SOAP: A Social Authentication Protocol

Felix Linker
Department of Computer Science, ETH Zurich

David Basin
Department of Computer Science, ETH Zurich

Abstract

Social authentication has been suggested as a usable authentication ceremony to replace manual key authentication in messaging applications. Using social authentication, chat partners authenticate their peers using digital identities managed by identity providers. In this paper, we formally define social authentication, present a protocol called SOAP that largely automates social authentication, formally prove SOAP's security, and demonstrate SOAP's practicality in two prototypes. One prototype is web-based, and the other is implemented in the open-source Signal messaging application.

Using SOAP, users can significantly raise the bar for compromising their messaging accounts. In contrast to the default security provided by messaging applications such as Signal and WhatsApp, attackers must compromise both the messaging account and all identity provider-managed identities to attack a victim. In addition to its security and automation, SOAP is straightforward to adopt as it is built on top of the well-established OpenID Connect protocol.

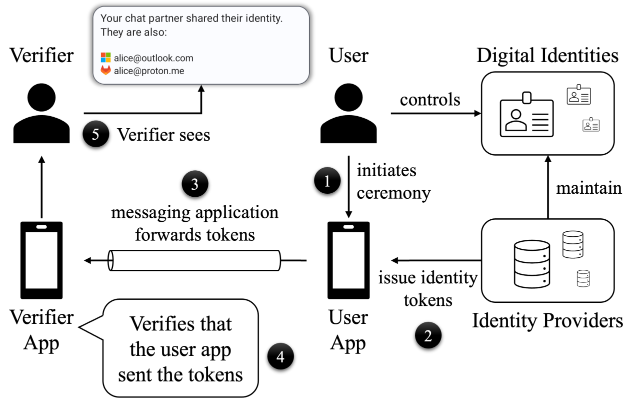


Figure 1: SOAP implements a social authentication ceremony. A user initiates the ceremony in their messaging application, which requests an identity token for each of the user's identities and forwards the tokens. The verifier's application verifies the token's sender. The verifier uses the identities to authenticate the user.

1 Introduction

Social authentication promises simple, usable, and remote key authentication for messaging applications [48] and was first implemented in the Keybase application [27]. Using Keybase, Alice can link her Keybase account to, for example, her Twitter account by tweeting a message signed with her Keybase account's key. This allows other users to *socially authenticate* Alice on Keybase via her Twitter account. More generally, when performing social authentication, users verify that their actual chat partner controls accounts at different identity providers (IdPs) which they know are controlled by their intended chat partner.

Authenticating chat partners is critical for user security: if not done properly, users risk that a Meddler-in-the-Middle (MITM) intercepts their messages. Existing authentication ceremonies do not sufficiently address this risk. Various studies have found that users are unwilling or unable to perform these authentication ceremonies [24, 41, 42, 49]. In particular, users are both challenged and constrained by the in-person comparison of safety numbers as implemented in the messaging applications Signal and WhatsApp. Not only must they understand how to perform this ceremony correctly, they must also be in close physical proximity with one another.

In contrast, automated social authentication was established as a usable authentication ceremony [48] that works remotely. Keybase was first to study social authentication beyond the idea, but Keybase requires manually posting key material, which requires non-trivial user effort. Moreover, the posting is public, which discloses account associations to everyone.

After Zoom acquired Keybase, Zoom published an end-to-end encryption whitepaper [7] which continued this line of work. In particular, it automated the authentication process using a modified version of the OpenID Connect protocol. Zoom's proposal, though, was designed in a setting where every account can be authenticated only by a single IdP and, moreover, Zoom's design requires IdP adoption.

Finally, although past works have presented designs, social authentication has never been studied as an authentication protocol. No prior work defined what security guarantees social authentication should provide, let alone considered whether a given design correctly provides these guarantees.

In this paper, we address all these shortcomings and present SOAP, a formally verified social authentication protocol. We make the following contributions:

- We precisely define the security objectives of social authentication and argue that it should provide a novel security property that we call *sender correspondence*. This is a strong security property in that messaging sessions can only be compromised if all digital identities and the application's key servers are compromised. This raises the bar for the adversary and distributes trust amongst many providers. In contrast to Signal's and WhatsApp's default security, neither the cellular provider nor the key servers nor any of the IdPs involved can individually intercept messaging sessions.
- We formally relate sender correspondence to existing notions of authentication, and show how sender correspondence applies to designs beyond secure messaging.
- We present SOAP, a secure and practical protocol implementing social authentication. Our protocol can be seen as an extension of Zoom's design that works without IdP adoption and for multiple IdPs. SOAP automates the authentication ceremony and provides a straightforward and immediate means for adoption. Figure 1 provides an overview of our design.
- Using the Tamarin model checker [34], we formally prove that SOAP satisfies our novel security property and that SOAP respects user privacy. Users can decide to whom they disclose which identities, and SOAP leaks no information to IdPs beyond that one is using the messaging app in question, e.g., Signal. By employing a salt-and-hash scheme, we avoid revealing key material to IdPs and, thus, leaking one's contacts to providers.
- We show that SOAP is straightforward to adopt by implementing it in two fully functional prototypes: a web-based application and an extension of the Signal Android application.[1] The former requires some user interaction whereas the latter functions mostly automatically.

To the best of our knowledge, SOAP is the first formally verified authentication ceremony for messaging applications that works remotely and does not require users to work with cryptographic objects like keys or fingerprints. By leveraging an existing and widely used standard, SOAP is easy to implement and can be used immediately with any IdP that already supports OpenID Connect. Finally, SOAP allows for authentication in useful ways, that are impossible using conventional solutions, which we explicate in two further use cases.

Use Case 1: Social Authentication as a Second Factor One may question SOAP's value whenever users cannot authenticate their chat partner's identities because they have no relationship to these identities. For example, someone who has never received an e-mail from a given Outlook address would be unable to verify that e-mail address as truly belonging to the person in question without further interaction.

In such cases, SOAP is still valuable as it can serve as a second factor, raising the bar for compromise. If one of your contacts authenticated themselves as in control of two accounts, and you are prompted that this contact's public key changed, you can check whether the "new" contact still controls both these accounts. This information can help to distinguish a public key maliciously associated to your contact's profile from a legitimate, fresh public key after key-rollover.

Use Case 2: Native Digital Authentication For some online interactions, users do not base the identification of their chat partners in the physical world, but rather in the digital world. For example, in the physical world, I might like to authenticate my chat partner as "my colleague Alice, who I eat lunch with every day." In contrast, in the digital world, I might like to authenticate my chat partner as "the open-source maintainer Alice123 on GitHub, who I have never met in real life, but writes beautiful JavaScript." In the latter case, SOAP promises to seamlessly bootstrap a secure communication channel from such a pre-existing relationship.

Structure We proceed with additional problem motivation in Section 2. In Section 3, we present SOAP's design idea, define our security goal, and provide our threat model. Section 4 explains SOAP's design, Section 5 presents our security analysis, and Section 6 reports on our two prototypes. Finally, Section 7 compares SOAP to related work and sender correspondence to existing notions of authentication and designs.

Artifacts Our formal model and proofs (Sec. 5), and prototypes (Sec. 6) are available at: `https://soap-wg.github.io/sources`. Both prototypes come with source files and compiled versions, e.g., .apk files for the Signal prototype.

2 Problem Motivation

Both Signal and WhatsApp allow their users to communicate without authenticating their chat partners. By default, users rely on the authentication performed by the application provider during registration and that the application's key server correctly reports other users' public keys to them. When registering, users first register a public key at the key

[1]The web-based prototype is hosted at `https://soap-proto.net`, and a video demo of the Signal prototype can be viewed at `https://youtu.be/Ip_RAF8PRrM`.

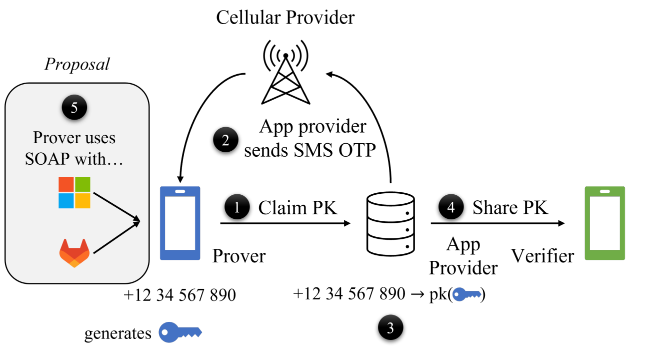


Figure 2: Steps of the prover to register at the messaging application and share their public key with other users. Our proposal is to additionally authenticate using multiple IdPs, here Microsoft and GitLab.

server (Step 1, Fig. 2). For that, they generate a public/private key pair locally and claim that the respective public key is associated with their phone number. The application then sends the user an SMS one-time password (OTP) to verify that the user controls the phone number (Step 2). When the user enters the correct OTP into the app, the key server associates the user's public key with their phone number (Step 3). Afterwards, the Signal server shares the user's public key with other users (Step 4).

This key distribution practice requires a high degree of trust in the security of SMSes and the application servers themselves. An adversary can eavesdrop or impersonate a user by compromising that user's cellular provider *or* the key server. More specifically, SMS-based attacks like SIM swaps [26] allow for impersonation, and by compromising a key server an adversary can eavesdrop on users as an MITM. Users must trust their messaging application and thus their messaging provider in general. But it is one thing to trust a provider to not implement a backdoor in a messaging application, and another to trust that their key servers are never compromised by insiders, attackers, or force by government authorities.

The breach of Signal's SMS OTP provider Twilio [46] calls into question whether the considerable trust placed in Signal's registration procedure is warranted. Through social engineering, attackers gained read access to SMS OTPs, and re-registered phone numbers, one of which belonged to a prominent journalist [20].

To prevent such attacks, Signal and WhatsApp users can compare *safety numbers* [36, 52] with their chat partners. Safety numbers are fingerprints of the two conversation participants' public keys. If two chat partners' safety numbers match, they are using the same public keys, and hence there is no MITM. To compare safety numbers though, users must rely on a trusted out-of-band channel, for example, the in-person scanning of the QR codes that encode these numbers.

