Stick: an End-to-End Encryption Protocol Tailored for Social Network Platforms

Omar Basem, Abrar Ullah, and Hani Ragab

Abstract—End-to-End Encryption (E2EE) has become a de facto standard in messengers, especially after the outbreak of the highly secure messaging protocol – Signal. However, this high adoption of secure end-to-end communications has been limited to messengers, and has not yet seen a noticeable trace in social network platforms. This work aims to design, verify, implement & evaluate an E2EE protocol tailored for social network platforms, based on the Signal protocol. The protocol should be able to re-establish encryption sessions in an asynchronous and multi-device setting while preserving the Signal protocol's security features and potentially improving on them. The proposed protocol is one of the first of its kind in the sense that it supports re-establishable encryption sessions while preserving forward secrecy and introducing backward secrecy. The development of this protocol was faced by some unconventional problems, which it was able to overcome through some innovative methodologies, such as: sticky sessions, multiple pairwise sessions, session lifecycle, refreshing identity keys and double hashing. The proposed protocol was successfully verified using Verifpal - a formal verification tool in the symbolic model. Our security analysis has shown that the Stick protocol is able to achieve a form of post-compromise security in many-to-many communications, the trait which most group protocols lack. Most importantly, the Stick protocol is able to authentically and confidentially re-establish encryption sessions. All of our design and verification has been successfully transformed into real world open-source implementation. Our evaluation has shown the Stick protocol can be used in real world social network app without having a noticeable compromise on performance or usability.

Index Terms—End-to-End Encryption, Security Protocol, Formal Verification, Social Networks

1 Introduction

We live in a more connected age than ever before. Such technological shift has revolutionized how we communicate, but it has also left our privacy more exposed to an increasing number of threats. These threats range from data mining and data selling to phishing attempts and identity theft. They also stem from different parties - including hackers, Internet providers, and even the application service providers. Recently, there have been multiple data leaks linked with some big names in the social networking field, like Facebook-Cambridge Analytica data scandal in 2018, in which the personal data of over 80 million Facebook users' were leaked [1]. Such incidents have led to an increased demand for secure communications. So, why E2EE is not utilized in social network platforms?

E2EE is being used for several applications from securing networking channels using SSL/TLS to private communications in messengers to signing digital certificates, but all of these applications can be summarized under one main use case: Establishing an end-to-end (e2e) encrypted short-term session between two parties for authenticity verification and/or exchanging data. A short-term session means that it does not require long-term persistence nor the ability to be re-established. For all of these kinds of applications establishing a new encryption session whenever needed is not a problem. In addition, the focus is always on communications between two parties only. This has limited the development of E2EE protocols towards this ideology of short-term sessions between two parties. Using these short-term sessions in a social network to provide E2EE would not work. In case of modern e2e encrypted messengers, when Alice wants to start chatting with Bob, she would create an e2e encrypted session with Bob. If Alice is going to use another phone or will reinstall the messenger application, she can create a new session with Bob. This will result in both of them being unable to decrypt any previously sent message — in practice these messages are usually deleted from the server anyway once received, so there is no way to redecrypt messages. However, this behaviour in messengers is acceptable. Indeed, it is the desired behaviour as normally the recipient would not need to decrypt your message more than once. On social networks, this behaviour would be problematic. If Alice shared a photo on a social network, and then a month later she wanted to reinstall the application, she would expect every photo she has shared or was shared to her to still be there and be able to decrypt it again — and not be gone. Also, Alice should be able to view these photos from any other phone using her account. As a result, using E2EE in a social network was never practically feasible. Building on that, we regard our contributions to be 4 fold:

- We attempt to solve the above problem by designing an E2EE protocol tailored for social network platforms, based on the Signal protocol. The protocol should be able to re-establish E2EE sessions, while preserving forward secrecy and introducing backward secrecy.
- We formally verify the proposed protocol in the symbolic model using Verifpal to prove that it is able to achieve its security properties.
- We implement the proposed protocol as iOS & Android

• *The authors are with the department of Mathematical and Computer Sciences, Heriot-Watt University, United Kingdom.*
Email: founder@stiiick.com, {A.Ullah, H.RagabHassen}@hw.ac.uk

libraries, in addition to a server library and a client-handlers library, all open-source available on Github.
- We practically evaluate the proposed protocol to provide validation supporting the fact that using the Stick protocol in a social network application is not going to have a compromise on performance or usability.

The rest of the paper is organized as follows. Section 2 provides background information about the Signal protocol, in addition to the related work. In section 3 we propose our design of the Stick protocol. We formally verify the protocol in section 4. Section 5 gives a brief overview of the Stick protocol's implementation. We evaluate the protocol's performance in section 6. In section 7, we identify the protcol's limitations and future work. Lastly, we conclude our work in section 8.

2 Background

This Signal protocol being a faily complex protocol, we aim through this section to give a brief dissection of it. The Signal Protocol is arguably one of the most secure E2EE protocols, was first introduced in the messaging app TextSecure, which later became known as Signal. Over 2 billion people use the Signal protocol everyday, as it is the protocol used to e2e encrypt WhatsApp's communications [2]. The Signal protocol glamorousness among the encryption protocols can be accredited to the following 2 assets that it posses:

- The X3DH (Extended Triple Diffie-Hellman) Key Agreement Protocol [3].
- The Double Ratchet Algorithm [4].

The following two subsections details a little more about both of them.

2.1 The X3DH Key Agreement Protocol

Overview. The X3DH protocol creates a shared secret key between two parties wishing to establish a secure communication session, through 4 DH calculations. X3DH is designed to be used in asynchronous environments where Alice uses pre uploaded information by Bob to create a shared secret.
Roles. There are 3 parties involved in the X3DH protocol: Alice, Bob and a server.

- Alice wants to start a conversation with Bob, and create a shared secret key which both of them can use to exchange messages.
- Bob wants to let other parties like Alice to start a conversation with him and create a shared secret key, even if Bob was offline.
- The server can keep messages that Alice has sent to Bob, until Bob gets online. Also, the server can pre-store data from Bob which can be provided to parties like Alice when needed.

Keys. Each party has 3 kinds of keys:

- Identity key (IK): a long-term identifying key.
- Signed prekey (SPK): used for signing, changes periodically.
- One-time prekey (OPK): every party would have a set of one-time prekeys.

Phases. The X3DH protocol goes through 3 phases. (i) Uploading Keys: Bob publishes his set of public keys to the server which consists of an identity key, signed prekey (with signature) and a list of one-time prekeys. (ii) Sending the first message: Alice fetches from the server a prekey bundle (PKB) of Bob, which consists of Bob's IK, SPK, and one of Bob's OPK, and verifies the signature. Alice will generate an ephemeral key *EK* and carry out 4 DH calculations as shown in figure 1, then generates the secret key *SK* where SK = KDF(DH1 ‖ DH2 ‖ DH3 ‖ DH4). Mutual authentication is provided by DH1 and DH2, while DH3 and DH4 assures forward secrecy. DH4 can be omitted if Bob has ran out of his pre-uploaded one-time prekeys and has not refreshed his store yet. The initiating ciphertext is encrypted with an AEAD (Authenticated-encryption with Associated-data) scheme where the encryption key is one of SK or the result of a cryptographic PRF (Pseudo Random Function) having SK as the input. (iii) Receiving the first message: Bob extract's from the initial message Alice's IK, EK and identifiers referring to which of his keys Alice has used to create *SK*. Bob carries out DH and KDF calculations like Alice to calculate SK, then decrypt the ciphertext. X3DH is now complete for Alice and Bob. They may keep using *SK* or keys derived from SK for subsequent communications within some post-X3DH protocol.

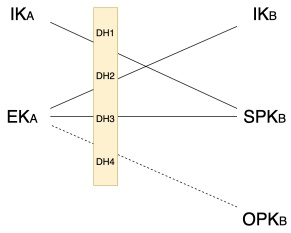

Fig. 1. X3DH Key Agreement

2.2 The Double Ratchet Algorithm

Going into details of the double ratchet algorithm is beyond the scope of this work, but here is how is works in short. After two parties have used the X3DH protocol to agree on a shared secret *SK*, they can use the Double Ratchet Algorithm as a post-X3DH protocol for subsequent encrypted messages. For every Double Ratchet message, the parties will derive new keys from a symmetric-key ratchet preventing the calculation of earlier keys from subsequent ones (forward secrecy). In addition, the symmetric ratchet is combined with a DH ratchet, which is used to derive DH outputs. These DH outputs are combined with the keys derived from the symmetic ratchet preventing the calculation of subsequent keys from earlier ones (backward secrecy).

2.3 Signal Protocol Group Messaging

Group messages in the Signal protocol build on the pairwise encrypted sessions explained above and uses server-side fan-out to send the message to all members of a group. This is accomplished using "Sender Keys". The first time Alice wants to send a message to a group:

1) Alice generates a **Chain Key** — a random 32-byte key.
2) Alice generates a **Signature Key** — a random Curve25519 key pair.

3) Alice combines the **Chain Key** and the public key of the **Signature Key** to make a **Sender Key**.
4) Alice individually encrypts the **Sender Key** to every group member using Signal's pairwise sessions.

