SARA: Secure Android Remote Authorization

Abdullah Imran, Habiba Farrukh, Muhammad Ibrahim, Z. Berkay Celik, and Antonio Bianchi
Purdue University, {imran8, hfarrukh, ibrahi23, zcelik, antoniob}@purdue.edu

Abstract

Modern smartphones are equipped with Trusted Execution Environments (TEEs), offering security features resilient even against attackers able to fully compromise the normal operating system (e.g., Linux in Android devices). The academic community, as well as the smartphone manufacturers, have proposed to use TEEs to strengthen the security of authorization protocols. However, the usage of these protocols has been hampered by both practicality issues and lack of completeness in terms of security.

To address these issues, in this paper, we design, implement, and evaluate SARA (Secure Android Remote Authorization), an Android library that uses the existing TEE-powered Android APIs to implement secure, end-to-end remote authorization for Android apps. SARA is *practical* in its design, as it makes use of Android APIs and TEE features that are already present in modern Android devices to implement a novel secure authorization protocol. In fact, SARA does not require any modifications to the Android operating system nor to the code running in TrustZone (the TEE powering existing Android devices). For this reason, it can be readily used in existing apps running on existing smartphones. Moreover, SARA is designed to ensure that even developers that have no experience in implementing security protocols can make use of it within their apps. At the same time, SARA is *secure*, since it allows implementing authorization protocols that are resilient even against attackers able to achieve root privileges on a compromised Android device.

We first evaluate SARA by conducting a user study to ascertain its usability. Then, we prove SARA's security features by formally verifying its security protocol using ProVerif.

1   Introduction

People are becoming increasingly reliant on their smartphones and other smart devices in their daily lives. Developers have fueled this reliance by designing solutions to most of the everyday problems in the form of applications (apps). Furthermore, an increasing number of applications deal with security-sensitive tasks such as online payments or online banking. New regulations, such as the European Banking Authority's Strong Customer Authentication [13], require that electronic payments be made secure by the use of two-factor authentication. Smartphone applications are often used to implement the second factor or sometimes even both factors. The security measures
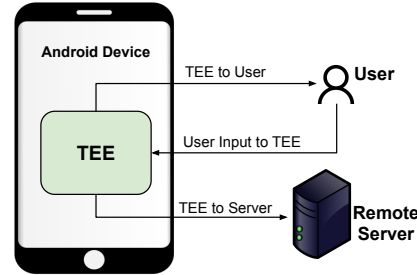

Figure 1: TEE-enforced communication channels

of these applications rely on the integrity of the underlying OS. Unfortunately, more attacks are being discovered that can compromise the entire OS, and, consequently, the security of the apps running on a mobile device [5].

Trusted Execution Environments (TEEs) provide a solution for this issue by running security-sensitive tasks in a separate execution environment. In the case of Android devices, ARM's TrustZone [15] serves as the TEE for most devices, and it offers hardware-backed, strong security services to third-party apps [8]. These security services, like hardware backed key storage and Secure UI, help in achieving stronger *security* for various functionalities, such as user authentication and device attestation, which are prerequisites for the authorization of security-sensitive operations.

As an example, consider a banking app that allows a user to make transactions. For the transaction to be authorized, the banking server needs to ensure that the user identifies themselves to prove they are the right person authorizing the transaction. We call this process User Authentication. The server also needs to carry out User Confirmation, a process ensuring that the user knows exactly what transaction is taking place and that the user willingly confirms that transaction. In addition, the server needs a guarantee that the client device has the security features that are prerequisites to perform such a transaction by means of a process known as Device Attestation. These processes allow backend servers to determine whether an action was legitimately authorized.

If the adversary obtains root privileges on an Android device, all of these steps are vulnerable. Stored credentials can be stolen, confirmation messages can be spoofed, and the presence of security features can be faked. However, using TEEs, it is

possible to implement remote authorization resilient against attackers capable of compromising the OS (i.e., *root attackers*), by securing all channels involved in the communication between the user, the app, and the app's remote backend (as shown in Figure 1).

Android itself provides developers with TEE-enforced features through dedicated APIs. Unfortunately, existing apps rarely use these features, as they require developers to have abundant experience with these complex cryptographic APIs. Furthermore, to achieve root-resilient authorization, these APIs need to be combined, which is an even harder task to accomplish. Prior works also proposed several solutions [31, 35, 36, 40, 44, 45] to implement root-resilient authorization protocols, but they require hardware, OS, or TEE modifications, making them impractical to be used on real devices.

To address these issues, we propose SARA, a library implementing a novel authorization protocol on top of the existing TEE-enforced APIs currently offered by Android. SARA is designed to be *practical* and *secure*.

SARA's practicality derives from the fact that any developer can integrate it within their Android apps with only a few lines of code. Moreover, the library does not require any hardware modifications or changes to the existing Android or TEE code, since it relies on the existing Android APIs, making it easy to deploy on existing smartphones. Lastly, SARA ensures that all the cryptographic procedures and checks are handled by the library, leaving less room for security flaws introduced by developers.

SARA's security is guaranteed by the fact that in its authorization protocol, all the channels in Figure 1 are under the control of the TEE and an app's backend server can verify that every authorized action has been handled by the TEE. In addition, it ensures that users are aware of the actions being authorized and have biometrically authenticated themselves.

We proved the security of SARA by using ProVerif [27]. Then, we evaluated the practicality of SARA with a user study. The results of this study show that using our library significantly improves the ability of developers to implement root-resilient remote authorization within Android apps.

In summary, these are the main contributions of our work:

- We analyze the existing TEE-enforced APIs in Android and how existing apps use them. Our analysis of 112,886 apps revealed that these APIs are currently not used to improve the security of authorization protocols.

- To address the lack of usage of such features, we design a new security protocol that utilizes the already existing Android APIs to perform a root-resilient form of remote authorization, and we verify its security using ProVerif.

- We develop an Android library and a server module to allow developers to easily implement our new security protocol, and we evaluate its usability with a user study.

In the spirit of open science, and to ensure our work benefits the entire Android community, we have publicly released SARA [21].

2   Background and Threat Model

This section introduces background information about TEEs and Android, which will be useful throughout the paper.

2.1   Trusted Execution Environments

A Trusted Execution Environment (TEE) is a "secure" isolated execution environment that runs concurrently with the non-secure "rich" Android OS. There are many different implementations of TEEs, e.g., ARM's TrustZone, which serves as the TEE for most Android devices. Systems that employ the use of a TEE can run code and store data in two separate environments. The secure environment consists of a trusted kernel along with trusted applications. They are both signed by the OEM and isolated from the non-secure environment. To allow third-party applications in the non-secure world to access services that run in the secure environment, Android provides APIs that can interact with the "trusted" applications without granting direct access to them.

2.2   Terminology

We provide a common terminology for some recurring terms in this paper. A *legitimate* user is the one that is adjudged by a system to be the unique individual that holds the authority to perform certain actions. *TEE-enforced APIs* are APIs exposed to the application by the Android OS. They give an app the ability to access TEE-controlled functionality, such as cryptographic key storage. *Authorization* is the process through which a system determines if the legitimate user wishes to perform a certain action.

2.3   TEE-enforced API Usage

Key Storage Android provides a Key Storage to store and use cryptographic keys. This Key Storage is implemented within the TEE, ensuring that cryptographic material (i.e., the stored cryptographic keys) cannot leave the TEE. In fact, keys inside the TEE are inaccessible directly by Android apps, and any cryptographic operation such as encryption or signing with these keys is performed inside the TEE. This feature allows apps to use these keys without ever exposing them to the OS, which ensures that even root attackers cannot access these keys. The Key Storage can generate keypairs. In this case, the private key remains stored in the TEE, while the public key is returned to an app. The app can later invoke the TEE to sign arbitrary data using the TEE-stored private key.

Key Attestation In version 7.0, Android introduced a key attestation mechanism. This allows a remote server to cryptographically verify different properties of the obtained public key, including whether the corresponding private key is stored safely inside the TEE. The server first sends an attestation
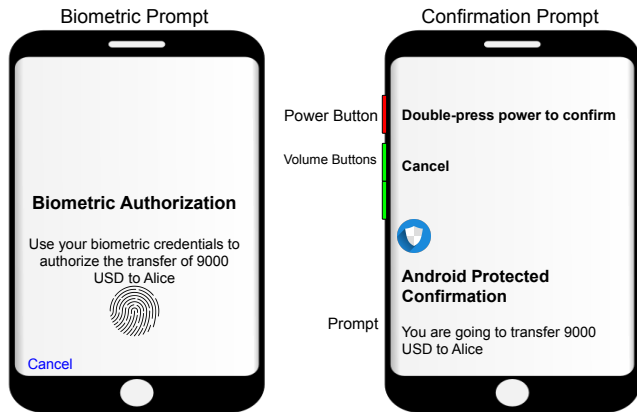
Figure 2: Sample display of Biometric Prompt (left) and Confirmation Prompt (right)

challenge to the client device. The keypair is then made by the client device by setting the attestation challenge to the one sent by the server. This attestation challenge is embedded within the key's corresponding attestation certificate, allowing the remote server to ascertain that the keypair was created in response to a specific request for which a specific attestation challenge was sent. The attestation certificate consists of a certificate chain, rooted to a Google-owned private key, signed with a secret key stored within the TEE. After the keypair is created, the corresponding certificate chain is sent to the server. The server then parses the certificate chain to check if it has the correct attestation challenge and shows that the keypair has the required properties. Specifically, obtaining an attestation certificate chain having, as a root certificate, a Google's Hardware Attestation Root Certificate, guarantees that the corresponding key has been generated within a TEE-based KeyStore.

