov::CMov>::cmov::hfb3b873ec633b014`				
`<u32 as aligned_cmov::CMov>::cmov::h8c83a0be31e669e8`	4	1	44	44
`<u64 as aligned_cmov::CMov>::cmov::h71348b2a0d35f12c`	4	1	44	44

Table D.2: Result for the `aligned-cmov` crate compiled in debug mode

Function	Number of traces	Number of distinct traces	Length of shortest trace	Length of longest trace
<sgx_oblivious_ram::position_map::ORAMU32PositionMap<ValueSize,O,R> as sgx_oblivious_traits::PositionMap>::len::hb260c402b5745a8f	3	1	10	10
<sgx_oblivious_ram::position_map::ORAMU32PositionMap<ValueSize,O,R> as sgx_oblivious_traits::PositionMap>::write::hc2b30620c6257070	60	5	414331	458465
sgx_oblivious_ram::path_oram::meta_is_vacant::h3e81349748fb227b	60	1	62	62
sgx_oblivious_ram::path_oram::meta_leaf_num_mut::hb718979d405b9243	60	1	34	34
sgx_oblivious_ram::path_oram::meta_block_num_mut::h4de3165b20cfe6dd	60	1	36	36
sgx_oblivious_ram::path_oram::BranchCheckout<ValueSize,Z>::ct_find_and_remove::hd21c26c33c8ebe14	60	1	46426	46426
sgx_oblivious_ram::path_oram::BranchCheckout<ValueSize,Z>::pack::h877869f91724078e	60	1	960199	960199
sgx_oblivious_ram::path_oram::BranchCheckout<ValueSize,Z>::checkin::h2571e4cae2e643b7	60	1	33531	33531
sgx_oblivious_ram::path_oram::BranchCheckout<ValueSize,Z>::checkout::hb999c775aceab2c0	60	3	33688	74115
sgx_oblivious_ram::path_oram::BranchCheckout<ValueSize,Z>::ct_insert::hd81b692e185150c6	960	1	43341	43341
sgx_oblivious_ram::path_oram::details::ct_find_and_remove::h8a624165c2bcf827	60	1	15291	15291
sgx_oblivious_ram::path_oram::details::ct_insert::h5f373baf9aa27cd9	60	1	13655	13655

Table D.3: Result for the `sgx-oblivious-ram` *crate compiled in debug mode.*

Function	Number of traces	Number of distinct traces	Length of shortest trace	Length of longest trace
<sgx_oblivious_ram::position_map::ORAMU32PositionMap<ValueSize,O,R> as sgx_oblivious_traits::PositionMap>::len::h465bf979c578025f	3	1	10	10
<sgx_oblivious_ram::position_map::ORAMU32PositionMap<ValueSize,O,R> as sgx_oblivious_traits::PositionMap>::write::h215a26ddc32ce63a	60	20	376285	391112
sgx_oblivious_ram::path_oram::meta_is_vacant::h1a2894b9116daef3	60	1	50	50
_ZN17sgx_oblivious_ram9path_oram17meta_leaf_num_mut17hede6405588b79af9E.llvm.1828173529510042346	60	1	34	34
sgx_oblivious_ram::path_oram::meta_block_num_mut::h1cb63120dd0d3885	60	1	36	36
sgx_oblivious_ram::path_oram::BranchCheckout<ValueSize,Z>::ct_find_and_remove::ha5898ee8293b869a	60	1	42394	42394
sgx_oblivious_ram::path_oram::BranchCheckout<ValueSize,Z>::pack::h93aa40d6e952a27c	60	22	886015	886019
sgx_oblivious_ram::path_oram::BranchCheckout<ValueSize,Z>::checkin::h34614744b385f640	60	1	33303	33303
sgx_oblivious_ram::path_oram::BranchCheckout<ValueSize,Z>::checkout::hb661404e92f99c2e	60	3	33398	73096
sgx_oblivious_ram::path_oram::BranchCheckout<ValueSize,Z>::ct_insert::h3596df4040fc06a7	960	2	40119	40120
sgx_oblivious_ram::path_oram::details::ct_find_and_remove::h6d9f8daf8429a42f	60	1	13947	13947
sgx_oblivious_ram::path_oram::details::ct_insert::hf8875fa4262af511	60	1	12851	12851