In Signal's Android app, the screen for safety number verification displays the safety number as a QR code and numerically. Signal users have four options: (i) They can mark the conversation as verified. (ii) They can tap the QR code to scan their partner's QR code. (iii) They can share the safety number using a separate out-of-band channel. To do this, they click on the appropriate button, which launches the sharing menu of the phone's operating system. (iv) Finally, they can press on the safety number itself to either copy it to the clipboard or to compare it with the clipboard's contents.

Although safety numbers provide a secure authentication ceremony, their actual benefit is questionable. In incidents like the Twilio breach, users are unlikely to reauthenticate their session in a timely way by scanning QR codes. Signal notified the impersonated journalist's contacts when the attackers associated a malicious public key with the journalist's phone number. However, to successfully thwart this attack, users must (i) notice and understand this warning, and (ii) compare safety numbers in-person, which requires physical proximity, or (iii) compare safety numbers using an ad-hoc out of band channel, which must be first agreed upon – while possibly talking to an attacker. But even if users were to engage in these authentication ceremonies, various studies suggest that an attack is still very likely to succeed.

To start with, [24, 41, 49] reported that, lacking explicit instructions, only around 15%-25% of user study participants manage to successfully authenticate their chat partners in person or using phone calls. When receiving instructions, these numbers rise to 75%-80%. Remarkably, only around 50% of [24]'s participants indicated that they would perform the authentication ceremony again in the future, even after its importance was explained to them. Another study investigated the usability of SMS-based authentication ceremonies supported by Signal's share button in the safety screen's top right corner [42]. The authors found that in 40%-60% of the cases, a difference in safety numbers went unnoticed. Notably, the study considered a best-case scenario by recruiting educated, technology-savvy users who were explicitly instructed to compare safety numbers.

The aforementioned user studies show that many users cannot correctly compare safety numbers and, more critically, they do not want to. To make matters even worse, none of these studies accounts for how few users initiate these ceremonies in the first place. The share of authenticated messaging sessions is likely much lower than the above figures suggest.

Users can also protect their own accounts using a *registration lock*, provided by both WhatsApp and Signal [2, 43]. When activating the registration lock, users configure a PIN or password that must be entered when attempting to associate a new public key with a phone number. This, for the most part, rules out SMS-based attacks. If users (or attackers) cannot provide the PIN, an inactivity of 7 days is required to disable the registration lock. Registration locks, though, still require trust in the application's key server, and, more critically, users cannot verify whether their chat partners utilize them. I can

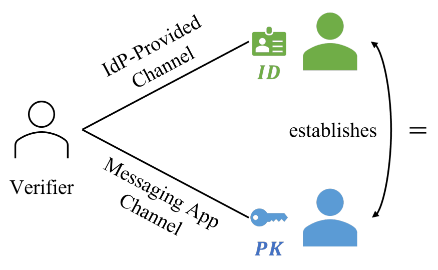


Figure 3: Social authentication establishes for a verifier that the digital identity *ID* and messaging application public key *PK* are controlled by the same person.

defend myself against impersonation attacks on my identity, but I cannot know whether my peers are impersonated or MITM-ed.

3 A Social Authentication Protocol

To address all aforementioned issues of modern messaging applications, we propose SOAP and present its security goals, its design idea, and our threat model.

3.1 Design Goals

The aim of social authentication is for a verifier to authenticate a prover (the verifier's chat partner) using one or more digital identities. Social authentication is appealing as: (i) many users have pre-existing relationships on social media, and (ii) by linking their social media presence to a different account, they can transfer the relationship from one medium to another. Note that these two notions are independent: even if you have no relationship with a given social media account, you could still be convinced that you are talking to the account holder. Formalizing this intuition, we define the security property of sender correspondence which, as we shall later see, is an authentication property.

Authentication Property (Sender correspondence)**.** A protocol P guarantees a verifier *sender correspondence* between two pseudonyms A and B if, whenever P successfully terminates, then all messages that appear to have been sent by A and all messages that appear to have been sent by B were sent by the same user.

Social authentication is simply an instantiation of sender correspondence, where the pseudonym A is the messaging application public key PK and B is an IdP-controlled digital identity ID. We define sender correspondence in more general terms than social authentication because sender correspondence finds application in other protocols, as we will discuss in Section 7.2. We illustrate social authentication's security guarantees in Figure 3.

In addition to social authentication, SOAP was also designed to provide privacy. For example, IdPs cannot learn with whom their users communicate. We define SOAP's privacy property in terms of the allowed leakage to an IdP. In particular, IdPs neither learn who the prover authenticates to nor which other IdPs the prover uses.

Privacy Property. IdPs can only learn that SOAP users (i) use SOAP, (ii) the messaging applications where they use it, and (iii) when they use it.

3.2 Design Idea

SOAP, our Social Authentication Protocol, works as follows. The prover requests an OpenID Connect identity token at an IdP and submits a hashed-and-salted conversation's safety number with that request. The IdP will then issue a token that includes a signature on the safety number and one of the prover's digital identities. At its core, this signature enables SOAP to provide social authentication and hashing-and-salting the safety number provides privacy. The prover forwards the token to the verifier, whose messaging application verifies it cryptographically and displays the prover's identity if all checks pass. In particular, this means that neither the verifier nor the prover must interact with cryptographic objects such as cryptographic keys or fingerprints thereof. In practice, users must run SOAP once per IdP to authenticate themselves to one contact, and only need to rerun SOAP should their long-term key material change.

We propose to run SOAP with multiple IdPs (as shown in Fig. 2), which substantially improves user security compared to Signal's and WhatsApp's default security. To the user, these multiple runs of SOAP (for multiple IdPs) will appear as one, which will become clear when we explain our prototypes in Section 6. Recall that, by default, a user's account can be attacked under the following condition: compromise the application's key server or compromise the cellular provider while the registration lock is not enabled (Sec. 2). Now suppose the prover runs SOAP (Step 5, Fig. 2), both with Microsoft and GitLab as the IdP. Both protocol runs are independent and, when completed successfully, the verifier's app will display both the prover's Microsoft and GitLab identity.

It is now much harder to impersonate or MITM the prover: the adversary must compromise both Microsoft *and* GitLab *and* either the prover's cellular provider *or* the application's key server. Critically, the compromise of the application's key server no longer suffices, and in contrast to the registration lock, users can authenticate their chat partners and need not rely on their chat partners activating the registration lock.

3.3 Threat Model

SOAP's security properties hold against two kinds of adversaries: We establish social authentication against an active network adversary and privacy against a malicious IdP. Whereas the social authentication-adversary can read, intercept, reorder, and replay all messages, the malicious IdP can do the same but only with messages sent to it directly. For example, the malicious IdP cannot observe whether the prover forwards tokens to the verifier. We restrict our analysis of SOAP's privacy property to a malicious IdP as we wish to show that adding IdPs to the messaging application ecosystem does not threaten user privacy.

Our threat model permits the compromise of the messaging application's key server, the leaking of OpenID Connect requests to IdPs, and the compromise of some of the IdPs integrated into the messaging application. Notably, we make no assumptions on user-behavior other than that users do not leak their credentials. Users may click any link sent to them, whenever an IdP asks them for consent, they may provide it, and whenever an IdP asks them to log in, they may do so, even if the adversary triggered that query. We only limit our adversaries' capabilities in the following ways:

1. Adversaries are bound by the security properties of the cryptographic primitives used and the TLS and Signal protocols (both Signal and WhatsApp use the Signal protocol). For example, adversaries can neither invert cryptographic hash functions nor eavesdrop on a TLS session.
2. User credentials at IdPs are uncompromised.
3. Whenever a user authenticates via a given IdP, that IdP's signing keys and TLS certificates are uncompromised.
4. The messaging application and the user's web browser are uncompromised. In particular, the parameters of browser redirects to the messaging application remain confidential until they expire, and messaging application key material remains uncompromised.

The necessity for Assumptions 1-3 is self-evident. Regarding Assumption 4, it should be clear that we must require the messaging application and the user's browser to be uncompromised. We will discuss why we require the parameters of browser redirects to remain confidential later in Section 5.2.3 as this assumption requires a deeper understanding of SOAP's design. Finally, we require that messaging application key material remains uncompromised because an adversary could otherwise launch a trivial impersonation attack. They could run SOAP with a victim's safety number as a parameter and use an attacker-controlled account to log in. The adversary could then inject that token into the conversation between the victim and one of the victim's contacts as they compromised the key material. However, note that (i) messaging secret keys are stored on-device only, i.e., a compromise of these keys likely comes with a compromise of the messaging application, and that (ii) the Signal protocol offers strong security guarantees against such key compromise, namely forward secrecy and post-compromise security [9]. SOAP was not designed to defend against key compromise, but rather against impersonation by associating malicious keys with pseudonyms such as

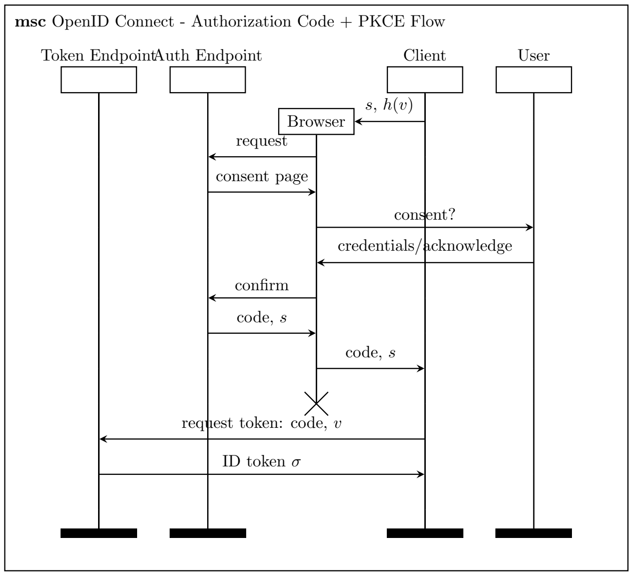


Figure 4: OpenID Connect authorization code flow with PKCE. $h(v)$ is the commitment to a random value v as specified by PKCE and s is the state parameter.

in the Twilio incident.