Then, for subsequent messages Alice sends to the group:

1) Alice derives a **Message Key** from the **Chain Key**, and updates the **Chain Key** - 1 Symmetric-key ratchet step.
2) Alice encrypts the message using **AES256-CBC**.
3) Alice signs the ciphertext using her **Sign. Key**.
4) Alice sends the encrypted message to server which does server-side fan-out.

You may have noticed that there is no "Double Ratchet" here, but only a single ratchet. This provides forward-secrecy without backward secrecy. This is true for almost all current group messaging protocols at the moment.

2.4 Related Work

Until now in 2021, to the best of our knowledge, we do not yet have an e2e encrypted social network platform. As a result the amount of research that has been done on "E2EE for social network platforms" in particular is scarce. Most of the research that has been done regarding privacy in social networks revolves around messengers or leveraging the users' privacy by means other than E2EE. This subsection presents some of the related work in that sense.

Blum et. al [5] presents a design for how Zoom's communications are e2e encrypted. The most significant contribution in their proposal is what they call "transparency tree". It is a methodolgy that forces Zoom servers to sign and immutably store mappings between a user and their public keys, and provide these mapping to all clients for signing, which builds a "transpareny tree" similar to those used by CA (Certificate Authorities). This methodology helps protect against identity spoofing. However, Zoom relies on an elementary secret key establishment mechanism that involves a single DH calculation. This makes it easier for an attacker to break the secret key, in comparison to X3DH, where there are more keys involved to break or compromise.

WhatsApp [2] followed the footsteps of Signal and integrated the Signal protocol into its messaging communications, making it the largest consumer of the Signal protocol with over 2 billion monthly active users. Nonetheless, WhatsApp is a product of Facebook, so should we trust WhatsApp? WhatsApp harvests and stores meta data about its users and their activity which is not e2e encrypted. Facebook can create a more accurate profile of your WhatsApp account through combining your WhatsApp metadata with your Facebook data, and then proceeding to serve you their targeted ads with a side of privacy violation. Also, Facebook's model is based on Big Data, user profiling and analytics. Running a huge and costly service like WhatsApp and not exploiting it to collect some data from its users, plus having already paid 19 billion to acquire it, can be hard to believe. Technically speaking, WhatsApp implementation of the Signal protocol for Group messaging suffers from having no backward secrecy. The group keys changes only if the group membership changes, otherwise, the same keys will be used till no end.

Cohn-Gordon et al. [6] have an interesting paper where they are addressing the above drawback of the Signal protocol (and almost every other messaging protocol) in group messaging, that is having no backward secrecy. They proposed a protocol design that provides both forward-secrecy and backward-secrecy in group messaging using Asynchronous Ratcheting Trees (ART) which uses tree-based Diffie-Hellman key exchange to let members of a group create a shared secret key without needing to be online at the same time. However, their design would work only for a messenger application, and not for a social network application that would require re-establishale sessions.

Barenghi et al. [7] proposal is perhaps the most related academic paper to our work. They are proposing an e2e encrypted social network as a single page HTML5 JavaScript application. Their work has some crucial security flaws. Their approach is protecting every account by a single master key which create a major threat to the users. If an adversary broke the master key of an account, they would be able to get all of the user's other keys and data. In addition, the master key is derived from the user's password using an outdated key derivation function PBKDF2, and with 1000 iterations only which is way below the recommended minimum number of iterations (10,000) by NIST [8]. Their secret key establishment relies on a question/answer pair. This would definitely make the encrypiton key weaker, as it is not based on a random function, and on top of that is a bad user experience. Group messaging uses one symmetric key, which again would be easier to break, plus tying all of the group's data to a single key is certainly not a very secure solution. Lastly, their proposal offers no advanced security features, such as: forward secrecy or backward secrecy.

Several papers [9], [10], [11] tried to tackle the privacy and confidentiality of user data in social networks by using Attribute-Based Encryption (ABE) allowing the users to apply customizable policies over who may view their data. In contrast to Identity-Based Encryption (IBE) where the keys of a user linked to their identity, in ABE the user's keys are linked to a set of attributes that the user have. In ABE systems, a ciphertext can be decrypted only if its attributes matches a user's key attributes. However, ABE suffers from 2 main disadvantages. Firstly, it is much slower than IBE because it needs to create a policy tree. Secondly, by default, it has no attribute revocation mechanism. Key revocation in cryptography is a non-trivial problem, and with ABE it is even more challenging, as each attribute can belong to multiple users, while in IBE a key belongs to one user.

Another approach for protecting the user's privacy on social networks is through decentralization. Multiple papers [12], [13], [14] have proposed decentralized architectures as a mean of keeping the users' data private. A decentralized system have no control over the day-to-day activities happening on the system making it difficult to achieve global big tasks. Also, it is not easy to find malicious and failing nodes. Moreover, In a decentralized system you have no control over the performance of the system, unlike in a centralized system where you can boost your system computing power on-demand at times of high traffic. In addition, [13], [14] do not employ any cryptographic methods to protect the user's data claiming security through the isolation of data. Isolation does not guarantee security nor privacy. It may protect the users from the service providers, but in case an attacker got access to the data it will be

available for them in plaintext.

TABLE 1
Social Networks Security Measures Comparison

App	2FA	TLS	Signed URLs	Data Center Enc.	Priv. Msg. E2EE	Platform E2EE
Facebook	Yes	Yes (1.3)	No	Partial	Yes (Opt-In)	No
Twitter	Yes	Yes (1.2)	No	Partial	No	No
LinkedIn	Yes	Yes (1.2)	No	Partial	No	No
Instagram	Yes	Yes (1.3)	Yes	Partial	No	No

The table below shows security measures taken by some of the popular social networks to protect the users' data. All of them supports 2FA, uses TLS and have partial data center encryption. Only Instagram makes use of signed URLs for private content. Facebook is the only one that uses E2EE for private messages as an opt-in. None of these applications uses E2EE for the platform content. Also, we can see that none of the above proposals were actually an "E2EE protocol for social networks". Building on that, this paper is proposing an E2EE protocol - the Stick protocol - that is specifically designed for social network platforms.

3 Stick Protocol Design

We present our design of the protocol through a use case description. This use case is considering a social network *SN* that meets the criteria detailed below. So, we revisit again Alice and Bob. On the social network *SN*, Alice has a **userId**, **partyId**, **phoneNumber** and a **password**. Also, Alice would be connected to a number of connections (friends), a number of groups, and her profile. When sharing a post Alice can choose to share to one of the following **parties**:

- A particular group
- A mix of certain group(s) and/or connection(s)
- Her profile (which includes all of Alice's connections)

Alice is connected with N users: [User0, ..., UserN]. Also, Alice is a member of 3 groups - A, B & C. Alice wants to share photo A with Group A, photo B with both of Groups A & B, and photo C with everyone she is connected with (her profile). Alice will share these photos from her phone *X*. Now, there are 4 main requirements that Alice needs:

1) Alice wants every photo to be e2e encrypted to the designated party.
2) Alice wants to be able to view the photos she has shared from her phone *X* on her other phone *Y*.
3) If Alice reinstalls the *SN* application, she wants to still be able to view any photos she has previously shared or was shared to her, i.e., not lose any data.
4) Alice wants to still benefit from the security features of the Signal protocol.

In addition, Alice would be interested in any extra security features on top of what is provided by the Signal protocol.

3.1 Preliminaries

As discussed in section 1, using common messaging protocols, such as the Signal protocol for a social network would be problematic. If Alice shared a photo on SN, and then a month later she wanted to reinstall the application, she would expect every photo she has shared or was shared to her to still be there and be able to decrypt it again — and not be gone. Also, Alice should be able to view these photos from any other phone using her account. The Stick protocol solves this problem by using "sticky sessions" (not referring to sticky sessions of load balancers, more on this in a bit) while preserving the security features of the Signal protocol. Moreover, the Stick protocol provides extra security advantages regarding "many-to-many" encryption.

3.1.1 Terms

User-Specific Key Types:

- **Identity Keys**: a key pair of type Curve25519 generated at registration time, refreshed every while.
- **Signed Pre Keys**: a signed key pair of type Curve25519 generated at registration time, refreshed every while.
- **One-Time Pre Keys**: a list of keys of type Curve25519 generated at registration time. Refilled as needed.

Session Key Types:

- **Encrypting Sender Key (ESK)**: consists of a 32-byte **Chain Key** and a Curve25519 Signature Key Pair. Acts as the root key of a sender's symmetric-key ratchet.
- **Decrypting Sender Key (DSK)**: consists of a 32-byte **Chain Key** and the public key of a Curve25519 Signature Key Pair. Acts as the root key of a receiver's symmetric-key ratchet.
- **Chain Key**: a 32-byte key used to derive **Message Keys**.
- **Message Key**: an 80-byte key which encrypts messages (posts). It consists of 32 bytes that are used for an AES-256 key, 32 bytes for a HMAC-SHA256 key and 16 bytes for padding.

Other Key Types:

- **Blob Key**: consists of a 32-byte AES256 key & a 32-byte HMAC-SHA256 key. Used to encrypt files.
- **Blob Secret**: (**Blob Key** $\|$ Hash of the encrypted file)

3.1.2 STATE RESET

Given an e2e encrypted application **X**, STATE RESET is in an event that occurs when a user reinstalls **X** wiping all the encryption sessions they had, or when installing **X** on another phone having to establish new encryption sessions, and being unable to decrypt the previously encrypted data.

