Leveraging remote attestation APIs for secure image sharing in messaging apps (extended version)

Joel Samper, Bernardo Ferreira
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
{jsamper,blferreira}@ciencias.ulisboa.pt

***Abstract*—Sensitive pictures such as passport photos and nudes are commonly shared through mobile chat applications. One popular strategy for the privacy protection of this material is to use ephemeral messaging features, such as the view once snaps in Snapchat. However, design limitations and implementation bugs in messaging apps may allow attackers to bypass the restrictions imposed by those features on the received material. One way by which attackers may accomplish so is by tampering with the software stack on their own devices. In this paper, we propose and test a protection strategy based on a multiplatform system that encrypts and decrypts sensitive pictures on top of messaging apps and performs remote attestation with available app integrity APIs to safeguard its security. Our analysis and experiments show that, compared to previous proposals for image encryption in a middleware, remote attestation offers increased security, adds privacy benefits, simplifies integration, and improves usability by not requiring users to exchange key material a priori. In our experiments, it incurs an added average latency of 3.8 and 4.5 seconds when sending and receiving private pictures, respectively.**

I. Introduction

In line with the trend of increased mobile phone usage and dependency [1]–[3], apps with chat features are pervasive, ranging from those specially purposed for messaging, to dating platforms and social networks. Most of these apps allow the exchange of pictures among users in private chats. While some media exchanges may not hold particularly sensitive information (such as sharing a picture of a beautiful sunset with a friend), others do. Such is the case, for example, of a passport photo that one has to share with the host of a touristic apartment to comply with vacation rental regulations, or self-generated nudes exchanged with strangers when flirting over a dating app or a social network. Relatedly, Grindr reported that 1 billion albums were shared in 2023 [4].

View once media (i.e., images and videos set to self-erase on the recipient once viewed) may end up giving a sense of heightened control, as the material is short-lived. Users may then feel confident that it is a one-off interaction that will remain private within the concerned parties. However, when privacy expectations are unmet this poses a problem, especially when the material is of sensitive nature. In this paper, we address the threat that a malicious receiver disrespects the sender's privacy by hacking into their own device to break the security of ephemeral messaging features. In the case of sexting, attackers may be collecting nudes for personal use, selling child pornography on the dark web, or doing cyberbullying or sextortion to senders [5], [6].

To showcase the feasibility of such a cyber attack, we first ran a little lab experiment with an Android phone rooted with Magisk [7], on which we installed Snapchat. We chose Snapchat because it is the most used app in sexting, set aside SMS [8], [9]. We then activated LSPosed for Zygisk [10] (which is an Android Runtime hooking framework), and integrated the SnapEnhance app module [11]. Snapchat prevents saving snaps with timer. It does not prevent taking screenshots, but at least the sender is notified so that appropriate action can be taken. However, with our lab setup, we were able to save snaps-with-timer. In our attack, the sender does not notice any difference in the UI nor receives any notification.

To address vulnerabilities in ephemeral messaging implementations, academic proposals (a) expect end users to verify keys manually, (b) require to deploy a new trusted third party, (c) rely solely on the secrecy of a key for their security, and/or (d) lack support for advanced privacy policies. To overcome the drawbacks of existing methods, we propose and test a middleware that encrypts and decrypts pictures on top of messaging apps, so that they never have access to image contents. Yet, the key idea of our approach is that images can be decrypted only if recipient endpoints attest the integrity of their middleware (including the operating system and the hardware they run on). This makes attacks like the SnapEnhance much more difficult. We achieve so by leveraging existing app integrity APIs, namely Play Integrity [12] in Android and DeviceCheck [13] in iOS (we provide a technical overview of these in §II). In the industry, such APIs have traditionally driven more focus in game cheat prevention and access control to premium features, but our work explores its usage for privacy-related issues.

As with previous proposals, ours requires updating the MOS (Mobile Operating System) to accommodate the middleware. We additionally require deploying a new trusted server, although this does not represent a new trusted third party since it would be provided by the MOS platform itself. In return, the main benefits of our approach are: (1) we do not introduce new trusted third parties, (2) user interactions can stay anonymous, (3) it requires minimal messaging app code adaptations (less than 10 lines of code) and no messaging server changes, (4) it preserves user-to-middleware privacy without having to regenerate keys, (5) it supports advanced privacy policies such as *recipient's subscriber line country verification*, and (6) it allows for extended security protections such as detection and contention of massive decryption requests. These benefits are provided without sacrificing user privacy.