Android Protected Confirmation An essential component of remote authorization is the ability to show the user the action being authorized and get a confirmation from them.

In Android 9, the Android Protected Confirmation [4] feature was introduced, which allows a user to perform TEE-enforced confirmation. Specifically, by using this feature, the TEE displays the prompt on a trusted screen (Figure 2) and takes the confirmation input through a hardware button. The TEE can store keypairs that can only be used to sign the prompt displayed on the trusted screen and only when the user presses the hardware confirmation button. If a remote server receives a prompt signed with such a keypair, it has the guarantee that this prompt was displayed via the trusted screen and a user pressed the hardware confirmation button in response to this prompt.

BiometricPrompt API Another crucial part of remote authorization is the ability to authenticate a user to verify that it is a *legitimate* user (and not, for instance, someone able to obtain physical access to a device). Android accomplishes this feature with the help of biometric authentication controlled by the TEE.

In this case, only the user whose biometrics are registered on the device is considered the *legitimate* user.

As of Android 9, Android's recommended API to use for biometric authentication is the BiometricPrompt API [6]. The BiometricPrompt API is used in conjunction with a keypair stored within the TEE. This keypair can be used for signing only upon successful biometric authentication. Whenever the keypair needs to be used for signing, a biometric prompt is displayed, asking the user to authenticate themselves biometrically. The title and description of the biometric prompt can be set using the `BiometricPrompt.Builder` class. At this point, the key can be used once for signing with a previously provided `CryptoObject`. This signature can then be verified by the server with the guarantee that a *legitimate* user is authenticated.

2.4 Threat Model

We assume that the adversary has completely compromised the OS and has all root privileges. These include control of input, output, and the code execution flow of any app.

Physical security is of paramount importance in mobile devices as they are more easily accessible to adversaries due to their portable nature. Hence, we also assume that the adversary has sporadic physical access to the device. The adversary's physical access and the user's physical access are mutually exclusive. The adversary can physically access the device if the user is not physically carrying the device. This can occur at any random time when the user leaves the device unattended, therefore, allowing the adversary to have physical access to it. We consider the duration of the physical access by the adversary to be no greater than a few minutes at a time. Furthermore, we assume that there is a time gap between the user accessing the device and the adversary accessing it, i.e., the adversary is unable to press a hardware button immediately after the user inputs their fingerprint.

We assume that the adversary has not compromised the code or data within the TEE. In fact, exploits that could grant this capability to the adversary are uncommon. From January 2020 till December 2021, there have been 453 Android CVE's that could result in an escalation of privilege but only 8 CVE's related to TEEs in Android [20].

The adversary does not have any control over the remote server. Hence, we do not consider any attacks that involve compromising the remote server directly.

Furthermore, we assume that the initial trust between the Android device and the remote server has already been established, i.e., the user was able to legitimately authenticate themselves at least once on that device (usually done with login credentials). Therefore, establishing the initial trust between the device and the remote server is out of scope for this paper. Consequently, attacks based on this lack of initial trust, like cuckoo attacks [39], are also out of scope for this paper.

In summary, we consider as a Trusted Computing Base (TCB), the data stored, and the code running, on a remote server and the TEE.

3   Motivation

Authorization involving mobile devices is becoming a necessity. Regulations like the European Banking Authority's Strong Customer Authentication [13] have made it almost impossible to process payments without using your mobile device. Financial applications that run on modern smartphones are ample, and without secure authorization protocols in place, users could suffer severe financial losses in the event of a security breach. Even local authorization use cases such as controlling insulin pumps using your smartphone [9] require that secure authorization take place to avoid any overdose. Under our threat model, for these authorizations to be secure, they have to use the TEE. Therefore, there is a need for a TEE-based solution that can handle all these forms of authorizations, and can be used in all modern mobile devices.

In this section, we first conduct a large-scale market analysis of Android apps to ascertain how apps in the market are using TEE-enforced APIs. Next, we discuss the issues and limitations of these TEE-enforced APIs. Lastly, we discuss possible authorization protocols that use TEE-enforced APIs and their flaws, which prevent them from fulfilling our goals.

3.1   TEE-enforced API's Market Analysis

We conduct a large-scale analysis of existing apps on the Google Play Store, using AndroZoo [25] to gather a dataset of Android apps. To limit our dataset to prominent applications, we used a web scraper to filter the apps based on their category and number of installs. The categories chosen for filtering were Business, Finance, Tools, Communication, and Medical. We selected all apps that belonged to these categories and had at least 1,000 installations on Google Play Store, resulting in a dataset of 112,886 apps downloaded in January 2021.

As discussed in Section 2.3, there are three TEE-enforced APIs relevant for authorization. Since Bianchi et al. [26] have already investigated the usage of the FingerPrint API, we focus our analysis on key attestation and Android Protected Confirmation APIs. The FingerPrint API is the deprecated version of the BiometricPrompt API but still has a very similar implementation. Hence, the older analysis results are still relevant.

First, we gauge the number of apps that are attempting to use these APIs. We use Apktool [19] to disassemble the apps into smali code [22]. We then search the smali code for methods necessary for the usage of these APIs. For key attestation, we search for the method `setAttestationChallenge`. For Android Protected Confirmation, we search for the method `setUserConfirmationRequired`.

We manually analyze the apps further to see how the APIs are being used. We decompile the apps using Jadx [10] and then search and trace within the Java code for the methods to see how they are used. We are interested in seeing what data is being sent to or from a remote server. In case no communication occurs with the server while these APIs are being used, we check how these APIs are then used locally.

We only found 5 apps that use key attestation, and none of them use it for remote attestation. The certificate chain containing the attestation data is never sent to any remote server. The apps are either checking the attestation data locally or are using the attestation data to determine the state of the device and display it to the user. Checking attestation data locally defeats the purpose as a root attacker will be able to alter the data to bypass any checks that use this attestation data as a measure of the device's integrity. We find no usage of Android Protected Confirmation within our dataset.

The user study we conduct, as discussed in Section 7, suggests that the lack of usage of these APIs is due to the complex nature of their implementation. From our analysis, we conclude that there is a need to provide developers with an easier way of incorporating these APIs into their apps.

3.2   TEE enforced API's Intrinsic Limitations

The Fake Prompt There is an intrinsic issue with the BiometricPrompt API, which lies in the fact that the prompt shown to the user requesting for their biometric scan does not guarantee that the key will be used for the purpose displayed on the prompt. A biometric prompt like the one shown in Figure 2 (left) can be manipulated by a root attacker, making the user think they are authorizing a certain action while the adversary silently uses the key for their own purposes [41].

The Illegitimate User Android Protected Confirmation used alone for authorization only ensures that a hardware button was pressed in response to a certain message, but gives no assurance of who pressed the button. A confirmation prompt like the one shown in Figure 2 (right) is controlled by the TEE. The key unlocked by pressing the hardware button can only be used to sign the prompt displayed. However, pressing the hardware button does not necessitate physical input from the legitimate user, as anyone can press the button.

Everlasting Biometric We found that using the Biometric Prompt to unlock a key for use with a `CryptoObject` results in the key being unlocked as long as the `CryptoObject` is not used. This means that if a user authenticates themselves to a BiometricPrompt, the resultant `CryptoObject` can be used long after the biometric input took place. A root attacker can exploit this issue by delaying the signing process after successful user authentication.

Overwriting Confirmation We found that a key requiring user confirmation to be unlocked can only sign the last accepted Confirmation Prompt. This is an issue because an accepted Confirmation Prompt can be overwritten by another accepted Confirmation Prompt. Even if a user accepts a Confirmation Prompt, the attacker can stop that particular prompt from being signed. The attacker can then display and accept a Confirmation Prompt of their own choosing and sign that instead. Further-

more, the accepted Confirmation Prompt does not expire due to time and is not bound to any particular key with any confirmation timeout (i.e., the confirmation must have been done within a certain duration of the signing). This results in an attacker having the capability to set the accepted Confirmation Prompt when they have physical access to the device and even use it for future attacks due to the lack of any timeout mechanism.

3.3 Vulnerable Protocols

This section will discuss possible authorization protocols based on the TEE-enforced APIs and their vulnerabilities.

Solitary Biometric Usage, Fake Prompt Attack A common protocol for authorization in Android is using Biometric Prompt alone. This protocol suffers from the *Fake Prompt* issue. After the user has provided the biometric input for authorization, the attacker can then sign any malicious prompt without alerting the user. If the protocol involves using a server generated nonce or timeout to avoid such attacks, then the root attacker can exploit the *Everlasting Biometric* issue.

Solitary Confirmation Usage, Fake User Attack An alternative protocol for authorization is using Android Protected Confirmation alone. This protocol suffers from the *Illegitimate User* issue. An attacker wishing to authorize a malicious action simply has to wait until they have physical access to the device and can then press the physical button when prompted to accept the malicious prompt.

Naive Combined Usage, Separation Attack The naive method of using both APIs together in an authorization protocol is to use them in parallel with separate keys, one after the other, to sign the same prompt and send both signatures back to the server. Since there is no mechanism to ensure both signings are done in an atomic manner (at the same time), each signing only guarantees the security properties as are in the case of solitary usage. Therefore, separate attacks can be carried out to bypass the two signing mechanisms, identical to the attacks as in the case of solitary usages.

For example, if the Android app used a Biometric Prompt before using a Confirmation prompt, the attacker can wait for the user to provide the biometric input in response to a legitimate Biometric Prompt. By exploiting the *Everlasting Biometric* issue, the root attacker can stop the original signing from happening and instead pause the process till they have physical access to the device. The root attacker will then generate a new malicious request to the server. The already unlocked Biometric key can then be used to sign the malicious request. The attacker can then provide the physical input for the confirmation prompt. Using server generated nonces and timeouts does not hamper this attack as the attacker can generate a new request to the server at the time of signing. Similarly, the attack also works in the case when the Confirmation Prompt is used before the Biometric Prompt.

