TECHNISCHE UNIVERSITÄT DARMSTADT

Implementing the algorithm proposed in "Recovering from key compromise in decentralised access control systems"

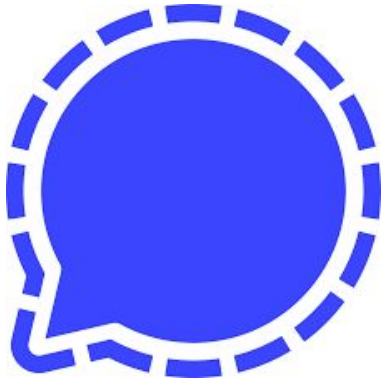
DARE 2024 - Second ACM Europe Summer School on Distributed and Replicated Environments

Dinis Sousa

Gustavo Costa

Decentralized systems impose challenges such as lack of a central authority and existence of concurrent operations

Decentralized message application systems, like **Whatsapp** and **Signal**, impose difficulties in managing users membership.



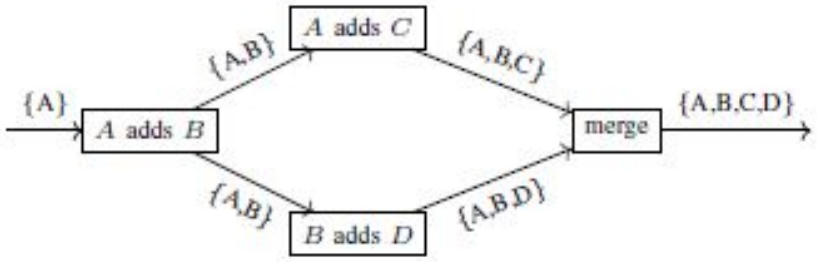
No Central Authority - Decentralized systems operate without a single trusted entity to enforce rules. This means members have to agree to whom the members of a messaging group are.

Exploitation of Stolen Keys - Adversaries using compromised keys can disrupt group membership, by adding other compromised members or by removing legitimate ones.

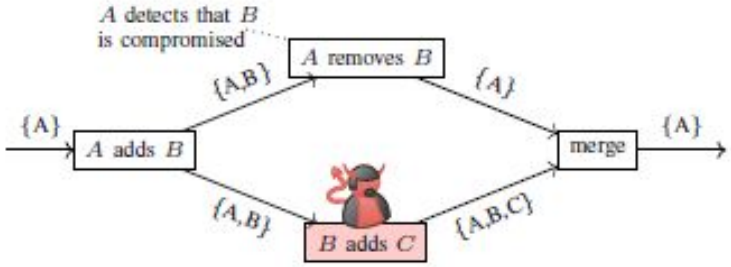
Conflict and Inconsistency - Concurrent conflicting operations, such as mutual removals, lead to divergent views across devices. It is important to ensure that, given the same history, members of the group agree on who is actually a member and has access to the chat.

Understanding the systems requirements: how to handle mutual addition and removal operations

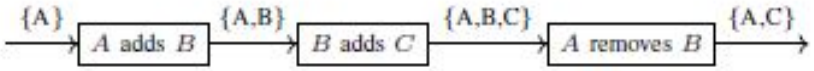
1) In concurrent “adds” both operations are accepted



2) While A is removing B, B tried to add C. The systems detects this and cancels the operation



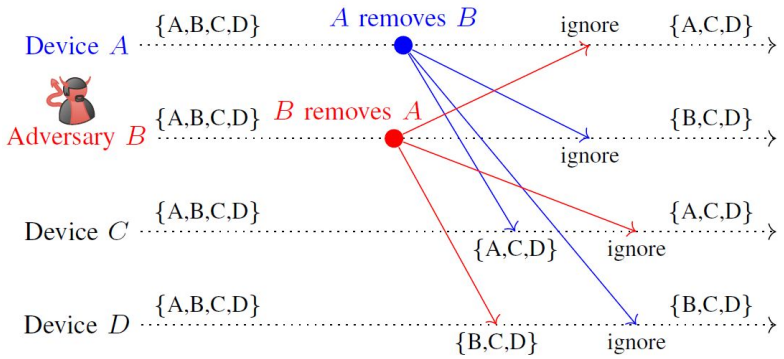
3) The same operations as in **2)**, but now in sequential order. C is now added to the group, has it was added by a valid group member at the time of the operation