One could argue that it would suffice that each messaging platform attests the integrity of its own app. While we consider so as a technically valid strategy, prior research has found that, in practice, app platforms do not use app integrity APIs, or do so incorrectly. Ibrahim et al. [14] studied 163,773 Android apps in 2021. They found that only 21 used SafetyNet (the predecessor of Play Integrity) but none of them implemented this API correctly. Beijnum [15] surveyed 1,836 popular apps in 2023, none of which were found to use Android's app integrity APIs correctly, and only 1.1% were found to use iOS' DeviceCheck (either correctly or incorrectly). All in all, performing attestation on a middleware instead of on the messaging app has the benefits of: concentrating implementation efforts on a single point, reducing the trusted computing base, and providing a unified user experience across messaging apps.

There are several potential motives for MOS companies to adopt our architecture. Firstly, increasing awareness about online privacy and safety puts social pressure to invest in new technologies. Secondly, companies may strategically decide to play a leading role in privacy-preserving technologies, for product differentiation. And thirdly, legislators may establish that the support with our architecture is mandatory for MOS platforms. Also, since our approach is permissive (i.e., it does not interfere with the will of those who do want to engage in private picture exchanges), and since it allows for anonymous interactions, this facilitates popular adoption.

In our threat model, the MOS and the messaging app are trusted, and the recipient is untrusted. Recipient users may have advanced computer science skills and may use them to try to circumvent ephemeral messaging features by hacking on the own device or the own network environment. Messaging apps may have design or implementation flaws, yet the focus of our work is on not depending on the security or correctness of the numerous messaging apps in use.

II. Background Information

Both Apple iOS and Google Android provide app integrity APIs to verify, from the server side of the app platform, that a remote mobile device runs unmodified app code. To achieve so, app integrity APIs leverage hardware-backed remote attestation[1] [13], [16], combined with Secure Boot [17]–[19] and a chain of trust mechanism. In Android, the API is called Play Integrity [12] (formerly, SafetyNet [20]), while in iOS it is called DeviceCheck (although the local service is called App Attest) [13]. We provide here a technical overview of Play Integrity and DeviceCheck.

A. Google Android's Play Integrity

Google Android provides two alternatives, namely Classic and Standard. The two variants provide security metrics about

[1]In Android, available depending on the phone model.

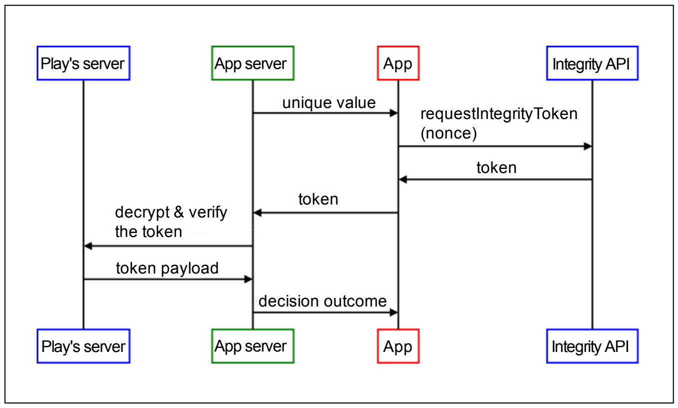


Fig. 1. Google Android
Classic Play Integrity sequence diagram
Modified from [21], Apache 2.0 license

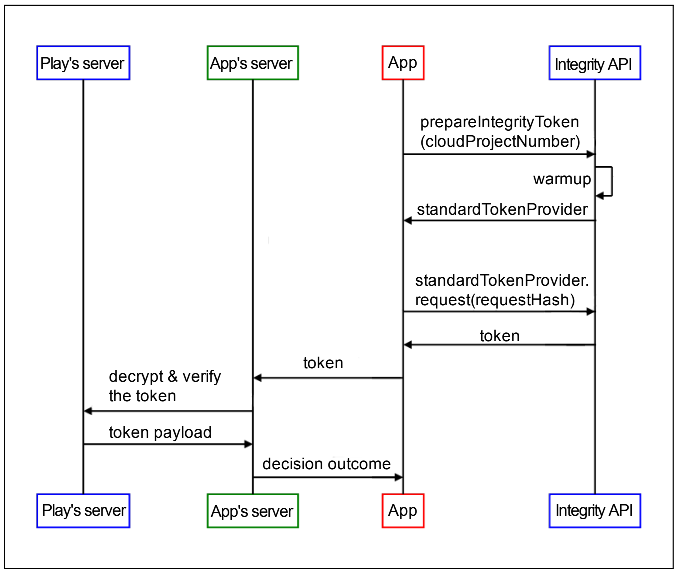


Fig. 2. Google Android
Standard Play Integrity sequence diagram
Modified from [22], Apache 2.0 license

the number of integrity tokens requested by our app on the device over the past hour, which corresponds to the recentDeviceActivity field of the token [23].

In the Classic API (Figure 1), there is no preparatory phase. When an app needs to provide attestation, it gets a nonce from the app server, it submits an attestation request to Google servers, and provides the obtained attestation token to the app server. The app server can either offload the validation to Google servers, or perform it locally. There is a limit of five integrity tokens per app instance per minute, and a maximum of 10,000 requests per app platform per day.