Rectified Combined Usage, Timed Attack Another method of combining BiometricPrompt API and Android Protected Confirmation is to use them both with a single key to sign a single prompt. The two prompts can be displayed in any order. However, in this case, an attack is possible by exploiting the *Everlasting Biometric* and *Overwriting Confirmation* issues. A root attacker can pause the process after the biometric input has been provided and wait till they have physical access. Afterward, the attacker can generate a new malicious request to the server, display a malicious confirmation prompt, accept it using the physical button and then sign it using the key as it has already been biometrically unlocked. Server generated nonces and timeouts do not protect against this attack due to the malicious request being made at the time of signing.

4 SARA Overview

In this section, we first discuss the goals and requirements of our work. Then, we introduce our new authorization protocol, which we use as the basis to implement a library that can be used by developers within their apps.

4.1 Design Goals and Requirements

The first goal for our library is to be *practical*. This requirement translates to two usability goals.

UG1: Developers who wish to perform remote authorization in their apps should only need to have minimalistic knowledge of the authorization framework to integrate it correctly into their apps. Developers should also not worry about the complicated cryptographic details that an average developer without any security experience would have no knowledge about. To ensure this property, the library should take care of all the intricacies of the authorization process.

UG2: The library must be compatible with all modern Android devices and be integrable within all existing and new apps. To ensure this property, our library has to make use of hardware features that already exist within devices, as well as APIs that are already provided by the Android OS. Using APIs that are provided by the Android OS ensures that the library will work across Android devices regardless of the TEE and code running inside the TEE. Our library has a minimum requirement of Android 9 which means that 74% of existing Android devices [17] can use our library. The only limitation is that necessary hardware features, like a biometric scanner or a TEE itself, must be available on the device. Since the APIs are provided by the Android OS and the necessary hardware features are recommended by the Android Compatibility Definition Document [3], the number of devices that have these hardware features is on the rise.

The second goal is that our library should provide a *secure* form of remote authorization that can be achieved by current Android devices. The library should both be resilient against root attackers and verify the physical presence of the legitimate user. In turn, to guarantee these two properties, we design our library to achieve the following five security goals:

Table 1: Fulfillment of security and practicality goals

Authorization Solution	UG2 (No impractical OS or Hardware Modifications)	SG3 (Secure UI)	SG4 (Biometric Authentication)
SecurePay [31]	✗	✓	✗
Truz-Droid [44]	✗	✓	✗
Yubikey [18]	✓ (uses external hardware)	✗	✓
SARA	✓	✓	✓

SG1: All authorized actions are signed by TEE backed keys

SG2: The server can attest to the state of the keys used for signing.

SG3: The legitimate user is aware of the action they are authorizing.

SG4: Physical presence of the legitimate user is verified at the time of authorization.

SG5: The server can verify that the legitimate user knowingly authorized an action.

As mentioned in Section 3.3, the existing Android TEE-enforced APIs, when used in isolation or simple combinations, have intrinsic issues that prevent them from achieving these goals. Therefore, there is a need for a new remote authorization protocol for Android devices that, while using the existing APIs, can achieve the aforementioned security goals.

4.2 Comparison with Similar Efforts

We analyze previously published works to ascertain if they can fulfill our design goals and requirements. Specifically, we focused on SecurePay [31] and Truz-Droid [44]. These two approaches are currently state of the art in the field, and, in their evaluation, they show they are better than older published works, such as TrustPay [45]. We also compare with Yubikey [18] which is currently being used in the industry.

SecurePay [31] proposes securing payments by leveraging the TEE to use the mobile device as a secure form of 2FA. SecurePay is composed of an Android library, as well as a Trusted Application (TA) which resides inside the TEE. In SecurePay, an encrypted transaction message is sent to the Android device which can only be decrypted by the SecurePay TA. The Secure-Pay TA then decrypts and displays the transaction details to the user, using a trusted UI. The user can then verify the transaction details and accept the transaction. SecurePay requires adding a new TA inside the TEE. In the current Android ecosystem, this is hard to accomplish in practice since only TAs signed by OEMs are allowed inside the TEE. For this reason, Secure-Pay violates **UG2**. Additionally, SecurePay does not verify the identity of the user accepting the transaction on the trusted UI. Therefore, under our threat model, it is possible that an attacker is able to process illegitimate transactions if they have physical access. Hence, SecurePay violates **SG4**.

Truz-Droid [44] suggests using a two-part solution. The first part secures the user's interaction with the device by providing a secure keyboard that feeds the input directly to the TEE. Moreover, Truz-Droid introduces a secure output screen that allows the TEE to show information directly to the user. The user can distinguish between the secure screen and the normal screen with the help of a LED. To handle the secure inputs and outputs inside the TEE, Truz-Droid requires a TA into the TEE. Since it requires modifications to the Android OS and hardware modifications to the device, Truz-Droid violates **UG2**. Additionally, Truz-Droid does not authenticate users using biometrics; hence it also violates **SG4**.

Yubikey [18] is one of the most popular hardware based 2FA authentication techniques [32] available on the market. To authenticate themselves, the user makes use of hardware token with an in-built finger touch sensor. Traditionally this sensor guarantees physical presence, but it does not biometrically authenticate the user. However, the newest available version of Yubikeys also supports fingerprint authentication [1], hence satisfying **SG4**. However, even though it does not require Android or hardware modifications, this solution requires a separate hardware token. For this reason, it partially violates **UG2**. Finally, Yubikeys do not offer any form of Secure UI, hence their usage violates **SG3**.

Table 1 summarizes how these solutions fulfill our practicality and security requirements. In the following sections, we will prove that SARA fulfills all these requirements. To the best of our knowledge, SARA is the only system that (1) can be practically used in modern Android devices without requiring any changes to hardware, OS or TEE code, and (2) is secure within our threat model where root attackers can also have limited physical access to the device.

4.3 SARA's Protocol Design

We now introduce the design for our new authorization protocol built on top of already existing Android APIs. The APIs have to be used together in a cryptographically secure manner to ensure that the legitimate user authorized the action. To this aim, our protocol uses the concept of double signing.

We will now describe each of the steps that take place within our protocol, as illustrated by Figure 3. The first two steps only need to be executed once, while steps 3 and 4 need to be executed for each authorized action.

Step ❶: Keypair Generation First, our protocol uses Android's KeyStore to create two asymmetric signing keypairs on the client Android app. Android's KeyStore uses the TEE to store the keypairs, which accomplishes **SG1**. We generate both of these keypairs in accordance with the key attestation protocol. Both of them have a separate attestation challenge that is sent by our server module. Furthermore, one of the
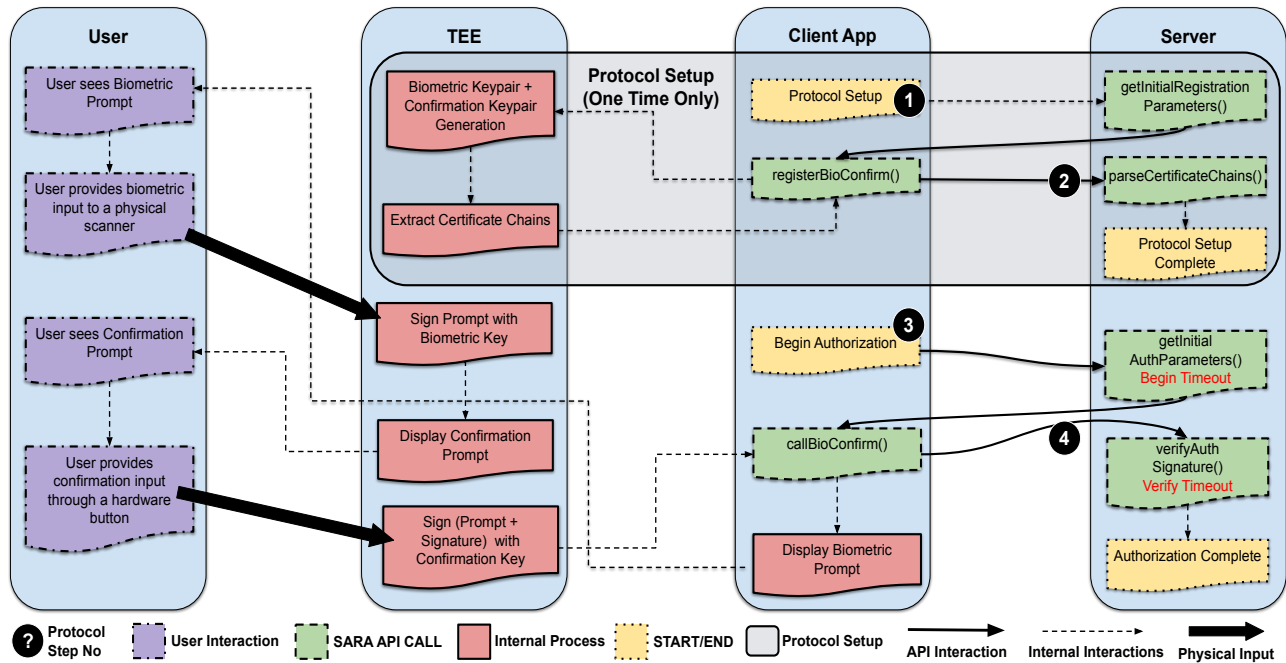
Figure 3: Biometric Confirmation Protocol with SARA's API interaction and internal processing handled by SARA

keypairs is made to be used with the BiometricPrompt API (`biometricKeypair`). The other keypair is created to be used in conjunction with the Android Protected Confirmation API (`confirmationKeypair`).

