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

This paper is published in an IEEE Journal – Transactions on Dependable and Secure Computing.

You can access the IEEE version from here.

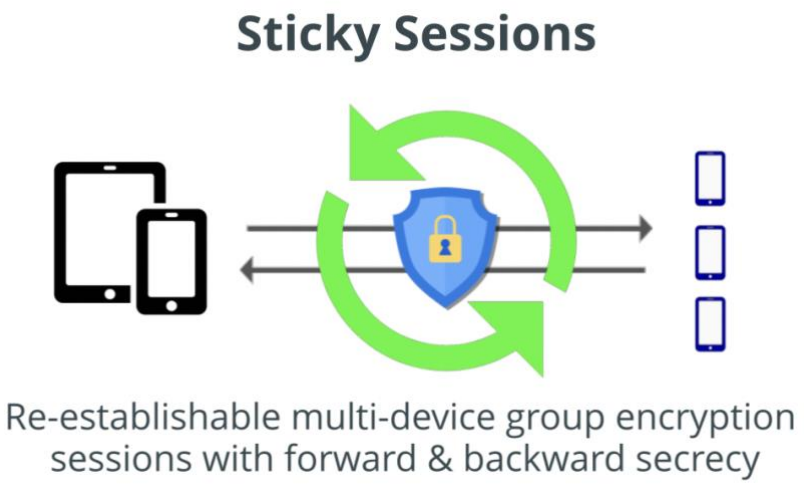

Figure – Stick Protocol teaser image

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

Omar Basem, Abrar Ullah, and Hani Ragab Hassen

Abstract—End-to-End Encryption (E2EE) has become a de facto standard in messengers, especially after the development of the secure messaging protocol – Signal. However, the adoption of E2EE has been limited to messengers, and has not yet seen a noticeable trace in social network platforms, despite the increase in users' privacy violations. In this paper, we propose, verify, implement and evaluate a novel E2EE protocol – Stick. Stick is a Signal-based protocol tailored for social network platforms. We believe our protocol is the first to support re-establishable encryption sessions in an asynchronous multi-device setting while preserving forward secrecy and introducing backward secrecy. Stick includes several innovative features, including a new session concept, multiple pairwise sessions and refreshing identity keys. We verified Stick using Verifpal - a formal verification tool in the symbolic model. Our security analysis shows our protocol does achieve a form of post-compromise security in many-to-many communications – the trait most group protocols lack. Most importantly, the Stick protocol can re-establish encryption sessions while ensuring authentication and confidentiality. We implemented our protocol as a stand-alone open-source API. Our evaluation shows the Stick protocol can be used in a real-world social network app with no noticeable compromise on usability or performance.

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

1 Introduction

Social network platforms (SNPs) have been rapidly developing, making us more connected than ever. Their pace of evolution is accelerating and is not slowing down anytime soon as we head into the era of virtual reality. While these technological advances have revolutionized how we communicate, they have 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 service providers, and application service providers. Recently, there have been multiple data leaks associated with social networking, i.e., 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. A potential solution is using E2EE in SNPs. But, what are the challenges for using E2EE in SNPs?

E2EE is being used for several applications from securing networking channels using SSL/TLS to private communications in messengers to signing digital certificates. 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 usage of E2EE protocols to short-term sessions between two parties. Using these short-term sessions in an SNP to provide E2EE would not work. In the 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 device or re-installs 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 messages. In practice, these messages are usually deleted from the server anyway once received, so there is no way to redecrypt those messages. However, this behavior in messengers is acceptable. Indeed, it is the desired behavior as normally the recipient would not need to decrypt your message more than once. In contrast, this behavior would be problematic on SNPs. If Alice shared a photo on an SNP, then a month later she wanted to reinstall the app, she would expect every photo she has shared or was shared with her to still be there and be able to decrypt it again. Also, Alice should be able to view these photos from other available devices using her account. As a result, using E2EE in an SNP was never practically feasible. Building on that, we regard our contributions as follows:

- We solve the above problem by designing an E2EE protocol tailored for SNPs. The protocol can re-establish E2EE sessions, while preserving Signal's forward secrecy and introducing backward secrecy.
- Our protocol introduces the following security features: (i) *Multiple pairwise sessions* for backward secrecy of exchanging re-establishable session keys. (ii) *Encryption sessions life cycle* for many-to-many (M2M) communications post-compromise security. (iii) *Double-Hashing* for complete hiding of the user's password. (iv) *Refreshing Identity Keys* to self-heal from long-term keys leakage.
- We formally verify the proposed protocol in the symbolic model using Verifpal to prove that it is able to achieve its security properties.

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

- We implement the proposed protocol as iOS and Android libraries, in addition to a server library and a client-handlers library.
- We evaluate the proposed protocol to provide validation supporting the fact that using the Stick protocol in an SNP does not compromise usability or performance.

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 discuss the protocol's main outcomes, limitations and future work. Lastly, we conclude our work in section 8.

2 Related Work

Given the complexity of Signal protocol, we aim through this section to give a brief overview on it, without going into deep details. The Signal Protocol is known to be one of the most secure E2EE protocols. Initially, it was 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 messages [2]. The Signal protocol uniqueness can be accredited to the following 2 assets:

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

2.1 The X3DH Key Agreement Protocol

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.

There are 3 parties involved in the X3DH protocol: Alice, Bob and a server. Alice wants to create a shared secret key with Bob that both can use to exchange messages. Bob wants to let other parties such as Alice start a conversation with him and create a shared secret key, even when Bob is offline. The server can keep messages that Alice has sent Bob, until Bob gets online. Also, the server can store data from Bob which can be provided to parties like Alice when needed.

A user has 3 types of keys. Identity key (IK): a long-term identifying key. Signed prekey (SPK): used for signing, changes periodically. One-time prekey (OPK): every user would have a set of one-time prekeys.

