NAME: A Research

Paper #XX

Abstract

Abstract stuff.

1 Introduction

Many people value and want privacy []. But achieving privacy has proven to be difficult and impractical. So much so that even billion-dollar companies get it wrong (insert data breaches here). This is largely due to the three-tiered architecture used to build applications today.

In traditional three-tier architected applications, a centralized server is fully trusted to implement necessary functionality (application logic, storing ground truth data, performing access control, etc). In other words, theserver must directly operate on user data. Consequently, user data is vulnerable to any entity that gains access to the server (curious administrator, hacker, government). Various solutions for hardening this architecture must rely on homomorphic encryption [5] or . . . Thus developers of applications that want privacy end up needing to be experts in cryptography, practically limiting the number of applications that can successfully uphold user privacy.

An increasingly common architecture for building applications primarily relies on the *client* to implement necessary functionality. Firebase [7], for instance, is an application framework that largely gets rid of the middle-tier and reduces server functionality to operate on a subset of data operations, pushing (e.g.) application logic to the client. A myriad of applications ranging from payment (Venmo) to games (Halfbrick) are built on top of Firebase, suggesting that a large class of applications can be built in this way.

Other work has taken this new architecture a few steps further, pushing even more functionality to the client such that server trust can be minimized. SPORC [1], a framework for OT applications, relies on the server for ordering and routing messages, storing access control groups, correctly updating access control groups, and storing encrypted data changes for offline clients. Thus, SPORC may be generalizable to the class of OT applications but still entrusts the server with user and group metadata. Signal [9], an end-to-end encrypted messaging application, relies on the server for some application logic, ordering and routing messages, storing encrypted access control groups, storing user-to-device groups, and authentication. However, Signal bakes in many application-specific assumptions or guarantees (weaker consistency and relatively simple application invariants like "append this message") into their protocol, making it very difficult for other applications to profit from their technique (it took WhatsApp–*another messaging application*–two years to adopt the Signal protocol [2]).

Support for applications in this new architecture's landscape is either general but not private or private but not general. We design and implement a framework, called NAME, that closes this gap and enables a large class of applications to get end-to-end privacy for free. NAME unifies existing end-to-end encryption protocols [9] with other common application functionality such that developers using NAME need only implement application-specific logic. Notably, NAME does not change the high-level semantics of any supported applications such that developers do not need to change how they think about their applications.

This paper makes the following contributions:

- NAME design
- NAME implementation
- NAME evaluation

2 Related Work

Private messaging. Signal [9] establishes a privacy protocol for instant messaging, in which messages between users are end-to-end encrypted with additional session keys to achieve forward secrecy in the presence of compromised keys. Group information is stored encrypted on the server, but the server is not entirely oblivious to all group information as it "special". The server also knows which devices belong to which users (a kind of group). Signal leverages other specific properties about messaging in their highly-specific protocol, such as weaker consistency needs and simple application invariants (messages should be appended).

Whatsapp [10]. Matrix [8]. Stadium [11].

Untrusted servers. SPORC [1] presents a similar application model that primarily leverages the server to establish a global order of events and remaining functionality is handled on the client. However, the SPORC server has a predefined notion of groups and any group modifications are visible to the server, whereas the NAME server is completely oblivious to any notion of groups and treats them just like all other data. SPORC also focuses on "operational transform" applications in which operations are meaningful regardless of the order they are applied in. This property is not the case for most applications (and most applications that NAME supports), where some operations can be invalidated by others. Furthermore, SPORC does not generalize their application model, although they hint that it could be done, nor does it examine how application invariants could be maintained in the face of malicious or buggy clients.

SUNDR [3]. Depot [4]. Mylar [6]. CryptDB [5].

3 Threat Model

We assume a threat model with ~~an honest but curious server and~~ potentially malicious ~~servers,~~ clients and network attackers. ~~An honest but curious server preserves integrity by working as expected but is able to compromise privacy by inspecting user data it processes.~~ Users trust the integrity of ~~servers and~~ the client-side library in the application model, as well as application code provided by developers.

