Coauthentication

Jay Ligatti Cagri Cetin Shamaria Engram Jean-Baptiste Subils
Dmitry Goldgof
Department of Computer Science and Engineering
University of South Florida
{ligatti,goldgof}@usf.edu {cagricetin,sengram,subils}@mail.usf.edu

USF Technical Report Auth-7-17-17

July 17, 2017

Abstract

Collaborative authentication, or coauthentication, is a single-factor technique in which multiple registered devices work together to authenticate a user. Coauthentication aims to provide security benefits similar to those of multi-factor techniques, such as mitigating theft of any one authentication device, without the inconveniences of multi-factor techniques, such as having to enter passwords or scan biometrics. Coauthentication can provide additional security benefits, including: preventing phishing and man-in-the-middle attacks, basing authentications on high-entropy secrets that can be generated and updated automatically, and availability protections against, for example, device misplacement or denial-of-service (DoS) attacks. This paper introduces coauthentication and discusses and evaluates applications, example protocols, and implementations.

1 Introduction

Authentication is one of the most common security activities users perform. Authentication is also a common target of attacks, through phishing, guessing, man-in-the-middle, token-theft, and related vectors. Due to the commonality of using and attacking authentication systems, even modest improvements to their security or usability may produce significant benefits.

1.1 Background

As is well understood, authentication techniques are based on factors, the three standard factors being what you know (human-entered secrets like passwords), what you have (physical tokens like keys, electronic remote controls, or smartcards), and what you are (biometrics like fingerprints).

Each authentication factor has advantages and disadvantages [22]. For example, tokens are susceptible to theft, but doing so in the obvious way requires physical access. Users will often notice physical theft of a token more readily than a remote theft or guessing of a password or biometrics. However, tokens have traditionally relied on special-purpose hardware and consequently been more expensive to implement and deploy than other factors. In addition, usability benefits of tokens have traditionally been offset by the costs of having to carry and handle the tokens [22, 29].

Multi-factor authentication attempts to improve security by requiring successful attacks to compromise every factor being used. One popular two-factor mechanism combines a username/password with a second password (a one-time password, OTP) texted to the user's phone [13]. Alternatively, instead of receiving an OTP from the authenticator, the phone may share a cryptographic key with the authenticator and generate its own OTP, called a time-based OTP or TOTP, as a cryptographic hash, using the shared key, of the current time (represented in coarse granularity to avoid synchronization problems between the authenticator and phone) [21]. A benefit of such mechanisms is that the physical-token factor is a device already possessed and carried by the user, thus avoiding expensive, dedicated hardware.

However, multi-factor techniques add the inconveniences of each factor required. For example, because OTP and TOTP techniques require users to enter two passwords and carry a registered device, they suffer from the nontrivial usability drawbacks of password-based authentication mechanisms (e.g., [24, 11, 14, 25, 19]) and the inconvenience of having to access one's mobile device to authenticate.

This latter inconvenience, of having to access one's registered mobile device to authenticate, has lessened over time, as the overwhelming majority of adults have gone from having zero personal smart devices accessible at all times to having one personal smart device—a smartphone—accessible at all times [12].

With the growth of the Internet of Things, ubiquitous computing, and wearable, edible, and implantable devices, the overwhelming majority of adults will soon have multiple personal smart devices accessible at all times, all of which can be registered and used to authenticate. For example, to log in to a website, open a door, or start an engine, *two* of a user's registered devices, perhaps a smartphone and smartwatch, might have to participate in the authentication. A gate or garage door might authenticate a request to open by requiring participation from both a registered car and a registered smartphone; then stealing only the car, or only the phone, would be insufficient for opening the door.

Even today many people only authenticate to certain services when multiple of their devices are present. For example, a user U may log in to banking services only from a certain laptop, tablet, or PC, while in the presence of U's smartphone. In this case the banking service could register these two devices to U and require their participation in every authentication of U.

We call this technique, in which multiple devices collaborate to authenticate a user, coauthentication.

1.2 Contributions

Coauthentication is a single-factor technique, using only the physical token (registered-device) factor. As a standalone technique, it avoids inconveniences of factors like passwords and biometrics. Coauthentication is however compatible with other factors and may be incorporated into multi-factor authentication, such as requiring a password as one factor and coauthentication as a second factor.

Although coauthentication is a single-factor technique, it requires multiple devices' participation in order for authentications to succeed. Coauthentication therefore provides benefits of robustness similar to multi-factor techniques. For example, if an attacker steals one of the two devices required for a coauthentication, then the attacker still cannot authenticate because the legitimate user, in possession of the second device, will not confirm the authentication. Having to compromise both devices, or more specifically the authentication keys accessible to both devices, is similar to the requirement in two-factor approaches to compromise both factors.

Coauthentication also provides security benefits beyond existing authentication techniques, including multi-factor techniques. By applying more sophisticated coauthentication policies, such as that any 3 out of 5 registered devices must participate, coauthentication can succeed even when up to 2 of a user's 5 devices have been lost, stolen, left at home, denied service maliciously (e.g., due to radio jamming or other DoS attacks), or are operating in environments that render the devices inoperable or incommunicado (e.g., due to co-channel or electromagnetic interference). Coauthentication can thus provide availability protections; to deny service for the overall authentication, an attacker must deny service to *all* (e.g., 3 of 5) required devices, which may be heterogeneous (e.g., communicating through different channels and protocols, running different operating systems, etc).

Another benefit of coauthentication relates to its ability to reset secrets automatically. Every authentication system—regardless of the factors used—is based on secrets, which could take the form of passwords, patterns of metallic teeth on keys, radio frequencies at which devices transmit data, codes stored on devices and transmitted, fingerprints, etc. Authentication systems aim to protect against attackers that do not obtain the required secrets. However, with existing systems, if a nonuser does obtain the required secrets, then resetting the secrets is laborious (e.g., for the victim to reset a password), expensive (e.g., to send the user a new physical token), or impractical (e.g., to give a user new fingerprints, retinas, vocal profile, etc).

In contrast, coauthentication secrets may be cryptographic keys stored on registered devices; these keys may be reset, and periodically updated, automatically. These keys can also be generated to have high entropy, without concern for whether users can create, memorize, or enter the high-entropy secrets.

Another significant benefit of coauthentication is its invulnerability to phishing and man-in-the-middle attacks. Although phishing is a widely exploited vector in existing authentication systems [11], coauthentication secrets cannot be phished because users never enter them.

Man-in-the-middle attacks on existing authentication systems may proceed as follows: the victim enters a username and password on a fake website; the fake website forwards this information to the real website, which may then issue an OTP, in which case the user also enters the OTP into the fake website; the attacker can now complete the authentication on the real website and masquerade as the user. Such attacks are not possible with coauthentication because its communications are encrypted to provide confidentiality and authenticity (under standard cryptographic assumptions). If a user's devices send a fake website coauthentication messages encrypted for the fake website, then authenticity ensures that the fake website cannot masquerade as the user to the real website. On the other hand, if a user's devices send a fake website coauthentication messages encrypted for the real website, then confidentiality ensures that the fake website cannot extract useful information from those messages. Section 3 reports formal-verification results showing that, under explicitly stated assumptions, coauthentication prevents man-in-the-middle attacks.

However, as with all security mechanisms, coauthentication's benefits are limited. Malware installed on a user's device may monitor, modify, and exfiltrate data accessible to that device. Such malware may exfiltrate session keys, enabling attackers to masquerade as the victim on other devices. Even worse, such malware may send and receive messages on the compromised device, enabling attackers to masquerade as the victim on the victim's own device. Malware on an authenticator may similarly exfiltrate session keys, insert backdoors, deny service, etc. Protecting against malware that is active during authentication is beyond the scope of coauthentication.

