CryptLog

Author: Dylan Visher, Supervisor: Mahesh Balakrishnan

Abstract

CryptLog builds on research on State Machine Replicated (SMR) Data Structures. CryptLog is a systems research project that aims to bring recent advances in homomorphic encryption into the field of SMR Data Structures. Previously SMR Data Structures were shown to be able to be scalable, performant and efficient with systems like Tango. However, as compute and storage shifts from personally owned and maintained machines to third-party controlled cloud providers like Amazon Web Services and Google Compute Engine, it becomes essential to start protecting private and sensitive data from these honest-but-curious cloud providers. In order to protect against this new adversary model, CryptLog leverages various types of partially homomorphic encryption schemes including Orderable, Additive, Deterministic encryption schemes, in order to yield an SMR system that is both efficient and secure. While naively encrypting each entry in the log would provide security, a VM would no longer be able to provide many of the optimizations like snapshots that are essential to making SMR Data Structures practical in the real world. CryptLog offers the best of both worlds by offering strong encryption while still offering these performance optimizations. Clients can choose for themselves how they want to trade off the guarantees of security with performance, while never having to fully sacrifice end-user privacy. CryptLog is built
using the new systems programming language, Rust, and has a storage backend using DynamoDB. It is built to have a pluggable database backend and makes it easy for clients to use either existing data structures or develop their own third-party data structures on top of CryptLog’s runtime and encryption layer.

**Crypto Schemes**

**Encryption**

As mentioned above, CryptLog leverages many cryptography systems, both symmetric and asymmetric alike. The clients are considered honest in our model and have both the public and the private keys of this system for encryption and decryption. In certain cases, like additively homomorphic encryption, the VM must have access to the public keys to encrypt an initial value. Therefore the VM does not have any ability to decrypt data and can only encrypt when it is given the public key for a subset of the crypto systems used. Furthermore, this can be limited by eliminating the usage of default values.

Non Deterministic Encryption is the most secure encryption scheme, and all the other encryption schemes that we offer are optional, opening the door to performance optimizations like snapshotting with a small security trade-off.

**Deterministic/Equable**

For deterministic encryption we use AES-256 with a SHA-256 prefix. The SHA-256 prefix allows for equality comparisons (given its assumptions regarding no hash colli-
sessions), while AES-256 is used for non-deterministic encryption of the data. Even if the data is encrypted with different private keys, the VM can compare all values for equality by comparing their hashes. Since the SHA-256 hashes are assumed to be uninvertable and collision free, it does not compromise the security much. It opens the door to chosen-plaintext attacks, but SHA-256 is assumed to not be vulnerable to these types of attacks. Also the VM can start performing frequency analysis on the encrypted data which in some cases can compromise the security of the data.

Additive

For additive encryption we use a Paillier encryption scheme. It is a probabilistic asymmetric, public key, encryption scheme. It allows for two encrypted values to be added together by multiplying their cipher-texts modulo a public key parameter.

The Paillier Encryption Scheme is based on the decisional composite residuosity assumption which is believed to be intractable. However, as a consequence of the homomorphic properties, the encryption system is malleable. Due to the inherent malleability of homomorphic encryption schemes, this does leave the system more vulnerable to “malicious” actors that may seek to mutate the data that clients are receiving, but this is not expected in the “honest-but-curious” cloud provider model. However, by using additively homomorphic encryption CryptLog could be vulnerable to attacks where an attacker can add existing cipher texts and compare them against existing cipher texts to discover their relationships to each other. The Paillier crypto system, however, is non-deterministic so \(3 + 2\) might not look equal to a separately encrypted \(3 + 2\), though they will both decrypt to 5. This offers some level of security against these types of attacks which seek to find these relations.
Data Structures

Counter

We implemented a simple Counter Register. Which offers a simple Get and Put operation, as well as an atomic add operation which atomically adds a given amount to the register. This register consists of a single integer value which can be encrypted by the aforementioned Paillier crypto-system. This means that when an “ADD R 5” operation is appended to the Log encrypted, the VM can, without decrypting the values, add X to counter register R, where R is an encrypted 5.

Ultimately this allows the VM to represent the counter by a single Paillier encrypted number rather than a log of operations done to the counter. This allows efficient snapshotting of the counter.

HashMap

Another, more complex data structure that we implemented was a hash map. The client side hash maps work exactly as one would expect, but when running on the VM, it runs a different implementation (though to a similar effect). This illustrates the fact that a log of operations can be interpreted differently by different implementations. For instance a hash map can be both implemented as a hash map with O(1) lookup, insertion and deletion or it can be implemented as a list of elements with O(N) lookup, insertion and deletion.

Since a hash map needs to hash elements, there is no way to hash values on the VM that is consistent with the hashes on the client side (since it doesn’t have the same
unencrypted data). Two options for how to implement the hash table are as follows: a list of buckets where insertion operations says exactly which bucket to append the element to (hashing with chaining); or to implement it using a list of elements that can be compared for equality, in other words a list of unique values.

The first implementation has the same performance advantages of a hash map, even when run on the VM, but requires more implementation details to leak into the values appended to the Log. This makes it harder to have multiple different object implementations running off of the same log: i.e. hashing with chaining and hashing with linear probing. Furthermore, when the VM sends a snapshot to the client, it could be decoded in-place.