The X3DH protocol goes through 3 phases. (i) Uploading Keys: Bob publishes his set of public keys mentioned above to the server. (ii) Sending the first message: Alice fetches a prekey bundle (PKB) of Bob, which consists of Bob's IK, SPK, and one of Bob's OPK. Alice will generate an ephemeral key EK and carry out 4 DH calculations as shown in Fig. 1, then generates the secret key XK defined as XK = KDF(DH1 ‖ DH2 ‖ DH3 ‖ DH4). Mutual authentication is provided by DH1 and DH2, while DH3 and DH4 assure forward secrecy. DH4 can be omitted if Bob ran out of his pre-uploaded one-time prekeys. (iii) Receiving the first

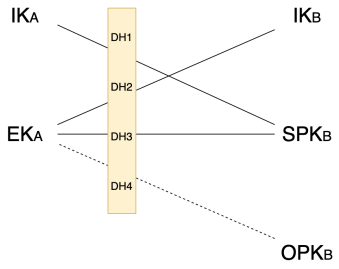

Fig. 1. X3DH Key Agreement

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 XK. Bob carries out DH and KDF calculations like Alice to calculate XK, then decrypt the ciphertext. X3DH is now complete for Alice and Bob. They may keep using XK or keys derived from XK for subsequent communications within some post-X3DH protocol.

2.2 The Double Ratchet Algorithm

After 2 parties have carried out X3DH to agree on a shared secret, they can use DRA as a post-X3DH protocol for subsequent messages. For every *Double Ratchet* message, the parties 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 outputs are combined with the keys derived from the symmetric 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 use server-side fan-out to send the message to all members of a group. This is accomplished using *Sender Keys* (SKs). 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 Pair*.
3) Alice combines the *Chain Key* and the public key of the *Signature Key* to make an SK.
4) Alice individually encrypts the SK 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 *Signature Key*.
4) Alice sends the encrypted message to the 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 Towards E2EE for Social Network Platforms

To our best knowledge, fully E2E encrypted SNPs do not exist. Therefore the amount of research that has been done

on E2EE for SNPs specifically is scarce. Most of the research that has been done regarding privacy in SNPs revolves around private messaging 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] present 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 methodology 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 "transparency 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 vulnerable for an attacker to break the secret key, in comparison to X3DH, where there are more keys involved to break.

Cohn-Gordon et al. [6] have an interesting paper where they are addressing the above drawback of the Signal protocol in group messaging, which 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 app, and not for an SNP app that would require re-establishable sessions.

Barenghi et al.'s [7] proposal is closely related to our research work. They are proposing an E2E encrypted SNP as a single-page HTML5 JavaScript application. However, their work has some crucial security flaws. Their approach is protecting every account with a single master key which creates a major threat to the users. If an adversary broke the master key of an account, they would be able to retrieve 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 the National Institute of Standards and Technology (NIST) [8]. Their secret key establishment relies on a question/answer pair. This makes the encryption key weaker, as it is not based on a random function. Group messaging uses one symmetric key, which again would be easier to break. Lastly, their proposal offers no advanced security features, such as: forward secrecy or backward secrecy.

Multiple papers [9], [10], [11] have proposed decentralized architectures as a means of keeping the users' data private. A decentralized system has 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, [10], [11] 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.

We can see that none of the above proposals were actually an E2EE protocol for SNPs. In addition, the mainstream SNPs we have today, such as Facebook and Twitter, none of them uses E2EE for the platform content. Building on that, this paper is proposing an E2EE protocol - the Stick protocol - that is specifically designed for SNPs.

3 Stick Protocol Design

We present our protocol design through a use case description that considers a social network *SN* with the criteria detailed below. On *SN*, Alice has a *userId*, *partyId*, *phoneNumber* (or *email*) and a *password*. Also, Alice would be connected to a number of connections (friends) and a number of groups. When sharing a post Alice can choose to share with one of the following *parties*: (i) A particular group, (ii) Selected group(s) and/or connection(s), or (iii) Her profile (self-party, which includes all of Alice's connections). Alice is currently a member of 3 groups - *G1*, *G2* and *G3*. Alice wants to share photo *A* with *G1*, photo *B* with both of *G1* and *G2*, and photo *C* with everyone she is connected with (her profile). Alice will share these photos from her device *X*. Now, there are 5 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 device *X* on her other device *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 with her, i.e., not lose any data, unlike the case with E2EE messaging protocols.
4) Alice wants to still benefit from the security features of the Signal protocol.
5) Alice is interested in 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 an SNP 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 with 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 device using her account. The Stick protocol solves this problem by using *sticky sessions* (not referring to sticky sessions of load balancers – discussed in section 3.1.1.) while preserving the security features of Signal protocol. Moreover, the Stick protocol provides extra security advantages regarding M2M encryption.

User-Specific Key Types:

- *Identity Keys*: a key pair of type Curve25519 generated at registration time, periodically refreshed.
- *Signed Prekeys*: a signed key pair of type Curve25519 generated at registration time, periodically refreshed.
- *One-Time Prekeys*: a list of key pairs of type Curve25519 generated at registration time. Refilled as needed.

Session Key Types:

- *Encryption 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.
- *Decryption Sender Key (DSK)*: consists of a 32-byte *Chain Key* and a Curve25519 Signature Public Key. 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 that encrypts messages (posts). It consists of an AES-256 key, an HMAC-SHA256 key, and 16 bytes for padding.

Other Terms

- *Collection*: a mix of groups and/or connections.
- *Party*: can be one of 3 – a *group*, a *collection* or *self-party*.
- 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*.

STATE RESET

Given an E2E encrypted application *SN*, STATE RESET is an event that occurs when a user re-installs *SN* wiping all the encryption sessions they had, or when installing *SN* on another device having to establish new encryption sessions, and being unable to decrypt the previously encrypted data.

