Faction Cooperative Report: Sensitive Data Management

Ari Rodriguez, Miles Thompson

March 13 2020

1 Introduction

At Faction, individuals collectively own, pool, and monetize their data. Secure data storage with strong privacy guarantees is paramount. We aim to give data owners safety and full control over their Faction data, with minimal technical requirements or potential for error on the side of co-op members. We further aim to give Faction members the ability to authorize data for use in Secure Multi-Party Computation (MPC). Using the cryptographic techniques of MPC, a given program can be *securely* evaluated, whereby the inputs remain strictly private and only the final output of the desired computation is public (to authorized parties). In this way Faction data can be computed on *without ever being revealed* even to the machines jointly processing this data.

Suppose we have a data owner O holding data d. The goal is to:

1. **Securely store** d in such a way that it is highly available to O (across any devices the owner may control and with fault tolerance) but only accessible by O and not readable by anyone else.
2. **Safely share** selected subsets of their data d with another trusted party, P, within a constrained context and only to the extent and within the time frame where the data owner O wishes to grant access to P.
3. **Securely Compute** on selected subsets of data d to obtain $f(d)$ using MPC techniques where d remains secure and indecipherable even in the process of computing $f(d)$.

The report below outlines Faction's secure methods for these three tasks.

2 Secure Data Storage and Access

End to end encrypted cloud storage is a well understood problem. The data owner O generates and securely stores a symmetric key k. O encrypts d with key k to produce ciphertext c. They then derive h, the content hash of the

ciphertext. The data owner can now safely give any third party a copy of the pair (h, c) to store on their behalf. They no longer have to maintain a copy of the actual data (d or c) and only have to retain control over the (much smaller) private key k and the content pointer h. The data owners can access their data (or store new data) from any device, provided only that they can access k and h, preferably on a hardware secure module (HSM).

One well known risk of this model is that while third parties cannot read their data, the data owners can still be denied access if the third party does not remit a copy of the ciphertext (either maliciously or due to failure). For this reason we recommend that Faction will distribute a copy of encrypted data c to the (decentralized) Inter-Planetary File System (IPFS) and then 'pin' a copy of that data on behalf of the member. IPFS is a robust peer-to-peer file storage system where continued storage can be incentivized by the use of Filecoin. The member can simply pay Faction who then ensures the data is stored in a redundant, resilient fashion across servers under the control of Faction as well as servers not in their control. As long as O retains, locally, the IPFS content hash h and their symmetric key k then d is both secure and available to O without the need for O to "trust" Faction servers or any third party.

Because the owner may wish to store more than one piece of data, for reasons of practicality we recommend that data owners should have a 'master key' k_{id} as a KEK (key encryption key) stored on their HSM. The master key can then be used to decrypt a ciphertext c_{id} whose plaintext p_{id} contains a list of tuples like so:

$[(r_1, h_1, k_1), (r_2, h_2, k_2), \ldots (r_i, h_i, k_i)]$ where:

- r_i = Reference information for data file d_i (identifier/name, data type, etc.)
- h_i = IPFS Content Hash for ciphertext c_i
- k_i = Decryption key for ciphertext c_i

In this way, a single key (isolated on an HSM) can be used to recover the entirety of a Faction member's stored data, however different data files can have different decryption keys which may be desirable in many cases (if a decryption key is compromised it will only compromise one piece of data, for example).

2.1 Storage

Data owner O wants to store medical data m with Faction:

1. O generates a fresh symmetric key k_m and specifies the reference information r_m (simple as a common name to identify data m i.e. 'my medical exam').
2. O locally computes encrypt(m, k_m) and obtains ciphertext c_m
3. O obtains IPFS Content Hash h_m by computing Hash(c_m)

4. O sends ciphertext c_m to Faction. Faction stores c_m on their servers under the reference id h_m. Faction also distributes a copy of c_m to IPFS. Faction responds upon success.

5. O appends tuple (r_m, h_m, k_m) to their list of stored data files p_{id} and obtains an updated c_{id} by computing encrypt(p_{id}, k_{id}).

6. O sends c_{id} to Faction who stores the updated ciphertext and pins it on IPFS.

2.2 Access

Data owner O wants to access medical data m stored with Faction:

1. O requests c_{id} from Faction (or IPFS) with reference h_{id}. O obtains p_{id} by running decrypt(c_{id}, k_{id}).

2. From decrypted p_{id}, O seeks the entry (r_m, h_m, k_m) for genetic data m.

3. O requests c_m from Faction/IPFS with reference h_m. O obtains m by running decrypt(c_m, k_m).

2.3 Key Recovery via Shamir's Secret Sharing

One noticeable vulnerability with the storage system above is that it relies heavily on key k_{id} to always remain under the control of data owner O. If the key is lost or compromised, O could be in deep trouble. In decentralized systems (or end to end encrypted systems, for that matter) there is no central authority one can appeal to for keys to be restored or data to be decrypted. Advanced cryptographic techniques like Shamir's Secret Sharing can help us to approximate the convenience of a centralized system's account recovery mechanism, while continuing to provide strong security guarantees that don't involve an insecure backdoor for a central authority or administrator.