Step ②: Double Attestation To achieve **SG2**, our protocol attests both the created keypairs on the remote server in accordance with the key attestation protocol available in Android devices. Specifically, once the remote server receives the certificate chains that correspond to both keypairs, our protocol analyzes the attestation data encoded in these chains. The server then verifies that the separate attestation challenges for the keypairs match the ones generated by our server module.

Our protocol ensures that attestation data from both certificate chains match since they come from the same device, except for the part that gives detail on the key properties. For the attestation to be successful, the attestation data must show that both keypairs are stored in secure hardware and that the device is not compromised. Furthermore, the remote server checks the attestation data to ensure that one key can only be used with biometric authentication, while the other key can only be used when a confirmation input is provided in response to a confirmation prompt.

Step ③: Double Signing To achieve **SG3** and **SG4**, our protocol needs to verify that the legitimate user is aware of the action they are authorizing and provides a physical confirmation. Therefore, for every action that requires authorization, both keypairs are used to sign a prompt that summarizes the action being authorized (i.e., *Are you sure you want to send $50 to Alice*), as well as a nonce to protect against replay attacks.

Both the prompt and the nonce are generated by our server module. After the prompt and nonce are sent to the client app, a timeout is started on the server.

More precisely, the signing is performed the following way:

```
intermediateSignature = sign(prompt+nonce,
    biometricPrivateKey)
finalSignature = sign(prompt+nonce+intermediateSignature,
    confirmationPrivateKey)
```

where `sign()` signs the first argument using the private key specified in the second argument. `biometricPrivateKey` is the private key part of `biometricKeypair` and `confirmationPrivateKey` is the private key part of `confirmationKeypair`.

The `biometricPrivateKey` is used to sign the prompt displayed by the BiometricPrompt API. Similarly, `confirmationPrivateKey` is used to sign the prompt displayed by using Android Protected Confirmation. Due to this double signing, our protocol gives a guarantee to the remote server that the user has both seen the prompt and biometrically authenticated. The double signing along with the server timeout ensure that an attacker cannot exploit the *Everlasting Biometric* or the *Overwriting Confirmation* issues. The confirmation fulfills the requirement for **SG3** by ensuring the user sees the action they are authorizing. The biometric authentication achieves **SG4** by verifying the user is legitimate and present, through a physical biometric input.

Step ④: Signature Verification To achieve **SG5**, our protocol sends back the prompt signed by both keypairs to the

server module along with both signing outputs. The server first ensures that the signatures were received within the timeout limit. We then verify both signatures using the certificate chains that were received by the remote server in **Step ❷**. The verification is performed in the following way:

```
verify(finalSignature, prompt+nonce+intermediateSignature,
    confirmationCertificateChain)
verify(intermediateSignature, prompt+nonce,
    biometricCertificateChain)
```

where `verify()` verifies that the first argument is a valid signature of the second argument using the certificate chain specified in the third argument. `biometricCertificate-Chain` is the public certificate chain of `biometricKeypair` and `confirmationCertificateChain` is the public certificate chain of the `confirmationKeypair`. Upon successfully verifying both signatures, our protocol ensures that the legitimate user has confirmed the action and **SG5** has been achieved.

4.4 Alternative Protocol

Prior to Android 11, we only had two options. Either biometric authentication was required for every use of the key, or a validity duration could be set for the key, in which case all forms of authentication can be used, such as a pin or password. The latter option does not enforce the use of biometric authentication and can be bypassed using any authentication method such as a screen lock as long as it is done within the set time limit. Even though biometric authentication for every use of the key enforces the need for biometric authentication, it does not set any time limit in which the signing must take place after the authentication (*Everlasting Biometric*).

Starting from Android 11, we could use the new method `setUserAuthenticationParameters` to enforce biometrics as well as set a time limit for the authentication. Using this method, we can create a single keypair in **Step ❶** of Figure 3 to be used with both BiometricPrompt API and Android Protected Confirmation API. The time limit, combined with a server generated nonce and timeout, is able to overcome the *Overwriting Confirmation* and *Everlasting Biometric* issues. However, since the method is not backward compatible with previous Android versions, SARA does not use this protocol. SARA's protocol allows more Android devices to perform secure authorization as its minimum requirement is Android 9. As shown ahead in Section 7, even with the Android 11 approach, it is not trivial for developers to implement it. Furthermore, without knowing about the issues we highlight in this paper, developers are unlikely to try and avoid these issues by using methods such as `setUserAuthenticationParameters`.

5 Implementation

In this section, we present the library we developed, which implements the remote authorization protocol as described in Section 4.3. The library is divided into two parts: An Android library package and a server module. First, we demonstrate how developers can use the library within their own apps. Then, we discuss the internal workings of our library and how our implementation follows the protocol outlined in Section 4.3.

5.1 SARA's API Usage

SARA's API uses the existing Android APIs to implement the authorization protocol discussed in Section 4.3. To use SARA to implement the authorization protocol, changes need to be made on both the server and the Android app. Listing 1 shows an example of how the SARA's API can be used by developers within the remote servers for their Android apps. Listing 2 shows an example of how the SARA's API can be used by developers within their Android apps. To use SARA's API on the server side, the developer must do the following:

Initialize `AttestationServer`: To start off, the developer initializes an instance of `AttestationServer`.

Setup Protocol: To set up the protocol for a certain user, the developer must call the method `getInitialRegistrationParameters` on `AttestationServer`. As an argument, this method requires a unique identifier for the user the protocol is being set up for. The method returns registration parameters that must be sent to the client Android app. The developer then must call the method `parseCertificateChains` on `AttestationServer`. This method requires the unique user identifier as well as the certificate chain received from the client Android app as arguments. The method returns `true` if the protocol has been set up successfully.

Authorize Action: To authorize an action for a certain user, the developer must call the method `getInitialAuthParameters` on `AttestationServer`. As an argument, this method requires a unique identifier for the user. The method returns authorization parameters that have to be sent to the client Android app. The developer then has to call the method `verifyAuthSignature` on `AttestationServer`. As arguments, this method requires the unique identifier for the user as well as the signed prompt received from the client Android app. The method returns `true` if the action is authorized successfully. The developer has to inform the client Android app of this success.

To use SARA's API on the Android app, a developer needs to perform the following steps:

Initialize `AttestationController`: To start off, the developer initializes an instance of `AttestationController`.

Setup Protocol: To setup the use of the protocol, the developer must call the method `registerBioConfirm` on `AttestationController`. As an argument, this method requires a reference to the current Android context and the registration parameters received from the server. The method returns the certificate chains that have to be sent to the server.

Authorize Action: To authorize an action, the developer must call the method `callBioConfirm` on `AttestationController`. As an argument, this method requires a reference to the current Android context and the authorization parameters received from the server. The method returns the signed prompt

```
def ServerCode()
{
  #Access library class
  myAS = AttestationServer()

  #Setup authorization protocol for user X (only once)
  regParams = myAS.getInitialRegistrationParameters(X_UniqueID)
  #Send regParams to Android client app
  #Receive certChains from Android client app
  myAS.parseCertificateChains(X_UniqueID,certChains)
  #Returns true if authorization protocol has been setup

  #Authorize action Y for user X
  authParams = myAS.getInitialAuthParameters(X_UniqueID, Y)
  #Send authParams to Android client app
  #Receive signedPrompt from Android client app
  myAS.verifyAuthSignature(X_UniqueID, signedPrompt)
  #Returns true if authorization is successful
}
```

Listing 1: Server code snippet to use SARA

```
public void AndroidCode()
{
  //Access library class
  myAC = new AttestationController();

  //Setup authorization protocol (only once)
  //Receive regParams from server
  String certChains = myAC.registerBioConfirm(mContext,regParams);
  //Send certChains to server

  //Authorize action
  //Receive authParams from server
  String signedPrompt = myAC.callBioConfirm(mContext,authParams);
  //Send signedPrompt to server
}
```

Listing 2: Android code snippet to use SARA

that has to be sent to the server. The developer has to have the app wait for a confirmation message from the server to indicate the action has been authorized successfully.

5.2 Android Library Implementation

We implemented the client part of our protocol in an open-source Android library that developers can easily plug in into their Android projects. We implement the protocol using APIs that are offered by modern Android versions (Android 9.0+).

Keypair Generation: To create the keypairs in accordance with **Step ❶** of our protocol, we use `KeyGenParameterSpec.Builder` with the purpose argument set as `Sign+Verify`. We refer to the keypair to be used with BiometricPrompt API as the biometric keypair and the keypair to be used with Android Protected Confirmation as the confirmation keypair. We use the methods of `KeyGenParameterSpec.Builder` to set the properties for our keypairs. The properties we use for both the keypairs are listed in Table 2.

Attestation challenges are received from the server, which ensures that the keypair was created in response to the server's request. Furthermore, the biometric keypair has the validity duration for authentication set to −1 to ensure it can only be used with BiometricPrompt API. If someone registers their

Table 2: Key properties for the Biometric Keypair and the Confirmation Keypair

Keypair Property	Biometric KP	Confirmation KP
AttestationChallenge	From Server	From Server
Digests	SHA256	SHA256
SignaturePaddings	RSA-PSS	RSA-PSS
UserAuthenticationRequired	✓	✓
InvalidatedByBiometricEnrollment	✓	✗
UserConfirmationRequired	✗	✓
UserAuthenticationValidityDuration	-1s	120s
StrongBoxBacked	✓	✓

biometrics after a key has already been created for biometric use, the existing key gets invalidated. We set the confirmation keypair to require user confirmation to ensure it is only usable with Android Protected Confirmation. Once the properties are set, we use `KeyPairGenerator` with `RSA` to build the keypair.