3.2 The Stick Protocol

3.2.1 User Registration

At registration time, Alice would generate an IK, an SPK and a list of OPKs. 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 2.3 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 *SK1* to Bob. Bob will not be able to decrypt *SK1* after STATE RESET unless he establishes 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 IKs, SPKs and OPKs 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 [12]. 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. Alice or Bob can encrypt a private key for backup as follows:

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 back up a private key, and no one else including the server may 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 Pairwise Sessions

At this point, Bob will not be able to re-establish every single session he has had with Alice, but the first session only. Signal pairwise sessions have backward secrecy where no later key can be derived from a previous key, i.e., when Bob re-establish a session he will be able to decrypt the first message (*SK1*) only. So, if Alice encrypted to Bob another sender key *SK2* of group *G2*, Bob will not be able to decrypt *SK2* or any subsequent SKs. 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 SK only. The next SK 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 SKs.

3.2.5 Session Life Cycle

In Signal protocol group messaging, users change their SK only when the group's membership changes, or when reinstalling the app, otherwise they will keep using the same SK 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 has 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 for more *Encryptions*, but it can still be used for decryptions as the message keys are saved into the session's internal state on the user's device.

3.2.6 Creating Sticky Sessions and Encrypting Data

Whenever Alice wants to make an *Encryption* she needs to provide the *stickId*, a sticky session id.

$$stickId = (targetPartyId \parallel activeChainId)$$

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. Fig. 2 shows the algorithm for creating sticky sessions and encrypting data. It goes through 3 phases. Phase A of the algorithm presents how to determine the *stickId*. The server returns to Alice the *stickId*, target party members that do not have Alice's sender key for that particular sticky session, and its *currentStep*. In phase B, Alice can check whether she already has a sticky session on her device associated with that *stickId* and create one if needed, encrypt her SK to the list of users individually using Signal's pairwise sessions, and upload her encrypted SKs to the server. Finally, in phase C, Alice can derive a new

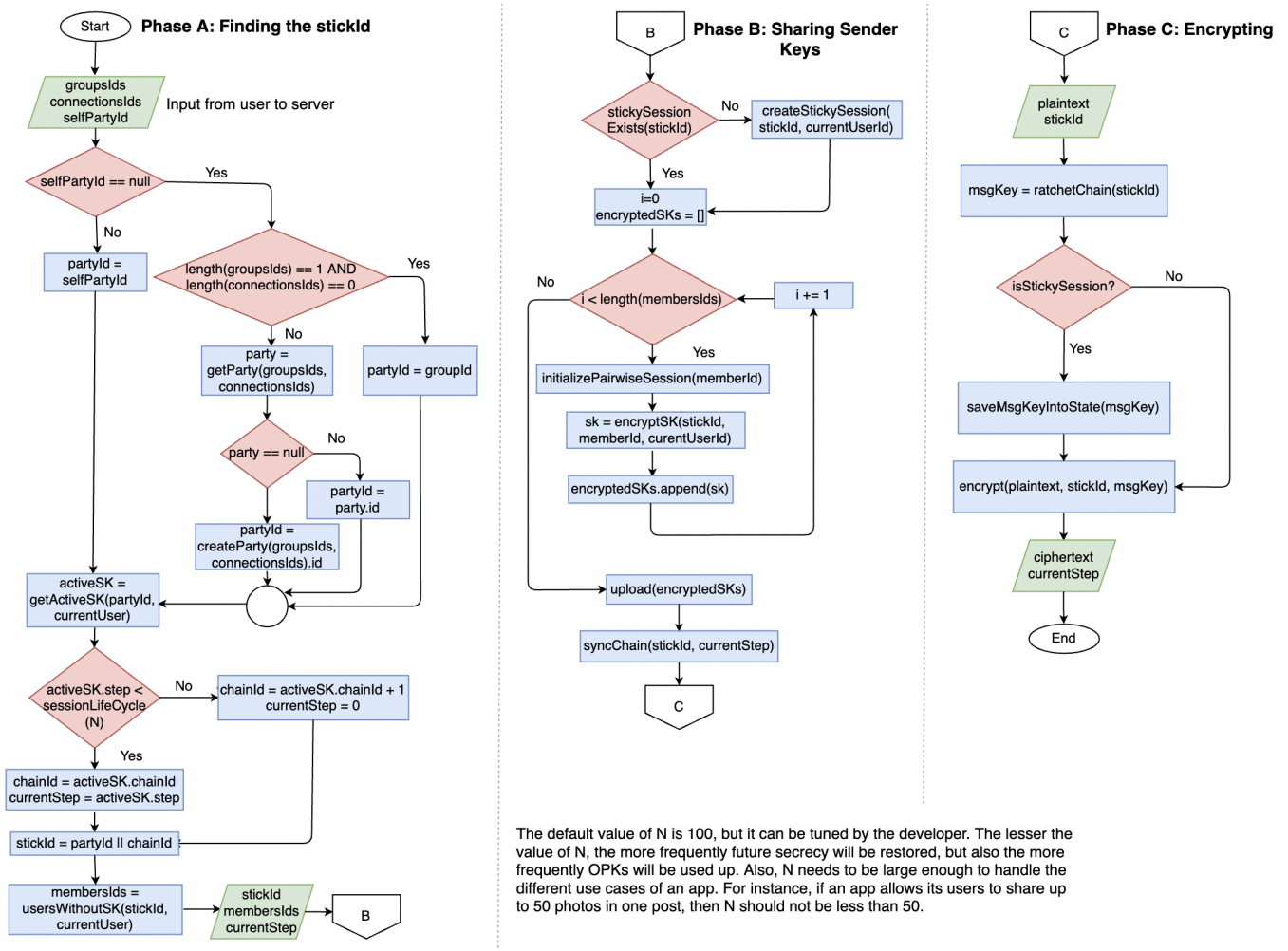

Fig. 2. Algorithm Flowchart 1: Creating Sticky Sessions and Encrypting Data

message key, encrypt the data, save the new state of the sticky session, and upload the encrypted data (along with the new *currentStep).*

3.2.7 Refreshing Identity Keys

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

3.2.8 Password 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 within a two-factor authentication. Although typically servers do not store plaintext passwords, 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 to create an *Initial Password Hash (IPH)* before being sent to the server. The server will treat the *IPH* as the plaintext password, and will rehash it. 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 leaving their device.