Adi Shamir's Secret Sharing Scheme (SSS) is an information-theoretically secure primitive for threshold cryptography based on polynomial interpolation over a finite field. In plain English: SSS allows a number of parties to each have a "share" of a secret value, in such a way that no share leaks any information about the underlying value, but whenever a large enough subset of shareholders (above the threshold) reveal their shares, the secret value can be reconstructed. SSS is used in many high risk key recovery scenarios: today, the crypto-currency exchange Coinbase uses SSS for private keys stored in their Vault, with shares locked in cold storage at different locations around the globe.

The recommendation here is that co-op members secret share their cryptographic key k_{id} to a number of parties on the Faction network. Some might be Faction administrators, others might be trusted individuals on Faction (i.e. friends and family). In the case that a co-op member O loses her k_{id} (imagine an HSM is lost or fails and a backup HSM does not exist), a petitioning process for recovery can be run. Parties that hold a share of k_{id}, upon being convinced

that the request legitimately comes from O (out of band confirmation, email verification etc.), may remit their share to O. When O collects enough shares, by proving their identity to enough shareholders, O can locally reconstruct k_{id} without ever revealing the value to anyone. As long as a large enough subset of those parties never collude against O (come together to reveal O 's secret k_{id} for themselves), the key stays absolutely private. Such collusion can be mitigated in practice by setting the threshold quite high, having a diverse set of shareholding participants, and not necessarily revealing the list of other counter-parties to each counter-party.

3 Secure Data Sharing

With end to end encrypted storage, how can we reveal our data to a trusted party without causing potential insecurities in transit? Let us continue the example where a data owner O has medical data m securely stored with Faction, and now wants to share parts of m with a doctor. Assuming doctor and patient may want to asynchronously request and then fulfill the access for data we outline the steps to do this. The full data sharing request and response lifecycle is below:

1. Doctor provides the query to Faction which includes: a request id; a reference to data owner O and her data m being queried; a verified public key pk_{doc} controlled by the doctor; and (if applicable) the relevant portions of m that the doctor actually wants access to.

2. The query is processed by Faction who waits for confirmation from data owner O.

3. When O accepts and is ready to grant the doctor access, O receives the doctor request and locally accesses m (as outlined in section 2.2).

4. O then takes pk_{doc} and the private part of her own verified public key pk_{owner} to derive a shared secret s (Diffie-Hellman Key Exchange). The shared secret s is used as a symmetric key in encrypt(m, s) which outputs c_{doc} (Integrated Encryption Scheme). O responds to Faction with: the request id (from doctor's request); her verified public key pk_{owner}; and the ciphertext c_{doc}.

5. Finally, the doctor receives the response O submitted to Faction. The doctor derives shared secret s from pk_{owner} and the private part of pk_{doc}, and runs decrypt(c_{doc}, s) to obtain g. The doctor confirms with Faction that the requested data was correctly received.

In this way Faction can cache requests and the responses, allowing the data owner to transfer the sensitive data to the doctor asynchronously, still without ever giving Faction access to sensitive information.

4 Secure Multi Party Computation

Finally, it is possible to allow Faction data to be processed on, without ever being directly revealed. Secure Multi Party Computation is a cutting edge approach for participants to collectively compute on private inputs to obtain a valid output without ever revealing the inputs or any internal state of the computation. In this way Faction members can authorize data to be used in computations, with cryptographic proof that a member's personal data input into a given MPC operation will stay absolutely private.

There are a number of different settings for Multi Party Computation depending on the threat model and the parties involved. We propose a three party computation in the semi-honest setting following in the footsteps of the peer-reveiwed and commercially viable Sharemind protocol:

specifics here: https://eprint.iacr.org/2008/289.pdf

On this model, data owners would use a secure Secret Sharing Scheme to give a share of their personal data to each of the three computing servers, which should each be under the control of distinct, trustworthy, non-colluding entities (a Faction Co-op controlled server could be one of these entities). The three servers collect data shares from all participants, and shares will only be collected and input into a pre-established computation which is a compiled program that participants could validate (if desired). Even if one server is entirely corrupted by an attacker during the computation, the data still remains information-theoretically secure (if the attacker deviates from the protocol it will simply fail, if the attacker tries to extract information from the leaked data it will learn nothing). However, if share information on two or more servers are exposed than security is trivially compromised. This is why non-collusion of computing parties is an absolute necessity and the handling of data shares by the servers is of utmost importance. However provided that two or more servers are not successfully and simultaneously attacked, and computing parties do not collude, arbitrary computation can be done on the input data in a secure, privacy preserving way.

To do this, we simply have to express the desired computation in a compiled circuit which is fed to the three computing parties along with the data shares from end users. The parties securely execute the circuit which involves a number of rounds of communication (the communication complexity can be diminished with an offline preprocessing phase as described in MPC literature).

5 Cryptographic Standards

1. Symmetric key k_{id} should be a 256-bit AES (advanced encryption standard) key, used in CBC mode. Ciphertext c_{id} should always have a MAC (message authentication code) or auxiliary signature scheme, for authentication.

2. Asymmetric keys (public/private keypairs) should be 256-bit ECDSA or Ed25519. Public keys should be "verified" with a valid cert from a CA (certificate authority). Public key types have to match in a Diffie-Hellman Key Exchange, so the key type should be standardized.

3. Decryption keys for data files may also use AES but may potentially be fine with One Time Pad encryption, which could be desirable in certain cases. The reference information for encrypted data files should specify the exact encryption methods used. Always follow exact specifications of encryption standards (OTP, AES, or whatever else).