Certificate Chain Extraction: Step ❷ of our protocol requires that the certificate chains of our keypairs be sent to the server. To extract these certificate chains, we use `KeyStore` object's method `getCertificateChain`. This returns a chain of X.509 certificates. We then encode the certificate chain into a single string that can be sent to the server.

Biometric Prompt Display: For **Step ❸** of our protocol, we first display a biometric prompt. We ensure that biometrics have to be used for this prompt and the user cannot revert to using device credentials such as a pin or password to authenticate to the prompt. We use a `Signature` object with `SHA256withRSA\PSS`. We then call the `Prompt.authenticate` method to display the prompt and pass our `Signature` object as an argument. This allows us to sign using that object later.

Biometric Signing: For the next part of **Step ❸** of our protocol, we sign the biometric prompt using our biometric keypair. We override the `onAuthenticationSucceeded` method. The method has a result as an argument which contains our `Signature` object. We use the `update` method to add our prompt and nonce to the `Signature` object, and then we use the `sign` method to complete the signing.

Confirmation Prompt Display: Continuing with **Step ❸** of our protocol, we display a confirmation prompt to the user. Within the overridden `onAuthenticationSucceeded` method, we use the result of our signing, concatenated with the prompt and nonce, as `extraData` in `ConfirmationPrompt.Builder`. We then use the method `presentPrompt` to display the prompt.

Confirmation Signing: Lastly, for **Step ❸** of our protocol, we sign using the confirmation keypair. We override the `onConfirmed` method for the `confirmationCallback`. This method has as an argument `dataThatWasConfirmed`. The argument `dataThatWasConfirmed` contains both the prompt as well as the `extraData`. We use a `Signature` object with

SHA256withRSA\PSS. We use the `update` method to add `dataThatWasConfirmed` to the `Signature` object, and then we use the `sign` method to complete the signing and send the result to the server.

5.3 Server Module Implementation

We implement our server module using Python 3. The server module can be imported and used within any existing Python server. The implementation is based on the Android guidelines [16] for the remote server.

Attestation Challenge Creation: For **Step ❶** of our protocol, our server module creates an attestation challenge for each keypair. These attestation challenges are then sent to the client Android app to be used for keypair creation.

Certificate Chain Verification: We begin **Step ❷** of our protocol by verifying the certificate chains received from the client Android app. First, we verify that the root certificates of the certificate chains are one of the Google certificates [16]. If the root certificates are not one of the Google certificates, then the keypairs are not stored within the TEE and the verification fails. Second, we verify that none of the certificates in the chains are part of Google's certificate revocation list [2] as they pose a security threat; hence, the verification fails. Finally, we verify the certificate chains themselves. We verify that the root certificate is self-signed, and we verify the following certificates using the preceding certificate in the chain. We use the `OpenSSL` module to verify the certificate chains.

Attestation Data Verification: Continuing with **Step ❷** of our protocol, our server module checks the attestation data embedded within the certificate chain. We implement our own ASN.1 parser to extract the attestation data from the certificate chain. We then verify that the attestation data contains the expected properties of the keypairs. The server module also checks the attestation data to verify the state of the device by checking if the `VerifiedBootState` value is set to Verified.

Prompt Preparation: For **Step ❸** of our protocol, our server module creates the prompt that is shown to the user. Furthermore, a nonce is also created using the `Random` module to be sent alongside the prompt to the client Android app. The timeout is started at this point by storing the current time. The duration of this timeout has to allow ample time for user interaction as well as client-side processing. Furthermore, the duration of the timeout must not exceed the duration of the time gap between the legitimate user access and the adversary access, as discussed in the threat model. Since screen timeout duration reflects the time a user is inactive before the device sleeps and locks, we use Android's screen timeout durations as a reference point. The median option for screen timeout in the latest Android versions is 2 minutes. Therefore, we set the duration of this timeout to be 2 minutes.

Signature Verification: For **Step ❹** of our protocol, our server module verifies the signatures received from the client Android app. We first check the time to see if we have received the signatures within the timeout limit. The server module
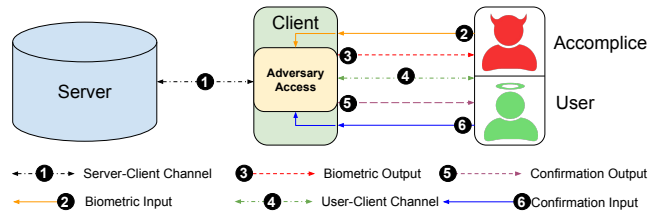

Figure 4: ProVerif Model with the channels that are used by the processes

receives `dataThatWasConfirmed` and the confirmation signature. First, we verify that `dataThatWasConfirmed` has been signed by the confirmation keypair, using the corresponding certificate chain and signature. Then, we extract the `extraData` from `dataThatWasConfirmed` and verify that it includes the prompt, nonce as well as the biometric signature. Finally, we verify that the prompt and nonce have been signed by the biometric keypair, using the corresponding certificate chain and signature. Upon successful verification, our server module guarantees that the user authorized the action.

6 Verification using ProVerif

We evaluated the security aspect of SARA's authorization protocol through an automated cryptographic protocol verifier called ProVerif [27]. In addition, we used ProVerif to verify our hypotheses regarding the weaknesses of the BiometricPrompt API and the Android Protected Confirmation APIs being used in vulnerable protocols. The entire ProVerif code is available in our public repository [21].

ProVerif uses "processes" to represent the different actors in a protocol. For instance, a server and a client will be modeled as two separate processes. These processes communicate with each other using "channels". Channels can be open (accessible by an attacker) or closed. A channel can be two-way or one-way. To verify security properties, ProVerif uses "queries" and "events". Queries are used to assess if an attacker can access certain information. Events are triggered when a process reaches a certain stage. Queries are also used to verify how events are triggered with respect to other events. ProVerif uses the Dolev-Yao model [30] to simulate the abilities of the adversary. That means the attacker can read, intercept, and write its own messages over channels.

6.1 Model Design

Figure 4 illustrates our ProVerif model for remote authorization. Our model uses 4 processes: **Server**, **Client**, **User**, and **Accomplice**. The **Server** and **Client** processes follow our authorization protocol by simulating a remote server and a client Android app. Even though there is no process representing the TEE itself, the **Client** process is designed in a manner where the attacker is only able to access data and channels that are not secured by the TEE. This means that the

Table 3: ProVerif queries and corresponding security goals and implications when using SARA for authorization

#	ProVerif Query	Security Goal	Security Goal Achieved	Security implication when not achieved
1	query f:bitstring; inj-event(serveraccepts(f)) ==> (inj-event(userSeesPrompt(f)))	The legitimate user sees the action the server performs (**SG3**)	Yes	Attacker can authorize an action the legitimate user is unaware of.
2	query f:bitstring; inj-event(serveraccepts(f)) ==> (inj-event(userAuthenticated(f)) && inj-event(userConfirmsPrompt(f)))	The legitimate user physically authorizes the action the server performs (**SG4**)	Yes	Attacker can authorize an action the legitimate user did not approve.
3	query a:biosskey,b:confsskey,e:biospkey,d:confspkey; inj-event(serverconfirms(e,d)) ==> (inj-event(biosigned(a)) && inj-event(confsigned(b)))	Server has a guarantee that the action has been authorized by the legitimate user (**SG5**)	Yes	The server performs an action that was not authorized by the legitimate user.

Client process is separated into a secure world (representing the TEE) and a non-secure world. Therefore, we did not need to create a separate process representing the TEE. Both the **Server** and the **Client** processes are running at all times in the model. The **User** process is modeled to behave like a legitimate user. The **Accomplice** process represents a physical adversary that provides the physical confirmation input in response to all prompts. The **User/Accomplice** interactions are mutually exclusive. We assume that if the user is present, then a physical attacker cannot be present at the same time and vice versa. Hence, our model does not allow the **User** and **Accomplice** processes to run simultaneously.

The **Server-Client** channel represents the network between the client app and the remote server. The **User-Client** channel represents the touch screen. This serves as the primary channel for user interaction with the app. Both these channels are modeled as two-way open channels.

The **Biometric Output** channel represents the prompt that is displayed at the time of biometric authentication. This is not controlled by the TEE and the attacker has complete access to this channel. The **Confirmation Output** channel represents the prompt that is displayed on the secure UI. This is under the control of the TEE. However, since the prompt is provided by the app, it can be manipulated by a root attacker. Hence, these channels are modeled as one-way open channels.

The **Biometric Input** channel represents the biometric scanner. This is a physical input directly to the TEE and even a root attacker cannot provide this input. The **Accomplice** process does have access to this channel but cannot provide the correct input. The **Confirmation Input** channel represents the hardware confirmation button. This is also a physical input that goes directly to the TEE and hence it is inaccessible to a root attacker. The **Accomplice** process has access to this channel and can provide the correct input to this channel as anyone can press the button. We model these channels as one-way closed channels. Even though the attacker cannot provide inputs on these channels, a root attacker can perceive that inputs have been given. To model this aspect, the **Client** process sends out the received inputs into an open channel, simulating an attacker accessing them.

To model the timeouts, we use ProVerif's "Phases". Phases in ProVerif allow us to separate information. Any data in one phase cannot be used in another phase. This is similar to a timeout in the sense that once the duration of the timeout expires, the corresponding data (i.e., a signature) is no longer valid. In our model, each phase represents the duration of one timeout. We use three phases in our model. In the first phase, the **Accomplice** process is running. In the second phase, the **User** process is running. Lastly, in the third phase, the **Accomplice** process is running again. The **Accomplice** process runs before and after the **User** process to simulate that an attacker can have physical access to the device before and after the actual user makes use of the device.