Storing Passwords. Every while, the user would need to refill their store of OPKs whenever it goes below a certain threshold. While doing so, they would also need to back up 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.

The Stick protocol employs E2E encrypted sensitive storage APIs from the underlying operating system (OS) to securely store the user's password persistently, such as Keychain API on iOS [13]. If the underlying OS does not support an E2E encrypted sensitive storage API, then an

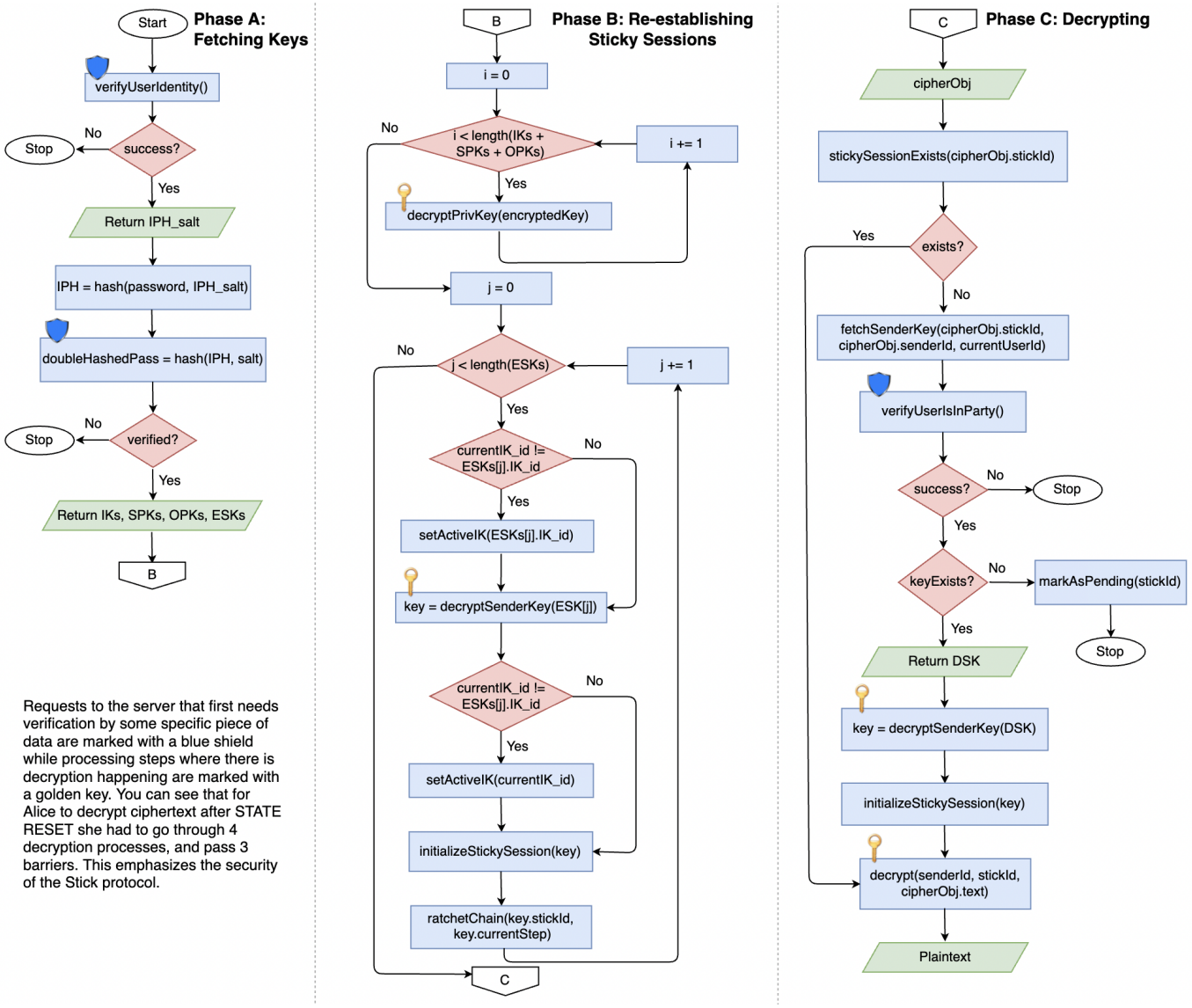

Fig. 3. Algorithm Flowchart 2: Re-establishing Sticky Sessions and Decrypting Data after STATE RESET

alternative solution is to encrypt the password using an AES256 key, then store the ciphertext with the user's OS account, and store the key with the user's *SN* application account. That way only the user and no one else – including the OS provider and the application service provider – can access the password. Storing the user's password persistently would also help the user recover it if they forgot it.

3.2.9 Decryption After STATE RESET

Fig. 3 shows the algorithm of re-establishing sticky sessions and decrypting data after STATE RESET. It goes through 3 phases. In phase A, firstly, Alice needs to verify her identity (ex.: phone). If verified, the server returns the *IPH salt*. Alice will create and send the *IPH*, then the server will use it to create the *double-hash*. If verified, the server returns to Alice her keys. Moving to phase B, Alice will decrypt the private keys of her *IK*s, *SPK*s and *OPK*s as discussed in section 3.2.3. Then, Alice will decrypt her ESKs, initialize her sticky sessions, and ratchet each chain to its *currentStep*. Finally, in phase C, when Alice needs to decrypt some data which she has not re-established its sticky session yet, she fetches the corresponding DSK from the server, initializes the sticky session, then decrypts the data (if the key does not exist, it will be marked as *pending* where the sender is notified that they should encrypt that SK to Alice).

