CryptLog: A cloud-driven approach to enabling clients to share and synchronize sensitive data

Iulia Tamas
Supervisor: Mahesh Balakrishnan

Abstract
Recent advancements in Cryptography show the viability of performing useful operations on encrypted data in a timely manner. Performing operations on encrypted data is useful in that it can allow a third party, such as a VM, to perform time consuming operations for the client without compromising the security and privacy of the handled data. Order revealing and order protecting encryption schemes are prime candidates for encrypting information in data structures that rely on ordering, such as BTrees, or Binary Search Trees, as they allow maintaining the essential property of key orderability. Additive Homomorphic Encryption (AHE) schemes allow addition over encrypted data to be reflected in the corresponding decryption. This makes AHE schemes useful when implementing counter-like structures. Equatable encryption, where equality checks between encrypted data blocks are speedy, allows implementing a protected HashTable, with both keys and data encrypted. Vanilla deterministic encryption allows users to protect all other data, that is intended to be stored but not manipulated while encrypted. We propose CryptLog, a way of extending the capabilities of a traditional shared-log approach to synchronization, to include the protection of sensitive data from an honest but curious cloud service provider, through the use of modern flexible encryption schemes. We implement the mentioned encryption schemes and show that adding encryption to the shared-log data comes at a marginal cost to the client. We show that the VM is able to perform efficient snapshotting and streaming, and implement efficient skip lists over encrypted data. This leads to network and not CPU times dominating, making CryptLog a viable alternative to unencrypted shared logs.

I. Introduction
Cloud-driven architectures and cloud service providers are growing increasingly popular and widespread. Their usefulness is varied, from allowing individual clients to outsource their work, to helping distributed systems servers communicate and synchronize.

Tango[1] proposes using a cloud-stored shared log for backing replicated in-memory data structures. The benefits for the clients of Tango are many: flexibility in choosing the data structures to represent their data or metadata, multi-object transactions, replication and coordination, log replay, object-based streaming. What Tango doesn’t offer, however, is protection against an honest but curious cloud service provider. As such, the benefits of Tango come at the cost of having to fully trust the cloud provider.

We propose CryptLog, a Tango inspired system which allows clients to synchronize the state of in-memory data structures via an encrypted shared log, without losing the ability to perform streaming, and snapshotting in the cloud. CryptLog, to the best of our knowledge, is the first SMR system that offers encryption of shared log data without a significant performance loss.
II. **Deliverables Summary**

The project was a team effort, that required integrating my work with that of another undergraduate. The git repository mentioned in the ‘Code’ section showcases our work.

I took lead on the following:

- SMR Runtime
- Transaction Logic, including Transaction Validation
- Order Revealing/ Order Preserving Encryption
- SMR Data Structure: BTreeMap (Encrypted and Unencrypted), Register
- Http Client and Server to allow communication with the SharedLog
- Marshaling and Unmarshaling logic for various data types

I collaboratively implemented the following:

- MetaEncryptor which allows the integration of Crypto primitives in the SMR Data Structures
- Logic to allow encrypted and unencrypted types to coexist and be handled properly in the SMR Runtime and SMR Data Structures
- Converters to allow transitions between encoded and unencoded types, as well as the various encrypted and unencrypted types.
- SMR Data Structure: HashMap (Encrypted and Unencrypted), Counter (Encrypted and Unencrypted)
- Application code to test CryptLog

I. Transactions

Transactions are composed of reads, writes, and operations.

The reads set contains pairs of (object id, version), capturing the object ids of the objects read during the transaction and their versions at the time the transaction started.

The writes set contains the object ids of the objects the transaction wants to modify.

The operations array describes all the operations performed. The operations are represented as enums specific to, and created internally by the SMR data structure. The enums are encoded, or encoded and encrypted.

A log entry holding a transaction additionally keeps track of the transaction type (begin: TxBegin, or end: TxEnd) and the transaction state (validated, aborted, or undecided). This allows the transaction validator to inspect whether the changes that happened between pairs of TxBegin and TxEnd invalidate the transaction.