Instead, coauthentication protects against theft of authentication secrets (keys) used by devices. This theft may occur in any way, even through malware, as long as the devices in the victim's possession run as intended during the coauthentication process (e.g., without malware exfiltrating keys). For example, besides remotely compromising devices to obtain keys, attackers may acquire keys through physical theft. After stealing a device, an attacker may attempt to authenticate on it or may copy all secrets accessible to the device and return the device to the user, who may be unaware of the theft. Coauthentication systems introduced here protect against all of these attacks.

Organization This paper presents coauthentication and considers and evaluates several example implementations. Section 2 describes coauthentication in greater detail and presents several example protocols that may be used to implement it. Sections 3 and 4 evaluate these implementations, first formally, with ProVerif [3, 4], in Section 3, and then empirically in Section 4. Section 5 contains additional discussion of related work, and Section 6 concludes.

2 Coauthentication

The devices involved in a coauthentication are the *authenticator* (e.g., a server deciding whether to authenticate a user), the *requestor* (on which the current authentication attempt is initiated), and one or more *collaborators*. The requestor and collaborator(s) are *registered* with the authenticator, meaning that the devices have access to a secret that the authenticator can use to verify the devices' participation in an authentication. This secret accessible to the requestor and collaborator(s) may, for example, be a secret key shared with the authenticator, or a private key K such that the authenticator can verify signatures created with K.

In some implementations, the authenticator, upon receiving an authentication *request*, issues one or more *challenges* and awaits one or more valid *responses* to the challenges. Other implementations avoid authentication challenges. In all cases, the authenticator verifies that multiple registered devices, more specifically the secret keys accessible to those devices, participate in authentications.

Description of the collaboration policy
Disallow by default (require user confirmation before collaborating)
Allow by default, with a warning or log of the collaboration
If co-located then tacitly allow, else allow by default with a warning
If co-located then tacitly allow, else disallow by default
If co-located then tacitly allow, else disallow entirely
If co-located then allow by default with a warning, else disallow by default
If co-located then allow by default with a warning, else disallow entirely
If co-located then disallow by default, else disallow entirely

Table 1: Example collaboration policies.

2.1 Collaboration Policies

Each collaborator may enforce its own policy specifying the circumstances under which it participates in a coauthentication.

As a first example, a collaborator may only participate in an authentication after a user has explicitly clicked a button to confirm participation. Under this policy, if an attacker steals or compromises the requestor and initiates a coauthentication, the legitimate user will not confirm the attacker-initiated authentication on the collaborator, so the authentication attempt will fail. The collaborator may further enforce a deadline for clicking the confirmation button. Voice commands or other inputs could be used in place of confirmation buttons.

As a second example, a collaborator may automatically participate in an authentication but warn the user, or log, that it has done so, for example by displaying a text alert with an audible warning sound (like a text message). The alert could state the date and time at which the collaboration occurred and provide a simple interface for the user to notify the authenticator if the collaboration was unauthorized (i.e., an attacker-initiated authentication).

The first of these example policies defaults to disallowing participation, and only collaborates when a user explicitly confirms the authentication. Let's call this first example the *disallow-by-default* collaboration policy. The second example policy defaults to allowing participation, and relies on users to observe a warning and handle unauthorized collaborations after the fact. Let's call this second example the *allow-by-default-with-warning* collaboration policy.

These two policies illustrate a security-usability tradeoff: the disallow-by-default policy prevents attackers from initiating authentications (increased security) but requires user confirmation on each authentication (decreased usability). The allow-by-default-with-warning policy allows attackers to successfully be logged in during the window of time between when the collaborator warns the user and when the user observes the warning and notifies the authenticator of the unauthorized authentication (decreased security), but users don't have to confirm the authentications on the collaborator (increased usability).

For many applications the usability benefits of the allow-by-default-with-warning policy may outweigh the security costs. For example, users of financial servers may prioritize usability because (1) legitimate authentications are far more common than device compromise, and (2) the financial institutions, due to legal requirements or to entice customers, may guarantee to reimburse customers for any losses incurred from unauthorized logins.

Many more collaboration policies are possible. For example, a collaborator could decide whether to participate in a coauthentication based on the requestor's proximity, that is, whether the requesting device is co-located with the collaborator. In applications where the attack vector of concern is device theft, a collaborator may presume that a co-located requestor has not been stolen (because stolen devices are commonly not in proximity to the victim's unstolen devices). Such a collaborator may tacitly allow collaborations with co-located requestors but show warnings for, require explicit confirmations for, or disallow entirely, collaborations with non-co-located requestors.

Table 1 lists several example collaboration policies, in accordance with the preceding discussion.

Collaboration policies may also take into account the source of authentication requests (e.g., only allowing collaborations with known requestors) or how much time has passed since earlier collaborations. For example, timing considerations might cause a collaborator C to require user confirmation only for collaborations with a requestor R when C has not collaborated with R within the last hour, or when C has not issued a warning about collaborating with R within the past 20 minutes.

2.2 Example Applications

Besides the more-obvious applications of authentication, such as logging in to operating systems and web services, authentication occurs in less-obvious, but common, ways, including pushing a button on a remote control to open a door or gate, using a physical key to unlock a door or start an engine, or swiping, scanning, or inserting a payment card, passport, or driver's license. Let's consider coauthentication in a few of these less-obvious applications.

A thief who successfully breaks into a car containing a garage-door controller can push a button to open the victim's garage. Once in the garage, the thief could shut the garage door and enter the victim's house without being observed by neighbors. Such attacks have occurred [1, 20]. Assuming the authenticator (garage-door opener) is not compromised, the requestor is the remote control (or a smart car), and the collaborator is a smartphone, successful attacks on the coauthentication system would require compromising both the car and phone. Because people rarely leave phones in cars unattended, coauthentication mitigates the car-theft vector. Similarly, compromising only the phone is insufficient for opening the door. The collaboration policy in this example might specify that the phone tacitly allows collaborations for co-located requestors and disallows, with warnings, all other collaborations.

Door locks are a similar application. Stealing a physical key provides access. With the door lock acting as an authenticator—and again assuming that the authenticator is not compromised, because if an attacker controls the authenticator then its authentication requirements are irrelevant—two of a user's devices may coauthenticate to open a door. These devices might be a remote control (as is currently used for remote keyless entry in automobiles), and a smartphone as a collaborator. A user wishing to unlock a door might initiate the coauthentication request by pressing a button on the remote control. Alternatively, the remote control might initiate coauthentication requests automatically for all locks within a certain proximity of the remote control; for example, all door locks within 2m of the remote control may automatically unlock, through coauthentication with the smartphone as collaborator, and automatically re-lock if not opened within 5s of unlocking.

In some payment-card systems, physical access to a victim's card enables an attacker to use the card successfully. Although the attacker may be caught after the fact, financial institutions and their customers may expend significant resources in the process. Payment cards might instead be coauthenticated, to require participation of a registered smartphone as a collaborator to the requesting payment card (which itself might be combined into an existing device like a smartwatch). Then this coauthenticated payment system would achieve security protections similar to systems in which an OTP, sent to the user's phone, is required for payment-card use. However, the coauthentication systems could run automatically, without requiring the user to enter any passwords, for example with the smartphone enforcing the second-to-last collaboration policy listed in Table 1.