3.1.3 Other Terms

- **Collection**: a mix of groups and/or connections.
- **Party**: a party can be one of 3 - a **group**, **self-profile** or a **collection**.
- **One *Encryption***: an encryption of a photo, video, comment, notification, status or any other data that needs to be e2e encrypted between a **user** and a **party**.

3.2 The Stick Protocol

3.2.1 User Registration

At registration time, Alice would generate an Identity key pair, a Signed prekey (with its signature) and a list of one-time prekeys. Alice would then transmit the public credentials of her keys to the server. The server would store those keys associated with Alice's identifiers, assign a **userId** and a **partyId** to Alice and return those ids to her.

3.2.2 Sticky Sessions Overview

A sticky session is an e2e encrypted session between a **user** and a **party** that can be re-established after **STATE RESET** in an asynchronous multi-device environment, and keeps track of the ratcheting **Message Keys** while preserving **forward secrecy** and introducing **backward secrecy**.

3.2.3 Re-establishing Sessions

As discussed in section 2.4 about group messaging, when Alice wants to share her sender key SK1 of a group G1 with Bob, she would establish a normal Signal pairwise encrypted session with Bob, then encrypt SK to Bob. Bob will not be able to decrypt that sender key after STATE RESET unless establishing the same encryption session with Alice which she used to encrypt that key.

In order to be able to re-establish the same session, the private keys of the identity keys, signed prekeys and one-time prekeys will be backed up encrypted with the help of **Argon2**, arguably the most secure hashing algorithm and the winner of the PHC (Password Hashing Competition) in 2015. Argon2 was designed to resist GPU cracking attacks as well as side-channel attacks. Argon2 summarizes the state of the art in the design of memory-hard functions and can be safely used to hash passwords for private credentials storage, key derivation, and other applications [15]. Alice or Bob can encrypt a private key for backup using the following procedure:

1) The user generates a securely random 32-byte **salt**.
2) The user generates a securely random 16-byte **IV**.
3) The user creates a **secret hash** of their password using **Argon2** with the **salt**.
4) The user uses the produced **secret hash** to encrypt the **private key** using AES256 in CBC mode.
5) The user pads the produced **cipher** using **IV**.

After that the user can securely backup a private key, and no one else including the server can decrypt their key. Similarly, a user can decrypt their private key by extracting IV, creating the secret hash again, then decrypting the cipher. Now, Bob will be able to re-establish the same session he had with Alice at any time in the future and decrypt SK1.

3.2.4 Multiple Sessions

But, there is still another problem. Signal pairwise communications have backward secrecy where no later key can be derived from a previous key, meaning when Bob re-establish a session he will be able to decrypt only the first message. So, if Alice encrypted to Bob another sender key SK2 of group G2 Bob will not be able to decrypt SK2 or any subsequent sender keys. To solve this problem, the Stick protocol allows two users to establish together multiple pairwise sessions. So, a Signal pairwise session will be used to encrypt one sender key only. The next sender key will be encrypted using a fresh pairwise session. This allows two users after STATE RESET to re-establish as many sticky sessions as they had before STATE RESET, while keeping the forward secrecy and backward secrecy of exchanging sender keys.

3.2.5 Session Life Cycle

In the Signal protocol group messaging, a user change their sender key when the group's membership changes, or when reinstalling the app. So, if the user never needed to reinstall the app and the group's membership did not change, they will keep using the same sender key till no end. This causes zero backward secrecy in group messaging. If an eavesdropper intercepted one message key of a sender, they will be able to find every message key of that sender in the future. The Stick protocol solves this problem by having a life cycle for its sticky sessions. A sticky session have a life cycle of N *Encryptions* (ex.: 100 *Encryptions*). After N *Encryptions*, the session will go from an "active" state to a "freeze" state. A sticky session in freeze state cannot be used do more *Encryptions*. However, it can still be used for decryptions as all the **message keys** were being saved into the session's internal state on the user's device.

Whenever Alice makes a new post from her phone X, she will make a few *Encryptions* (photo, caption, notification, ...). This will cause the chain of her **Encrypting Sender Key** SK-X to ratchet a few steps as it is generating new **Message Keys**. SK-X current chain step will be updated on the server. After that, when Alice wants to make another post from her other phone Y encrypted using SK-X, she will know from which step she should start (by contacting the server). When the chain of SK-X reaches N steps, that session will go into "freeze" state. Then, Alice will create a new sticky session, generate a new sender key, and encrypt her new **Decrypting Sender Key** to the target **party** members. The end result is that a user's Sender Key of a party will be automatically changing every N *Encryptions*. This introduces **backward secrecy** every a maximum of N *Encryptions*, and still keeps **forward secrecy** for every single *Encryption*.

3.2.6 Creating Sticky Sessions

When Alice wants to make an *Encryption* of a photo she will call a function *encryptFile()* providing 3 main parameters: (i) filePath (ii) userId (iii) stickId

$$\textbf{stickId} = (\text{targetPartyId} \parallel \text{activeChainId})$$

But, first Alice needs to know what is the stickId. To find the stickId, Alice will send a request to the server that includes the list of connections and groups she wishes to share a post with. The server will return the following: (i) The stickId. (ii) A list of *usersIds* for which Alice needs to share her sender key of her active sticky session for that party. For the server to find the stickId, it first needs to determine the *targetPartyId*. If the target party is Alice's profile, then it will simply be her own partyId, and if it is a group, then it will be the groupId. If the target party is a **collection** then, the server will check whether there is a party object associated with that collection. If so, the *targetPartyId* will be this party's id, otherwise, the server will create a new **party** object and associate it with that collection. After determining the partyId, the server needs to find the chainId. The server will fetch the last senderKey SK-L associated with Alice and that partyId. If its current chain step is below N, then the activeChainId will be SK-L's chainId, otherwise it will be SK-L's chainId plus one. Next, the server will loop over the target party members to find which of them do not have Alice's sender key for that particular stickId. Finally, the server will return to Alice the stickId and the list of users that she needs to encrypt her sender key to. Now, Alice can check whether she already has a sticky session on her device associated with that stickId

and create one if needed, encrypt her sender key to the list of users individually using Signal's pairwise sessions, and upload her encrypted sender keys to the server.

3.2.7 *Making an* Encryption

Now that Alice knows the right **stickId**, she can execute *encryptFile(filePath, userId, stickId)* which will do two things:

A. Encrypt the photo file and get a **Blob Secret**
B. Encrypt the **Blob Secret** like a normal message.

Alice will encrypt the photo file as follows:

1) Alice generates an ephemeral 32-byte **AES256** key.
2) Alice gen. an ephemeral 32-byte **HMAC-SHA256** key.
3) Alice gen. a 16-byte securely random **IV**.
4) Alice encrypts the photo file using the **AES256** key in CBC mode with the random **IV**.
5) Alice appends a MAC of the ciphertext using the **HMAC-SHA256** key.
6) Alice makes a **Blob Secret** as (AES256 key ‖ HMAC-SHA256 key ‖ hash of the encrypted file)

Next, Alice will encrypt the **Blob Secret** as follows:

1) Alice loads the sticky session associated with the **stickId** and her **userId**.
2) Alice calculates a new **Message Key** as:
   Message Key = HMAC-SHA256(Chain Key, 0x01)
3) Alice encrypts the **Blob Secret** with the **Message Key** using AES256 in CBC mode.
4) Alice signs the ciphertext using the private sign. key of her **Encrypting Sender Key** for that sticky session.
5) Alice adds the **Message Key** in the sticky session state.
6) Alice ratchets the Chain Key one step:
   Chain Key = HMAC-SHA256(Chain Key, 0x02)
7) Alice saves the new state of the sticky session and can start uploading the encrypted data.

3.2.8 *Refreshing Identity Keys*

A compromise of a user's identity key can have a devastating effect on the security of future communications. An attacker with a user's identity private key can impersonate the compromised user. Within the Signal protocol, if a user finds out that their identity private key has been compromised, they can replace their identity key by reinstalling the app. However, a compromised user may never find out that their identity private key has been leaked. To mitigate this, the Stick protocol refreshes the identity key every while. Similar to signed prekeys, a user can have multiple identity keys where only one is active. Every encrypted sender key will have an associated identity key id from the receiver. This allows the receiver to know which of their identity keys was used to encrypt that sender key.

3.2.9 *Passwords Security*

Double Hashing *(not referring to the collision resolving technique)***.** In the Stick protocol, the user's private keys are backed up encrypted using secret keys derived from password hashes. Also, the password is used as a 2FA. Although, typically a server does not store the user's password, but only hashes, the user should not be forced to trust the server. In addition, since the user's password is used in backing up their private keys, therefore, the user's password should never be sent to the server, as this can make it vulnerable to attacks by eavesdroppers or even the server itself. To avoid such attacks, the Stick protocol uses "double-hashing", where the password is hashed on the user's device before being sent to the server. The server will treat that hash as the plaintext password, and will hash it again. The server will store the resultant double-hashed password as the password hash. That way, the server can verify the user's password without it having to leave the user's device.