4 Protocol Design

In this section, we present SOAP's design in detail. SOAP utilizes the OpenID Connect protocol [38, 39] to facilitate adoption, which we explain first.

4.1 OpenID Connect

OpenID Connect [38, 39] is an authentication protocol that itself builds on the OAuth 2.0 authorization protocol [23]. OpenID Connect is used to implement the well-known *"Login with Google/Microsoft/Apple/... "* buttons. OpenID Connect involves three parties: a *user*, an *IdP* managing the user's identity, and a *relying party* seeking to authenticate the user. The OpenID Connect protocol is executed by a *client* operated by the relying party. At the end of a successful protocol run, the client receives an *ID token* through a browser redirect from the IdP. The ID token is a cryptographically signed message, proving that the IdP authenticated the respective user, and which the client can use to identify the user. Prior to issuing requests, relying parties must register at the respective IdP. During registration, relying parties whitelist redirect URLs, and IdPs issue *client IDs*.

OpenID Connect supports multiple *flows*, which are protocol variants aiming at specific types of software clients. We use the authorization code flow extended with the Proof Key for Code Exchange (PKCE) standard [40], which is currently recommended as best practice for clients such as mobile applications [31]. The authorization code flow with PKCE implements a commitment reveal scheme, which we depict in Figure 4 and works as follows. The client first issues an authorization request by launching the device's browser at a specific URL called the *authorization endpoint*. The request encodes various parameters in the URL: a client ID, redirect URL, code challenge (the commitment, which is the hash of a random string), and optionally a state parameter and a nonce. The state parameter and nonce protect against replay and cross-side request forgery (CSRF) attacks.

After receiving an authorization request, the IdP verifies the redirect URL as whitelisted for the given client ID, authenticates the user, and asks the user for consent. Users usually authenticate by logging in, or through a session cookie already stored in their browser. Depending on the user's history with the IdP, the user may not need to grant consent.

Once a user consents to logging in, the IdP forwards the browser to the redirect URL given in the request. In the redirect URL, the IdP encodes an *authorization code* and the state parameter sent earlier. The client verifies that the state matches the state it previously issued, and exchanges the authorization code for an ID token. The client does this by sending a POST request to the IdP's *token endpoint*, including the authorization code and the *code verifier*. The code verifier opens the code challenge commitment sent earlier to the IdP. This allows the IdP to determine that the ID token is requested from the same client that issued the initial request.

The ID token is encoded as a JSON Web Signature (JWS), which is a signed object that maps keys to values. Amongst other values, the object includes the issuer, the audience (identifying the client), the subject (the user who was authenticated), the nonce, and a validity period. Usually, ID tokens are short-lived, with lifetimes typically ranging from two minutes to two hours. According to the OpenID Connect specification [38], ID tokens must only be accepted by the intended audience. This prevents a service to which a user logged in from using the ID token with another service.

4.2 Protocol Description

We next present SOAP, which provides an automated social authentication ceremony that requires no IdP adoption.[2] We suggest running SOAP with multiple IdPs (Section 3.2), and that applications provide a user experience where these multiple runs appear as one. Users are then only required to select IdPs and to provide consent. We base SOAP on the OpenID Connect authorization code flow with PKCE [38]. This makes its adoption straightforward, and it can be used immediately with any IdP that supports OpenID Connect.

The idea behind SOAP is to utilize the signed ID token to convince the verifier that the prover (i) authenticated to the IdP using the identity included in the token, and (ii) submitted their session's safety number when logging in. The verifier must only check that the token was intended for them by comparing the safety number included in the token to the safety number of the session through which they received the token. SOAP consists of three steps for the prover (request, validation, and forwarding) and one step for the verifier (validation). Figure 5 sketches our protocol.

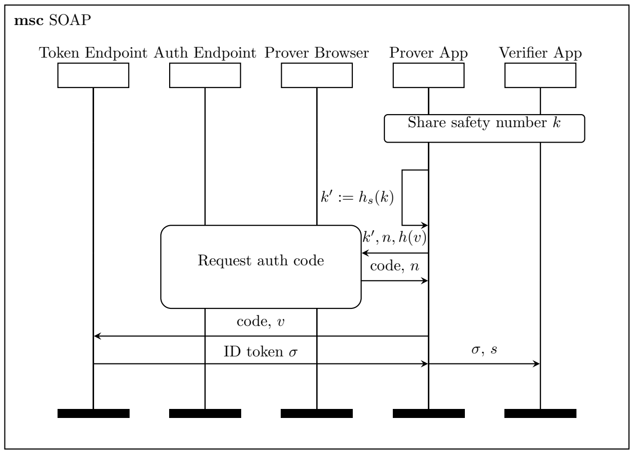


Figure 5: SOAP running between the prover, the IdP, and the verifier. Here, h_s is a password hashing algorithm using a salt s and h is SHA-256 as specified by PKCE. The application randomly samples a code verifier v, a salt s, and nonce n. Finally, σ is the OpenID Connect token, which is forwarded to the verifier and includes a signature on $h_s(k,s)$ and n.

To start a run of SOAP, the prover's messaging application prepares the request. It generates two random values: a code verifier and a nonce n. The application then uses a secure password-hashing algorithm to calculate a salted hash h of the safety number. This hash serves to blind the safety number to the IdP. To defend against CSRFs attacks, the application stores the nonce, the salt, the IdP's ID, and the code verifier as the most recently issued request. Then, the application launches the authorization code flow, passing n as the nonce and as the state, h, the code challenge, and a redirect URL. This redirect URL must be distinct for each IdP and should use the HTTPS scheme (preferred) or a custom scheme. The application then launches the system's browser with the request URL, which takes the prover to the consent/login page.

If the prover consents, the IdP redirects the browser back to the application. The redirect passes the application a state parameter and an authorization code, which can be exchanged for an ID token. Before the application uses the authorization code, it must verify that the state value it just received equals the nonce stored with the most recently issued request, and that the response originates from the expected IdP.

If both checks pass, the application uses the authorization code and stored code verifier to request the ID token from the IdP. It verifies the token's signature and fields, e.g., that it correctly encodes the hashed and salted safety number. Finally,

[2]For a technical description, see Appendix A.

the application clears its storage for the most recently issued request, and stores the nonce in a replay cache. Recording nonces defends against reflection attacks; the description of our web-based prototype in Section 6.1 illustrates this threat.

The application forwards the ID token and the salt to the verifier and the verifier applies the same checks. It also verifies that the safety number encoded in the ID token encodes the prover and verifier's keys, and that it did not request this token itself by looking up the stored nonces from runs where it was the prover. If these checks pass, the verifier can obtain the sender's identity from the token.

While Figure 5 may suggest that SOAP simply "calls OpenID Connect," previous work [31, 17, 19] highlighted many subtleties in implementing an OpenID Connect-based protocol securely. Hence, we next present our formal proof that SOAP indeed provides social authentication.

5 Security Analysis

SOAP is designed to implement social authentication and to protect user privacy, as defined in Section 3. In the following, we prove its security using the protocol-verifier Tamarin [34]. Tamarin has been used to verify many critical protocols, such as 5G-AKA, TLS 1.3, and the credit card protocol EMV [5, 10, 11]. We first introduce Tamarin, and then present the formal proofs of SOAP's social authentication and privacy property.

5.1 The Tamarin Prover

Tamarin [34] is an infinite-state protocol verifier that supports both fully automated and guided proof construction in the symbolic model. Tamarin models have three parts: rules modeling protocol steps, an equational theory modeling cryptographic primitives, and lemmas modeling protocol properties.

Rules have the form `l --lbl-> r`, where `l` reads from the current state, `r` updates the new state, and `lbl` labels this transition for reference in properties. Typically, `l` encodes reading incoming messages and parts of the participant's state and `r` encodes sending messages and updating the participant's state. For example, the rule below models that a participant receives a message, looks up a key, and sends the message symmetrically encrypted under the respective key. The label adds `Enc(m)` to the protocol trace for reference in properties.

```
[In(m),!K(k)] --[Enc(m)]-> [Out(senc(m,k))]
```

Equations model cryptographic primitives. For example, the equation `sdec(senc(m, k), k) = m` models that the symmetric decryption of a ciphertext using the correct key yields the respective plain text. Equational theories allow one to formalize a strong network adversary that can derive new knowledge from previously sent messages. For example, if a participant sent both a symmetrically encrypted message and the respective key, the aforementioned equation would allow the adversary to learn that message's content.

Tamarin supports two types of properties: universally quantified and existentially quantified trace properties. Tamarin verifies both kinds of properties similarly. Universally quantified properties are first negated, whereas existentially properties are left as is. Then, Tamarin tries to construct a trace that satisfies the resulting property using backwards constraint solving. For universally quantified properties, Tamarin hence tries to construct a counterexample, and for existentially quantified properties, Tamarin tries to construct a positive example.

5.2 Social Authentication

We next provide high-level intuition on why SOAP provides social authentication and then describe our formal proof.

5.2.1 Informal Analysis

We previously defined social authentication as an implication: "If the verifier associates an account A with the public key PK and received a message from each of these pseudonyms, then these messages were sent by the same party."

There are two ways to violate social authentication. Either, there is no send event for one of the messages or the send events have different senders. In our formal model, we modelled the message-exchange channels as authentic (i.e., given a receive event, there will be a send event) and thus focus on the latter case. Given SOAP's design, the attacker can achieve this by either making the prover send a malicious token through the messaging application, linking an attacker-chosen account to the prover's messaging channel (identity substitution), or making the prover authenticate in an attacker-controlled OpenID Connect flow, linking the prover's account to an attacker-chosen channel (impersonation).