II. SMR Runtime

The Runtime handles communication between the SMR data structures and the shared log. It sends the data structure’s writes to the shared log to be appended, and polls the shared log for the latest log entries to update the data structure’s local view.
RegisterObject(obj_id, callback)

Clients have to register the object (data structure) with the Runtime by providing an object_id and a callback.

Append(obj_id, encoded_operation)

If we are operating within a transaction, the runtime adds the encoded_operation to an array of pending transaction operations, and adds the object_id to our current set of writes.

If we are not operating within a transaction, the runtime creates a log entry around the encoded_operation and obj_id and sends it to the shared log.

The encoded operation can also be encrypted.

Sync(obj_id)

The sync operation is called by data structure read operations before reading their local state. Calls InternalSync(obj_id, None)

InternalSync(obj_id option, tx_idx option)

The runtime streams log entries from the shared log. The logic of stream is abstracted away, and is currently implemented as http requests to the shared log. The http responses are fed to the runtime through a channel.

As log entries are streamed, the operation(s) within are sent to all the SMR data structures affected by them (these are the data structures that registered callbacks for the object_ids of the operations through the RegisterObject function above).

If we are in a transaction mode, the pair(obj_id, version) is added to our current set of reads.

If as we are streaming, we encounter a transaction, we attempt to validate it. If the transaction needs to be aborted, we continue with the next log entry, otherwise we proceed with the callbacks.

BeginTx

Sets transaction mode to true and calls sync(None). As a result, the sync will lead to the updating of all objects that the runtime has had registered. This behavior allows us to have one synchronization point, namely the beginning of the transaction, for all the objects the transaction might want to operate on.

EndTx

The Runtime assembles a LogEntry of type End Transaction, packs it with the current reads, writes and operations and sends it to the Shared Log. Then the runtime clears its reads, writes and operations. Lastly, the runtime calls InternalSync(None, its transaction idx), a blocking call that allows the runtime to return to the user only after the transaction has made it to the log and back.

CatchUp(obj_id, callback)

Streams the shared log and reports the operations to the interested objected through the callback. Called from RegisterHandle to make sure a new view into an existing objected gets up to speed. It provides a nice way of frontloading the work of catching up.
ValidateTx(log_entry)

Goes through the (object_id, version) pairs in the read set of the log_entry and checks that none of the object versions are older (smaller) than the ones we currently have. In other words a transaction is valid if in the time between it started and the time it got appended to the log, no changes occurred to the transaction’s readset.

III. Order Revealing/Order Preserving Encryption

We implemented ‘Practical Order-Revealing Encryption with Limited Leakage’ as described in the Chenette paper [2]. Here is a summary of the encryption scheme proposed by Chenette et al [2]:

To encrypt a message m made of bits \(b_1, b_2, \ldots, b_n\) we make use of a pseudo-random function \(F\) and a secret key \(K\), as follows:

\[
u_i = F(K, (i, b_1 b_2 \ldots b_{i-1} || 0^{n-1})) + b_i \mod M.\]

\(M\) is set to \(2^{40}\) as recommended by the authors. \(u_i\) is the encryption of \(b_i\). Note that our expansion factor, encrypted message to plaintext is 40 to 1. When we compare two encrypted messages \(X\) and \(Y\), we find the first pair of both \(X_i\) and \(Y_i\) that are different [2].

If \(X_i + 1 = Y_i\) then \(X < Y\) [2]. Otherwise \(Y < X\) [2]. Of course, if no such pair of bits exist, the two messages are equal.

We implemented the pseudo-random function \(F\), using the Goldreich-Goldwasser-Micali method [3].

IV. SMR Data Structures

We allow clients to synchronize state via a shared log in the cloud. The abstraction provided to the end application is that of a shared data structure that provides strong consistency.

Each SMR Data Structure contains: an object id, a pointer to its Runtime, a local view into the SMR Data Structure, and a MetaEncryptor. The object id serves to uniquely identify the data structure in the Runtime. The Runtime facilitates the communication with the cloud-stored encrypted shared log. The local view is an instance of a standard library object that contains a local copy our SMR Data Structure, reflecting all the writes to the data structure, effected by any of the clients, as containted in the shared log at a point in time (For example, an SMR HashMap implemented in rust has a rust HashMap as its local view. An SMR counter has an integer as its local view).