Storing Passwords. Every while, the user would need to refill their store of one-time prekeys whenever it goes below a certain threshold. While doing so, they would also need to backup the corresponding private keys encrypted using secret keys derived from the password. Obviously, the user cannot be asked to enter their password every time this process needs to happen. Therefore, the password must be stored somewhere on the user's device securely. How the password is stored depends on the underlying OS:

- **iOS**: the keychain services API is used to securely store small chunks of data (e.g.: passwords) persistently in an encrypted local database [16]. On iOS, if the user opts to backup their keychain, it is backed up e2e encrypted using a key derived from information unique to their device, combined with their device passcode, which only the user knows. No one else can access this data.
- **Android**: the equivalent of keychain on Android is the Block Store API. Similarly, it is used to store sensitive data locally, persistently and encrypted, and the user can opt to have its data backed up e2e encrypted. Worth noting that the Block Store API is yet to be released this year, 2021. An alternative for the Block Store API, is the Keystore API, but it lacks data persistence [17].

Storing the user's password persistently using Keychain API on iOS or Block Store API on Android would also help the user recover their password in case they forgot it.

3.2.10 *Decrypting After STATE RESET*

The following is the procedure that Alice will go through after STATE RESET to re-establish the sticky sessions, then decrypt a photo X:

1) A user wanting to login would enter his **phone number** and sends a request to the server.
2) The server verifies that this **phone number** is associated with a user, then sends to it a verification **code**.
3) User enters the verification **code** and sends a request to the server asking for verification.
4) The server verifies the **code**, then returns to the user the password's salt used in **Double Hashing** and **Limited Access Token (LAT)**, which the user can use to send a password verification request.
5) The user then creates the **Initial Password Hash (IPH)** and the returned salt, and sends a request to the server that includes the **LAT**.
6) The server verifies the **LAT**, then creates the password's **double hash** and verify it, then returns to the user his **IKs**, **SPKs**, **OPKs** and **ESKs** along with an **Auth Token** which the user can use for future requests to the server.
7) The user decrypts the private keys of his **IKs**, **SPKs** and **OPKs** as discussed in 3.2.3.
8) The user re-establishes the pairwise sessions.
9) The user decrypts his **Encrypting Sender Keys** and re-establishes his sticky sessions.

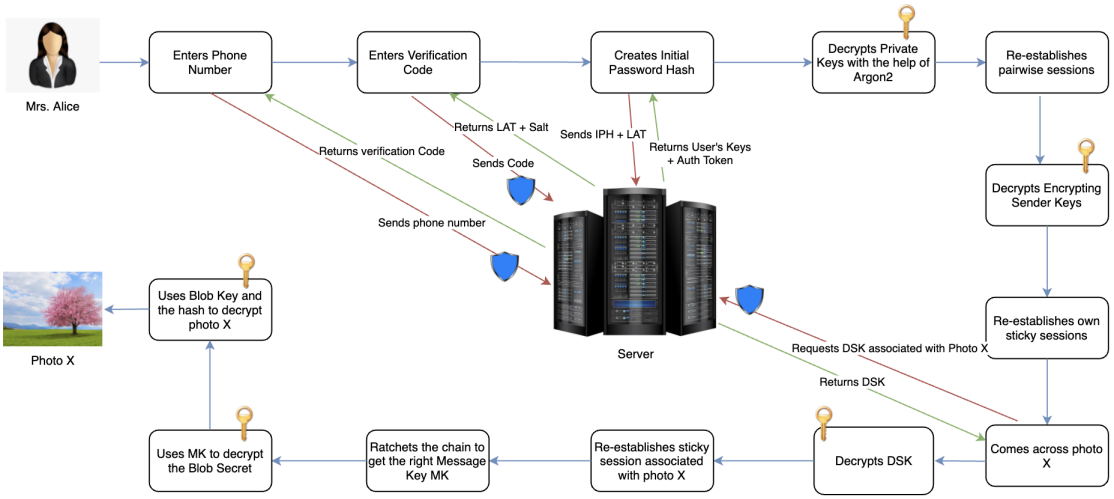

Fig. 2. A flow diagram showing showing decryption process after STATE RESET

10) The user comes across photo X that they have not re-established its sticky session yet, so they fetch **DSK** associated with photo X's **userId** and **stickId** from the server.
11) The user decrypts **DSK** using the corresponding pairwise session, and re-establishes the sticky session for photo X.
12) The user ratchets the chain the necessary number of steps to get the right **Message Key** MK of photo X.
13) The user uses MK to decrypt the **Blob Secret** of photo X to get the **Blob Key** and the **hash**.
14) Finally, the user uses the produced **Blob Key** and the **hash** to decrypt and verify photo X.

The above diagram summarizes the procedure. Requests to the server that first needs verification by some specific piece of data are marked with a blue shield. The first shield verifies the phone code, second one verifies the password + LAT, third one verifies the Auth token. Also, processing steps where there is decryption happening are marked with a golden key. You can see that for Alice to finally decrypt photo X after STATE RESET she had to go through 5 different decryption processes (and pass 3 barriers). This emphasizes the security of the Stick protocol.

3.3 Use Case Problem Solved!

Now, let's reflect back on the use case we had. Alice can share photo A with group A using group A's id as the stickId, and photo B to both of groups A & B using the partyId associated with that collection as the stickId, and photo C to her profile using her own partyId as the stickId. As discussed throughout the Stick protocol, all of Alice's photos will be e2e encrypted to the designated parties. Alice can view her posts after reinstalling the application or from another phone as shown in the above flow diagram. Alice still benefits from the security features of the Signal protocol, such as X3DH. Sharing sender keys has perfect forward secrecy and perfect backward secrecy using multiple pairwise sessions. In addition, sharing posts using sticky sessions provides perfect forward secrecy as well as backward secrecy up to a maximum of N *Encryptions*.

4 Formal Verification

Throughout this work, we were aiming to have the development and implementation of the Stick protocol done hand-in-hand with the formal security analysis. To help us in achieving this, we used Verifpal [18], a formal cryptographic verification tool, designed with a more intuitive language, in order to bring development and verification closer together. Verifpal is inspired by two decades of seminal work on formal verification by Prof. Bruno Blanchet, the author of ProVerif - which is considered to be state-of-the-art in formal verification of cryptographic protocols. Verifpal is based on the Dolev-Yao model which is the most well-known modelling technique for verifying cryptographic protocols [19]. Verifpal is able to model protocols under an active attacker with unbounded sessions and fresh values, and supports queries for advanced security features such as forward secrecy and backward secrecy or key compromise impersonation. Furthermore, Verifpal's semantics have been formalized within the Coq theorem prover [20]. Verifpal has already been used to verify security properties for Signal, Scuttlebutt, TLS 1.3, Telegram and other protocols, and it has obtained matching results to that of ProVerif [18].

4.1 Security Model

4.1.1 Cryptographic Models

When attempting to verify cryptographic protocols, there are 2 kinds of formal verification models to use: the symbolic model, and the computational model. Generally, symbolic cryptographic models have been particularly suitable for automated protocol analysis and verification. Symbolic models ignore attacks with negligible probability and treat each cryptographic function as a perfect black-box. For instance, in these models, encryption is a message constructor method that can only be reversed by decryption, and hash functions cannot collide [21]. In the computational model, the messages are bitstrings, the cryptographic primitives are functions over bitstrings, and the adversary is any probabilistic Turing machine [22]. It aims to verify the low-level implementation details. When we want to find attacks

```
attacker[active] // Declare an active attacker
principal Alice[
    knows private aIkPriv // Alice private IK
    aIkPub = G^aIkPriv // Alice public IK
]
principal Bob[
    knows private bIkPriv, bSpkPriv // Bob priv IK & SPK
    generates bOpkPriv // Bob private OPK
    bIkPub = G^bIkPriv // Bob public IK
    bSpkPub = G^bSpkPriv // Bob public SPK
    bOpkPub = G^bOpkPriv // Bob public OPK
    bSig = SIGN(bIkPriv, bSpkPub) // Bob's signature
]
// Alice fetches Bob's prekey bundle
Bob -> Alice: [bIkPub], bSig, bSpkPub, bOpkPub
principal Alice[
    generates aEk1Priv // Alice ephemeral key
    aEk1Pub = G^aEk1Priv
    // Derive the master secret & then root key aRK1
    aMasterSec = HASH(bSpkPub^aIkPriv, bIkPub^aEk1Priv,
      bSpkPub^aEk1Priv, bOpkPub^aEk1Priv)
    aRK1, aCkBA1 = HKDF(aMasterSec, nil, nil)
]
principal Alice[ // Encrypting msg1
    generates msg1, aEk2Priv // Generates msg1 & eph. key
    aEk2Pub = G^aEk2Priv // Ephemeral public key
    // Verify Bob's signature
    valid = SIGNVERIF(bIkPub, bSpkPub, bSig)?
    aDH1 = bSpkPub^aEk2Priv // DH output for the DH ratchet
    // Derive new root and sending chain keys
    aRK2, aCkAB1 = HKDF(aDH1, aRK1, nil)
    // Derive msg key aMK1
    aCkAB2, aMk1 = HKDF(MAC(aCkAB1, nil), nil, nil)
    msg1Enc = AEAD_ENC(aMk1, msg1, HASH(aIkPub,
     bIkPub, aEk2Pub))// Encrypt msg1
]
// Alice sends encrypted msg to Bob
Alice -> Bob: [aIkPub], aEk1Pub, aEk2Pub, msg1Enc
principal Bob[ // Bob derive's master secret and root key
    bMaster = HASH(aIkPub^bSpkPriv, aEk1Pub^bIkPriv,
     aEk1Pub^bSpkPriv, aEk1Pub^bOpkPriv)
    brkba1, bckba1 = HKDF(bmaster, nil, nil)
]
principal Bob[ // Bob decrypts msg1
    bDH1 = aEk2Pub^bSpkPriv
    bRkAB1, bCkAB1 = HKDF(bDH1, brkba1, nil)
    bCkAB2, bMk1 = HKDF(MAC(bCkAB1, nil), nil, nil)
    msg1Dec = AEAD_DEC(bMk1, msg1Enc, HASH(aIkPub,
     bIkPub, aEk2Pub))
]
phase[1]
principal Alice[leaks aIkPriv]
principal Bob[leaks bIkPriv, bSpkPriv]
queries[
    authentication? Alice -> Bob: msg1Enc
    confidentiality? msg1
]
```

Fig. 3. Verifpal Model for a Pairwise Session

that rely on logical flaws in the protocol and in its use of cryptographic primitives, we use the symbolic model, and that is our goal here.