3.3 Reflecting Back: Use Case Problem Solved!

Alice can share photo *A* with group *G1* using its *id* as the *stickId*, and photo *B* with both of groups *G1* and *G2* using the *partyId* associated with that collection as the *stickId*, and photo *C* to her profile using her own *partyId* as the *stickId*. 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 device through the procedure shown in fig. 3. Alice still benefits from the security features of the Signal protocol, such as X3DH. Sharing SKs 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 every *N Encryptions* at max.

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 [14], a modern formal cryptographic verification tool, designed with a more intuitive language, in order to bring development and verification closer together. Verifpal is inspired by the two-decades-old formal verification tool - ProVerif. Verifpal is based on the Dolev-Yao model which is the most well-known modeling technique for verifying cryptographic protocols [15].

4.1 Threat Model

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

- *Untrusted Network*: The attacker will have control over the network, and hence can intercept and tamper with data sent over the network. In addition, we treat the server of the application 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 IK for its controlled principals. It may even pretend to own someone else's IK.
- *User Keys Compromise*: The attacker may compromise a particular user to obtain any of their keys.
- *Session State Compromise*: The attacker may compromise a user device to obtain the full session state at some intermediate stage of the protocol.

4.2 Analysis in Verifpal

To verify cryptographic protocols using verification tools like Verifpal, the protocol is broken down into several smaller models, where each model represents a scenario. In each Verifpal scenario, 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 different *principals* across the network. Finally, we ask Verifpal our queries which we would like to test, for example, confidentiality of a message.