Images taken from: “Recovering from key compromise in decentralised access control systems”

The existing approaches lack to fulfill all the requirements of our system

In a situation as seen below, were two members try to remove each other from the group, several strategies can be made



Remove both

Remove neither

Arbitrate by timestamp

External arbiter

Voting

Smart contract

Images taken from: “Recovering from key compromise in decentralised access control systems”

Implementing a seniority-based conflict resolution, given a DAG of signed and hashed operations

The solution presented bases on the following principles:

Seniority-based conflict resolutions - When in the presence of conflicting operations, take the most senior member side. The group creator is, if still in the group, the most senior member. Seniority is based on the first addition to the group.

Cryptographically signed and hashed operations - A DAG containing all the operations, signed and hashed. Similar to a tree of git commits, were each operation's parent is the one that logically preceded it.

DAG-based representation of membership operations (Authority Graph) - A graph that represents the dependencies between the operations. "There is an edge from op1 to op2 in this graph if operation op1 may affect whether op2 is authorised".

CRDT like convergence guarantees - The algorithm for conflict resolution is causally consistent, meaning the group membership (as seen by each individual member) will converge.

Conflict Scenarios and Resolution

The proposed algorithm aims to solve the following conflict scenarios:

- Conflicting remove operations
  - A removes B and B removes A
  - A removes B and B removes/adds C

The conflicting scenarios can be simplified to: there's a conflict if an acting user is concurrently removed.

To solve these issues, the seniority of the user's devices is taken into consideration, with the most senior device taking precedence. **However, this means that the utmost senior device (the group's creator) can, technically, avoid being removed, by issuing a concurrent remove operation for the device that tried to remove them.**

Security and Robustness

As previously discussed, the presented approach has both security and robustness guarantees.

Security guarantees:
- Adversarial actions cannot override legitimate operations.
- Consistency is maintained even under adversarial conditions.

Robustness guarantees:
- Works in offline or partitioned network scenarios.
- Resilient to concurrent operations and timing issues.

Implementation Overview

Our implementation was based off of Martin's lecture available code.

On top of development, we decided to restructure the initial code, and ended up with the following structure:

acl_operations.py : Defines the operations for group management:
- *create_op* : Creates a new group.
- *add_op* : Adds a new member to the group.
- *remove_op* : Removes an existing member from the group. All operations are signed and include dependencies to ensure causal consistency.

acl_test_operations.py : Contains unit tests to validate the access control implementation. It:
- Tests scenarios like adding/removing members and checking operation precedence.
- Includes methods to simulate and validate the system's behavior under various conditions, such as conflicting operations.

acl_validation.py : Implements algorithms to validate and manage operations:
- *compute_seniority* : Assigns seniority based on the operation graph.
- *authority_graph* : Builds a graph to track the influence of operations.
- *compute_validity* : Determines valid operations based on graph analysis.

acl_helpers.py : Provides utility functions:
- Cryptographic hashing (*hex_hash*) and message signing/verification (*sign_msg* , *verify_msg*).
- Helpers for transitive closure in operation graphs.

Contributions

The original paper presents 3 algorithms. The first and second algorithms were implemented by Gustavo and Dinis, respectively. Since the third algorithm combines both the first and second algorithms, it was implemented cooperatively.

The deliverables such as the report and the presentation were split among both of us by sections.

Effectively, it was a 50/50 split throughout the entire project.

Conclusion

The development of the project gave us **insights on the inner workings of decentralized systems**, more specifically in:

- How to address key compromises in such systems;
- How to ensure reliability through seniority-based conflict resolution;
- How such systems are suitable for real-world applications with high security demands;

Future work was identified, for later exploration:

- Enhancements like time-based key invalidation;
- Hierarchical or multi-level permissions (Role-based access control);

We found the overall DARE experience very **entertaining** and **informative**, allowing us to better understand the complex world of decentralized systems.

Implementation found at https://github.com/DinisSousa2003/DARE-2024

THANK YOU!

Q&A