4.1.2 Threat Model

Threat models vary for different protocols. For the Stick protocol we consider the following threats:

- **Untrusted Network**: In our analysis, the attacker will have control over the network, thus can intercept and tamper with data sent over the network. In addition, we treat the server of the service provider as untrusted.
- **Malicious Principals**: The attacker controls a set of valid protocol participants, for whom it knows the long-term secrets. The attacker may advertise any identity key for its controlled principals; it may even pretend to own someone else's identity keys.
- **User Keys Compromise**: The attacker may compromise a particular user to obtain their identity key, signed prekey or one-time prekey.
- **Session State Compromise**: The attacker may compromise a user device to obtain the full session state at some intermediate stage of the protocol.

4.1.3 Security Goals

Unless otherwise specified, the following security properties assume that the users are honest, meaning their long-term secrets have not been compromised. We begin with several variants of authenticity goals:

- **Post Authenticity**: If Bob sees a post from Alice, then Alice must have shared that post, and to a party that includes Bob.
- **No Replays**: Each post shared by Alice is unique - the attacker should not be able to make a message-replay-like behaviour on Alice's posts.
- **No Key Compromise Impersonation**: Even if Alice's identity keys are compromised, post authenticity must hold. Meaning the attacker should not be able to forge a post from Alice to Bob.

Our definition of post (message) authenticity covers integrity as well as sender and recipient authentication. Obtaining post (message) authenticity also helps prevent unknown key share attacks, where B receives a message M from A, but A sent that message to a different intended recipient C. We define six confidentiality goals:

- **Secrecy**: If Alice shares a post to party P, then nobody except the members of P be able to see that post.
- **Indistinguishability**: If Alice randomly chooses between two posts of the same size, x and y, and shares one of them, then the attacker should be able to indentify which of x or y was shared.
- **Pairwise Forward Secrecy**: If Alice sends Bob a sender key SK, then Alice's and Bob's Identity keys or the session state is compromised, SK remains secret.
- **Pairwise Backward Secrecy**: a compromise of a session state at any point should not reveal any future communications.
- **Sticky Session Forward Secrecy**: a compromise of a message key *MK* from a sticky session should not reveal any communications from the past.
- **Sticky Session Backward Secrecy**: a compromise of a sticky session state should self-heal after a maximum of *N Encryptions*.

4.2 Analysis in Verifpal

In order to verify cryptographic protocols using verifications tools like Verifpal, the protocol is broken down into several smaller models, where each model represent a scenario. To describe a protocol model in Verifpal, firstly we define whether the model is going to be analyzed under a passive or active attacker. Secondly, we define the different *principals* taking part in that protocol model other than the attacker, for example: Alice, Bob and Charlie. Then, we need to describe the *messages* being communicated between the

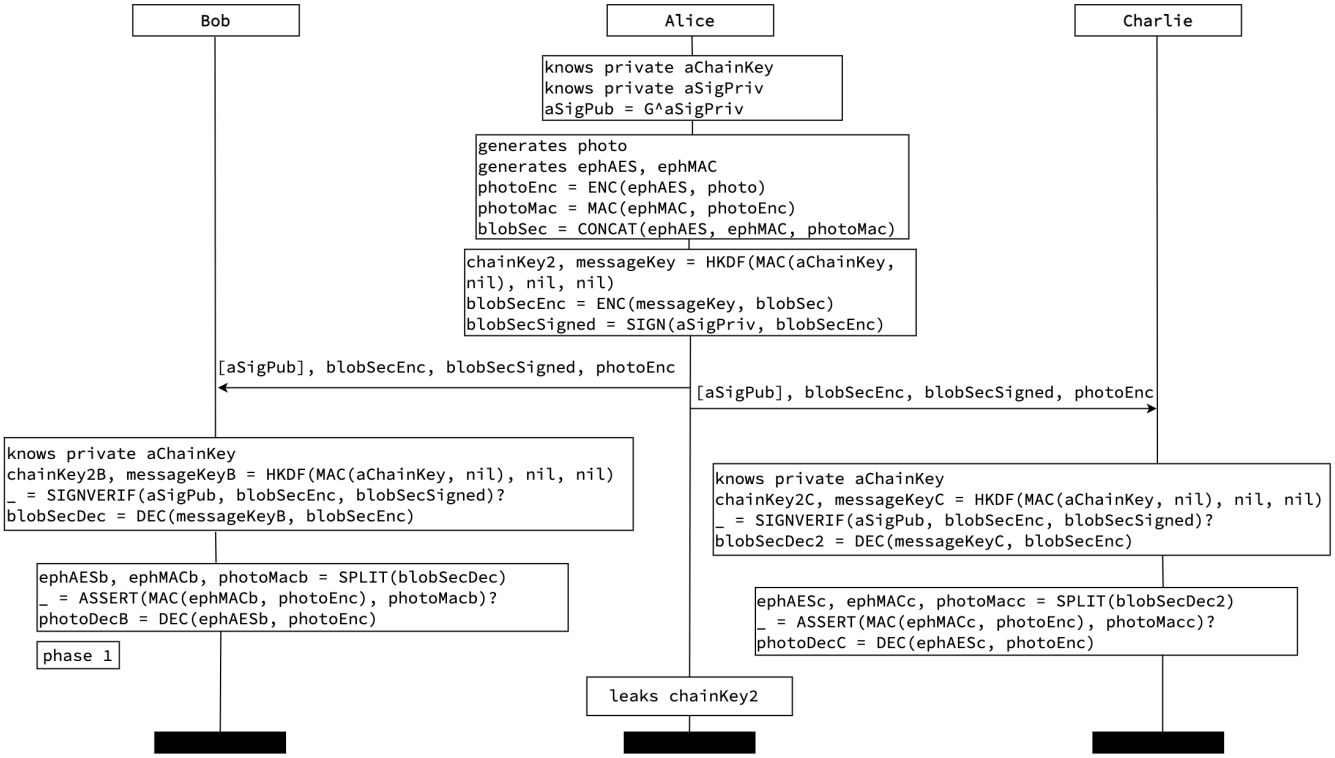

Fig. 4. Sticky Sesison Verifpal Model

different *principals* across the network. Finally, we ask Verifpal our queries which we would like to test, for example confidentiality of a message. The Verifpal user manual has full explanation of how it works [23].

Scenario 1: Exchanging Sender Keys through Pairwise Sessions. Here, we are trying to verify the authenticity and confidentiality of pairwise sessions in the Stick protocol which are used to communicate the sticky sessions' sender keys. Figure 3 shows a Verifpal model of a pairwise session in the Stick protocol, which essentially is a Signal pairwise session. At the end of the model we have two queries which we would like to test. First, we want to verify the authenticity of *msg1* (a sender key) that it really came from Alice. Second, we want to verify *msg1* confidentiality. So firstly, we start by declaring Alice and Bob. Alice has an *IK*, and Bob has an *IK*, *SPK* & *OPK*. Alice will fetch a PKB of Bob, and initiate a session with him by deriving a master secret *aMasterSec*, and then the root key *aRK1* for her sending chain. Next, Alice will encrypt *msg1* to Bob after carrying out the "double-ratchet", and sends it. In order for Bob to decrypt msg1, he will derive the master secret. Then, Bob will decrypt msg1 after carrying out the "double-ratchet" as well. Finally, we would like to declare that at some point in the future Alice and Bob will have their phones stolen, thus revealing their *IK* & *SPK* private keys to the attacker. We uses phases to express this (phases allow Verifpal to reliably model post-compromise security properties such as forward secrecy or backward secrecy). Our 2 queries passing Verifpal's analysis proves that whenever Alice sends a sticky session sender key to Bob, no one can tamper with it, no one will be able to decrypt it other than Bob, and it must have come from Alice. In addition, if their pairwise session is compromised, the sender key will stay confidential.