Let us first focus on impersonation attacks, and presume Eve attempts to attack Alice. This means that Eve has a messaging session with Alice and wants to convince Alice that she, Eve, controls one or more of Bob's accounts at IdPs. To achieve this, Eve must send a token that includes a reference to Bob's account and the safety number of Eve's and Alice's public keys. As we assume that Bob's account is uncompromised (Assumption 2, Sec. 3.3), Eve must craft a malicious OpenID Connect request and forward it to Bob. This is straightforward as it requires nothing more than convincing Bob to click on a malicious link encoding such a request. However, Eve cannot obtain a token from this: Bob's application will discard the authorization response if it did not issue the request itself. Moreover, if it did issue the request itself, it would include one of Bob's safety number and not Alice and Eve's.

Similarly, Eve cannot launch an identity substitution attack. Eve would need to run SOAP herself, using a victim's safety number as parameter, log in with her own account, and then make the victim's application accept the resulting browser forward. However, the application would again discard that forward as it did not issue the corresponding request.

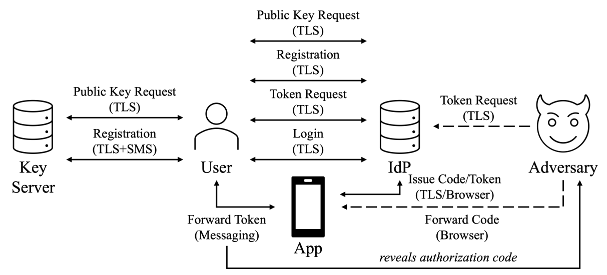


Figure 6: Sketch of our formal model of SOAP. Arrows indicate message exchanges, denoted with the respective channels used. Dashed arrows indicate that the adversary can initiate the respective request on the user's behalf. We omit the IdP-controlled bulletin board and the messaging channel.

This argument spells out the main idea behind SOAP's security. In reality, the security of protocols based on OAuth 2.0 and OpenID Connect is much more subtle. Attackers can in general access authorization code responses through other means than capturing the redirects. For example, [17] first described the IdP-mixup attack, in which applications leak an authorization code by sending it to the wrong IdP. Therefore, we formally evaluate SOAP's security next.

5.2.2 Formal Proofs

Our formal model comprehensively captures SOAP and its heterogeneous environment. Beyond the messaging application and TLS, we modelled all security-critical aspects and auxiliary protocols such as public key requests and distribution. Figure 6 depicts a sketch of what we modelled. Our model permits arbitrarily many participants to communicate with each other in arbitrarily many parallel protocol sessions. The adversary can corrupt any party in a fine-grained manner, e.g., a user's account could be compromised independently of their messaging long term keys, only constrained by our security assumptions from Section 3.3. For example, IdP corruption is possible for every IdP except the one with which the prover intends to authenticate themselves.

More specifically, our model includes channels for SMS, TLS, browser redirects, and messaging applications with different security properties and distinct keys. We also modelled communication associated with IdP-controlled pseudonyms (usernames) as a bulletin board where users can post messages associated with their pseudonym publicly after the IdP authenticates them using a password. We modelled SMSes as insecure and the messaging application as secure (confidential and authentic). We modelled TLS as a secure channel without client authentication, i.e., the adversary can always initiate new sessions with servers.

The adversary can compromise any TLS server, which allows it to read client queries and respond to them. TLS queries can have one of two methods, GET and POST, whereby GET requests can be initiated by the adversary on a user's behalf, modelling that the adversary can trick users into clicking any link. In practice, this allows the adversary to launch the OpenID Connect protocol at various points (e.g., initial request and code forwarding) for non-compromised clients. Browser redirects are modelled as GET requests using an existing session to connect to a new server, initiated by the previous server. This allows us to model, e.g., the redirect to a mobile application by modelling that application as a server.

We modelled the messaging provider, messaging application, end users, and IdPs as different parties. Our model includes SOAP itself, messaging application registration (including SMS OTP verification), IdP account registration, as well as messaging key server and IdP public key requests and responses. Moreover, we fully modelled OpenID Connect and we make no assumptions on this protocol's security.

Within this model, we prove that SOAP implements social authentication. Figure 7 shows our Tamarin specification formalizing social authentication as a trace property. It has three parts. Lines 2-7 formalize social authentication: If the verifier associates two pseudonyms with each other (`Correspond`), then all messages received from those two pseudonyms (`ReceiveMessaging`/`IdP`) originate from the same sender. We formalize the latter by showing that there exist two send events (`SendMessaging`/`IdP`) for which the sender (`Sender`) is the same party `s`.

Lines 8-15 formalize Assumptions 2-4 from our threat model (Sec. 3.3). Assumption 1 is covered implicitly as Tamarin operates in the symbolic model. There is just one subtle difference from social authentication as presented earlier. Namely, since the messaging channel is not just sender-authenticated, but also receiver-authenticated, we can include the recipient's pseudonym `rcvKey` in the receive and send event in lines 3 and 6. This means that our formalization of social authentication is stronger than sender correspondence.

The size and complexity of our model put its security properties out of reach for fully automated verification. For example, modelling that browser redirects remain confidential *until they expire* (Assumption 4) proved to be challenging. In part, we modelled this assumption by revealing authorization codes to the adversary after receiving the respective ID token. This led to infinite looping in Tamarin's proof construction, which we avoided by proving an inductive, auxiliary lemma showing that authorization codes can only be used once. In total, we verified nine auxiliary lemmas and programmed custom proof heuristics to aid Tamarin's proof construction.

5.2.3 Discussion of Threat Model

With our formal analysis of the authentication property completed, we briefly return to Assumption 4 of our threat model, where we require that the parameters of browser redirects to the messaging application remain confidential. Without this assumption, the adversary could easily obtain an identity to-

```
All v sendKey rcvKey m1 idp acc m2 #t #r1 #r2.
    ( Correspond(v, sendKey, idp, acc) @ #t
    & ReceiveMessaging(sendKey, rcvKey, m1) @ #r1
    & ReceiveIdP(idp, acc, m2) @ #r2)
==> ( (Ex s #x1 #x2.
      ( SendMessaging(sendKey, rcvKey, m1) @ #x1 & Sender(s) @ #x1 & #x1 < #r1
      & SendIdP(idp, acc, m2) @ #x2 & Sender(s) @ #x2 & #x2 < #r2))
    | (Ex p #x. CompromisedAccount(p, idp, acc) @ #x)
    | (Ex #x. CompromisedIdP(idp) @ #x)
    | (Ex #x. CompromisedDomain(idp) @ #x)
    | (Ex app redirectURL #x #y #z.
        IsMessagingApp(app) @ #x
      & IsRedirectURL(idp, app, redirectURL) @ #y
      & CompromisedDomain(redirectURL) @ #z)
    | (Ex p #x. CompromisedMessaging(p, sendKey) @ #x))
```

Figure 7: Formalization of social authentication (Sec. 3), also encoding the threat model (Sec. 3.3). Note that the two `Sender` facts are bound to the respective `SendMessaging` and `SendIdP` events, as they occur at the same time points (`x1` and `x2`).

ken that binds an attacker-chosen safety number to a victim's account. To achieve this, they would only need to trick their victim into clicking a SOAP-request link that includes a malicious safety number as parameter. As soon as the victim logs in and consents (which they will do under our liberal threat model), the adversary could learn the authorization code from the redirect URL and request the ID token themselves.

Users can theoretically protect themselves from this attack by only granting consent to requests they initiated themselves. However, (i) we find it unrealistic to assume users are resistant to social engineering, and (ii) we experienced during our prototype development that some IdPs immediately acknowledge requests without user involvement whenever the user was already logged in and had previously granted consent. In this case, users could be attacked easily, e.g., with malicious URLs obfuscated with a URL shortener.

In practice, though, capturing redirects requires the adversary to have access to a user's browsing history while an attack is launched. This requires the compromise of the user's browser, or the installation of a malicious application handler on the user's device. In case of a compromised browser, it is nigh impossible to protect the user's account credentials at the same time. To address malicious application handlers, we recommend that the application should use HTTPS redirect URLs as described in Section 4. With HTTPS URLs, application developers can utilize the security features provided by modern operating systems to ensure that authorization codes do not leak. For example, both Windows and Android support applications to handle HTTPS URLs, but only as long as these applications have been delegated to do so by the respective URL [22, 13]. This way, application developers can rely on the security features provided by the Web PKI to protect authorization responses. Without HTTPS URLs, an attacker would still need to install a malicious application handler and in turn a malicious application on their victim's device to capture redirects to custom schemes.

5.3 Privacy

SOAP protects users' privacy against the IdPs in that it only reveals that a prover is using SOAP, which messaging application is being used, and at what times SOAP is used. We formally proved SOAP's privacy as an observational equivalence property using Tamarin. Implementing our threat model (Sec. 3.3), we proved privacy in a simplified model (compared to the model presented in the previous section) that only includes communication between users and the IdP. We modeled the malicious IdP as the adversary and consequently replaced TLS with an insecure channel. Our observational equivalence property shows that IdPs cannot distinguish protocol runs where a user submits the correct salted-and-hashed prover/verifier safety number from runs where the user submits a different safety number. In the symbolic model, our privacy property is straightforward as we will lay out next.

SOAP only includes two requests to the IdP's servers, and these requests include the following parameters: the messaging application ID with the IdP, a redirect URL, a nonce, the code challenge (the hashed code verifier), the code verifier, the authorization code, and the salted-and-hashed safety number. The messaging application's ID and redirect URL reveal the messaging application and that the prover is using SOAP. The nonce, the code challenge, and the code verifier are randomly generated values that change with every request, and hence, leak nothing about the prover. The authorization code is issued by the IdP and therefore allows the IdP to connect the initial authorization request with the token request. Finally, the salted-and-hashed safety number leaks nothing about the prover under the assumption that the IdP cannot break cryptographic primitives such as salt-and-hashing, and because the salt is only shared with the verifier.