When run automatically, without requiring user interaction, coauthentication is a zero-interaction authentication system [6]. By making authentications transparent and unobtrusive, zero-interaction systems enable more devices to take advantage of the benefits of authentication, without fatiguing users with authentication activities. These benefits include enforcing access controls and adjusting to the preferences of each registered user. For example, smart home assistants, smart appliances, and computer components (microphones, keyboards, cameras, memory modules, ALUs) may coauthenticate users to mitigate unauthorized use; televisions may coauthenticate users to enforce parental controls; and chairs, televisions, lights, HVAC systems, etc., may coauthenticate users to adjust to their individualized preferences. We envision coauthentication operating in this way, with zero or low user interactions, to implement pervasive authentication.

For this same reason of coauthentication being able to run transparently, it is well suited to continuous authentication [6, 2].

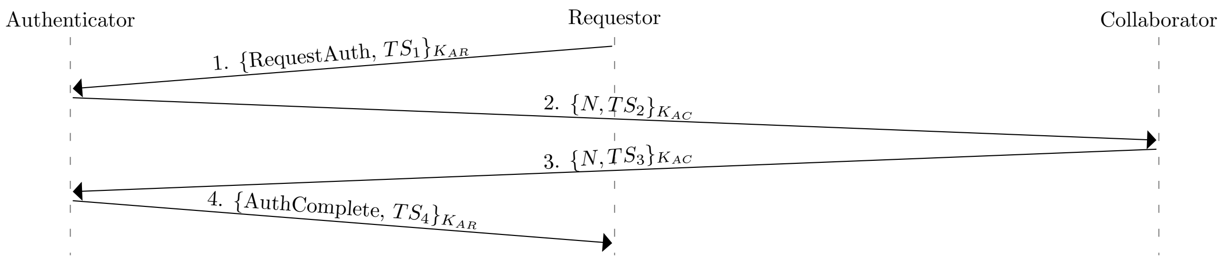
Authenticator
Requestor
Collaborator
1. {RequestAuth, TS_1}$_{K_{AR}}$
2. {N, TS_2}$_{K_{AC}}$
3. {N, TS_3}$_{K_{AC}}$
4. {AuthComplete, TS_4}$_{K_{AR}}$

Figure 1: The canonical coauthentication protocol, showing communications between authenticator, requestor, and collaborator. Secret key K_{AR} (K_{AC}) is shared between authenticator and requestor (collaborator). Each TS_i is a timestamp, $\{M\}_K$ is the encryption of M using key K, and N is a nonce. Timestamps enable timeouts and replay-attack detection.

2.3 A Canonical Coauthentication Protocol

There are many ways to instantiate coauthentication. For convenience of exposition, the first instantiation detailed here will be called the canonical version. This version operates according to the protocol shown in Figure 1.

Following the flow of data in Figure 1, canonical coauthentication operates as follows. Assume that during device registration, the authenticator A and requestor R share a secret key K_{AR}, and the authenticator A and collaborator C share a secret key K_{AC}. Also assume that all messages include timestamps and timeouts as appropriate (for example, so the authenticator terminates authentication attempts when collaborators fail to respond to challenges within a reasonable amount of time).

1. Requestor R initiates the coauthentication by sending an encrypted request message to the authenticator A.

2. Authenticator A receives and decrypts the request message and verifies its freshness. Authenticator A thus ensures that secret key K_{AR} has been used to generate the request. Authenticator A then finds that the requestor R is registered to a user having collaborating device C, creates a nonce N, encrypts N with secret key K_{AC}, and sends this *challenge* ciphertext to C.

3. Collaborator C receives the challenge encrypted with K_{AC}, decrypts, verifies the freshness of nonce N, re-encrypts N—with a newer timestamp—using K_{AC}, and sends this *response* ciphertext back to the authenticator A.

4. Authenticator A receives the response encrypted with K_{AC}, decrypts, verifies that the received nonce matches the N it sent C, and verifies that the received timestamp is newer than the one it sent C. Authenticator A thus ensures that secret key K_{AC} has been used to respond to its challenge. Because A has now verified participation of both keys K_{AR} and K_{AC}, it sends an authentication-complete message, for example containing a session identifier, to the requestor R.

In this way, coauthentication requires participation of both authentication keys.

2.4 Protecting Against Attackers Who Have Acquired a Key

Suppose an attacker acquires the key K_{AC} used in the canonical protocol, possibly by stealing the collaborator. Acquiring K_{AC} only gives an attacker the ability to allow or deny authentications initiated by the victim. Attackers are already assumed to be active and consequently capable of denying service by dropping network messages. Acquiring K_{AC} therefore provides an attacker with no new capabilities (and Section 2.7 describes generalizations of canonical coauthentication that mitigate DoS attacks). In fact, if an attacker possesses the collaborator, it is preferable for the attacker to deny the victim's authentications, because doing so alerts the victim to the theft.

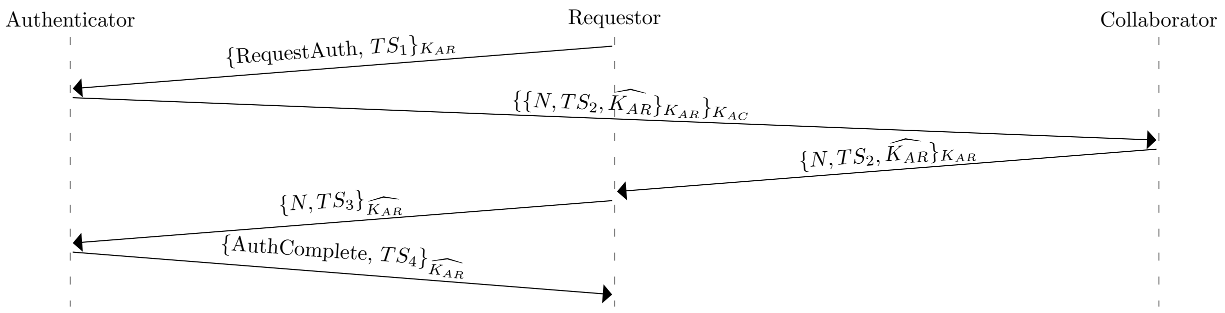
Authenticator
Requestor
Collaborator
$\{\text{RequestAuth}, TS_1\}_{K_{AR}}$
$\{\{N, TS_2, \widehat{K_{AR}}\}_{K_{AR}}\}_{K_{AC}}$
$\{N, TS_2, \widehat{K_{AR}}\}_{K_{AR}}$
$\{N, TS_3\}_{\widehat{K_{AR}}}$
$\{\text{AuthComplete}, TS_4\}_{\widehat{K_{AR}}}$

Figure 2: A version of the canonical coauthentication protocol that also updates K_{AR} to $\widehat{K_{AR}}$. The third message, which goes from collaborator to requestor, is sent through a private channel.

Attacks based on acquiring K_{AR} are divided into two cases, depending on how the attacker uses K_{AR}. In the first case, an attacker uses K_{AR} to initiate a coauthentication. For example, the attacker may have stolen the requestor and is attempting to coauthenticate with it. Assuming an appropriate collaboration policy, the victim is notified of the authentication attempt (on the collaborator device) and disallows the authentication. Acquiring and using K_{AR} in this way therefore provides an attacker with no new capabilities.

Finally, suppose an attacker acquires K_{AR}, waits for the victim to initiate a coauthentication, and then uses K_{AR} to decrypt the authentication-complete message sent to the requestor. For example, the attacker may have "borrowed" the requestor R, copied K_{AR} from R's memory, returned R to the victim, and waited for, observed, and decrypted the final message in a legitimate coauthentication. In this case, if the victim is still using the stolen K_{AR}, then the attacker obtains a capability, such as a session ID in the authentication-complete message, enabling the attacker to masquerade as the victim.