Scenario 2: Sharing a Photo in a Sticky Session. In this scenario, we would like to verify the confidentiality and authenticity of Alice sharing a photo to Bob and Charlie in a sticky session. We want to query both of the photo and its blob secret. We assume that Alice has already sent her sender key to Bob and Charlie. Figure 4 shows a Verifpal model diagram for a better visualization of the process. Looking at figure 4, Alice has her sender key, which is composed of a chain key and a signature key pair. Alice will first encrypt the photo file using an ephemeral AES256 key and an ephemeral HMAC-SHA256 key as described in section 3.2.7. Then, Alice will ratchet her chain to obtain a message key, encrypt the blob secret and sign it. Alice, will send the encrypted data to Bob and Charlie (since we are assuming that Alice has already communicated her sender key, Alice's public signature key *aSigPub* is guarded with [], and Bob & Charlie already *knows* the chain key *aChainKey*). Both Bob and Charlie will derive the message key, verify the signature, and decrypt the blob secret. Then, they will verify the encrypted photo file and decrypt it. At last, we assume that Alice leaks her current chain key *chainKey2*. This scenario had 6 queries for the authenticity of the blob secret and the photo file from Alice to Bob and Charlie, in addition to their confidentiality. These queries passing prove that whenever Alice shares a photo to a party, the members of that party will be able to verify that both of the blob secret and the photo are from Alice. Also, only the members of that party will be able to decrypt the blob secret and the photo. Moreover, if Alice ever leaks her chain key it will not affect the secrecy of her past communications (forward secrecy).

Scenario 3: Re-establishing Sessions. In this scenario,

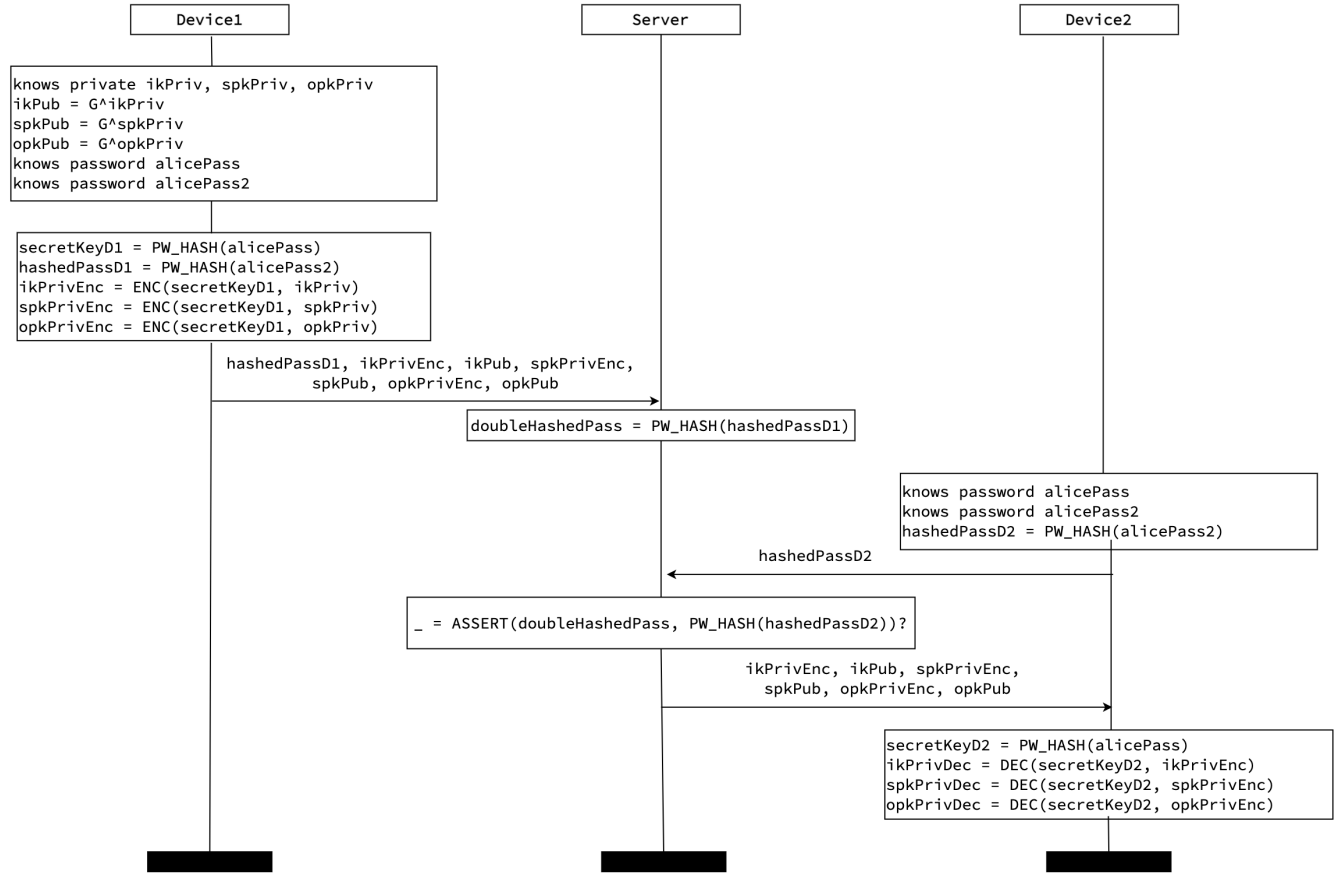

Fig. 5. Re-establishing Sessions Verifpal Model

we want to prove the confidentiality of the backed up private keys, and the user's password, which is a key part to decrypting the ciphered private keys. Let's say Alice has 2 devices, and has created her account on *device-1* and would like to access her content on *device-2*. Looking at figure 5, we initialize *device-1* with an *IK*, *SPK*, *OPK* & a password. In practice, Alice has one password, but here *alicePass2* is used so that we can have a different derived key or hash within Verifpal. Alice will create a secret key (in practice, derived secret keys are unique) using the *PW_HASH* function (can be used to represent Argon2 hashing within Verifpal), in addition to the IPH. Alice will use the secret key to encrypt her private keys. Alice can now safely send her encrypted private keys to the server. The server will hash again the IPH creating a double-hashed password. Now, Alice wants to login from her *device-2*. Alice will create the IPH, send it to the server. The server will create double-hashed password and verify it, then return to Alice her keys. Alice will recreate the secret key and decrypt the ciphered private keys. Now that Alice has gotten her keys on device-2, she can re-establish any pairwise sessions she had, then re-establish her sticky sessions by decrypting any sender keys that was sent to her (like in Scenario 1). This scenario queried about the confidentiality of the private keys and the password. Successful verification of these queries shows that Alice was able to get back her private keys on device-2, where no one other than Alice was able to decrypt those keys. Also, her password stayed secret and never left any of her devices.

Scenario 4: Sticky Session Backward Secrecy. Sticky sessions have a lifecycle of *N Encryptions*. This lifecycle trait of sticky sessions provides backward secrecy every a maximum of *N Encryptions*. In this scenario, we aim to test sticky sessions' backward secrecy. Looking at figure 6, we start by initializing Alice having a sender key for a sticky session *X*. At some point, Alice will have her chain key leaked. When Alice's sticky session X reaches the end of its lifecycle it will expire, and Alice will create a new sticky session Y. Using sticky session Y, Alice will share a post with Bob. Bob will verify and decrypt the post (again assuming Alice have already communicated her sender key). We are querying for the authenticity and confidentiality of *postY*. The 2 queries passing Verifpal's analysis proves the backward secrecy of the sticky sessions every a maximum of *N* Encryptions.

Scenario 5: Malicious Server. In this scenario, we want to test if a malicious server will be detected. We have a malicious server that is trying to tamper with a post shared by Alice. We expect the *SIGNVERIF* method to fail when Bob is trying to verify Alice's signature. Looking at figure 7, we start by initializing Alice with a sender key. Alice will encrypt a post, sign it, and upload it to the server. Now, the server will modify the encrypted post before sending it to Bob. When Bob tries to verify Alice's signature, it should fail. This proves that if a server tried to tamper with an encrypted post, it will be detected. The receiver is able to detect whenever encrypted data has been tampered with, and as such not consider it authentic nor confidential.

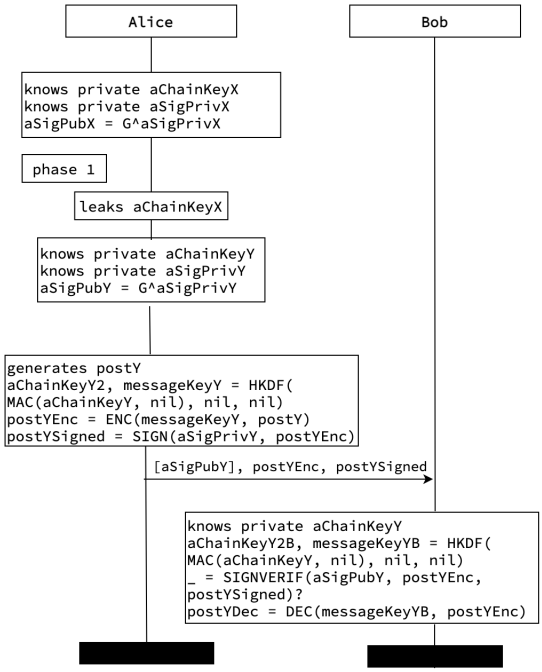

Fig. 6. Sticky Session Backward Secrecy Verifpal Model

```
attacker[active]
principal Alice[
    // Alice's chain key for a sticky session
    knows private aChainKey
    // Alice's signature key for a sticky session
    knows private aSigPriv
    aSigPub = G^aSigPriv
]
principal Alice[
    generates post
    chainKey2, messageKey = HKDF(
      MAC(aChainKey, nil), nil, nil) // Derive msg key
    postEnc = ENC(messageKey, post) // Encrypt post
    postSigned = SIGN(aSigPriv, postEnc) // Sign post
]
// Alice uploads to server her sender key and the encrypted data
Alice -> Server: [aChainKey], [aSigPub], postEnc,
                 postSigned
principal Server[ // Server tampering with the data
    generates malPart
    malpost = CONCAT(postEnc, malPart)
]
// Server sending to Bob the modified data
Server -> Bob: [aChainKey], [aSigPub], malpost,
               postSigned
principal Bob[
    // Bob verifying the signature. The protocol should fail here!
    _ = SIGNVERIF(aSigPub, malpost, postSigned)?
    chainKey2B, messageKeyB = HKDF(
      MAC(aChainKey, nil), nil, nil)
    postDec = DEC(messageKeyB, malpost)
]
```