In theory, a verifier could share the salt with an IdP to reveal that the prover intended to communicate with the verifier. However, a verifier could leak this information even without SOAP. The IdP could also attempt to correlate public key requests with issued tokens. Fetching public keys, however, is not part of SOAP itself as applications will likely cache public keys. Additionally, the OpenID Connect specification [39] requires that IdPs distribute their public keys publicly and application-independently via HTTPS. For both of these reasons, we deem such a correlation to be practically infeasible. To summarize: SOAP prevents the surveillance by an IdP of its user's contact graphs in messaging applications.

6 Implementation

We implemented our proposal in two prototypes: as a stand-alone web application[3] and as a fork of the Signal open-source Android application. Both prototypes support GitLab and Microsoft as IdPs. The stand-alone version does not require messaging application adoption but requires more user interaction. In its current design, it can be used to associate arbitrary statements to a user's account.

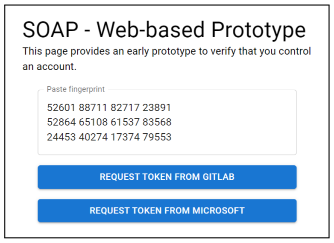
Our prototypes demonstrate that social authentication can be realized practically, with provable security guarantees, and without OpenID Connect-IdP adoption. Our web-based prototype shows that social authentication can be implemented even without messaging application adoption. Library support for OpenID Connect is abundant and, hence, adoption is straightforward. One of the authors could implement an initial prototype within a day. With just a few clicks and in a couple of seconds, users can verifiably share their identity.

6.1 Web-based Prototype

Figure 8 depicts the interaction with our web-based prototype. When started, the application renders a text input and a list of IdPs to select from (Fig. 8a). After the prover selects an IdP, the application assumes that the input contains the safety number to authenticate, and initiates SOAP. After they complete the OpenID Connect flow (see Sec. 4.1), the prover is forwarded to our application, which requests the ID token (Fig. 8b). The web application verifies the ID token, and, if all checks pass, displays a success message. The prover can then copy a link to send to their chat partners.

This link encodes the ID token and everything needed to verify it. When the verifier clicks the link, the application verifies the token, and if all checks pass, displays the prover's identity, the IdP, and the safety number. The verifier can copy this safety number to their clipboard. The Signal Android

[3] The prototype is hosted at https://soap-proto.net.

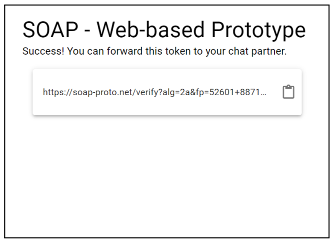

SOAP - Web-based Prototype

This page provides an early prototype to verify that you control an account.

Paste fingerprint

52601 88711 82717 23891
52864 65108 61537 83568
24453 40274 17374 79553

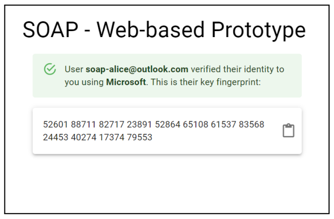
(a) Prover pastes safety number into the application and selects one of two IdPs.


SOAP - Web-based Prototype

Success! You can forward this token to your chat partner.

https://soap-proto.net/verify?alg=2a&fp=52601+8871...

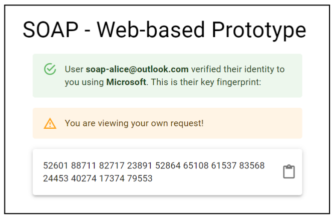
(b) Prover receives ID token and is presented a URL to forward to their chat partner.


SOAP - Web-based Prototype

User **soap-alice@outlook.com** verified their identity to you using **Microsoft**. This is their key fingerprint:

52601 88711 82717 23891 52864 65108 61537 83568
24453 40274 17374 79553

(c) Verifier sees the prover's identity and safety number when clicking the link.


SOAP - Web-based Prototype

User **soap-alice@outlook.com** verified their identity to you using **Microsoft**. This is their key fingerprint:

You are viewing your own request!

52601 88711 82717 23891 52864 65108 61537 83568
24453 40274 17374 79553

(d) Prover sees a warning when clicking a link generated within the same browser.

Figure 8: Screenshots depicting the web application prototype. The prover must initiate this flow for each IdP.

application offers its users to compare the safety number with the clipboard, giving the verifier an easy way to check the safety number (Sec. 2).

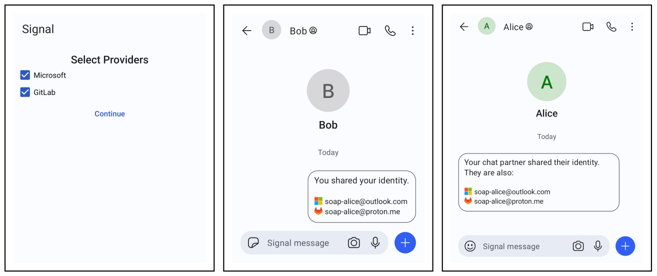
Note that the application will render a success message to anyone clicking a link containing an ID token. Only the user can determine whether the safety numbers match. This allows for social engineering attacks whenever users click links they earlier forwarded themselves. To mitigate this threat, the web application warns users that they issued this request themselves whenever they click a link that was issued within the same browser. Figure 8d shows the view after clicking on a URL that encodes an ID token.

6.2 Signal Prototype

Our Signal prototype, depicted in Figure 9, provides a more streamlined experience compared to the web-based prototype.[4] In particular, the Signal prototype requires significantly less interaction from the prover and no interaction from the verifier. Users need not actively examine and insert safety numbers, and flows need not be initiated for each IdP.

When a user wishes to authenticate themselves to their chat partner (becoming a prover), they must first select the new "Authenticate" option within the attachment menu. Next, the prover selects all the IdPs they wish to authenticate themselves with (Fig. 9a). When they press continue, they will run SOAP for each of the IdPs. The application verifies all tokens and

[4] A video demo of the Signal prototype can be viewed at https://youtu.be/Ip_RAF8PRrM.



You shared your identity.
soap-alice@outlook.com
soap-alice@proton.me

Your chat partner shared their identity.
They are also:
soap-alice@outlook.com
soap-alice@proton.me

(a) Prover selects the IdPs to authenticate with.
(b) Prover sees that they shared their identities.
(c) Verifier sees the prover's identities.

Figure 9: Screenshots depicting the Signal prototype. The interaction with our prototype is depicted from left to right.

displays a distinctly styled message to both the prover and verifier that shows the shared identities (Fig. 9b, 9c).

Our Signal prototype demonstrates that SOAP is a practical design that requires little user interaction. While it is adequate for a proof-of-concept, we suggest further enhancements before it is deployed in production. First, previously performed social authentications should be recorded as such in the application. The application could display a contact's identities within the chat header, rather than mark them as "Verified." This would additionally highlight that our proposal is intended to augment in-person safety number comparison, not supersede it. Second, the messaging application could compare the identities provided through SOAP with identities linked to the contact's address book entry on the smartphone to automatically combat impersonation attacks. Messaging applications are usually granted access to address books anyways, providing a straightforward means to automate social authentication further.

6.3 Development

The web-based prototype was written as a single-page JavaScript application in React [37], i.e., all code runs in the user's browser. The prototype consists of roughly 500 lines of code and the first version was developed within a day by one of this paper's authors. In contrast to the web-based prototype, the Signal prototype's development was more involved and required around three person weeks. Understanding the meagerly documented Signal Android codebase demanded most of the time. We changed 21 files and added around 1000 lines of Java and Kotlin code for the application logic, and also changed 17 files with around 200 additional lines of code for configuration changes, like layout and localization.

During prototype development, we noticed that not all IdPs support OpenID Connect ideally for our use case. Microsoft, for example, does not support HTTPS redirect URLs for Android applications. We tried working around this restriction by registering our Android application as a Single-Page Application, which permitted us to configure an HTTPS redirect URL. However, requesting the ID token failed as cross-origin request headers were missing.

GitLab supports HTTPS redirects and does not distinguish the types of applications upon registration. However, GitLab does not ask for consent again after the user consented once to log in. This causes HTTPS redirects to Android applications to fail. A Chrome policy requires user interaction in order to redirect users to Android applications through HTTPS URLs [3]. This left us with using custom schemes, e.g., `auth://`, in redirect URLs to support GitLab as an IdP.

Finally, Google neither allows one to configure a redirect URL nor lists a redirect URL when registering Android applications as an OpenID Connect client. In this way, Google hides their OpenID Connect API. We suspect this practice is intended to force developers to implement "Sign-In with Google" using Google's SDK [44]. This SDK need not be configured with a redirect URL to request ID tokens and does not allow one to specify a nonce.

These findings suggest that while SOAP does not require adapting the OpenID Connect specification, explicating our use case in the OpenID Connect specification could still benefit users and developers. Currently, users will only consent to log in, and developers have to use the nonce field outside its specified intent. If OpenID Connect were to recognize user-submitted claims as a request parameter, users could grant consent to show that they control the given account and developers could use APIs supporting user-submitted claims.

6.4 Performance

The performance of both prototypes is mainly constrained by the page load times of OpenID Connect authorization endpoints, consent screens, and HTTP redirects. We timed the prototypes with a user that was logged in and had authorized our application already. In the web-based prototype, running SOAP for a single IdP was nearly instantaneous (<1 second), and in the Signal prototype, running SOAP for two IdPs consistently took less than 10 seconds.

As we described in Section 4, SOAP requires local, persistent storage to protect against replay and CSRF attacks. SOAP stores the latest issued request, which only requires a small, constant amount of space, and the nonces generated by the application. The number of nonces stored is limited by the number of SOAP sessions initiated by the user, and the nonces can be discarded after a token expires, which in our experience happens within two hours.

7 Related Work

7.1 Key Authentication

In this section, we examine three research strands aimed at improving key authentication: key transparency, authentication ceremonies within messaging applications, and social authentication.