6.2 Verifying SARA Security

We use the ProVerif model to verify the security properties of our SARA's authorization protocol, as shown in Table 3. The model simulates the protocol setup by creating the keypairs, sending the certificate chains, and performing the attestation. Keys in ProVerif are inaccessible to the attacker and therefore, mimic the behavior of keys inside a TEE (**SG1**). The model assumes that the attestation data retrieved by the server from the certificate chain is acceptable since, otherwise, the remote authorization procedure cannot proceed (**SG2**). We use ProVerif queries to assert the correspondence of certain events to verify that our security goals are achieved.

The event userSeesPrompt is triggered after the confirmation prompt is received by the **User** process. The event userAuthenticated is triggered after valid biometric input is given by the **User** process. The event userConfirmsPrompt is triggered after confirmation input is given by the **User** process. The event serverConfirms is triggered after the **Server** process verifies the signatures. The event serverAccepts is triggered when the **Server** process accepts the authorization for an action.

Query 1: If the event serverAccepts is triggered for a certain authorization request then that can only happen if and only if the event userSeesPrompt was triggered for the same authorization request. This verifies that the legitimate user is aware of the action that is authorized (**SG3**).

Query 2: If the event serverAccepts is triggered for a certain authorization request then that can only happen if and only if the event userAuthenticated, and the event userConfirmsPrompt were triggered for the same authorization request. This verifies that the legitimate user authorized the action using a physical input (**SG4**).

Query 3: If the event `serverConfirms` is triggered for a certain confirmation public key and biometric public key then that can only happen if and only if the event `confsigned` was triggered with the corresponding confirmation private key and the event `biosigned` was triggered with the corresponding biometric private key. This verifies if the server has a guarantee the legitimate user authorized the action (**SG5**).

The result from ProVerif shows that all our queries are true, and no attack is possible.

6.3 Attacks on Incomplete Protocols

To further corroborate our design, we use our ProVerif model to show possible attacks on the vulnerable protocols discussed in Section 3.3

First, we removed TEE-enforced user confirmation from our protocol and used ProVerif (as shown in Appendix, Table 8) to prove that an attack is possible when the *Solitary Biometric* protocol is used. Second, we removed TEE-enforced user confirmation from our protocol and used ProVerif (as shown in Appendix, Table 9) to prove an attack is possible when the *Solitary Confirmation* protocol is used. Next, we disjoined the two signing operations and performed them separately. We used ProVerif (as shown in Appendix, Table 10), to prove an attack is possible when the *Naive Combined* protocol is used. Lastly, we conjoined the two signings to signify a single key being used. We used ProVerif (as shown in Appendix, Table 11), to prove an attack is possible when the *Rectified Combined* protocol is used. In all the four cases, ProVerif was able to generate a counterexample, showing a possible attack that might occur if the tested variation of the protocol was used.

7 Usability and Practicality

In this section, we describe the evaluation of SARA's practicality, gauged by a user study involving Android developers. The goal of the user study was to answer the following research questions:

RQ1: Does using SARA make it easier for developers to use Android's TEE-enforced APIs?

RQ2: How long does it take for a developer to learn how to use SARA?

The results, as described below, show that the use of our library significantly reduced the time and effort consumed for implementing a secure payment application as compared to implementing it with the TEE-enforced APIs directly.

7.1 Participants Recruitment

We conducted a qualitative comparative user study to evaluate the practicality of our library compared to using existing TEE-enforced APIs. For this purpose, we obtained approval from our university's IRB and recruited participants by advertising on internal university/department groups and mailing lists. We designed a questionnaire consisting of 10 Android development related questions (see Appendix, Table 6) to gauge the

expertise level of the participants. We assigned three different expertise levels to the participants based on the number of questions they correctly answered (3-5 as beginner, 6-7 as intermediate, and 8-10 as expert). Prospective participants with a score less than 3 were excluded as we deemed them not to be Android developers). In total, we recruited 14 participants, which included 5 beginners, 4 intermediates, and 5 experts. While having a knowledge requirement for participation affected the number of participants recruited, we believe these conditions were necessary to obtain meaningful conclusions from the user study. Every participant was compensated for the completion of the study.

7.2 Experiment Design

Each participant was given access to a skeleton Android payment application. The participants were asked to integrate Android Protected Confirmation with key attestation and biometric authentication into the skeleton application by performing two tasks i.e., `Task-A:` using the native Android APIs and `Task-B:` using the SARA library. Furthermore, we also asked them to implement related server-side functionality in a sample server-side code provided alongside the application.

We asked each participant to perform both tasks and the time limit to complete each task was set to 105 minutes. Given the complexity of implementing the desired functionality with the native APIs, we divided the implementation of each task into three subtasks to evaluate participants' progress in each task.

In `Subtask-1`, we asked the participants to create a keypair(s) needed to integrate Android Protected Confirmation in the application along with biometric authentication. `Subtask-2` asked the participants to display a confirmation and biometric prompt corresponding to the payment request made by a user. `Subtask-3` required them to verify the keypair(s) on the server side and ensure that the attestation data showed that the keypair was generated by secure hardware and had the requisite properties.

For each subtask, the code sections that needed to be implemented were clearly marked in the skeleton code. In order to ensure that the participants spent ample time attempting the server side subtask (`Subtask-3`), we asked the participants to spend the last 35 minutes of the allotted time for each task on `Subtask-3`, in case they could not complete the first two subtasks. This was necessary to evaluate how developers deal with server side code when implementing such protocols. We allowed the developers to use either Java or Python when attempting `Subtask-3`.

During the experiment, the participants could consult any online resources. Additionally, we provided links to the Android documentation and asked the participants to consult the official Android training articles on key attestation and user confirmation to complete the tasks. To effectively compare the learning times for the two tasks, we also provided the documentation for our library for completing `Task-B`.

Table 4: Summary of the subtask completion results

Completed after 90 minutes	Task-A	Task-B
Subtask-1: Successfully created the keypair(s) with the required properties	0/14	14/14
Subtask-2: Successfully created the confirmation prompt and the biometric prompt	0/14	14/14
Subtask-3: Successfully attested the keypair(s) on the server	0/14	14/14

Table 5: Summary of the survey results of the user study

Evaluation of the Native APIs	
Please rate your experience trying to implement Task-A	
1 or 2 (Very Complicated)	13/14
3	1/14
4 or 5 (Very Straightforward)	0/14
Evaluation of SARA's API	
Please rate your experience trying to implement Task-B	
1 or 2 (Very Complicated)	0/14
3	0/14
4 or 5 (Very Straightforward)	14/14
Evaluation of the Native API Documentation	
Please rate the complexity of the documentation for Task-A	
1 or 2 (Very Complicated)	12/14
3	0/14
4 or 5 (Very Straightforward)	2/14
Evaluation of SARA's API Documentation	
Please rate the complexity of the documentation for Task-B	
1 or 2 (Very Complicated)	2/14
3	0/14
4 or 5 (Very Straightforward)	12/14
Learning Overhead	
In your experience, what was the difference in time taken to learn the usage of both APIs?	
Native APIs took long/very long	14/14
Equal time	0/14
SARA's API took long/very long	0/14
Ease of Implementation	
In your experience, which of the two tasks you just performed was easier to implement?	
Task-A	0/14
Both were equally difficult	0/14
Task-B	14/14
Developer Preference	
If you were to implement this task within your own application, which method would you prefer to use.	
Use SARA's API	14/14
Use the Native APIs	0/14

Since the two tasks required implementing the same functionality, some of the knowledge and understanding about the implementation gained in the first task could decrease the difficulty level of the second task. To remove such bias, we randomly selected half of the participants to perform Task-A first and the other to perform Task-B first.

At the end of the experiment, the participants were asked to complete a survey with questions about their overall experience with the two tasks. This survey collected information about the developer's preferences, ease of usage, learning overhead and their personal feedback concerning the two tasks. The survey also included the System Usability Score [28] questionnaire for both tasks to evaluate SARA and the Native APIs usability.

7.3 Results

Usability (RQ1) We checked the implementation of the participants to verify if they successfully completed a subtask. Upon successful completion of a subtask, we also measured the time it took for them to complete the subtask. Table 4 presents the number of participants that completed the three subtasks within the given time. There is a stark difference between the completion rates for the subtasks with the Native APIs and SARA. All the participants were able to complete all three subtasks using SARA within the allotted time frame. In contrast, within the same time frame, none of the participants were able to complete any subtask using the Native APIs.

We performed a Wilcoxon signed rank test [42] on the subtask completion time with the null hypothesis that the completion time for subtasks using the Native API and SARA are equal. In cases where participants could not finish a subtask, we used the total available time (i.e., 35 minutes) as the user's completion time. We compared the task completion time for the Native API and SARA, and we observed a p-value of $0.0005 < 0.05$. Thus, we reject our null hypothesis and conclude that the task completion time with SARA is lower, demonstrating its practicality.

Participants' responses to the survey questions (summarized in Table 5), show that 93% of the participants had a positive experience with using SARA, while 0% had a positive experience with the Native APIs. All the participants agreed that it was easier to implement the given task using SARA and expressed a preference to use SARA over the Native API in their own applications. Based on the participants' responses (as shown in the Appendix, Table 7), the System Usability Score (SUS) [28] for the Native API was only 11.61 as compared to SARA's score of 95.18. Generally, a system with a SUS score above 68 is considered above average [12]. Therefore, we can conclude that SARA is highly usable.