Fig. 7. Malicious Server Verifpal Model

4.3 Analysis Conclusion

This section has provided formal security analysis of the Stick protocol. We opted to use a more intuitive tool - Verifpal - which can be understood by a wider audience, and help us have the design and implementation of the Stick protocol done hand-in-hand with the formal verification. Doing so have helped us detect some flaws in the design which we successfully fixed during the development of the protocol (discussed in a bit). While a protocol being analyzed and verified using verification tools like Verifpal or ProVerif does not gurantee protection against all possible attacks, it helps in eliminating certain well-defined classes of attacks. Our analysis has shown that the Stick protocol provides useful security properties under a variety of adversarial compromise scenarios. The Stick protocol is able to achieve a form of post-compromise security in many-to-many communications, the trait which most group protocols lack. Moreover, the Stick protocol is able to securely re-establish the pairwise sessions and thus the sticky sessions. Being able to securely re-establish encryption sessions is the goal towards having E2EE in social network platforms. Also, our analysis featured verification of Signal's pairwise sessions. Although, Signal's pairwise sessions has already been verified in a few papers before using ProVerif [24], our analysis included a Verifpal model verifying the authenticity and confidentiality of signal's pairwise sessions for exchanging the sticky sessions' sender keys. And most importantly, we have verified the authenticity and secrecy of communciations (including blob files) within sticky sessions, directly in scenario 2, and by corollary in scenario 5.

The verification process has led to some changes in the protocol design. The following are the main changes made as a result of the verification process:

Refreshing Identity Keys. In scenario 1 for exchanging sender keys through pairwise sessions, we wanted specify that at some point Alice and Bob may have their identity private keys compromised. In the signal protocol, Alice and Bob may recover from such a compromise by reinstalling the app causing them to start a new phase with fresh identity keys. In the initial design of the Stick protocol, a user would have the same identity for key every phase. To mitigate this, the Stick protocol introduced refreshing identity keys as discussed in section 3.2.8.

Double Hashing. In the initial design of the Stick protocol, the user would traditionally send their plaintext password to the server in order to verify it. This presents multiple threats to the user, as the password is used for authorization, in addition to deriving secret keys for encrypting the private keys. Since in our threat model we are assuming an untrusted network, therefore the user's password should never leave the user's device, as this can make it vulnerable to attacks by eavesdroppers or even the server itself (in scenario 3 for re-establishing sessions). As such, the Stick protocol introduced the double hashing technique discussed in section 3.2.9.

5 Implementation

This section gives a brief overview of the Stick protocol's implementation. The Stick protocol implementation is open-source available on Github.

The Stick protocol was implemented to be a superset to the Signal protocol making the Stick protocol logic external to the Signal protocol. This creates a well defined borders for the Stick protocol when trying to verify it. In addition, this allows the Signal protocol to be used in parallel with the Stick protocol, from just the Stick protocol library.

A common trend we noticed across the Signal protocol's Github issues across their 3 repositories [25], is that many developers struggle to get started with the Signal protocol, as the Signal protocol does not provide a detailed implementation documentation. Furthermore, the Signal protocol leaves a lot of logic and interfaces for the developer to implement themselves. While this creates some flexibility, it can be very difficult and incomprehensible for many developers to implement who simply wants to have some sort of E2EE in their app. In addition, this leaves a lot of room for errors, such as storing private keys in an unsecure manner on the user's device — which can cause a catastrophic compromise on security. The stick protocol was implemented to be a fully comprehensive Android and iOS library (rather than just a Java and C library) which can be simply dropped into a social network application, and provide E2EE using re-establishable sticky sessions, with as low development overhead as possible. The Stick protocol implementation is composed of 4 libraries:

- Android library (Gradle package)
- iOS library (CocoaPod framework)
- Server library (PIP package)
- Client handlers library (NPM package)

The Android library and the iOS library are the 2 main libraries of the Stick protocol. They have most of the logic needed on the client-side. In an attempt to make the Stick protocol implementation comprehensive both on the frontend and the backend, the implementation includes a server library in Python for Django. Moreover, the implementation features a client handlers library in JavaScript which contains common handler methods needed for the Stick protocol client-side.

6 Performance Evaluation

One might think that the Stick protocol causing every user to have tens or hundreds of keys would have some computational and networking overhead and take up excessive storage space on the user's device and the server, but this is not true. First of all, when you look at how much data a social networking server process and store per day, in addition to the amount of processing happening on the user's device due to tracking and other services from the service provider, and the gigs of data uploaded by each user from photos and videos, you would realize how negligible the overhead of encryption keys would be. In addition, with the introduction of the ever powerful cellular networking standard 5G, and we already have phones with as big storage as 1TB equipped with powerful processors that are capable of doing highly intensive computations, such as machine learning processing, makes this overhead even smaller. In this section, we aim to provide experimental validation to support the fact that using the Stick protocol in a social network application is not going to have a compromise on performance or usability.

6.1 Experimental Setup

We aim to make as realistic as possible. The Stick protocol is already being used for a social network application called "STiiiCK" which will be released to production soon. We will use STiiiCK for our experiment. To make sure our implementation will run smoothly on a wide range of devices, we will be testing on 2 average devices: an iOS device - iPhone 6s, and an Android device - Samsung Galaxy Note 10 Lite, with benchmarks of 531 and 533 respectively (to put that into perspective, that's only 1/3 of the top ranking phone [26]). The devices will be communicating with a remote server running Python 3.8 on Amazon Linux 2 and connected to a PostgreSQL database - about 4900km away. The application running on the mobile devices is in *release mode*. All tests were done under the same Internet conditions. Each experiment was repeated 20 times on each device, then the average was taken.

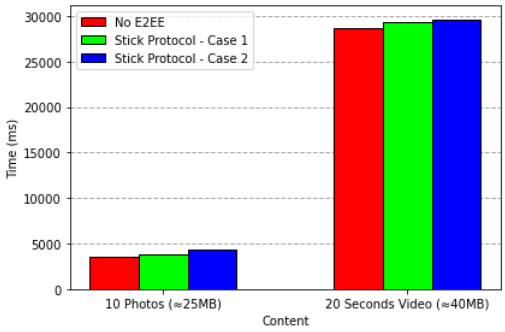

Fig. 8. Stick Protocol vs No E2EE - Sharing Content

6.2 Sharing Content

In the first experiment, we aim to verify that sharing of content using the Stick protocol causes negligible performance overhead in comparison to not using E2EE. There are 3 cases to test:

- **Case 0**: no E2EE being used.
- **Case 1**: Stick protocol E2EE - the user makes a request to the server to get the stickId, then encrypts & shares the content. No extra sender keys needs to be shared.
- **Case 2**: Stick protocol E2EE (less common) - the user needs to make a request to the server to get the stickId of a new sticky session, in addition to a list of users (2 in this case) to share the sender key with. The user will need to fetch PKBs, initialize new pairwise sessions, initialize a new sticky session, encrypt the sender keys and upload them. Then, the user can encrypt & share the content.

For each case, a user is trying to share 10 photos (≈25MB) and a 20 seconds video (≈40MB). To keep the measurements as realistic as possible, the tests includes standard media pre-processing such as: compression & creating thumbnails. Figure 8 shows the results. Sharing 10 photos with no E2EE took ≈3.5s. Introducing E2EE using the Stick protocol in case 1 took an extra ≈0.3s. Stick protocol worst case took ≈4.3s (less than one extra second). Sharing a 20 seconds video with no E2EE took ≈28.6s. Stick protocol case 1 took an extra ≈0.7s. Stick protocol worst case took ≈29.7s. Only 1.1 extra seconds. We can see that the overhead is negligible.

6.3 Receiving Content

In this experiment, we aim to verify that receiving content that is e2e encrypted using the Stick protocol causes negligi-

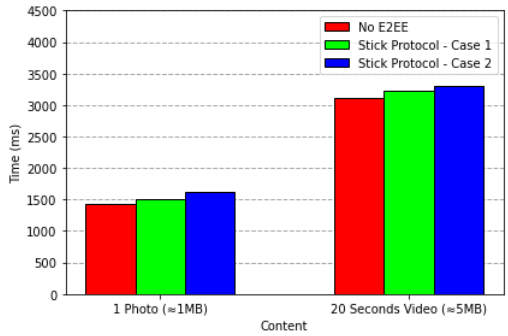

Fig. 9. Stick Protocol vs No E2EE - Receving Content

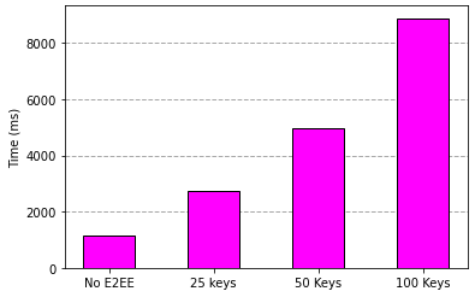

Fig. 10. Stick Protocol vs No E2EE - Reinitializing (login time)

ble performance overhead in comparison to not using E2EE. Again, there are 3 cases to test:

- **Case 0**: no E2EE being used.
- **Case 1**: Stick protocol E2EE - the user already has the corresponding sticky session initialized. The user can download and decrypt the content.
- **Case 2**: Stick protocol E2EE (less common) - the user does not have the corresponding sticky session initialized. The user will need to fetch the corresponding sender key from the server, decrypt it and initialize the sticky session. Then, the user can download and decrypt the content.