Table D.4: Result for the `sgx-oblivious-ram` crate compiled in release mode

```
...
0x400001400a:  48 89 44 24 10           movq    %rax, 0x10(%rsp)
0x400001400f:  48 8d 7c 24 08           leaq    8(%rsp), %rdi
0x4000014014:  48 8d 74 24 10           leaq    0x10(%rsp), %rsi
0x4000014019:  ff 15 e9 97 0c 00        callq   *0xc97e9(%rip)
0x400009fa90:  48 8b 06                 movq    (%rsi), %rax
0x400009fa93:  48 33 07                 xorq    (%rdi), %rax
0x400009fa96:  74 10                    je      0x400009faa8
0x400009faa8:  b9 40 00 00 00           movl    $0x40, %ecx
0x400009faad:  b8 40 00 00 00           movl    $0x40, %eax
0x400009fab2:  29 c8                    subl    %ecx, %eax
0x400009fab4:  c3                       retq
0x400001401f:  29 c5                    subl    %eax, %ebp
0x4000014021:  89 e8                    movl    %ebp, %eax
0x4000014023:  48 83 c4 18              addq    $0x18, %rsp
...
```

```
...
0x400001400a:  48 89 44 24 10           movq    %rax, 0x10(%rsp)
0x400001400f:  48 8d 7c 24 08           leaq    8(%rsp), %rdi
0x4000014014:  48 8d 74 24 10           leaq    0x10(%rsp), %rsi
0x4000014019:  ff 15 e9 97 0c 00        callq   *0xc97e9(%rip)
0x400009fa90:  48 8b 06                 movq    (%rsi), %rax
0x400009fa93:  48 33 07                 xorq    (%rdi), %rax
0x400009fa96:  74 10                    je      0x400009faa8
0x400009fa98:  48 0f bd c8              bsrq    %rax, %rcx
0x400009fa9c:  48 83 f1 3f              xorq    $0x3f, %rcx
0x400009faa0:  b8 40 00 00 00           movl    $0x40, %eax
0x400009faa5:  29 c8                    subl    %ecx, %eax
0x400009faa7:  c3                       retq
0x400001401f:  29 c5                    subl    %eax, %ebp
0x4000014021:  89 e8                    movl    %ebp, %eax
0x4000014023:  48 83 c4 18              addq    $0x18, %rsp
...
```

Figure D.1: Instruction traces leading to [TOB-MCF-008]

Discussion

Table D.3 paints a somewhat rosier picture than the reality. While `write` is not expected to be constant time, there are calls to `ct_find_and_remove` and `ct_insert` buried within its instruction traces. A manual review of those traces leads us to suspect that other bugs like TOB-MCF-008 may reside within the code.

One impediment to applying this analysis to MobileCoin Fog is the following. Several functions of interest are not constant time in the strict sense, as their runtimes may vary with the size of their arguments. However, their runtimes should not vary with their arguments' contents.

The `details::ct_find_and_remove` function (figure D.2) is an example. The function loops over its argument `src_meta`, and thus is clearly not constant time in the strict sense. However, one would expect its runtime not to be affected by the contents of `src_meta`.

```
    pub fn ct_find_and_remove<ValueSize: ArrayLength<u8>>(
        mut condition: Choice,
        query: &u64,
        dest_data: &mut A64Bytes<ValueSize>,
        dest_meta: &mut A8Bytes<MetaSize>,
        src_data: &mut [A64Bytes<ValueSize>],
        src_meta: &mut [A8Bytes<MetaSize>],
    ) {
        debug_assert!(src_data.len() == src_meta.len());
        for idx in 0..src_meta.len() {
            // XXX: Must be constant time and not optimized, may need a better barrier
here
            // Maybe just use subtle::Choice
            let test = condition
                & (query.ct_eq(meta_block_num(&src_meta[idx])))
                & !meta_is_vacant(&src_meta[idx]);
            dest_meta.cmov(test, &src_meta[idx]);
            dest_data.cmov(test, &src_data[idx]);
            // Zero out the src[meta] if we moved it
            meta_set_vacant(test, &mut src_meta[idx]);
            condition &= !test;
        }
    }
```

Figure D.2: sgx-oblivious-ram/src/path_oram/mod.rs#L475-L496

For such a function, grouping all of its instruction traces together and expecting them to be the same is misguided. Rather, one should group together all of its instruction traces for arguments of the same size.

How could our instruction tracing tools be adapted to determine the size of a function of interest's arguments? We can imagine two possibilities.

First, the tools could inspect the internal state of QEMU (e.g., its registers, call stack, memory, etc.). However, such a solution seems fragile. It could break from either updates to QEMU or to how the compiler generates code for the function of interest.

Another possibility would be to have the code explicitly tell the tool the size of its arguments. For example, the code might contain special instructions that have no other effect than to say, "Hey, tool, if you are watching, I'm about to operate in arguments of size N." The x86_64's existing `nop` instruction seems to be an ideal candidate for this purpose.

References

- nop with argument in x86_64