There are at least two approaches for preventing such attacks:

1. Try to prevent the attacker from acquiring K_{AR} in the first place, using mechanisms like storing K_{AR} in a trusted platform module [16]. This approach would assume that attackers, who possibly have physical access to the requestor R, can *use* R's key K_{AR} to initiate authentications on R but cannot *extract* K_{AR} from R. Without acquiring K_{AR}, the attacker cannot access the capability (session ID) in the authentication-complete message.

2. Update K_{AR} at any point between when the attacker acquires K_{AR} and when the victim uses K_{AR} to coauthenticate. Without access to the up-to-date version of K_{AR}, the attacker again cannot access the capability (session ID) in the authentication-complete message.

Regarding the second of these two approaches, there are a variety of ways to update K_{AR} on the requestor. If we were only concerned with passive attackers, then a Diffie-Hellman key exchange [9] would suffice.

In the presence of active attackers, K_{AR} can be updated if the requestor and collaborator can communicate privately—the authenticator can send the collaborator an updated K_{AR} (encrypted with both K_{AC} and the old K_{AR}, so an attacker with access to at most one of these secrets cannot obtain the new key), and then the collaborator can forward the new K_{AR} to the requestor through a private channel. Private channels may be implemented with short-range communication channels, such as NFC, zigbee, wireless USB, infrared, or near-field magnetic induction, under the assumption that attackers cannot access such communications because they are on direct, device-to-device channels.

This method of updating K_{AR} can be merged into coauthentication protocols. Figure 2 shows such a merging of K_{AR} updates into the canonical protocol. Assuming the third message in Figure 2 is sent through a private channel, the protocol of Figure 2 coauthenticates the requestor while preventing active attackers who obtain either K_{AC} or K_{AR} (but not both) from accessing the session ID in the authentication-complete message. This security property has been formally verified, as discussed in Section 3. It is straightforward to modify the protocol in Figure 2 to update K_{AC} in addition to K_{AR}.

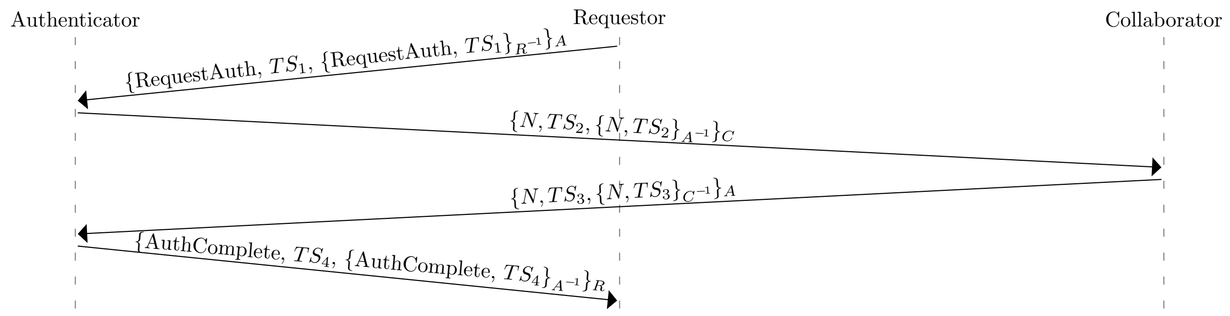
Authenticator
Requestor
Collaborator
$\{\text{RequestAuth}, TS_1, \{\text{RequestAuth}, TS_1\}_{R^{-1}}\}_A$
$\{N, TS_2, \{N, TS_2\}_{A^{-1}}\}_C$
$\{N, TS_3, \{N, TS_3\}_{C^{-1}}\}_A$
$\{\text{AuthComplete}, TS_4, \{\text{AuthComplete}, TS_4\}_{A^{-1}}\}_R$

Figure 3: A public-key version of the canonical coauthentication protocol. The requestor's digital signature of M is notated $\{M\}_{R^{-1}}$, and the encryption of M using the requestor's public key is notated $\{M\}_R$ (and similar notation for the authenticator A and collaborator C).

Ideally, every coauthentication would include an update to the secret keys, through a protocol like the one shown in Figure 2, or, if the requestor and collaborator cannot communicate privately, through some other mechanism. In practice however, a security-performance tradeoff may be desirable, with keys updated periodically, perhaps during times of low network and CPU usage. Analogously, password-based authentication systems would ideally require users to update their passwords on every authentication, to limit attackers who have acquired passwords. In practice security-performance and security-usability tradeoffs are made, and passwords are typically updated only rarely, through a manual process [11].

2.5 Variations on the Canonical Protocol

Besides merging key updates into the canonical protocol, many other variations are possible. Figure 3 for example replaces the symmetric-cryptographic operations with asymmetric (public-key) operations.

Even limiting consideration to symmetric-key coauthentication, an infinite variety of challenge-response processes exist, as the authenticator may issue arbitrarily many, arbitrarily complex challenges, with more complex challenges requiring additional collaboration between the requestor and collaborator. For example, instead of the second message in the canonical protocol of Figure 1, the authenticator could send a challenge $\{\{N, TS_2\}_{K_{AC}}\}_{K_{AR}}$ to the requestor. The requestor solves half the challenge by decrypting and forwarding the result to the collaborator, which responds as in the third message of the canonical protocol. Upon receiving a valid response from the collaborator, the authenticator has assurance that both the requestor and collaborator keys were used to generate the response. Instead of full encryptions of challenges, valid responses could be message authentication codes (MACs).

In some applications it may be useful to make some of the communications multicast or broadcast. For example, to require additional user interaction, the requesting device may display and transmit an authentication challenge as a QR code [15]. The user could scan this code on the collaborating device, which could then respond to the challenge in the canonical way.

Additional variations exist. An authenticator may treat valid responses to its authentication challenges as authenticating both the requestor and the collaborator, or even just the collaborator, in which case the requestor acts as a requestor-proxy for the collaborator. Authenticators may also transmit authentication-complete (e.g., session-ID) messages differently, for example by encrypting a session ID with both shared keys and sending the resulting ciphertext to the collaborator, which decrypts and then forwards to the requestor. This extra layer of collaboration mitigates attacks in which the requestor is noticeably compromised after the collaborator responds to the initial challenge, giving the collaborator one more chance to confirm the authentication.

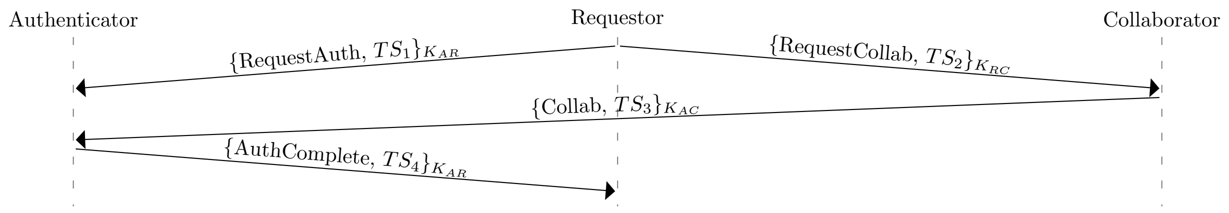
Authenticator
Requestor
Collaborator
$\{\text{RequestAuth}, TS_1\}_{K_{AR}}$
$\{\text{RequestCollab}, TS_2\}_{K_{RC}}$
$\{\text{Collab}, TS_3\}_{K_{AC}}$
$\{\text{AuthComplete}, TS_4\}_{K_{AR}}$

Figure 4: A symmetric-key coauthentication without authenticator challenges.