In each case, the user is receiving a photo (≈1MB) and a video (≈5MB). Figure 9 summarizes the results. Case 0 took ≈1.4s and ≈3.1s for the photo and the video respectively. Introducing E2EE using the Stick protocol took an extra ≈0.1s only. Stick protocol worst case took ≈1.6s and ≈3.3s for the photo and the video respectively (≈+0.2s). The overhead of receiving encrypted content is even more negligible.

6.4 Reinitializing

The only considerable overhead introduced by the Stick protocol is while recreating the Argon2 hashes from the password at login time in order to decrypt the private keys. In this experiment we aim to evaluate how big of an overhead it is. In the Stick protocol, the default Argon2 parameters are: 4096KiB memory, 3 iterations, running on 2 threads in "id" mode. They provide a balance between security and usability. A developer can tune these parameters if they wish. We will test 4 cases: (i) No E2EE, (ii) 25 keys needs to be decrypted, (iii) 50 keys and (iv) 100 keys. Figure 10 presents the results. Having no E2EE took about 1.2 seconds to login. When introducing E2EE using the Stick protocol, the login time increases linearly in proportion with the number of keys. Having 25 keys took ≈2.7s, 50 keys took ≈5.0s and 100 keys took ≈8.9s. This increase is still just a few seconds and can be mitigated by decrypting the latest N keys at login time, and then decrypting any further keys on demand. Bear in mind these tests were carried out on average to below average devices.

We can conclude from these experiments that running the Stick protocol in a real work social network application is feasible with negligible overhead when sharing and receiving content. The only felt overhead is at login, which does not need to happen often, and we can deem such a small overhead acceptable.

7 Limitations & Future Work

Tying the user's private keys with his password can have some cost, not security related, but rather UX related. As discussed is section 3.2.9, passwords are securely and persistently stored on the user's device using Keychain API or Block Store API. This allows the user to recover their password as long as they have their device. Also, the user can opt to have their Keychain or Block Store backed up e2e encrypted by the OS. The problem lies when the following 3 events happen at the same time: (1) The user forgets their password. (2) The user loses their device. (3) The user's Keychain or Block Store are not backed up. Although these 3 events happening simultaneously would not be very common, but in case they happen the user would not be able to decrypt their private keys, and as a result will not be able to re-establish their encryption sessions. A fix for this problem is to use a Biometric-Based KDF (BB-KDF) instead of a Password-Based KDF (PB-KDF). Biometrics are tied to the user physically (e.g.: fingerprint) making them more secure than a password as they cannot be guessed, stolen or forgotten! However, it is not yet possible to use biometrics to derive a key. A KDF expects a discrete input, whereas a biometric vector is continuous. Also, different devices have different sensors, which would represent the same biometric differently. Even so, some research [27] have been conducted over the past few years on using BB-KDF.

The Stick protocol improves on the current standard of many-to-many encryption where there is mostly no post-compromise security, by having backward secrecy every a maximum of N *Encryptions*. There are some papers with possible solutions to have perfect backward secrecy in group messaging [6], but they are yet to be implemented and verified in a real world app, besides, they are not applicable for a social network. The ultimate goal would be to have perfect backward secrecy for a re-establishable session.

In this work, we have successfully verified the Stick protocol in the symbolic model using Verifpal. A key future work point would be to verify the Stick protocol using ProVerif. In addition, the verification process can be further extended to the computational model using CryptoVerif.

8 Conclusion

In this work, we have proposed an E2EE protocol tailored for social network platforms, based on the Signal protocol. This work was end-to-end process of developing the proposed protocol from designing and verification to

implementation and evaluation. Our verification has proved that the proposed protocol is able to support re-establishable sessions with forward secrecy, and achive a form of post-compromise security. In addition, our evaluation has shown that using such a protocol in a real world social network app would have no noticeable compromise on usability or performance.

REFERENCES

[1] J. Isaak and M. J. Hanna, "User data privacy: Facebook, cambridge analytica, and privacy protection," *Computer*, vol. 51, no. 8, pp. 56–59, 2018.

[2] WhatsApp, "Whatsapp encryption overview," WhatsApp Inc., Menlo Park, CA, Tech. Rep. Revision 2, 2017.

[3] M. Marlinspike and T. Perrin, "The x3dh key agreement protocol," Open Whisper Systems, Mountain View, CA, Tech. Rep. Revision 1, 2016.

[4] ——, "The double ratchet algorithm," Open Whisper Systems, Mountain View, CA, Tech. Rep. Revision 1, 2016.

[5] J. Blum, S. Booth, O. Gal, M. Krohn, J. Len, K. Lyons, A. Marcedone, M. Maxim, M. E. Mou, J. O'Connor *et al.*, "E2e encryption for zoom meetings," Zoom Video Communications, Inc., San Jose, CA, Tech. Rep. Version 2.3.1, 2020.

[6] K. Cohn-Gordon, C. Cremers, L. Garratt, J. Millican, and K. Milner, "On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1802–1819.

[7] A. Barenghi, M. Beretta, A. Di Federico, and G. Pelosi, "Snake: An end-to-end encrypted online social network," in *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS)*. IEEE, 2014, pp. 763–770.

[8] P. Grassi, J. Fenton, E. Newton, R. Perlner, A. Regenscheid, W. Burr, J. Richer, N. Lefkovitz, J. Danker, Y.-Y. Choong, K. Greene, and M. Theofanos, "Digital identity guidelines: authentication and lifecycle management," National Institute of Standards and Technology, Tech. Rep., 2017.

[9] E. Luo, Q. Liu, and G. Wang, "Hierarchical multi-authority and attribute-based encryption friend discovery scheme in mobile social networks," *IEEE Communications Letters*, vol. 20, no. 9, pp. 1772–1775, 2016.

[10] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin, "Persona: an online social network with user-defined privacy," in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, 2009, pp. 135–146.

[11] S. Jahid, P. Mittal, and N. Borisov, "Easier: Encryption-based access control in social networks with efficient revocation," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011, pp. 411–415.

[12] S. Jahid, S. Nilizadeh, P. Mittal, N. Borisov, and A. Kapadia, "Decent: A decentralized architecture for enforcing privacy in online social networks," in *2012 IEEE International Conference on Pervasive Computing and Communications Workshops*. IEEE, 2012, pp. 326–332.

[13] A. Shakimov, H. Lim, R. Cáceres, L. P. Cox, K. Li, D. Liu, and A. Varshavsky, "Vis-a-vis: Privacy-preserving online social networking via virtual individual servers," in *2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011)*. IEEE, 2011, pp. 1–10.

[14] D. Liu, A. Shakimov, R. Cáceres, A. Varshavsky, and L. P. Cox, "Confidant: protecting osn data without locking it up," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2011, pp. 61–80.

[15] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon2: new generation of memory-hard functions for password hashing and other applications," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 292–302.

[16] Apple, "Keychain services," [online] Available at: https://developer.apple.com/documentation/security/keychain_services/, 2021, [Accessed 1 March 2021].

[17] Android, "Android keystore system," [online] Available at: https://developer.android.com/training/articles/keystore, 2021, [Accessed 1 March 2021].

[18] N. Kobeissi, G. Nicolas, and M. Tiwari, "Verifpal: Cryptographic protocol analysis for the real world," in *International Conference on Cryptology in India*. Springer, 2020, pp. 151–202.

[19] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on information theory*, vol. 29, no. 2, pp. 198–208, 1983.

[20] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[21] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *2017 IEEE European symposium on security and privacy (EuroS&P)*. IEEE, 2017, pp. 435–450.

[22] B. Blanchet, "Security protocol verification: Symbolic and computational models," in *International Conference on Principles of Security and Trust*. Springer, 2012, pp. 3–29.

[23] N. Kobeissi, *Verifpal User Manual*. BoD-Books on Demand, 2020.

[24] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, "A formal security analysis of the signal messaging protocol," *Journal of Cryptology*, vol. 33, no. 4, pp. 1914–1983, 2020.

[25] SignalApp, "libsignal-protocol-java," [online] Available: https://github.com/signalapp/libsignal-protocol-java, 2019, [Accessed 12 April 2020].

[26] Geekbench, "Mobile benchmarks," [online] Available: https://browser.geekbench.com/mobile-benchmarks, 2021, [Accessed 30 March 2021].

[27] M. Seo, J. H. Park, Y. Kim, S. Cho, D. H. Lee, and J. Y. Hwang, "Construction of a new biometric-based key derivation function and its application," *Security and Communication Networks*, vol. 2018, 2018.