NAME servers are responsible for routing encrypted messages across clients. A malicious server can compromise cross-client communication but does not expose private user data. A malicious server can drop messages to all clients, drop messages to a subset of clients, or reorder messages for specific destination clients. NAME also assumes that a network attacker may attempt to read or alter messages, but does not perform timing analysis on traffic patterns. Byzantine clients may attempt to write unauthorized or incorrect data to shared data.

4 Design

NAME enables developers to easily build secure applications. It does so by exposing a simple interface that handles interactions with the application data store and with other clients. All application data is stored in on-device local storage, and all cross-client communication is encrypted. Thus, no private information is exposed or persisted on a central server or database, ensuring user privacy. NAME is aimed to provide a similar, if not easier, application development process to tools such as Firebase [firebase] or ReactDB [], where much of developer onus with regards to implementing common application functionality in the new architecture is alleviated.

The two main components of NAME are the *client library* and the *NAME server*. All NAME applications utilize the library's exposed functions to enable off-device communication and consistency for any shared data. The server facilitates communication across clients while enforcing an order on messages, and is application-agnostic, meaning it does not need to be modified by application developers and can service multiple applications simultaneously.

4.1 Server

The server is responsible for forwarding messages and imposing a relative order on all routed messages. The server forwards messages to each client with an esablished connection to the server. Each client, regardless of the application or user, is dedicated a *mailbox* on the server through which all messages destined for the client are routed through. The mailbox's address is a unique idenitifier and in our case, each client's public key. Each message delivered to the server is a batch of encrypted messages, each destined to a different client. In addition, the server must uphold a realtive order of messages between any two clients. More specifically, for any two clients A and B, all messages exchanged between clients A and B must be totally ordered. To achievee this, the server must atomically add all messages to their destined mailboxes.

small wordy proof about how serializable transactions guarantee relative (if not total) order

4.2 Client-side Library

Most application functionality in the proposed two-tier architecture is performed on clients, including storing data, facilitating interactions with different users, and managing permissions. The NAME client-side library exposes a simple API that ensures these functionalities are implemented correctly and consistently across clients and applications. The library facilitates interactions with the local data store (including permissions and maintaining data invariants), sharing across devices and users, and encryption.

Developer API. do this when specifics are finalized

- New User
- New device under user
- establishing connection between to users / defining group
- new shared data object
- imposing data invariants

- logical invariant
- conflict resolution
- group and permissions

Sharing Protocol. Trust is established between any two clients by exchanging public keys. talk about discovery, establishing relationship and exchanging user group information

When a change to any data object is initiated on a client, the client references the *group* field of the data and gets a list of all clients this data is shared with, each with their own copies. Groups are discussed more thoroughly later on in this section. The client encrypts the message describing the change multiple times, once for each destination client and once for itself. The client then sends this batch of messages to the server. The server forwards each encrypted message to the appropriate destination clients.

A client recieves messages through its mailbox on the server. Each message describes a change to be performed to a local shared object. A client can choose to accept or reject this change based on the conflict resolution scheme of the object. Whether a change is accepted and committed is deterministic on all clients, as changes are only committed after they've been read from the server, even on the client that initiated the data change. Since the server guarantees an ordering of messages across releavant clients, all clients will reach the same decision on whether to apply a given change, so that data is consistent across client copies.

It is entirely possible that a client will initiate a change, send the encrypted messages to the server, then recieve a message that invalidates the original change. The client will only commit the change once it reads it back from its mailbox. If the change is aborted, a client is able to re-initiate it. Conflict resolution takes multiple forms and NAME clients can enforce multiple policies such as first-writer-wins and last-writer-wins easily, since a change can be easily compared to a previous committed version. This model easily extends to commutative data structures, where changes would never be rejected and always committ in the same order on all devices.

Encryption. As described above, all client communication is end-to-end encrypted. NAME implements double-ratchet encryption []. elaborate

Groups. All shared data objects are associated with a *group* defining access permissions for the specific data object. A group is a list of pointers to any combination of other groups or clients. By default, all NAME data objects are assigned to a *self* group. Any group can be recursively enumerated to a list of destination clients by the address of their mailbox. Instinctively, a group is the set of clients that own a local copy of some data. Each client locally stores all group information for groups it is in, and thus stores all potential groups associated with its local data.