Key Transparency CONIKS, SEEMless, Parakeet, and KTACA [35, 8, 33, 53] implement *key transparency* as an alternative to key authentication. In key transparency, providers commit to a publicly auditable key directory, and users (or rather their messaging applications) both fetch their contacts' public keys from this directory and check that the public key associated with their pseudonym in the directory matches their actual public key. Key transparency aims to obviate the need for key authentication as other users can assume that their peers would have taken action should their application detect a malicious public key associated with their pseudonym. Recently, Meta announced that WhatsApp will deploy key transparency services and published an open-source key transparency service based on SEEMless and Parakeet [29, 14].

In contrast to SOAP, key transparency has the upside that it requires no user-interaction when there is no compromise, but this comes at the cost of a significant engineering effort and can require recruiting external auditors. Moreover, key transparency only aims to protect against malicious providers and not outside attackers, such as in the Twilio incident (Sec. 2), and key transparency cannot prevent compromise but only make it detectable. Finally, key transparency provides no notion of authentication, which is desirable in its own right, e.g., to verify that one chats with the person controlling a given account on a different platform. We see SOAP as complementing key transparency and not rivaling it. Moreover, SOAP is far less complex than key transparency and, thus, it is simpler to deploy SOAP, especially for smaller organizations.

Authentication Ceremonies for Messaging Applications We already extensively compared SOAP to numerous manual authentication ceremonies in Section 2. Other researchers additionally investigated how one could improve authentication ceremonies to increase success rates. [47] proposed UI changes, and [15] proposed new ceremonies altogether, but they did not consider whether those ceremonies were secure. Both [15, 47] focus on manual verification, where users compare two pieces of information. In contrast, SOAP lifts this burden from the user and instead asks them to authenticate a set of identities.

Social Authentication The idea of authenticating users using their profiles at IdPs was pioneered by Keybase [27]. There, users can bind their social media accounts, e.g., at Twitter, to their Keybase account using so-called *proofs* [28]. Users do this by posting a signed message on a social media platform. Other users can verify that a user linked their accounts by checking that the signature was generated using the key associated with the Keybase account, and posted by the claimed social media account.

[48] coined the term "social authentication," proposed its application to Signal, and conducted a user study. The authors found that users regard social authentication as understandable, easy to use, working asynchronously and remotely (in contrast to, e.g., in-person verification), and that it enhances their security when using multiple providers. However, users gave social authentication a lower trust score than in-person verification, partially stemming from their limited understanding of the mechanism. For example, users feared that a compromise of their social media accounts could lead to a compromise of their Signal account, and they distrusted social media providers in general. Users also mentioned the risk of social engineering with fake accounts.

The authors of [48] did not implement a working prototype for social authentication. They justified this decision with the complicated process of acquiring approval from the application providers to access social media accounts. Instead, they evaluated a mock-up prototype without designing or implementing any protocol. The prototype communicated with servers storing the association of social media profiles to Signal keys, simulating a perfectly secure world. Nonetheless, the authors note that safety numbers should be posted publicly on the social media platforms, thereby disclosing the association of keys with accounts. Through such public posts, users have no control over who can associate keys with accounts.

In contrast to both of the above works, SOAP is a privacy-preserving protocol that allows for the selective disclosure of key-to-account associations. Whereas the proposal of [48] requires users to authorize Signal to access the respective social media platform, SOAP neither requires Signal to access accounts at IdPs nor vice versa. Additionally, we demonstrated SOAP's feasibility with two fully functional prototypes, and we rigorously evaluated SOAP's security.

Key Authentication with OpenID Connect Zoom's cryptography whitepaper [7] proposes "Identity Provider Attestations," which authenticate Zoom's end-to-end encryption keys using OpenID Connect in conjunction with DNS. Organizations can delegate an IdP via DNS as eligible to authenticate that organization's users. The Zoom client can then (i) upload a commitment to a user's public key at the IdP on the user's behalf using OAuth and a custom API, and (ii) request an ID token that includes that commitment. Both these steps require adoption by the IdP, which is not the case for SOAP. Moreover, Zoom's design considers the case in which an account delegates authentication to a trusted IdP. In contrast, we propose to utilize multiple IdPs, which makes it strictly more difficult to compromise a messaging account.

7.2 Sender Correspondence

We next focus on sender correspondence, our formalization of social authentication. We show that this notion has application beyond secure messaging, and we relate it to existing notions of authentication. Namely, we show that sender correspondence establishes non-injective agreement for each of the two associated pseudonyms.

7.2.1 Sender Correspondence in the Wild

To show that sender correspondence applies beyond social authentication as presented in this paper, consider the Automatic Certificate Management Environment (ACME) protocol [4, 1], powering the free certificate authority (CA) *Let's Encrypt*. ACME automates the certificate request and issuance procedure. Using ACME, CAs verify certificate requests using a challenge-response mechanism in three steps. First, a CA receives a certificate request for a given domain name, signed by a private key. Second, the CA sends a challenge to the respective public keyholder and asks them to return it using DNS or HTTP. Third, the CA verifies that they receive the signed challenge through the DNS or HTTP channel, at which point it issues a certificate for the respective public key.

The ACME protocol can be seen as establishing sender correspondence. Namely, the requesting public key is the pseudonym A and the domain name is the pseudonym B. Related literature [6, 25] verifying the ACME protocol only considered key establishment properties. Namely, they require that for any attack on ACME, the adversary must know the certificate's corresponding secret key. However, they do not consider identity misbinding attacks, where the adversary could provide the domain name that gets associated with an honest key. [4] analyzed ACME's domain validation algorithm and considered an authentication-style property. But as the authors only analyzed this one part of ACME, they did not consider ACME's overarching security goals.

In general, sender correspondence can be applied to any two channels that allow for information exchange associated to pseudonyms. An IdP-managed online version-control system like GitLab is a channel where users exchange information (commits, comments, etc.) associated with a pseudonym (usernames). Similarly, DNS is a channel where information (DNS records) is associated to pseudonyms (domain names).

7.2.2 Relationship to Other Authentication Properties

Non-Injective Agreement In his analysis of authentication properties [32], Lowe defined the notion of *non-injective agreement*, which we recast slightly as follows and formalize in Tamarin's property language below:[5]

Authentication Property (Non-injective agreement; adapted from [32])**.** A protocol guarantees a responder A *non-injective agreement* if whenever A receives a message m, apparently from initiator B, then B was previously running the protocol as the initiator, and the two agents agreed on m.

[5][32] also requires that A and B agree on the intended recipient (A), which we drop so that we can also consider only sender-authenticated protocols.

```
All R S m #tr. Receive(R, S, m) @ #tr
  ==> (Ex #ts. Send(S, m) @ #ts
        & #ts < #tr)
```

Compare this to the following formalization of sender correspondence, a simplified variant of our formalization of social authentication (Sec. 5.2.2).

```
1  All V R1 R2 PX PY mx my #ta #trx #try.
2        ( Correspond(V, PX, PY) @ #ta
3        & ReceiveChX(R1, PX, mx) @ #trx
4        & ReceiveChY(R2, PY, my) @ #try)
5    ==> (Ex S #tsx #tsy.
6            SendChX(PX, mx) @ #tsx
7          & Sender(S) @ #tsx
8          & #tsx < #trx
9          & SendChY(PY, my) @ #tsy
10         & Sender(S) @ #tsy
11         & #tsy < #try)
```

`Receive[ChX/Y](R, S, m)` models that `R` received message `m` from `S` (on channel `X` or `Y`), `Send[ChX/Y](S, m)` that `S` sent message `m` (on channel `X` or `Y`). Note that `S` could be a pseudonym. `Correspond(V, PX, PY)` formalizes that the verifier `V` identifies pseudonyms `PX` and `PY` with the same party, and `Sender(S) @ #t` that the message sent at time point `t` was sent by agent `S`.

Intuitively, sender correspondence relates to non-injective agreement for two reasons: (i) Sender correspondence only works when the two associated pseudonyms can be used for authentic communication. Otherwise, it would make little sense to "tie" them together. (ii) The formalizations of both properties are very similar: lines 3, 6, 8 and lines 4, 9, 11 exactly match our formalization of non-injective agreement.

We can express (ii) formally. Namely, sender correspondence establishes non-injective agreement on channel `X` and `Y` for the pseudonyms `PX` and `PY`. Specifically, we show that whenever there is a successful run of a protocol providing sender correspondence between `PX` and `PY`, that trace also satisfies non-injective agreement for both of these pseudonyms (formalized using the respective `ChX` and `ChY` events). If such a trace were a counterexample to non-injective agreement, e.g., for channel `X` (the other case follows symmetrically), there must be an event `ReceiveChX(R, PX, m)` for which there is no corresponding `SendChX` event. In that case, since `R1` and `mx` in the formalization of sender correspondence are universally quantified, that trace would be a counterexample to sender correspondence as well. This contradicts our assumption that the protocol provides sender correspondence.

This relationship between sender correspondence and non-injective agreement highlights that sender correspondence is a desirable authentication property. When successfully running a protocol providing sender correspondence, we know that both pseudonyms can be used for authentic communication *and* that they are controlled by the same sender.

Sender Invariance In [12], the authors distinguish two notions of authentication that we conflate: *(non-injective) agreement* and *sender invariance*. Whereas non-injective agreement (in [12]) is defined for all kinds of agents, sender invariance is defined only for pseudonyms, capturing the security guarantees behind authenticated channels that use unauthenticated public keys. One does not know who one is connected with, but it must always be the same agent, provided their corresponding private key does not leak. The authors show that non-injective agreement implies sender invariance. Thus, in the formalism of [12], sender correspondence establishes both non-injective agreement and sender invariance.

7.3 Formal Analyses of OAuth Protocols

Our formal analysis of SOAP (Sec. 5.2.2) was influenced by the recommendations of *OAuth 2.0 Security Best Current Practice* standard [31] and the formal analyses of the OAuth 2.0 and OpenID Connect protocols conducted in [17, 19]. The latter works, [17, 19], conducted pen-and-paper proofs in the Web Infrastructure Model [18], which captures more details of the browser environment than our model, e.g., HTTP status codes and their semantics. In contrast, we utilize the Tamarin prover [34], generating machine-checked proofs and consider a strictly stronger adversary than both [17, 19] and the original specifications [23, 38, 30]. Neither of these considered the leakage of authorization requests.