All data structures are implemented based on the same two principles:

- all reads first perform a blocking sync operation through the runtime. This operation updates the smr data structure’s local view. Then they carry out the read operation on their local view and return the result
- all writes create a write operation that they encode and pass to the runtime through calling the runtime’s append method. Additionally the writes can also encrypt the the operation with one of the encryption methods described in the MetaEncryptor.

V. SMR BTreeMap

The BTreeMap respects the rules outlined above, and is a structure that contains:

- a constructor that takes as arguments: a pointer to the SMR Runtime we want responsible for the object, the object id, the initial state of the data structure. The object id allows registering
this object with the runtime so that it can keep track which log entries are relevant for this
data structure. In the constructor we also instantiate converters specific to the data structure
types. This allows us to seamlessly convert between the format the data has locally and the
format the data has in the log (for eg. decoded and unencrypted vs. encoded and encrypted)

- a start function that registers a callback with the runtime, to allow the runtime to communicate
to the data structure updates coming from the shared-log;

- a register object function which registers the current object with the runtime and makes a
  synchronous call to the catch up function to make sure the object is up to date before returning
to application code

- an in memory BTMap (a local representation of data persisted and shared through the
  shared-log)

- get(key): a read method which first synchronizes local state through the runtime (via calling
  runtime.sync(obj_id)), and then evaluates the get on the in memory data structure

- insert(key, value): a write method which first converts the key and value data types to the
  data format we want kept in the log (for example encoded and encrypted or just encoded),
  then assembles the key and value in a MapOp::Insert operation and passes it to the runtime,
  where it gets further assembled in the a log entry and sent to the shared log.

- callback method receives as argument an operation passed to the data structure by the runtime;
  if the operation is a MapOp::Insert, the operation gets decoded and decrypted, and the key and
  data are pulled out and inserted in the local BTMap; if the operation is a MapOp::Snapshot,
  the snapshot is decrypted, a new BTMap is created from the snapshot, and the local BTreeMap
  is replaced by this new BTMap

III. Results

We tested our system’s performance with Encrypted vs. Unencrypted BTreeMaps, in the absence or
in the presence of a VM performing snapshotting and implementing skip lists.

I. Plots

The first figure shows that the benefit of encrypting data comes at almost no cost to the client.

The green-blue pair of lines show the system operating in the absence of a VM, with and
without encryption, respectively. The red-purple pair of lines show the system operating with VM
functionality enabled, with and without encryption, respectively. The lines in the two pairs are very
close to each other, showing the small cost of adding encryption. Figure 1 also shows the benefit of
‘outsourcing’ client work to the VM, by allowing the VM to ease the burden of log replay on the
client-side through its snapshotting mechanism and skip-lists. Without the VM the read latency of
the client grows linearly with the number of writes, as the client has no choice but to replay the log.

The second plot emphasizes the benefit of enabling the VM, and further proves that the VM
functionality of snapshotting, streaming and skip lists can be offered on encrypted data, at a similar
cost to that of working on unencrypted data.

Thus, our results suggest that using CryptLog is a viable way of protecting shared data, while
maintaining performance. We think that the encrypting comes at a small cost because the networking
time ends up being higher than the cpu time.
Figure 1: Read latency of one client reading every 5 seconds vs. # of clients continuously writing to log

Figure 2: Recovery time of one client catching up vs. number of write operations between consecutive reads
IV. Work and Collaborators

My work on CryptLog was carried out under the supervision of Professor Balakrishnan and in direct collaboration with another undergraduate. Advice and guidance on the use of Cryptography was kindly provided by Professor Mariana Raykova. The graphs showing the performance of CryptLog were not created by me, but by the other undergraduate working on the project.

V. Code

More details about our work, including the parts not addressed in detail in the report (such as the http client-server implementation, the specifics of converters and marshallers, the testing infrastructure) can be found in the git repository. The code, written in rust, is available here:

https://github.com/iuliatamas/CryptLog

References