4.2.1 Scenario 1: Exchanging Sender Keys

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' SKs. Fig. 4 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* (an SK) 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* and *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 send it. In order for Bob to decrypt *msg1*, he will derive

```
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. 4. Pairwise Session Verifpal Model

the master secret, then 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 *IK* and *SPK* private keys leaked. We use 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 prove that whenever Alice sends an SK to Bob, no one can decrypt it other than Bob, or tamper with it, and it must have come from Alice. Moreover, if their pairwise session is compromised, the SK will stay confidential.

4.2.2 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 with Bob and Charlie in a sticky session. We want to query both of the

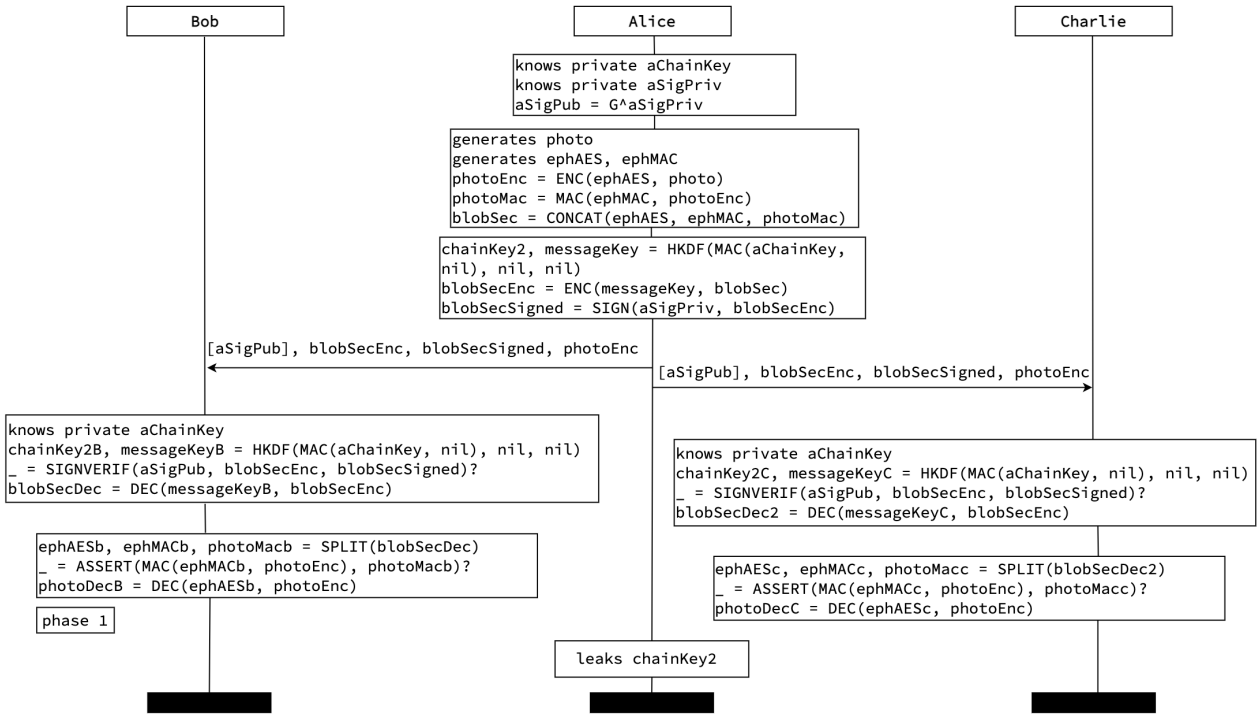

Fig. 5. Sticky Session Verifpal Model

photo and its blob secret. We assume that Alice has already sent her SK to Bob and Charlie. Figure 5 shows a Verifpal model diagram for better visualization of the process. Looking at Fig. 5, Alice has her SK, 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. 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 SK, Alice's public signature key *aSigPub* is guarded with [], and Bob and 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 with a party, its members 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).

4.2.3 Scenario 3: Re-establishing Sessions

Here, we want to prove the confidentiality of the backed-up private keys, and user's password, which is a key part of decrypting the ciphered private keys. Alice has 2 devices, and has created her account on *device-1* and wants to access her content on *device-2*. Looking at Fig. 6, we initialize *device-1* with *IK*, *SPK*, *OPK* and a *password*. In practice, Alice has one password, but here *alicePass2* is used so that we can have a different hash within Verifpal. Alice will create a secret key (in practice, derived secret keys are unique) and the *IPH* using the *PW_HASH* function (can be used to represent Argon2 hashing within Verifpal). 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 log in from her *device-2*. Alice will create the *IPH*, send it to the server. The server will create a *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 has had, then re-establish her sticky sessions by decrypting any SKs that were 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.

4.2.4 Scenario 4: Sticky Session Backward Secrecy

Sticky sessions have a lifecycle of *N Encryptions*. This provides backward secrecy every *N Encryptions* at max. In this scenario, we aim to test sticky sessions' backward secrecy. Looking at Fig. 7, we start by initializing Alice having an SK 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 has already communicated her SK).

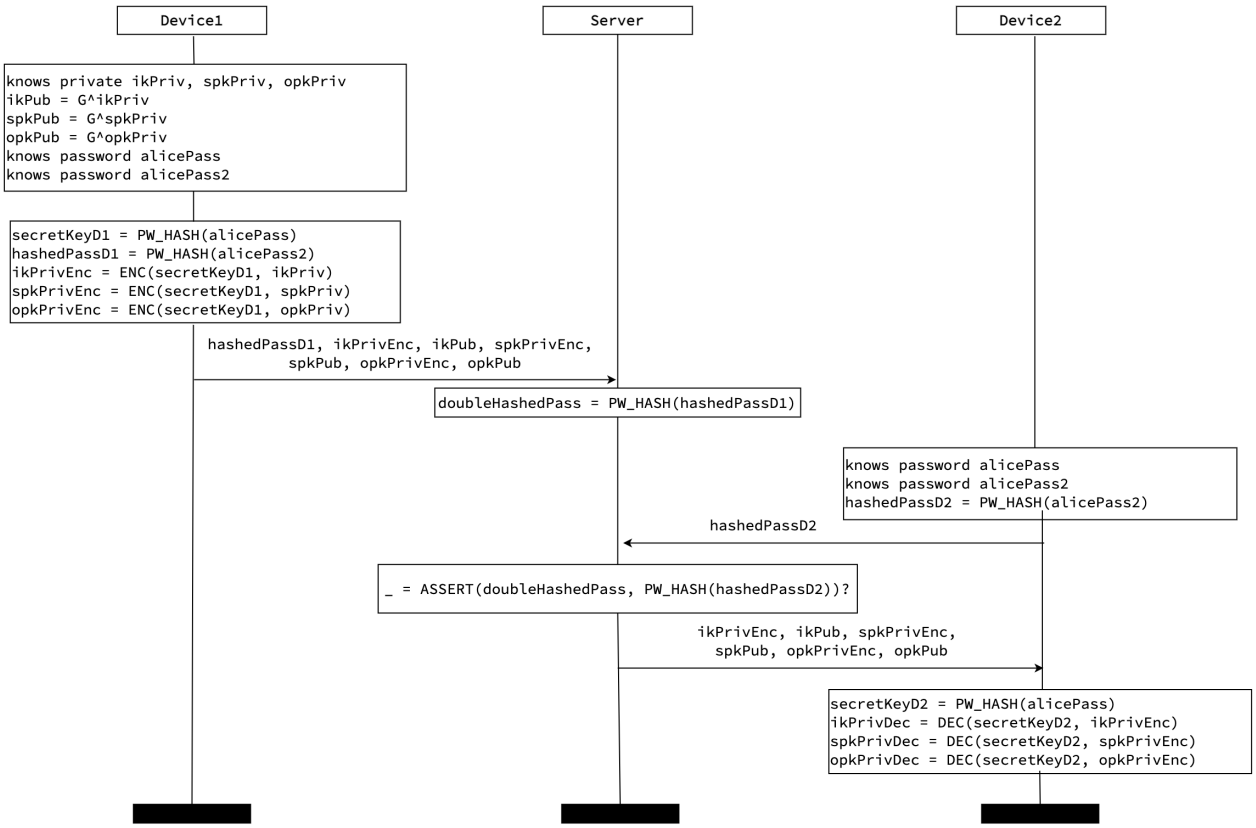

Fig. 6. Re-establishing Sessions Verifpal Model

We queried for authenticity and confidentiality of *postY*. The queries passing Verifpal's analysis prove the backward secrecy of sticky sessions every a max of N Encryptions.

4.3 Analysis Conclusion

This section has provided formal security analysis of the Stick protocol. We opted to use Verifpal - which can be understood by a wider audience, and help us develop and verify the protocol simultaneously. Doing so has helped us detect some flaws in the design which we successfully fixed during the development of the protocol. While a protocol being analyzed and verified using formal verification tools does not guarantee protection against all possible attacks, it helps to eliminate 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 protocol is able to provide backward secrecy in M2M communications, the trait most group protocols lack. Moreover, the Stick protocol can securely re-establish pairwise sessions and thus sticky sessions. Being able to securely re-establish encryption sessions is the goal towards having E2EE in SNPs. Also, our analysis featured verification of Signal's pairwise sessions. Although they have been verified in a few papers before using ProVerif [16], our analysis included a Verifpal model verifying the authenticity and confidentiality of Signal's pairwise sessions for exchanging sticky sessions' SKs. And most importantly, we verified the authenticity and secrecy of communications (including blob files) within sticky sessions. The verification process has led to some design changes:

Refreshing Identity Keys. In scenario 1, we specified that the identity private keys may be compromised. In the signal protocol, Alice and Bob may recover from such a compromise by reinstalling the app. In the initial design of the Stick protocol, a user would have the same IK for every phase. To mitigate this, the Stick protocol introduced *refreshing identity keys* as discussed in section 3.2.7.

Double-Hashing. In the initial design of the Stick protocol, the user traditionally sends their plaintext password to the server to verify it. This presents multiple threats to the user, as the password is used for authentication, as well as in encrypting the private keys. Since in our threat model we are assuming an untrusted network, therefore the user's password should never leave the device, as this can make it vulnerable to attacks by eavesdroppers or even the server itself (in scenario 3). As such, the Stick protocol introduced the *double-hashing* technique discussed in section 3.2.8.

5 Implementation