Only [16], the formal analysis of the OpenID Financial-grade API, considers a stronger attacker model not requiring Assumption 4 (browser redirect parameters remain confidential). However, [16] analyzes a different profile of OAuth 2.0 that does not match our setting as it assumes that applications can protect secrets, which enables the IdP to authenticate clients. Dropping Assumption 4 for SOAP would allow the adversary to capture redirects and thus effectively allow them to run the protocol themselves altogether (see Sec. 5.2.3).

[21] proposed the Privacy-Preserving OpenID Connect (POIDC) protocol, which enhances OpenID Connect's privacy guarantees, and also analyzed the security of their proposal in Tamarin. While [21] models the process of users granting consent more explicitly (logging in and providing consent are two steps, which we model as one), they make stronger assumptions on user behavior. Namely, they require that users only log in and consent to OpenID Connect flows when they themselves launched the protocol. We do not make this assumption and it is unrealistically strong. Some IdPs neither require a login (given an existing session) nor require consent (given that consent has been granted in the past; see Sec. 4.1). Moreover, [21] does not model the authorization code flow with PKCE, which our design relies upon. Nevertheless, our design's privacy guarantees could be enhanced if designs such as POIDC were adopted.

8 Conclusion

Social authentication is an exciting authentication paradigm promising usable [48], remote, and automated authentication in messaging applications. In this paper, we precisely and formally defined social authentication (Sec. 3), we presented SOAP, a secure and practical protocol implementing social authentication (Sec. 4), we formally proved that SOAP implements social authentication even in the presence of a strong adversary (Sec. 5), and we demonstrated SOAP's practicality in two prototypes (Sec. 6).

Note that while we targeted Signal in our prototype development, and additionally WhatsApp in our problem motivation (Sec. 2), SOAP can be applied to any application to authenticate key material and, more generally, applied to any kind of pseudonyms. For example, the applications Telegram, Threema, and Viber [45, 51, 50] all provide contact verification mechanisms similar to Signal and WhatsApp. Hence, SOAP can also be applied to these applications.

SOAP is automated to a large degree and can immediately be adopted by IdPs (indeed, it may not require adoption at all) because it relies on the well-established OpenID Connect protocol. It implements a secure and complete in-application ceremony that requires nothing more of users than their consent. Widespread adoption in messaging applications would be a cost-effective measure to increase their robustness against impersonation attacks and eavesdropping.

Future Work. Our results suggest several next steps. First, we argue that social authentication should be supported by modern messaging applications. Improving our open-source Signal prototype such that it could be deployed in production is a promising first step in that direction. Second, social authentication should be applicable far beyond secure messaging. For example, it could be used to secure other communication such as e-mail, or video conferencing, or it could be used as a second factor. Third, we suggest amending the OpenID Connect specification to support user-submitted claims such that users can consent unambiguously and developers can use streamlined APIs. Finally, while [48] established social authentication as a usable authentication ceremony and our prototypes require little interaction, a user study would help to finalize our design, accounting for users' understanding and preferences regarding social authentication.

Acknowledgements

This research was funded by the Werner Siemens-Stiftung (WSS) as part of the Centre for Cyber Trust (CECYT). We thank the WSS for their contribution. We also thank our colleagues Mihael Liskij and Jorge Luis Toro Pozo for their valuable feedback in the early stages of this project.

References

[1] Josh Aas et al. "Let's Encrypt: An Automated Certificate Authority to Encrypt the Entire Web". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. New York, NY, USA: ACM, Nov. 6, 2019, pp. 2473–2487. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3363192.

[2] *About Registration and Two-Step Verification | WhatsApp Help Center*. URL: https://faq.whatsapp.com/506595211487528/?helpref=hc_fnav (visited on 04/06/2023).

[3] *Android Intents with Chrome*. Chrome Developers. URL: https://developer.chrome.com/docs/multidevice/android/intents/ (visited on 10/03/2022).

[4] Richard Barnes et al. *Automatic Certificate Management Environment (ACME)*. Request for Comments RFC 8555. Internet Engineering Task Force, Mar. 2019. 95 pp. DOI: 10.17487/RFC8555.

[5] David Basin, Ralf Sasse, and Jorge Toro-Pozo. "The EMV Standard: Break, Fix, Verify". In: *2021 IEEE Symposium on Security and Privacy (S&P)*. May 2021, pp. 1766–1781. DOI: 10.1109/SP40001.2021.00037.

[6] Karthikeyan Bhargavan et al. "An In-Depth Symbolic Security Analysis of the ACME Standard". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. New York, NY, USA: ACM, Nov. 12, 2021, pp. 2601–2617. ISBN: 978-1-4503-8454-4. DOI: 10.1145/3460120.3484588.

[7] Josh Blum et al. *Zoom Cryptography Whitepaper*. Zoom Video Communications, Inc., Nov. 21, 2023. URL: https://github.com/zoom/zoom-e2e-whitepaper/blob/v4.3/zoom_e2e.pdf (visited on 12/05/2023).

[8] Melissa Chase et al. "SEEMless: Secure End-to-End Encrypted Messaging with Less Trust". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS 2019. Ed. by Lorenzo Cavallaro et al. London, UK: ACM, Nov. 11–15, 2019, pp. 1639–1656. DOI: 10.1145/3319535.3363202.

[9] Katriel Cohn-Gordon et al. "A Formal Security Analysis of the Signal Messaging Protocol". In: *2017 IEEE European Symposium on Security and Privacy*. EuroS&P. Apr. 2017, pp. 451–466. DOI: 10.1109/EuroSP.2017.27.

[10] C. Cremers and M. Dehnel-Wild. "Component-Based Formal Analysis of 5G-AKA: Channel Assumptions and Session Confusion". In: *26th Annual Network and Distributed System Security Symposium*. NDSS 2019. The Internet Society, 2019. URL: https://www.ndss-symposium.org/ndss-paper/component-based-formal-analysis-of-5g-aka-channel-assumptions-and-session-confusion/ (visited on 10/10/2022).

[11] Cas Cremers et al. "A Comprehensive Symbolic Analysis of TLS 1.3". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. New York, NY, USA: ACM, Oct. 30, 2017, pp. 1773–1788. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134063.

[12] Paul Hankes Drielsma et al. "Formalizing and Analyzing Sender Invariance". In: *Proceedings of the 4th International Conference on Formal Aspects in Security and Trust*. FAST'06. Berlin, Heidelberg: Springer-Verlag, Aug. 26, 2006, pp. 80–95. ISBN: 978-3-540-75226-4.

[13] *Enable Apps for Websites Using App URI Handlers*. URL: https://docs.microsoft.com/en-us/windows/uwp/launch-resume/web-to-app-linking (visited on 07/15/2022).

[14] *Facebook/Akd*. Meta, Apr. 21, 2023. URL: https://github.com/facebook/akd (visited on 04/24/2023).

[15] Matthias Fassl, Lea Theresa Gröber, and Katharina Krombholz. "Exploring User-Centered Security Design for Usable Authentication Ceremonies". In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI '21. New York, NY, USA: ACM, May 6, 2021, pp. 1–15. ISBN: 978-1-4503-8096-6. DOI: 10.1145/3411764.3445164.

[16] Daniel Fett, Pedram Hosseyni, and Ralf Küsters. "An Extensive Formal Security Analysis of the OpenID Financial-Grade API". In: *2019 IEEE Symposium on Security and Privacy (S&P)*. May 2019, pp. 453–471. DOI: 10.1109/SP.2019.00067.

[17] Daniel Fett, Ralf Küsters, and Guido Schmitz. "A Comprehensive Formal Security Analysis of OAuth 2.0". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. New York, NY, USA: ACM, Oct. 24, 2016, pp. 1204–1215. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978385.

[18] Daniel Fett, Ralf Küsters, and Guido Schmitz. "An Expressive Model for the Web Infrastructure: Definition and Application to the Browser ID SSO System". In: *2014 IEEE Symposium on Security and Privacy*. May 2014, pp. 673–688. DOI: 10.1109/SP.2014.49.

[19] Daniel Fett, Ralf Küsters, and Guido Schmitz. “The Web SSO Standard OpenID Connect: In-depth Formal Security Analysis and Security Guidelines”. In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. 2017 IEEE 30th Computer Security Foundations Symposium (CSF). Aug. 2017, pp. 189–202. DOI: 10.1109/CSF.2017.20.

[20] Lorenzo Franceschi-Bicchierai. *How a Third-Party SMS Service Was Used to Take Over Signal Accounts*. Vice. Aug. 17, 2022. URL: https://www.vice.com/en/article/qjkvxv/how-a-third-party-sms-service-was-used-to-take-over-signal-accounts (visited on 08/30/2022).

[21] Sven Hammann, Ralf Sasse, and David Basin. “Privacy-Preserving OpenID Connect”. In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. ASIA CCS ’20. New York, NY, USA: ACM, Oct. 5, 2020, pp. 277–289. ISBN: 978-1-4503-6750-9. DOI: 10.1145/3320269.3384724.

[22] *Handling Android App Links*. Android Developers. URL: https://developer.android.com/training/app-links (visited on 07/15/2022).

[23] Dick Hardt. *The OAuth 2.0 Authorization Framework*. Request for Comments RFC 6749. Internet Engineering Task Force, Oct. 2012. 76 pp. DOI: 10.17487/RFC6749.

[24] Amir Herzberg and Hemi Leibowitz. “Can Johnny Finally Encrypt? Evaluating E2E-encryption in Popular IM Applications”. In: *Proceedings of the 6th Workshop on Socio-Technical Aspects in Security and Trust*. STAST ’16. New York, NY, USA: ACM, Dec. 5, 2016, pp. 17–28. ISBN: 978-1-4503-4826-3. DOI: 10.1145/3046055.3046059.