Learning overhead (RQ2) Participants could access online resources, including Android documentation, while implementing the task with the Native APIs (Task-A). However, in the case of SARA (Task-B), the participants could only use the provided SARA's documentation as a reference. Despite this, all

participants found that the time taken to learn the usage of the Native APIs as compared to SARA was much longer. Furthermore, only 14% of the participants felt that the documentation for SARA was complicated, whereas 86% of the participants felt that the Native API documentation was complex.

8 Performance Evaluation

In this section, we evaluate the performance of the implementation of our authorization protocol within the Android library. The evaluation is conducted on a Pixel 3a with Android 11. We divide the protocol into two parts: the initial registration phase and the per-authorization phase.

The initial registration phase consists of keypair generation and certificate chain extraction. The overhead of these processes (averaged over 100 trials) is 12.29 seconds with a standard deviation of 3.82 seconds. The vast majority of this time is spent on the creation of the keypairs themselves as we store the keypairs within the StrongBox [14]. Specifically, more than 99% of this time was spent calling just one method: `KeyPair-Generator.generateKeyPair`. This method takes only 1.05 seconds per keypair generation if StrongBox is not used. We speculate that the StrongBox internally requires a significant amount of processing time to create the keypairs and that, in addition, there could be a communication delay between the main chip and the StrongBox chip. The usage of StrongBox to store the keys is, however, unavoidable as Android Protected Confirmation requires using a keypair stored within the StrongBox. We note that this is a one-time process that takes place in the background when the authorization protocol is enabled. Therefore, this overhead will not have a major effect on the user's experience of an app using SARA.

The per-authorization phase consists of the creation and display of the prompts, user's biometric and confirmation input, and the two signings. The user input times vary between users; therefore, we focus on the prompts and the signing for measuring time overhead. The overhead of these processes (averaged over 10 trials) is 1.76 seconds with a standard deviation of 30 milliseconds. This time is negligible compared to the time needed for a user to interact with the prompts.

9 Limitations and Discussion

This paper focuses on implementing root-resilient end-to-end remote authorization in Android devices using TEE-enforced APIs. However, other popular mobile ecosystems like Apple have their own versions of these APIs. Contrasting the usage of these APIs across platforms, and developing a solution that is compatible with non-Android devices is the principal focus of our future work.

Our solution relies on presenting two prompts for the user to confirm in order to authorize an action. However, this solution only works if the user can be trusted to correctly identify that both prompts are equivalent. If the user only verifies the biometric prompt and does not verify the confirmation prompt

before providing the confirmation input, then a root attacker can take advantage of this oversight to authorize an arbitrary action. We plan to conduct a user study to determine the effect of this human-induced vulnerability.

SARA includes a server module to allow developers to implement the protocol easily. However, this module is currently only for Python. We plan to create server modules for other popular languages such as Java.

The new authorization protocol gives us the opportunity to use regular mobile devices as a form of biometric 2-Factor Authentication. We plan to compare our protocol to other various modes of 2-Factor Authentication to gauge the efficacy of their security as well as their ease of use.

10 Related Work

The issue of remote authorization has been tackled by both industry and academia. However, these works usually use the TEE to secure only the individual channels that are required for authorization (Figure 1).

TEE-enforced User Interactions. Several existing academic works aim to secure the user interactions by utilizing a TEE. TrustUI [34] performs color and keyboard randomization by leveraging TrustZone to provide a trusted UI on an untrusted device. DroidVault [35] ensures secure input and display by providing APIs for keyboard and display that utilize ARM TrustZone. Light-SPD [43] emulates an SPD (Secure Personal Device) by utilizing TrustZone on boards with ARM. TrustOTP [40] provides users with an OTP (One-Time Password) secured by TrustZone using GUI that is also secured by TrustZone. Using TrustZone ensures that the user is provided with a secure OTP even on a malicious OS. TrustPAY [45] utilizes TrustZone to provide the user with a secure GUI, keyboard, and mouse for making payments on an untrusted OS. VeriUI [36] provides users with secure login interfaces by generating them using a WebKit engine from within the TrustZone. Akowuah et al. [24] present an SQLite Database secured by TrustZone. Their framework enables users to securely edit an SQLite Database on an untrusted device by leveraging TEE for data encryption and showing keyboard. TruWalletM [29] provides a password manager backed by TrustZone. TruWalletM also provides a User Interface secured by the TEE. VButton [33] leverages TrustZone to provide the user with secure UI and secure input, and it allows the server to attest to the authenticity of the user action. BitE (Bump in the Ether) [37] proxies user input through a mobile device using encrypted tunnels. Oprea et al. [38] proposed using PDAs as a form of secure input to untrusted terminals.

Samsung Knox [11] provides hardware-backed storage for data by leveraging TrustZone on Android devices. It also handles Biometric authentication, which can be used to enforce 2-Factor Authentication. However, Samsung Knox is only available for certain Samsung devices [7].

TEE-enforced Server Communication DroidVault [35] provides a mechanism for storing and processing sensitive data from a remote server on an untrusted Android device. DroidVault leverages TrustZone to prevent end-user from directly accessing sensitive data. TrustPAY's [45] framework allows users to make privacy-preserving transactions with a remote service. TruzCall [23] utilizes TEE to provide secure VoIP on a compromised device. TruWalletM [29] splits the SSL channel with the server into a secure and a non-secure channel. The secure channel is backed by TEE and used for communicating only sensitive information like authentication tokens and passwords.

11   Conclusions

This work presents the first practical implementation of TEE-enforced root-resilient remote authorization in Android devices. We extensively evaluated our work with both a user study and a formal cryptographic model. Our results showed that not only is SARA more secure as compared to the existing native API, but it is also easier to use. Hence, we believe that SARA should be used for all forms of authorization in Android devices and hope that Android will integrate the SARA's protocol as part of its official API.

Acknowledgments

References

[1] YubiKey Bio. https://www.yubico.com/products/yubikey-bio-series/, 2020.

[2] Android Certificate Revocation List. https://android.googleapis.com/attestation/status, 2021.

[3] Android Compatibility Definition Document. https://source.android.com/compatibility/cdd, 2021.

[4] Android Protected Confirmation. https://developer.android.com/training/articles/security-android-protected-confirmation, 2021.

[5] Android Security Bulletin-February 2021. https://source.android.com/security/bulletin/2021-02-01, 2021.

[6] Biometric Prompt. https://developer.android.com/reference/android/hardware/biometrics/BiometricPrompt, 2021.

[7] Devices Secured by Knox. https://www.samsungknox.com/en/knox-platform/supported-devices, 2021.

[8] Hardware-backed Keystore. https://source.android.com/security/keystore, 2021.

[9] Insulin Pumps using Android Protected Confirmation. https://android-developers.googleblog.com/2018/10/android-protected-confirmation.html, 2021.

[10] Jadx. https://github.com/skylot/jadx, 2021.

[11] Knox Platform for Enterprise White Paper. https://images.samsung.com/is/content/samsung/p5/ch/business/enterprise-edition/Knox_Platform_for_Enterprise_Whitepaper_2019.pdf, 2021.

[12] Measuring SUS Score. https://measuringu.com/sus/, 2021.

[13] Strong Customer Authentication EU. https://www.eba.europa.eu/regulation-and-policy/payment-services-and-electronic-money/regulatory-technical-standards-on-strong-customer-authentication-and-secure-communication-under-psd2, 2021.

[14] StrongBox. https://developer.android.com/training/articles/keystore#HardwareSecurityModule, 2021.

[15] TrustZone. https://developer.arm.com/ip-products/security-ip/trustzone, 2021.

[16] Verifying Hardware-backed Keypairs with Key Attestation. https://developer.android.com/training/articles/security-key-attestation, 2021.

[17] Version Stats. https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide, 2021.

[18] Yubikey. https://www.yubico.com/, 2021.

[19] Apktool: A tool for Reverse Engineering Android Apk Files. https://ibotpeaches.github.io/Apktool/, 2022.

[20] CVE Mitre. https://cve.mitre.org/, 2022.

[21] SARA Public Repository. https://github.com/purseclab/SARA-Secure-Android-Remote-Authorization, 2022.

[22] Smali. https://github.com/JesusFreke/smali/wiki, 2022.

[23] Amit Ahlawat and Wenliang Du. TruzCall: Secure VoIP Calling on Android using ARM TrustZone. In *Proceedings of the International Conference on Mobile And Secure Services (MobiSecServ)*, 2020.

[24] Francis Akowuah and Amit Ahlawat. Protecting Sensitive Data in Android Sqlite Databases using TrustZone. In *Proceedings of the International Conference on Security & Management (SAM)*, 2018.

[25] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2016.

[26] Antonio Bianchi, Yanick Fratantonio, Aravind Machiry, Christopher Kruegel, Giovanni Vigna, Simon Pak Ho Chung, and Wenke Lee. Broken Fingers: On the Usage of the Fingerprint API in Android. In *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2018.

[27] Bruno Blanchet. Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif. In *Foundations of Security Analysis and Design VII (FOSAD)*. 2013.

[28] John Brooke et al. SUS-A quick and dirty usability scale. *Usability Evaluation in Industry*, 1995.

[29] Sven Bugiel, Alexandra Dmitrienko, Kari Kostiainen, Ahmad-Reza Sadeghi, and Marcel Winandy. TruWalletM: Secure Web Authentication on Mobile Platforms. In *Proceedings of the Conference on Trusted Systems (INTRUST)*, 2010.

[30] Danny Dolev and Andrew Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 1983.

[31] Radhesh Krishnan Konoth, Björn Fischer, Wan Fokkink, Elias Athanasopoulos, Kaveh Razavi, and Herbert Bos. SecurePay: Strengthening Two-Factor Authentication for Arbitrary Transactions. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020.

[32] Juan Lang, Alexei Czeskis, Dirk Balfanz, Marius Schilder, and Sampath Srinivas. Security keys: Practical Cryptographic Second Factors for the Modern Web. In *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, 2016.