This section gives a brief overview of the Stick protocol's implementation [17]. The Stick protocol was implemented to be a superset to Signal protocol making the Stick protocol logic external to Signal protocol. This creates well-defined borders for the Stick protocol when trying to verify it. Also, it 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 [18], is that many developers struggle to get started with the Signal protocol,

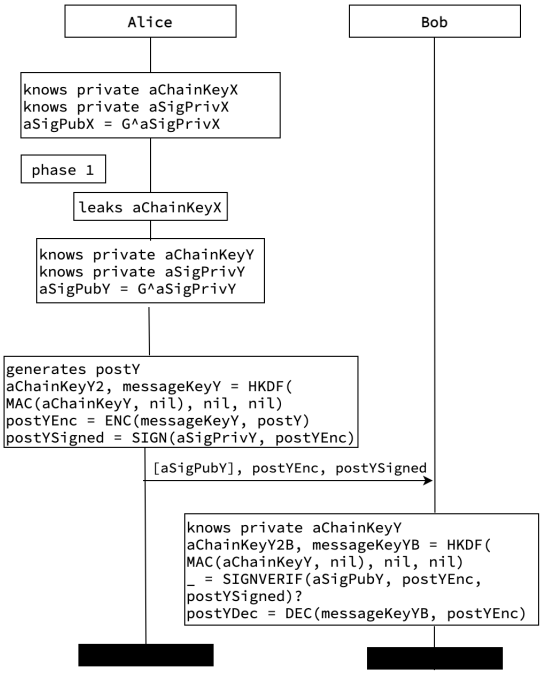

Fig. 7. Sticky Session Backward Secrecy Verifpal Model

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 challenging and incomprehensible for many developers for basic implementation of E2EE in their app. In addition, this can be prone to errors, such as storing private keys in an insecure 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 incorporated into an SNP 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, iOS library, Server library and Client handlers library.

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. 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

In this section, we want to test whether using the Stick protocol in an SNP app would compromise its usability or performance, compared to not having E2EE.

6.1 Experimental Setup

We aim to make our experiment as realistic as possible. The Stick protocol is already being used for an SNP app called StickNet available on the App Store and the Play Store [19]. This study uses StickNet for evaluation. To ensure the implementation runs smoothly on a wide range of devices, it was tested 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 (putting into perspective, that's only 1/3 of the top-ranking phone [20]).

The devices were communicating with a remote server running Amazon Linux 2 and connected to a PostgreSQL database. The backend server is implemented in Python 3.8. 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.

6.2 Sharing Content

In the first experiment, we aim to measure the overhead when sharing content using the Stick protocol in comparison to not using E2EE. There are 3 cases to test:

- *Case 0*: no E2EE being used (1 round).
- *Case 1*: Stick protocol E2EE (2 rounds) - the user makes a request to the server to get the *stickId*, then encrypts and shares the content. No extra SKs to be shared.
- *Case 2*: Stick protocol E2EE (4 rounds, less frequent) - the user needs to request the server for the *stickId* of a new sticky session, plus a list of users (2 in this case) to share the SK with. The user will need to fetch PKBs, initialize new pairwise sessions, initialize a new sticky session, encrypt the SKs and upload them. Then, the user can encrypt and 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 include standard media preprocessing such as: compression and creating thumbnails. Graph A in Fig. 8 shows the results. Sharing 10 photos with no E2EE took ≈3.5s. Introducing E2EE using the Stick protocol in case 1 took extra ≈0.3s. Stick protocol worst case took ≈4.3s (<1 extra second). Sharing a 20 seconds video with no E2EE took ≈28.6s. Stick protocol case 1 took extra ≈0.7s. Stick protocol worst case took ≈29.7s. Only 1.1 extra seconds. We can see that the overhead is fractional.

6.3 Receiving Content

In this experiment, we aim to measure the overhead when receiving content using the Stick protocol in comparison to not using E2EE. Again, there are 3 cases to test:

- *Case 0*: no E2EE being used (1 round).
- *Case 1*: Stick protocol E2EE (1 round) - the user already has the corresponding sticky session initialized. The user can download and decrypt the content.
- *Case 2*: Stick protocol E2EE (2 rounds, less frequent) - the user needs to fetch the SK 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). Graph B in Fig. 8 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 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 smaller.

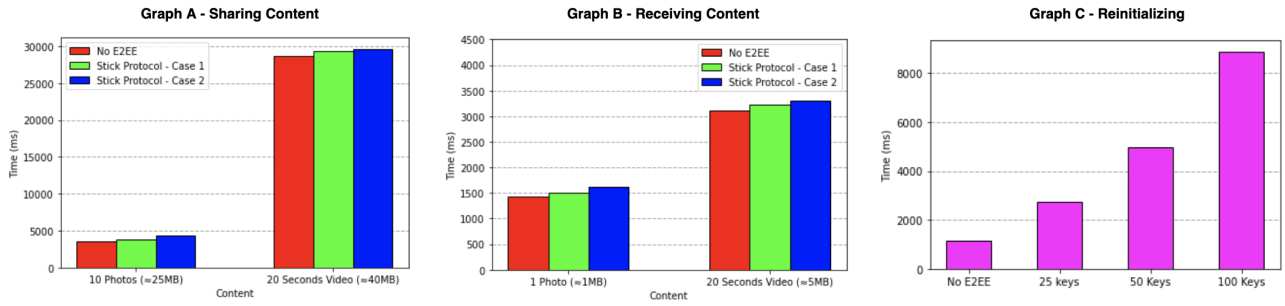

Fig. 8. Stick Protocol vs No E2EE - Performance Evaluation

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 measure that overhead. 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 need to be decrypted, (iii) 50 keys, and (iv) 100 keys. Graph C in Fig. 8 presents the results. Having no E2EE took about 1.2 seconds to log in. 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 can be easily mitigated by decrypting the latest X keys at login time, and then decrypting any further keys on demand.

6.5 Storage

This subsection analyzes whether using our protocol in an SNP imposes considerable storage usage. A user decrypting a small photo of size 1MB (1,048,576 bytes) would need an extra 80 bytes of space used by the message key (and if sticky session was not initialized an additional 160 bytes at max used by PKB and 64 bytes used by SK). That's less than 0.0003% extra space. The same goes for the extra data size of sending or receiving encrypted content. In addition, with the introduction of the ever-powerful cellular networking standard 5G, and we already have phones with as big storage as 1TB, makes the storage overhead even smaller.

6.6 Conclusion

One might think that the Stick protocol causing every user to have tens or hundreds of keys would have some computational, networking or storage overhead, but these experiments have proven otherwise. We can conclude from these experiments that running the Stick protocol in a real-world SNP 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 Discussion

Table 1 compares the privacy and security measures taken by mainstream SNPs versus an app utilizing the Stick protocol. Normally, when a user logs into their SNP account, their password can be seen by the backend server. In contrast, the Stick protocol hides the user password using *double-hashing*. As for private messaging E2EE, only Facebook supports it as an opt-in. On the other hand, *StickProtocolApp* can benefit from the Signal protocol's messaging E2EE as the Stick protocol is built to be a superset to the Signal protocol. Only *StickProtocolApp* can support E2EE for the platform content using sticky sessions. And as a result, the data center will be fully encrypted as well.