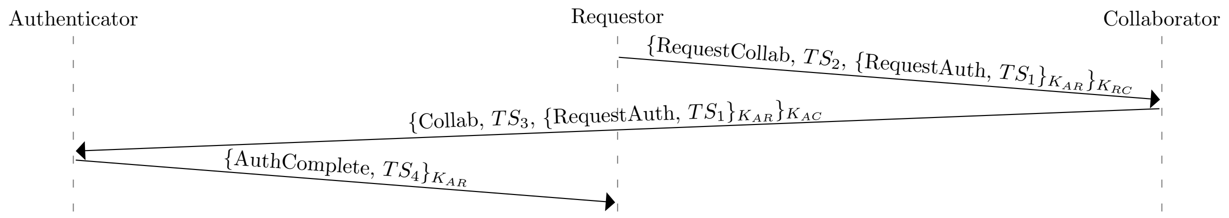
Authenticator
Requestor
Collaborator
$\{\text{RequestCollab}, TS_2, \{\text{RequestAuth}, TS_1\}_{K_{AR}}\}_{K_{RC}}$
$\{\text{Collab}, TS_3, \{\text{RequestAuth}, TS_1\}_{K_{AR}}\}_{K_{AC}}$
$\{\text{AuthComplete}, TS_4\}_{K_{AR}}$

Figure 5: A symmetric-key coauthentication without authenticator challenges, wherein the collaborator forwards the requestor's participation to the authenticator.

2.6 Avoiding the Challenge-Response Process

It is also possible to avoid the challenge-response portion of canonical coauthentication entirely, by having the requestor send two requests, one to the authenticator (to request authentication) and another to the collaborator (to request collaboration).

Figure 4 shows one such a possibility, where the requestor and authenticator share a secret key K_{AR}, the authenticator and collaborator share K_{AC}, and the requestor and collaborator share K_{RC}. The requestor sends its two requests, one to the authenticator and another to the collaborator, possibly with (same or different) timestamps. By including timestamps in all the messages, the participating machines can verify the freshness of every message received. Here freshness means that the timestamp has not been used before (at least by the device from which the timestamp is received, to prevent replay attacks) or falls within an allowed window of time, for example the past 5 seconds. After verifying that both the requestor and collaborator have participated in an authentication by sending fresh messages, the authenticator notifies the requestor that it is successfully authenticated.

Just as the canonical protocol in Figure 1 was modified to incorporate key updates (Figure 2) and asymmetric-cryptographic operations (Figure 3), the challengeless coauthentication protocol in Figure 4 can be modified straightforwardly to incorporate the same features. Furthermore, in all these protocols, devices may forward the messages of other devices. For example, Figure 5 shows the same protocol shown in Figure 4 but with the collaborator forwarding the requestor's participation message.

The protocol of Figure 5 is interesting because it minimizes the messages needed to coauthenticate. In general, coauthentication protocols that avoid the challenge-response process are expected to be more efficient than the challenge-response protocols, due to the omission of challenge creation and the parallelization or batching of some of the communications (e.g., the authentication and collaboration requests).

The drawbacks of the challengeless coauthentication protocols include their additional use of communication between the requestor and collaborator. For some applications, such as IoT devices with few resources (memory, energy, computing power, etc.) and many potential, or dynamically changing, collaborators, these extra communications may be inefficient or impracticable.

2.7 More General Coauthentication Policies, and Availability Benefits

There are advantages to systems in which users register more than two devices. Suppose a user has registered n devices and the authenticator requires any m of the n devices to coauthenticate, where $2 \leq m \leq n$. In the coauthentication protocols described so far (Sections 2.3–2.6), m=n=2, but now suppose m=2 and n=3. In this case, compromising only one of the user's devices is still insufficient for authenticating as that user, because m=2. At the same time, because m<n, the user can be authenticated even after forgetting or losing a device, or having a device become inoperable, for example due to a DoS attack.

This m-out-of-n-device policy, enforced at the authenticator, tolerates the absence of $n-m$ devices. When the user devices communicate through heterogenous channels, DoS attacks on coauthentication become more difficult to mount, as more channels have to be jammed or otherwise interfered with.

To prevent attackers from using these $n-m$ devices to coauthenticate, m may be further constrained to be greater than $n-m$, that is, $m > n/2$. For example, a system that requires only 2 out of 4 devices to coauthenticate (i.e., m=2=$n/2$) tolerates the absence of 2 devices, but if those 2 devices are absent due to theft, then the thief can use them to coauthenticate. To prevent such attacks, the m-out-of-n-device policy may be constrained to $2 \leq m \leq n < 2m$

The m-out-of-n-device policy can be generalized further, to policies in which devices are, for example, (1) weighted in various ways to get above a threshold (e.g., 2 "votes" are required to authenticate the current user, but each smart shoe only gets half a vote), (2) required (e.g., 2 devices are required but one must be the user's smartphone), or (3) excluded (e.g., for certain users, smartphones may not be used for authentication).

2.8 Group Coauthentication

Users may also be coauthenticated simultaneously, as a group. Such authentication subsumes the famous two-person concept for authenticating users who will have access to nuclear and other weapons [8, 30], or to bank vaults. For example, a two-person policy may require two users to simultaneously turn four keys, one in each hand, to gain access to a weapon-deployment system. The goal is to require both users to participate in the authentication.

Because coauthentication requires participation of multiple devices in an authentication, it may require participation of multiple users in an authentication, where each user has at least one registered device. The same communication protocols discussed in Sections 2.3–2.6 could be followed to authenticate multiple users simultaneously, with one user's device being the requestor and another user's device being the collaborator. More sophisticated group coauthentications could, for example, require participation of m-out-of-n devices from each of j-out-of-k users.

2.9 Device Sharing and Anonymous Coauthentication

Devices may be shared between users. For example, a garage-door authenticator may receive a request from a shared family car and send challenges to all the smartphones of drivers in the family. The smartphones might enforce the collaboration policy of tacitly participating if co-located with the requestor (or authenticator) and not participating otherwise. Alternatively, the authenticator might employ one of the protocol variations described earlier, in which the authenticator only communicates with the requestor. For example, the authenticator may send challenges to the car, which forwards challenges to all collaborators located inside of it and then forwards the first collaborator response it receives back to the authenticator for verification.

An interesting aspect of the coauthentication process just described, where the authenticator only communicates with the requestor, is the anonymity it provides. The authenticator does not know which user has been authenticated, and each user has plausible deniability for the authentication. Users may even share their secret keys with each other, to ensure that they are using identical secret keys and the authenticator cannot distinguish their responses to challenges. In this way, a single user authenticates as an anonymous member of a registered group of users. Authentications are still protected against attackers acquiring one of the secret keys.

It is also possible to achieve anonymous coauthentication for systems in which devices are not necessarily shared. Such authentication may be desirable for:

- a school's suicide-prevention or crime-tip hotline or website, which may, due to limited resources and prank users, want to authenticate that users are affiliated with the school;

- an organization's whistleblowing website, which may want to authenticate that users are employees; and
- a social network or online forum for members of specific communities, advertised as storing no user data because users may be reluctant to use the service otherwise.

These applications are good candidates for anonymous authentication because they require authenticating a user as a member of a group, without identifying the user. Moreover, users are not incentivized to leak their secrets to nonmembers of the group.

Anonymous coauthentication can be achieved by sharing the same key K_{AR} across all potential requestors in a group and the same key K_{AC} across all potential collaborators. Then when the authenticator receives a ciphertext (or MAC) request properly generated with K_{AR}, which identifies the group, the authenticator may send the requestor a challenge (using the requestor as a proxy for communicating with the collaborator) and await a valid response, generated with K_{AC} on a collaborator, from the requestor. Upon receiving a valid response, the authenticator completes the authentication, communicates only with the requestor, and when the session is finished, deletes all data related to the requestor, including for example its IP address. Users who do not trust the authenticator to delete information identifying the requestor device may use existing tools to anonymize this information, for example by sending their requests through an onion network [23].