[33] Wenhao Li, Shiyu Luo, Zhichuang Sun, Yubin Xia, Long Lu, Haibo Chen, Binyu Zang, and Haibing Guan. Vbutton: Practical Attestation of User-driven Operations in Mobile Apps. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2018.

[34] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. Building Trusted Path on Untrusted Device Drivers for Mobile Devices. In *Proceedings of the Asia-Pacific Workshop on Systems (APSys)*, 2014.

[35] Xiaolei Li, Hong Hu, Guangdong Bai, Yaoqi Jia, Zhenkai Liang, and Prateek Saxena. DroidVault: A Trusted Data Vault for Android devices. In *Proceedings of the International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2014.

[36] Dongtao Liu and Landon P Cox. Veriui: Attested Login for Mobile Devices. In *Proceedings of the Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2014.

[37] Jonathan M McCune, Adrian Perrig, and Michael K Reiter. Bump in the Ether: A Framework for Securing Sensitive User Input. In *Proceedings of the USENIX Annual Technical Conference (Usenix ATC)*, 2006.

[38] Alina Oprea, Dirk Balfanz, Glenn Durfee, and Diana K Smetters. Securing a Remote Terminal Application with a Mobile Trusted Device. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2004.

[39] Bryan Parno. Bootstrapping Trust in a "Trusted" Platform. In *Proceedings of the Annual Summit on Hot Topics in Security (HotSec)*, 2008.

[40] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. TrustOTP: Transforming Smartphones into Secure One-time Password Tokens. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[41] Xianbo Wang, Yikang Chen, Ronghai Yang, Shangcheng Shi, and Wing Cheong Lau. Fingerprint-jacking: Practical fingerprint Authorization Hijacking in Android apps. *Blackhat Europe*, 2020.

[42] Robert F Woolson. Wilcoxon signed-rank test. *Wiley Encyclopedia of Clinical Trials*, 2007.

[43] Sileshi Demesie Yalew, Gerald Q Maguire, and Miguel Correia. Light-SPD: A Platform to Prototype Secure Mobile Applications. In *Proceedings of the ACM Workshop on Privacy-Aware Mobile Computing (PAMCO)*, 2016.

[44] Kailiang Ying, Amit Ahlawat, Bilal Alsharifi, Yuexin Jiang, Priyank Thavai, and Wenliang Du. Truz-droid: Integrating TrustZone with Mobile Operating System. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2018.

[45] Xianyi Zheng, Lulu Yang, Jiangang Ma, Gang Shi, and Dan Meng. TrustPAY: Trusted Mobile Payment on Security Enhanced ARM TrustZone Platforms. In *Proceedings of the IEEE Symposium on Computers and Communication (ISCC)*, 2016.

Appendix

Table 6 shows the pre-screening questionnaire that the prospective participants had to fill out to sign up for the study. Table 7 shows the participant responses to the SUS questions in the survey regarding both APIs. Table 8 shows the ProVerif queries for the model which simulates the *Solitary Biometric* protocol. Table 9, Table 10, and Table 11 show, respectively, the ProVerif queries for the models simulating the *Solitary Confirmation* protocol, the *Naive Combined* protocol, and the *Rectified Combined* protocol.

Table 6: Pre-screening questionnaire for the user study

What of the following methods are not used with the AsyncTask class in Android?			
OnPreExecution()	OnPostExecution()	DoInForeground()	OnProgressUpdate()
Which of the following methods are used to get a response from an activity in Android?			
startActivityToResult()	startActivityForResult()	Bundle()	None of the above
What is the difference between services and thread in Android?			
Services performs functionalities in the background. By default services run on main thread only	Thread and services have the same functionalities	Thread works on services	None of the above
Which of the following methods can be used as broadcast receivers in Android?			
sendIntent()	onRecieve()	implicitBroadcast()	sendBroadcast(), sendOrderBroadcast(), and sendStickyBroadcast()
What is a fragment in android?			
JSON	Piece of Activity	Layout	None of the above
Which component is not activated by an Intent?			
Activity	Services	contentProvider	broadcastReceiver
Which of the following is the first callback method that is invoked by the system during an activity life-cycle?			
onClick() method	onCreate() method	onStart() method	onRestart() method
What is the use of content provider in Android?			
For storing the data in the database	For sharing the data between applications	For sending the data from an application to another application	None of the above
Which of the following android component displays the part of an activity on screen?			
View	Manifest	Intent	Fragment
The layout or design of an Android application is saved in a:			
.text file	.xml file	.dex file	.java file

Table 7: Summary of participant answers to the SUS questions for both APIs

SUS Questions	Native API					SARA API				
1: Strongly Disagree, 5: Strongly Agree	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**
Assuming you wanted to integrate these security features into your own app, would you like to use this API frequently	9	4	1	0	0	0	0	0	2	12
I found this API unnecessarily complex.	1	0	0	6	7	13	1	0	0	0
I thought this API was easy to use.	12	2	0	0	0	0	0	0	1	13
I think that I would need the support of a technical person to be able to use this API.	1	0	0	2	9	11	2	1	0	0
I found that various functions in this API were well integrated and were easy to use together	9	3	0	2	0	0	0	0	0	14
I thought there was too much inconsistency in this API's functions which made it harder to use them together.	0	1	3	3	7	12	2	0	0	0
I would imagine that most people would learn to use this API very quickly.	11	2	1	0	0	0	0	0	1	13
I found this API very cumbersome to use.	0	0	1	3	10	11	2	0	0	1
I felt very confident using this API.	11	2	0	0	1	0	0	2	1	11
I needed to learn a lot of things before I could get going with this API.	0	0	0	1	13	8	5	1	0	0

Table 8: ProVerif queries and corresponding security goals and implications for the Solitary Biometric Protocol

#	ProVerif Query	Security Goal	Security Goal Achieved	Security implication when not achieved
1	query f:bitstring; inj-event(serveraccepts(f)) ==>(inj-event(userSeesPrompt(f))).	The legitimate user sees the action the server performs (**SG3**)	No	Attacker can authorize an action the legitimate user is unaware of.
2	query f:bitstring; inj-event(serveraccepts(f)) ==>(inj-event(userAuthenticated(f))).	The legitimate user physically authorizes the action the server performs (**SG4**)	No	Attacker can authorize an action the legitimate user did not approve.
3	query a:biosskey,b:confsskey,e:biospkey,d:confspkey; inj-event(serverconfirms(e,d)) ==>(inj-event(biosigned(a)) && inj-event(confsigned(b))).	Server has a guarantee that the action has been authorized by the legitimate user (**SG5**)	No	The server performs an action that was not authorized by the legitimate user.

Table 9: ProVerif queries and corresponding security goals and implications for the Solitary Confirmation Protocol

#	ProVerif Query	Security Goal	Security Goal Achieved	Security implication when not achieved
1	query f:bitstring; inj-event(serveraccepts(f)) ==>(inj-event(userSeesPrompt(f))).	The legitimate user sees the action the server performs (**SG3**)	No	Attacker can authorize an action the legitimate user is unaware of.
2	query f:bitstring; inj-event(serveraccepts(f)) ==>(inj-event(userConfirmsPrompt(f))).	The legitimate user physically authorizes the action the server performs (**SG4**)	No	Attacker can authorize an action the legitimate user did not approve.
3	query a:biosskey,b:confsskey,e:biospkey,d:confspkey; inj-event(serverconfirms(e,d)) ==>(inj-event(biosigned(a)) && inj-event(confsigned(b))).	Server has a guarantee that the action has been authorized by the legitimate user (**SG5**)	No	The server performs an action that was not authorized by the legitimate user.

Table 10: ProVerif queries and corresponding security goals and implications for the Naive Combined Protocol

#	ProVerif Query	Security Goal	Security Goal Achieved	Security implication when not achieved
1	query f:bitstring; inj-event(serveraccepts(f)) ==>(inj-event(userSeesPrompt(f))).	The legitimate user sees the action the server performs (**SG3**)	No	Attacker can authorize an action the legitimate user is unaware of.
2	query f:bitstring; inj-event(serveraccepts(f)) ==>(inj-event(userAuthenticated(f)) && inj-event(userConfirmsPrompt(f))).	The legitimate user physically authorizes the action the server performs (**SG4**)	No	Attacker can authorize an action the legitimate user did not approve.
3	query a:biosskey,b:confsskey,e:biospkey,d:confspkey; inj-event(serverconfirms(e,d)) ==>(inj-event(biosigned(a)) && inj-event(confsigned(b))).	Server has a guarantee that the action has been authorized by the legitimate user (**SG5**)	No	The server performs an action that was not authorized by the legitimate user.

Table 11: ProVerif queries and corresponding security goals and implications for the Rectified Combined Protocol

#	ProVerif Query	Security Goal	Security Goal Achieved	Security implication when not achieved
1	query f:bitstring; inj-event(serveraccepts(f)) ==>(inj-event(userSeesPrompt(f))).	The legitimate user sees the action the server performs (**SG3**)	No	Attacker can authorize an action the legitimate user is unaware of.
2	query f:bitstring; inj-event(serveraccepts(f)) ==>(inj-event(userAuthenticated(f)) && inj-event(userConfirmsPrompt(f))).	The legitimate user physically authorizes the action the server performs (**SG4**)	No	Attacker can authorize an action the legitimate user did not approve.
3	query a:bioconfsskey,b:bioconfspkey; inj-event(serverconfirms(b)) ==>(inj-event(bioconfsigned(a))).	Server has a guarantee that the action has been authorized by the legitimate user (**SG5**)	No	The server performs an action that was not authorized by the legitimate user.