The second implementation has a naive solution where the VM just performs \(O(N)\) insertions and deletions in a list of values, but also leaves the door open for a more optimized solution where values can inserted into a local hash map, in a different location than the client might have inserted it. Both of these second implementations have the disadvantage decoding of values cannot be done in place, and require \(O(N)\) additional space to construct the client's view of the hash map. But the optimized version also has the advantage of providing just as fast of operations in the VM and being flexible to multiple different possible implementations of the underlying hash map on the VM.

Ultimately due to the elegance of the opaque log entries with the flexibility of multiple underlying VM implementations we chose to implement the second implementation.
VM

The VM is expected to be run on a centralized, honest-but-curious cloud provider, but could easily be run on a distributed or decentralized system as well. The VM leverages many properties of homomorphic encryption to increase performance.

The VM performs two main functions, facilitating client log streaming, and facilitating client log writing.

To facilitate client log streaming:

1. The VM batches operations to the cloud storage and has a small local cache of entries
2. It maintains internal skip-list of log entries such that it can quickly return only the operations relevant to the object the client is replicating.
3. It maintains an array of per object snapshots so that it can quickly catch up clients and can compact the logs.

To facilitate client log writing:

1. The VM accepts write requests from clients and linearizes the appends to the log locally to reduce contention on the Log by ensuring that multiple clients do not obstruct each others’ writes.

Batching

Clients, without the vm, currently stream the log by requesting a bucket that stores the log index, then serially reading from the index where they left off up to the up to the log index they just read. Depending on the cloud-storage provider used, this could be
optimized by doing a batch get request to the cloud-storage provider. For now, the VM offers a fast way of caching the last log entries and returning it all to the client in a single request.

Even if the cloud storage provider did offer a batch get request it would still take longer than the VM since the client would first have to read the log index entry, which would cost one full round-trip, whereas, using the VM, it would not.

**SkipLists**

In addition to returning all the log entries in a single response, the VM also helps remove unnecessary data from being sent to the client. Since CryptLog supports multiple objects being operated on concurrently, a common scenario is that one client replicating object A which is not busy, does not want to have its performance impacted by a very busy object B. The VM helps in this scenario by maintaining an internal skip list that stores, for each object id, the log entries that are relevant to that object. This way when a client replicating object A requests to stream the most recent update for it from the VM, the VM can send them a much smaller set of log entries for it to replicate locally.

SkipLists are implemented using a simple hash map, mapping object ids, to sorted lists of entry indices. These entry indices then can be looked up in the local log cache, or via a network request to the cloud based log. The skip list is updated in the runtime’s per object callbacks. The log cache is update pre object callbacks to ensure that any update to the skip list is already in the log cache.
**Snapshotting**

To handle the possibility that clients are catching up from being very far behind, or perhaps they are just starting up for the first time, the VM also provides a snapshotting service. The batching that the VM offers provides log entries in a minimal number of network requests, however, this data can be highly compressible. For instance if there is a map with 100 unique keys, but 10,000 write operations have been performed on it, there are only 100 writes that actually matter. By snapshotting objects, the VM provides a way of compressing these log entries before sending them to the client.

Snapshots are implemented by the VM replicating client objects locally. The VM performs the exact same operations that the client would, except on encrypted data. This is done by leveraging homomorphic encryption schemes. For example, for a BTreeMap, the VM performs operations over keys encrypted using an orderability constraint. This allows the VM to do logarithmic lookup, insertions, and deletions, just like the clients, all while being oblivious to the actual data that it is operating on.

These objects are then periodically snapshotted by the VM. The VM serializes the object and stores the snapshot of it. This way when a client is attempting to catch up and it is sufficiently far behind, the VM can just send it the snapshot to catch it up most of the way, and then send only the log entries not included in the snapshot.

The taking of the snapshots is performed after the object callbacks in a post_callback hook. This ensures that the snapshot is taken after the object has updated with respect to the most recent log entry.
Linearized Appends

The writers to the log have to busy wait while appending to a log by repeatedly trying to append at a given index and advancing their index if they discover the log index is already taken. This polling architecture is due to the most simple storage providers not offering an atomic append operation to a large array. If N clients are attempting to perform a write concurrently, in the worst case this can degenerate to a O(N^2) time complexity as they all try to write to index i, and only one makes it, so n-1 attempt to write at i+1, and n-2 attempt to write at i+2, etc. Though potentially an unlikely scenario, this type of worst case behavior should be protected against.

The VM offers the ability to efficiently linearize appends. By sending all write requests through the VM, the writers eliminate the contention that they would have experienced otherwise. This yields a worst case time complexity of O(N), given that the VM does not fail. And since all the log appends are passing encrypted data to the VM, there is no increased danger of this honest-but-curious compute provider revealing private information.
Benchmarks

The above graph graphs read time when there are N concurrent writers to a single BTreeMap. This shows that as the number of writes increases, the amount of time it takes to perform a read increases close to linearly if not using the VM, and is roughly
constant if using the VM. This also shows that the encryption adds very little overhead and its effect is mostly dwarfed by networking and other costs.

This graph formalizes the observations above, by showing directly how read time is impacted by the number of intervening writes. Since the VM batches responses and has snapshots to keep the message size relatively small, the time per operation is
roughly constant for the VM, while without using the VM the time per operation increases linearly (as one would expect).