TABLE 1
Mainstream SNPs VS an App Using Stick Protocol

App	Hidden Pass	Data Center Enc.	Priv. Msg. E2EE	Platform E2EE
Facebook	No	Partial	Yes (opt-in)	No
Twitter	No	Partial	No	No
LinkedIn	No	Partial	No	No
Instagram	No	Partial	No	No
StickProtocolApp	Yes	Yes	Yes	Yes

TABLE 2
Signal Protocol Group Messaging (SPGM) VS Stick Protocol

Protocol	Forward Sec.	Backward Sec.	IK Self-Healing
SPGM	Yes	No	No
Stick	Yes	Yes (up to N)	Yes

Table 2 compares the privacy features of SPGM vs Stick protocol. Both have perfect forward secrecy. SPGM lacks backward secrecy, while the Stick protocol provides backward secrecy every N Encryptions at max. SPGM also lacks IK self-healing, while the Stick protocol can self-heal after an IK leakage due to its *refreshing identity keys* feature.

Now, let's discuss the Stick protocol limitations and future work. Tying the user's private keys with their password can have some cost, not security-related, but rather UX-related. Consider the following 3 events: (1) The user forgets their password. (2) The user loses their device. (3) The user opted out of using the OS sensitive storage on their device. These 3 events occurring simultaneously would be unlikely, however, in the event, the user would be unable 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 [21] has been conducted over the past few years on using BB-KDF.

Our protocol improves on the current standard of M2M encryption where there is no post-compromise security, by having backward secrecy every *N Encryptions* at max. There are some papers proposing solutions to have M2M perfect backward secrecy [6], but they are yet to be implemented and verified in a real-world app. Furthermore, they are not applicable in an SNP scenario. The ultimate goal is 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.

Lastly, the Stick protocol can be extended to areas other than social networking where E2E encrypted re-establishable sessions would be useful. This includes IoTs, health care and banking systems.

8 Conclusion

In this work, we proposed an E2EE protocol tailored for SNPs, based on Signal protocol. This work was an E2E process of developing the proposed protocol from design 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 achieve a form of post-compromise security. In addition, our evaluation has shown that using our protocol in a real-world SNP app will not compromise 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 3, 2020.

[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 Commun., 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 secur. guarantees," in *Proc. 2018 ACM SIGSAC Conf. on Computer and Commun. Secur.*, 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 Comput. and Commun., 2014 IEEE 6th Intl Symp on Cyberspace Safety and Secur., 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS)*, 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] 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 Int. Conf. on Pervasive Comput. and Commun. Workshops*, 2012, pp. 326–332.

[10] 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 Int. Conf. on Communication Systems and Networks*, 2011, pp. 1–10.

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

[12] 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 Secur. and Privacy (EuroS&P)*. IEEE, 2016, pp. 292–302.

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

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

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

[16] 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.

[17] StickNet, "stick-protocol," [online] Available: https://github.com/sticknet/stick-protocol, 2021.

[18] SignalApp, "libsignal-protocol-java," [online] Available: https://github.com/signalapp/libsignal-protocol-java, 2019.

[19] StickNet, [online] Available at: https://www.sticknet.org/, 2021.

[20] Geekbench, "Mobile benchmarks," [online] Available: https://browser.geekbench.com/mobile-benchmarks, 2021.

[21] 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," *Secur. and Commun. Networks*, vol. 2018.

Omar Basem received the Bachelor of Science honours degree in Computer Science from Heriot-Watt University, United Kingdom. Since 2018, he has been working on creating StickNet – a fully end-to-end encrypted social network platform running the Stick protocol, and has been released on July 2021. His research interests include security protocols, formal verification and applied cryptography.

Abrar Ullah received his PhD Computer Science from the University of Hertfordshire, United Kingdom in 2017. In 2002, he worked as a lecturer at the University of Peshawar. In 2011, he joined Cardiff University as Senior Systems Analyst. In 2017, He worked as a lecturer at Cardiff Metropolitan University. He joined Heriot-watt University in 2019. His research interests include information security, usable security, authentication and access control, software engineering and machine learning for security.

Hani Ragab Hassen obtained his PhD from the University of Technology of Compiegne, France in 2007. He had several security-related roles, including security architect, before joining the University of Kent, United Kingdom, as a lecturer in Information Security. He joined Heriot-Watt University in 2015 where he is the director of the Institute of Applied Information Security. His research interests include security data science, access control systems, blockchains, P2P and secure group communications.