Anonymous coauthentication can also be achieved without the authenticator issuing challenges, by combining these methods of sharing keys across devices in a group with the methods of Section 2.6, in which the requestor also requests collaboration and may forward the collaborator's participation to the authenticator.

In these ways, authenticators ensure the participation of multiple group keys during authentication, without identifying the specific user or devices being authenticated. Of course, the designs only protect anonymity during authentication; after authentication, services frequently have other opportunities to de-anonymize users.

3 Formal Evaluation

The principal security properties of the example coauthentication protocols shown in Figures 1–5 have been formally verified with ProVerif [3, 4]. ProVerif uses a resolution-based strategy to verify that protocols satisfy desired security properties. A benefit of using ProVerif is that it can model arbitrarily many instances of a protocol running concurrently.

Our ProVerif encodings of the coauthentication protocols, and the properties verified, are available online [5]. The protocols and properties are specified in a typed pi-calculus.

The protocols were verified under the following assumptions.

1. All keys used in the protocols are already accessible to the devices. For example, in the symmetric-cryptographic protocols, both the authenticator and requestor begin with access to K_{AR}. In the asymmetric-cryptographic protocol, each device can access its own private key and all devices' public keys.
2. Attackers may be active, and all communications go through public channels. Attackers may eavesdrop on, modify, generate, replay, and drop messages arbitrarily. Attackers may mount replay and man-in-the-middle attacks. The only exception to this assumption are messages exchanged between the requestor and collaborator to update keys (e.g., the third message in Figure 2), which are assumed to go through a private channel.
3. Attackers cannot perform cryptanalysis successfully. Attackers can only infer plaintexts from ciphertexts when also having access to the required secret key.
4. Collaborators do not participate in authentications initiated by attackers. If an attacker initiates an authentication, then the collaboration policy is assumed to enable the victim to recognize the attack and disallow the collaborator from participating.

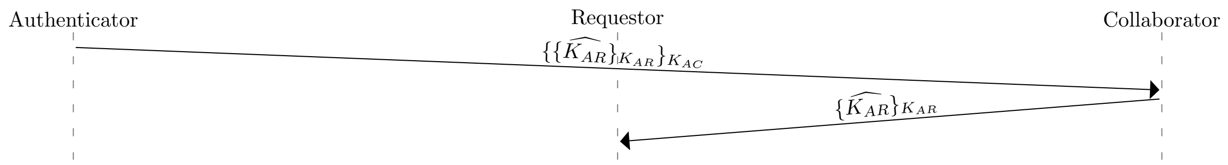
Authenticator
Requestor
Collaborator
$\{\{\widehat{K_{AR}}\}_{K_{AR}}\}_{K_{AC}}$
$\{\widehat{K_{AR}}\}_{K_{AR}}$

Figure 6: Propagation of a key update, from the old K_{AR} to the new $\widehat{K_{AR}}$, assuming a private communication channel between requestor and collaborator. This key update could be performed immediately before or during a symmetric-key coauthentication protocol, or periodically at times of low network usage. The protocol can be straightforwardly extended to propagate an update to K_{AC} as well.

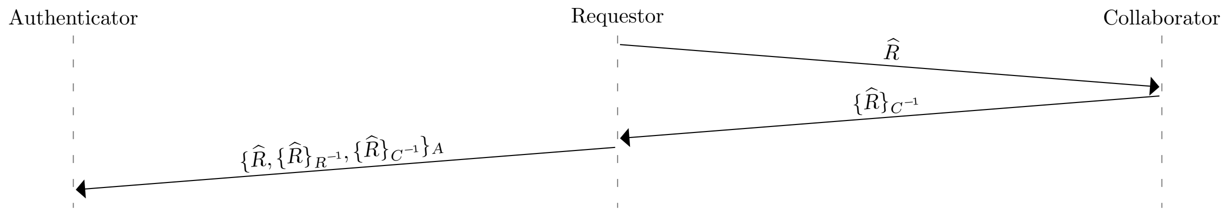
Authenticator
Requestor
Collaborator
$\widehat{R}$
$\{\widehat{R}\}_{C^{-1}}$
$\{\widehat{R}, \{\widehat{R}\}_{R^{-1}}, \{\widehat{R}\}_{C^{-1}}\}_A$

Figure 7: Propagation of a public-key update, from the old R to the new $\widehat{R}$, assuming a private communication channel between requestor and collaborator. This key update could be performed immediately before or during an asymmetric-key coauthentication protocol, or periodically at times of low network usage. The protocol can be straightforwardly extended to propagate an update to the collaborator's public key as well.

5. Before the protocols begin running, attackers may have acquired all the secret keys accessible to the requestor or the collaborator, but not both. For example, in the protocol shown in Figure 5, attackers may begin with K_{AR} and K_{RC}, both accessible to the requestor, or K_{RC} and K_{AC}, both accessible to the collaborator, but not all three of these keys. In the asymmetric-cryptographic protocol shown in Figure 3, attackers may begin with all public keys and either private key R^{-1} or C^{-1}, but not both private keys.

To implement these assumptions, each protocol was verified in multiple runs.

- The first run began with attackers knowing no secret keys.
- The second run began with attackers knowing all the secret keys accessible to the collaborator.
- The third run began with attackers knowing all the secret keys accessible to the requestor.
- The fourth run repeated the third run but on new versions of the protocols that update the requestor's secret key before initiating a coauthentication. Specifically, the new versions run the communications in Figure 6—where the authenticator sends the requestor an updated K_{AR}—before the communications in Figures 1 and 4–5, and the communications in Figure 7—where the requestor sends the authenticator an updated public key R—before the communications in Figure 3. As always, attackers are not constrained by these protocols and were free to try coauthenticating without first updating keys.

In all runs of all protocols, we attempted to verify the secrecy of authentication-complete messages, which contain session IDs encoded in ProVerif as nonces. In these implementations, session IDs are capabilities, and a user is authenticated if and only if it acquires a session ID. Hence, a successful attacker would have to acquire a session ID, which may be for an attacker-initiated, or a victim-initiated, authentication.

As expected, ProVerif found that attackers could obtain session IDs in the third run of the protocols shown in Figures 1 and 3–5. These attacks are exactly the ones described in Section 2.4, where a victim initiates a coauthentication on a requestor whose secret key is known to an attacker. As explained in Section 2.4, one method for preventing such attacks is to update the requestor's secret before completing a coauthentication.

Because the protocols shown in Figures 2 and 6–7 update keys in this way, ProVerif found no attacks in the third run of the protocol shown in Figure 2, nor in the fourth run of any of the protocols.

Besides these attacks against the non-key-updating versions of the protocols in Figures 1 and 3–5, ProVerif found no other attacks in any runs of any of the protocols.

4 Empirical Evaluation

Four versions of coauthentication were implemented, and their performance compared with that of a password-based system.

4.1 Implementations

In total, five authentication systems were implemented and evaluated:

1. To establish a baseline of performance, a password-based system composed of a requestor and authenticator. The requestor sends an encrypted username and password to the server, the username and password each being 8 hardcoded characters, as such length is common [7]; the server receives and decrypts the username and password, adds salt to the password, hashes, and checks a database to verify that the password hash matches the expected hash for the given username.

2. The canonical coauthentication shown in Figure 1.

3. The K_{AR}-updating version of coauthentication shown in Figure 2. Recall that this version adds protection against attackers who are able to acquire K_{AR} and monitor victim-initiated coauthentication communications.