A group is a node in a directed graph with pointers to any other groups or to client identifiers. A node representing a client identifier only points to itself. When group information is needed, the graph is traversed such that each group is recursively enumerated to produce a list of client identifiers.

The group model lends itself well to commonly used sharing schemes. For example, adding multiple clients to the *self* group enables users to use the application across multiple devices. Likewise, any application with a notion of friends will have a group per friendship. If used correctly, a friendship group contains two user groups, each containing each user's list of devices using their client identifiers.

Groups are also responsible for defining permissions. Each group member has a set of attached permissions within the group. For example, a group used to share class material may give a teachers group write permissions and student groups read permissions. These are defined within the group structure. describe notations for how this is done

All group information is stored locally using a client's data store and is continuously modified as more objects are shared across users and devices. Changes to the group field of an object are tracked the same way as changes to any other data, and propagated similarly to relevant clients. Groups are treated as NAME native data objects, with the group itself as the permission field. For example, a client with read permissions in a group is unable to modify the permissions.

Permissions. Permissions for data modification are stored locally. Both readers and writers access local copies of data.

If a client is malicious and attempts to write and dispatch a change to other clients, those clients will cross-check client permissions before applying a change locally and choose not to commit the change.

Data Invariants. The NAME client-side library facilitates application interactions with a local storage mechanism, e.g. localStorage in browsers, sqlite on smartphones. Applications read and write to storage as usual, but or any shared data, the library will track the change and propagate them to other clients. Likewise, any remote changes recieved from other clients will be applied automatically by the library.

Fascilitating interactions with the underlying data storage mechanism allows NAME applications to check data invariants before applying changes. what is the notation for this

4.3 Security

Byzantine Clients. Detecting malicious interference by clients is handled on the data invariant level.

Reordering Server. Detecting a malicious server, *i.e.* a server that reorders or drops message, is handled on a lower level than the object tracking level. describe scheme once we finalize it, small wordy proof

5 Implementation

- lines of code
- languages + library and features
- data storage -> localstorage

5.1 Client-Side Library

notes about how this is written, lines of code, etc...

5.2 Server

notes about how this is built, how we ensure serializability, lines of code

6 Evaluation

6.1 Performance

- Server
  - explain dummy app, knobs we tune
  - end-to-end w/little conflict
  - conflict -> how bad is it? are txn overheads for conflicting groups horrible? where does bottleneck come from?
- Client
  - storage overheads -> storing X amount of data w/tracking metadata requirex X*5 data on device
  - latency for change to be committed w/no conflict

6.2 Expressiveness

Four main criteria we need to test for applications:

1. user-generated data
2. small enough data
3. reasonable sharing circles
4. n

Health Tracker.
Social Media.
IoT.
Text Messaging.

7 Conclusion

Conclusion stuff.

Acknowledgements

ACKS.

Availability

Available stuff.

References

[1] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association.

[2] Andy Greenberg. Hacker Lexicon: What Is the Signal Encryption Protocol? https://www.wired.com/story/signal-encryption-protocol-hacker-lexicon/, 2020. [Online; accessed September-2022].

[3] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004. USENIX Association.

[4] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association.

[5] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 85–100, New York, NY, USA, 2011. Association for Computing Machinery.

[6] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. Building web applications on top of encrypted data using mylar. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 157–172, Seattle, WA, April 2014. USENIX Association.

[7] Firebase Team. Firebase. https://firebase.google.com/, 2022. [Online; accessed September-2022].

[8] Matrix Team. Matrix. https://matrix.org/, 2022. [Online; accessed September-2022].

[9] Signal Team. Signal. https://signal.org/, 2022. [Online; accessed September-2022].

[10] WhatsApp Team. WhatsApp. https://www.whatsapp.com/, 2022. [Online; accessed September-2022].

[11] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 423–440, New York, NY, USA, 2017. Association for Computing Machinery.