[25] Dennis Jackson et al. “Seems Legit: Automated Analysis of Subtle Attacks on Protocols That Use Signatures”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’19. New York, NY, USA: ACM, Nov. 6, 2019, pp. 2165–2180. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3339813.

[26] Roger Piqueras Jover. “Security Analysis of SMS as a Second Factor of Authentication: The Challenges of Multifactor Authentication Based on SMS, Including Cellular Security Deficiencies, SS7 Exploits, and SIM Swapping”. In: *Queue* 18.4 (Aug. 31, 2020), Pages 20:37–Pages 20:60. ISSN: 1542-7730. DOI: 10.1145/3424302.3425909.

[27] *Keybase*. URL: https://keybase.io/ (visited on 07/18/2022).

[28] *Keybase Book: Learn about Your Keybase Account*. URL: https://book.keybase.io/account#proofs (visited on 07/18/2022).

[29] Sean Lawlor Lewi Kevin. *Deploying Key Transparency at WhatsApp*. Engineering at Meta. Apr. 13, 2023. URL: https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/ (visited on 04/24/2023).

[30] Torsten Lodderstedt, Mark McGloin, and Phil Hunt. *OAuth 2.0 Threat Model and Security Considerations*. Request for Comments RFC 6819. Internet Engineering Task Force, Jan. 2013. 71 pp. DOI: 10.17487/RFC6819.

[31] Torsten Lodderstedt et al. *OAuth 2.0 Security Best Current Practice*. Internet Draft draft-ietf-oauth-security-topics-19. Internet Engineering Task Force, Dec. 16, 2021. 52 pp. URL: https://www.ietf.org/archive/id/draft-ietf-oauth-security-topics-24.html (visited on 12/22/2023).

[32] G. Lowe. “A Hierarchy of Authentication Specifications”. In: *Proceedings 10th Computer Security Foundations Workshop*. Proceedings 10th Computer Security Foundations Workshop. June 1997, pp. 31–43. DOI: 10.1109/CSFW.1997.596782.

[33] Harjasleen Malvai et al. “Parakeet: Practical Key Transparency for End-to-End Encrypted Messaging”. In: *30th Annual Network and Distributed System Security Symposium*. NDSS 2023. San Diego, California, USA: The Internet Society, Feb. 27–Mar. 3, 2023. URL: https://www.ndss-symposium.org/ndss-paper/parakeet-practical-key-transparency-for-end-to-end-encrypted-messaging/ (visited on 04/24/2023).

[34] Simon Meier et al. “The TAMARIN Prover for the Symbolic Analysis of Security Protocols”. In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 696–701. ISBN: 978-3-642-39799-8. DOI: 10.1007/978-3-642-39799-8_48.

[35] Marcela S. Melara et al. “CONIKS: Bringing Key Transparency to End Users”. In: *24th USENIX Security Symposium*. USENIX Security 15. Washington, D.C., USA: USENIX Association, 2015, pp. 383–398. ISBN: 978-1-939133-11-3. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara (visited on 09/24/2021).

[36] moxie0. *Safety Number Updates*. Signal Messenger. URL: https://signal.org/blog/safety-number-updates/ (visited on 01/17/2022).

[37] *React*. Meta, July 19, 2022. URL: https://github.com/facebook/react (visited on 07/19/2022).

[38] N. Sakimura et al. *OpenID Connect Core 1.0 Incorporating Errata Set 1*. Nov. 8, 2014. URL: https://openid.net/specs/openid-connect-core-1_0.html (visited on 09/24/2021).

[39] N. Sakimura et al. *OpenID Connect Discovery 1.0 Incorporating Errata Set 1*. Nov. 8, 2014. URL: https://openid.net/specs/openid-connect-discovery-1_0.html (visited on 07/15/2022).

[40] Nat Sakimura, John Bradley, and Naveen Agarwal. *Proof Key for Code Exchange by OAuth Public Clients*. Request for Comments RFC 7636. Internet Engineering Task Force, Sept. 2015. DOI: 10.17487/RFC7636.

[41] Svenja Schröder et al. "When SIGNAL Hits the Fan: On the Usability and Security of State-of-the-Art Secure Mobile Messaging – NDSS Symposium". In: *23rd Annual Network and Distributed System Security Symposium*. NDSS 2016. San Diego, California, USA: The Internet Society, Aug. 12, 2016. URL: https://www.ndss-symposium.org/ndss2016/eurousec-2016-workshop/when-signal-hits-fan-usability-and-security-state-art-secure-mobile-messaging/ (visited on 04/19/2022).

[42] Maliheh Shirvanian, Nitesh Saxena, and Jesvin James George. "On the Pitfalls of End-to-End Encrypted Communications: A Study of Remote Key-Fingerprint Verification". In: *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACSAC '17. New York, NY, USA: ACM, Dec. 4, 2017, pp. 499–511. ISBN: 978-1-4503-5345-8. DOI: 10.1145/3134600.3134610.

[43] *Signal PIN*. Signal Support. URL: https://support.signal.org/hc/en-us/articles/360007059792-Signal-PIN (visited on 04/06/2023).

[44] *Start Integrating Google Sign-In into Your Android App | Google Sign-In for Android | Google Developers*. URL: https://developers.google.com/identity/sign-in/android/start-integrating (visited on 10/04/2022).

[45] *Telegram FAQ*. Telegram. URL: https://telegram.org/faq?setln=en#q-what-is-this-39encryption-key-39-thing (visited on 12/19/2023).

[46] *Twilio Incident: What Signal Users Need to Know*. Signal Support. URL: https://support.signal.org/hc/en-us/articles/4850133017242-Twilio-Incident-What-Signal-Users-Need-to-Know- (visited on 08/30/2022).

[47] Elham Vaziripour et al. "Action Needed! Helping Users Find and Complete the Authentication Ceremony in Signal". In: Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018). 2018, pp. 47–62. ISBN: 978-1-939133-10-6. URL: https://www.usenix.org/conference/soups2018/presentation/vaziripour (visited on 07/18/2022).

[48] Elham Vaziripour et al. "I Don't Even Have to Bother Them!: Using Social Media to Automate the Authentication Ceremony in Secure Messaging". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. New York, NY, USA, May 2019. URL: https://doi.org/10.1145/3290605.3300323 (visited on 06/16/2022).

[49] Elham Vaziripour et al. "Is That You, Alice? A Usability Study of the Authentication Ceremony of Secure Messaging Applications". In: Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017). 2017, pp. 29–47. ISBN: 978-1-931971-39-3. URL: https://www.usenix.org/conference/soups2017/technical-sessions/presentation/vaziripour (visited on 04/19/2022).

[50] *Verify End-To-End Encryption: Trusted Contacts List*. Viber. URL: https://help.viber.com/hc/en-us/articles/9061180581661-Verify-End-To-End-Encryption-Trusted-Contacts-List (visited on 12/19/2023).

[51] *What Do the Three Colored Dots next to a Contact Mean? – Threema*. URL: https://threema.ch/en/faq/levels_expl (visited on 12/19/2023).

[52] *WhatsApp's Signal Protocol Integration Is Now Complete*. Signal Messenger. URL: https://signal.org/blog/whatsapp-complete/ (visited on 08/04/2022).

[53] Tarun Kumar Yadav et al. "Automatic Detection of Fake Key Attacks in Secure Messaging". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. New York, NY, USA: Association for Computing Machinery, Nov. 7, 2022, pp. 3019–3032. ISBN: 978-1-4503-9450-5. DOI: 10.1145/3548606.3560588.

A Detailed Protocol Description

To run SOAP, applications must periodically load the IdPs's token and authorization endpoints, as well as signing keys as specified in [39]. The simplest option to do so is whenever a application engages in SOAP.

The prover's application starts SOAP by generating three random values: a code verifier *cv* as per the PKCE specification [40], a salt *s*, and a nonce *n*. We suggest generating each of these random values with at least 256-bits of entropy, following [40]'s security requirements. The application then uses a secure password-hashing algorithm h to calculate a salted hash $h(k,s)$ of the safety number k. Next, the application uses these values to issue an OpenID Connect authentication request to the selected IdP with the following parameters:

scope: "openid email"; depending on the IdP, other scopes than "email" may be desirable.

response_type: "code"

nonce: $n \,||\, h(k,s)$; the application must ensure that it does not include the salt. $||$ denotes concatenation. The application must ensure the parsing is unambiguous, e.g., by adding a delimiter character.

state: n

code_challenge: $\mathrm{S256}(cv)$; S256 marks the SHA-256 hashing algorithm.

code_challenge_method: "S256"

Naturally, the application also includes its IdP-issued application ID, and an appropriate redirect URL. Redirect URLs must use the HTTPS scheme and must be distinct per IdP. The application stores the salted hash $h(k,s)$, the salt s, the nonce n, the redirect URL used, and the code verifier cv as the most recently issued request. Then, the application launches the system's browser with the request URL, which in turn takes the user to the consent and login page.

When the user consents, the IdP redirects the browser back to the application. The application verifies that it received the authorization code through the expected redirect URL and with the expected state by comparing these values to those stored as most recently issued request. If both checks pass, the application uses the authorization code and stored code verifier to request the ID token from the IdP as specified in [38], i.e., using a POST request. When receiving the token in the response, the application verifies the token as follows:

1. Verify that the issuer matches the redirect URL stored.
2. Verify that the token's audience matches the application ID.
3. Verify that the token's nonce includes the hash stored.
4. Verify that the token is not expired.
5. Verify the token's signature using a key loaded from the IdP's discovery document.

If all checks pass, the application clears its storage for the most recently issued request, stores the nonce as issued by itself, and forwards the token and salt to the verifier. The verifier's application applies checks 4 and 5 and makes the following additional checks:

1. Verify that the nonce encoded in the token has not been generated by itself, comparing it to the nonces stored locally.
2. Verify the safety number's hash encoded in the token by recomputing it, using the salt provided by the prover and the safety number of the channel through which it received the token, and comparing it for equality.