4. The challengeless version of coauthentication shown in Figure 4.

5. The minimal-message version of coauthentication shown in Figure 5.

To make performance comparisons more meaningful, the five implementations were uniform to the extent possible. Each authenticator was implemented as a Java server application that queries a MySQL database storing relevant authentication secrets. Each requestor and collaborator was implemented as an Android application. All nonces, including session IDs in authentication-complete messages, were strings of 64 bits generated dynamically with Java's cryptographically strong random number generator class `java.security.SecureRandom`. Only the third implementation involved a key generation and update (for K_{AR}); all other cryptographic keys were hardcoded and assumed to be shared before the implementation began running. All cryptographic operations were implemented with 256-bit AES using standard `javax.crypto` libraries.

All messages were sent through TCP sockets over standard Wi-Fi channels, except the message from the collaborator to the requestor in the K_{AR}-updating implementation (Figure 2), which had to be sent through a private communication channel. For ease of testing, we used Bluetooth as the private channel, though it has known vulnerabilities [10].

Each run of each implementation opened new connections, including a new Bluetooth connection in the K_{AR}-updating implementation. Connections were never reused between runs of the implementations, and the Android applications were restarted for each run.

4.2 Experimental Setup and Results

The implementations were executed on the following devices. The authenticator was always a MacBook Pro laptop running macOS Sierra version 10.12.5 and having 16GB of memory and a 2.2GHz Intel quad-core i7 processor. Due to the popularity of mobile access to authentication services, the requestor was always a smartphone, a Samsung Galaxy s8 Plus running Android 7.0 and having 4GB of memory and a Qualcomm

Implementation	Bytes Transmitted	Non-I/O Authentication Time (ms) Authenticator	Requestor	Collaborator	Total	Authentication Time (ms)
Password Authentication	1074	5.50	10.9	—	16.4	282
Canonical Coauth. (Fig. 1)	2468	7.92	10.8	6.76	25.5	324
K_{AR}-updating Coauth. (Fig. 2)	10979	7.62	8.94	3.10	19.7	1462
Challengeless Coauth. (Fig. 4)	2546	5.37	4.39	5.61	15.4	317
Minimal-message Coauth. (Fig. 5)	2651	8.53	4.55	5.46	18.5	301

Table 2: Average performance of five authentication systems over 100 runs.

MSM 8998 octa-core (a 2.35GHz quad-core and a 1.9GHz quad-core) processor. The collaborator was always a Motorola Nexus 6 running Android 7.0 and having 3GB of memory and a 2.7GHz quad-core Qualcomm Snapdragon 805 processor.

Each of the five implementations was run 100 times, in a uniform environment of normal (workday) university-network usage and standard loads of kernel and user-level applications running.

The following measurements were made for each run:

- The network usage, that is, the number of bytes transmitted over the course of the run. Due to unreliability in the communication channels, the number of bytes transmitted varied with each run.
- The non-I/O real time each device consumed. This measurement was made by starting a timer when beginning to process any newly received message or request, stopping the timer when finished preparing a response, taking the difference, and summing all of these times for each device. For example, the non-I/O real time consumed by the authenticator in canonical coauthentication is the sum of the real times it consumes processing the requestor's and the collaborator's messages, including generating a challenge and verifying the collaborator's response.
- The total authentication time. This is the real time, measured on the requestor, from beginning to prepare an authentication request until finishing obtaining a plaintext session ID.

Tables 2–3 summarize the results of running each implementation 100 times.

4.3 Performance Analysis

As shown in Table 2, network-I/O activities, including connection establishment and message transmission, dominated the performance of all implementations. Across the five implementations, I/O activities comprised 92.1% to 98.7% of the total authentication time, on average. Of the 10979 bytes transmitted on average by the K_{AR}-updating implementation, 2362 bytes (22%) were attributable to the TCP connections over Wi-Fi, and 8617 bytes (78%) to the Bluetooth connection used to send one message.

As shown in Table 3, these network-I/O activities not only dominated the total authentication time, but also took a highly variable amount of time to complete, over different runs of the same implementation. The coefficients of variation (CVs) for total authentication time ranged from 39% to 65%, indicating high variance. This variance explains the sometimes-substantial differences between median and average total

	Total Non-I/O Authentication Time			Total Authentication Time		
Implementation	Average (ms)	Median (ms)	CV	Average (ms)	Median (ms)	CV
Password Authentication	16.4	17.2	0.26	282	279	0.39
Canonical Coauth. (Fig. 1)	25.5	31.6	0.19	324	251	0.47
K_{AR}-updating Coauth. (Fig. 2)	19.7	18.9	0.23	1462	1222	0.65
Challengeless Coauth. (Fig. 4)	15.4	15.0	0.25	317	269	0.54
Minimal-message Coauth. (Fig. 5)	18.5	18.5	0.14	301	251	0.60

Table 3: Statistics on the performance of five authentication systems over 100 runs. CV refers to the coefficient of variation.

authentication times observed for the same implementation. In contrast, the coefficients of variation for total non-I/O times ranged from 14% to 27%, indicating significantly less variance in non-I/O performance than in overall performance.

When excluding I/O activities, the original challengeless (Figure-4) version of coauthentication was the fastest authentication method, followed by username-password authentication.

When including I/O activities, the minimal-message (Figure-5) version of coauthentication outperformed all other versions of coauthentication. This result can be explained by the fact that the minimal-message version requires the least network I/O, and network I/O dominates the total authentication time.

Also when including I/O activities and considering *median* performance, the three non-key-updating coauthentication implementations outperformed the password-based implementation. However, when including I/O activities and considering *average* performance, the password-based implementation outperformed the coauthentication implementations. In absolute terms, the coauthentication implementations took between 19ms and 1.2s more total time to authenticate than the password-based implementation, on average.

These performance results exclude human time, though it is known to be substantial for password-based authentication systems. Human entry of a password is expected to take on the order of several seconds [19, 28, 25].

Care should also be exercised when comparing the performance of the password-based system with the performance of the K_{AR}-updating version of coauthentication. The advantages of updating K_{AR} are analogous to the advantages of updating a password, so a fairer comparison would take into account the time required to update passwords. Password update is expected to take on the order of a minute of human time [27], significantly longer than an automatic coauthentication-key update.

We conclude from these results that coauthentication performs well relative to password-based authentication systems.

5 Additional Discussion of Related Work

An (m, n) threshold scheme enables a secret to be divided amongst n entities, such that each entity has one piece of the secret and m of the n pieces are required to determine the secret [26]. An (m, n) threshold scheme has cryptographic benefits analogous to the authentication benefits of an m-out-of-n-device coauthentication policy; both protect against fewer-than-m entities acting maliciously and at-most-n-minus-m entities being unavailable to participate.

However, threshold schemes do not provide coauthentication systems, and vice versa, as they differ in goals and techniques. At a high level, threshold schemes provide ways to combine secret-pieces into a master secret. Coauthentication systems require no master secret and do not combine secrets. Coauthentication secrets (i.e., keys) are instead used independently, to indicate participation in authentications, by generating verifiable ciphertexts, MACs, signatures, etc. The m-out-of-n-device policy is also just one class of policies a coauthenticator may enforce.

The existing authentication techniques most related to coauthentication use OTPs, as discussed in Section 1. The standard use of OTPs is as follows. A user enters a username and password on a requestor device, the authenticator SMS-texts an OTP to the user's phone (which may also be the requestor device), and the user sees the OTP and enters it on the requestor device as a second password required for authentication. This use of OTPs differs from coauthentication in several ways, perhaps the most significant being that the OTPs are used in two-factor systems, while coauthentication is a single-factor system. Hence, attackers can break the OTP portion of authentications by compromising one device, the victim's phone, or even just reading the SMS messages sent to the phone. Such attacks on OTPs have received much attention lately [17, 13, 18].

More broadly, coauthentication, like other zero- or low-interaction authentication systems [6], differs from systems that require users to interact with authentication secrets, in that it shields users from those interactions, and from attacks based on those interactions (e.g., password phishing). The authentication secrets can be generated automatically, with high entropy and without concern for whether humans have the resources (cognitive ability, time, etc.) to generate, store, and enter the secrets, or to know when and where to enter them.

6 Summary and Ongoing Work

Coauthentication provides multi-factor-like security benefits of protecting against compromise of any one authentication secret, without the inconveniences of existing multi-factor techniques, such as having to enter passwords (including OTPs) or scan biometrics. Coauthentication can provide additional security benefits, including: preventing phishing and man-in-the-middle attacks; basing authentications on high-entropy secrets that can be generated, exchanged, and updated automatically; and authentication-system availability (for example by requiring participation of m-out-of-n secrets, with $m < n$). Coauthentication is amenable to many applications, including continuous, group, shared-device, and anonymous authentications. The principal security properties of coauthentication have been verified formally, and implementations have performed well compared to a password-based system.

Ongoing work is investigating the usability of various authentication mechanisms, including coauthentication. Because coauthentication can be implemented to be transparent or require a small number of voice commands or button clicks (e.g., to request and confirm authentications), we hypothesize that coauthentication mechanisms have improved usability compared to existing authentication mechanisms, particularly multi-factor mechanisms. Coupling these hypothesized usability benefits with the security benefits outlined in this paper, coauthentication—or a multi-factor authentication with coauthentication as the physical-token factor—may be a preferred authentication method for many applications.

References

[1] 3 suspects arrested in garage door opener burglary, October 2016. http://www.wtsp.com/news/suspects-sneak-into-home-with-garage-clicker-steal-items-while-residents-slept/338411648.

[2] Geneva Belford, Steve Bunch, John Day, Peter Alsberg, Deborah Brown, Enrique Grapa, David Healy, and John Mullen. A state-of-the-art report on network data management and related technology.

Technical Report 150, Center for Advanced Computation, University of Illinois at Urbana-Champaign, April 1975. Page 132. `https://archive.org/details/stateoftheartrep150belf`.

[3] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 82–96, June 2001.

[4] Bruno Blanchet. ProVerif: Cryptographic protocol verifier in the formal model, 2016. `http://prosecco.gforge.inria.fr/personal/bblanche/proverif/`.

[5] Cagri Cetin and Jay Ligatti. ProVerif coauthentication files, July 2017. `http://www.cse.usf.edu/~ligatti/projects/coauthentication/pv.zip`.

[6] Mark D Corner and Brian D Noble. Zero-Interaction authentication. In *Proceedings of the ACM International Conference on Mobile Computing and Networking*, pages 1–11, September 2002.

[7] Matteo Dell'Amico, Pietro Michiardi, and Yves Roudier. Password strength: An empirical analysis. In *Proceedings of IEEE INFOCOM*, pages 1–9, March 2010.

[8] Department of Defense. *Nuclear Weapon Accident Response Procedures (NARP)*, September 1990. DoD 5100.52-M. `https://fas.org/nuke/guide/usa/doctrine/dod/5100-52m/chap15.pdf`.

[9] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[10] John Dunning. Taming the blue beast: A survey of bluetooth-based threats. *IEEE Security & Privacy*, 8(2):20–27, 2010.

[11] Dinei Florencio and Cormac Herley. A large-scale study of web password habits. In *Proceedings of the International Conference on World Wide Web*, pages 657–666, May 2007.

[12] Nancy Gibbs. Your life is fully mobile. *TIME*, August 2012. `http://techland.time.com/2012/08/16/your-life-is-fully-mobile/`.

[13] Paul Grassi, James Fenton, Elaine Newton, Ray Perlner, Andrew Regenscheid, William Burr, Justin Richer, Naomi Lefkovitz, Jamie Danker, Yee-Yin Choong, Kristen Greene, and Mary Theofanos. NIST special publication 800-63B digital authentication guideline, June 2017. `https://doi.org/10.6028/NIST.SP.800-63b`.

[14] Philip G Inglesant and M Angela Sasse. The true cost of unusable password policies: Password use in the wild. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 383–392, April 2010.

[15] International Standards Organization. Information technology – Automatic identification and data capture techniques – QR Code bar code symbology specification. Technical report, February 2015. ISO/IEC 18004:2015. `https://www.iso.org/standard/62021.html`.

[16] International Standards Organization. Information technology – Trusted platform module library – Part 1: Architecture. Technical report, August 2015. ISO/IEC 11889-1:2015. `https://www.iso.org/standard/66510.html`.

[17] Radhesh Krishnan Konoth, Victor van der Veen, and Herbert Bos. How anywhere computing just killed your phone-based two-factor authentication, 2016. `http://fc16.ifca.ai/preproceedings/24_Konoth.pdf`.

[18] Charles McColgan. Issues with SMS deprecation rationale, September 2016. `https://github.com/usnistgov/800-63-3/issues/351`.

[19] William Melicher, Darya Kurilova, Sean M Segreti, Pranshu Kalvani, Richard Shay, Blase Ur, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Michelle L Mazurek. Usability and security of text passwords on mobile devices. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 527–539, May 2016.

[20] Jorge Milian. Burglars using stolen garage-door openers in boynton beach, August 2015. `http://www.mypalmbeachpost.com/news/crime--law/burglars-using-stolen-garage-door-openers-boynton-beach/ZU0J1uARHfHTpRnGA6A4ZJ/`.

[21] David M'Raihi, Salah Machani, Mingliang Pei, and Johan Rydell. TOTP: Time-based one-time password algorithm. RFC 6238, May 2011. `http://www.rfc-editor.org/rfc/rfc6238.txt`.

[22] Lawrence O'Gorman. Comparing passwords, tokens, and biometrics for user authentication. *Proceedings of the IEEE*, 91(12):2021–2040, December 2003.

[23] Michael Reed, Paul Syverson, and David Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected areas in Communications*, 16(4):482–494, 1998.

[24] M Angela Sasse, Sacha Brostoff, and Dirk Weirich. Transforming the 'weakest link'–a human/computer interaction approach to usable and effective security. *BT Technology Journal*, 19(3):122–131, July 2001.

[25] Florian Schaub, Ruben Deyhle, and Michael Weber. Password entry usability and shoulder surfing susceptibility on different smartphone platforms. In *Proceedings of the International Conference on Mobile and Ubiquitous Multimedia*, pages 13:1–13:10, December 2012.

[26] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[27] Richard Shay, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Alain Forget, Saranga Komanduri, Michelle L Mazurek, William Melicher, Sean M Segreti, and Blase Ur. A spoonful of sugar?: The impact of guidance and feedback on password-creation behavior. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 2903–2912, April 2015.

[28] Richard Shay, Saranga Komanduri, Adam L Durity, Phillip Seyoung Huh, Michelle L Mazurek, Sean M Segreti, Blase Ur, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Can long passwords be secure and usable? In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 2927–2936, April 2014.

[29] Catherine S Weir, Gary Douglas, Martin Carruthers, and Mervyn Jack. User perceptions of security, convenience and usability for ebanking authentication tokens. *Computers & Security*, 28(1):47–62, 2009.

[30] Margaret Woodward. Air force instruction 91-104, April 2013. `https://fas.org/irp/doddir/usaf/afi91-104.pdf`.