In the Standard API (Figure 2), the process is divided in two mandatory phases, namely *warmup* and *attestation*. The warmup phase is a preparatory step in which the API triggers a communication to the Google servers if needed, and caches

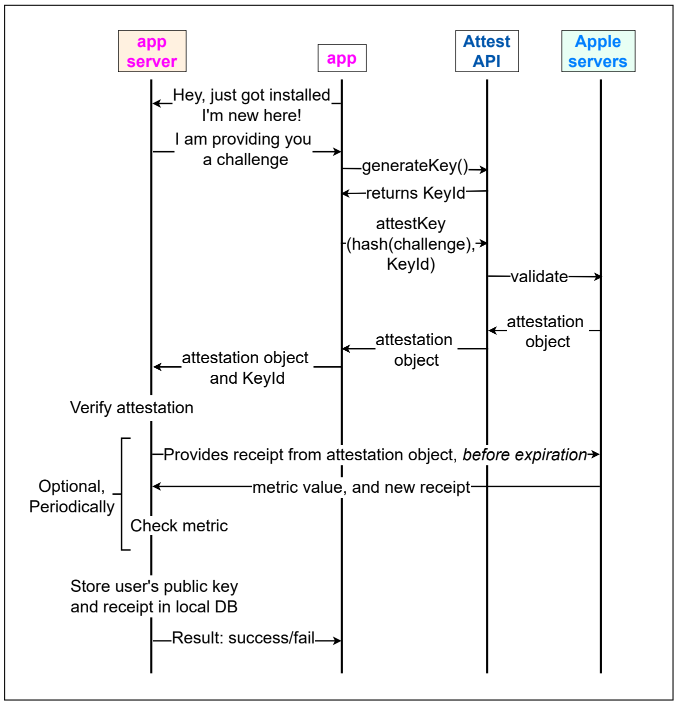

app server
app
Attest API
Apple servers
Hey, just got installed I'm new here!
I am providing you a challenge
generateKey()
returns KeyId
attestKey (hash(challenge), KeyId)
validate
attestation object
attestation object
attestation object and KeyId
Verify attestation
Provides receipt from attestation object, before expiration
Optional, Periodically
metric value, and new receipt
Check metric
Store user's public key and receipt in local DB
Result: success/fail

Fig. 3. Apple iOS
DeviceCheck sequence diagram
Attestation enrollment phase

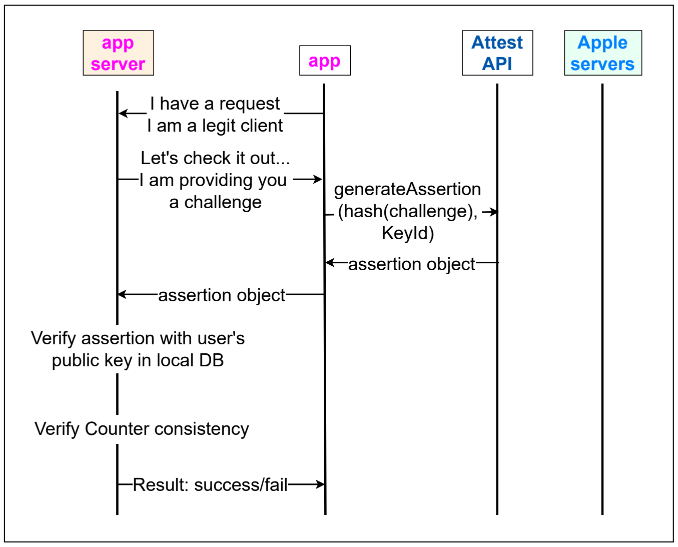

app server
app
Attest API
Apple servers
I have a request I am a legit client
Let's check it out... I am providing you a challenge
generateAssertion (hash(challenge), KeyId)
assertion object
assertion object
Verify assertion with user's public key in local DB
Verify Counter consistency
Result: success/fail

Fig. 4. Apple iOS
DeviceCheck sequence diagram
Assertion operation

attestation data. This can be done, for example, every time the app is launched. In the attestation phase, the app gets a nonce from the app server, then generates the attestation token locally using the cached data, and provides it to the app server. In this case, the app server needs to contact Google servers to verify the token. There is a maximum of five warmups per app instance per minute, and a maximum of 10,000 requests per app platform per day (where 'requests' include both warmups and verifications of attestation tokens).

B. Apple iOS' DeviceCheck

In Apple iOS, the DeviceCheck API works differently. The user first needs to generate a key-pair locally. Note that keys generated for development will not work in production, and vice versa.

Next, the user needs to enroll, which in Apple's terminology is called *attestation* (Figure 3). This is typically done only once, after a fresh install and every time the app is reinstalled or restored from a backup. This enrollment phase produces a public key associated with the device, which needs to be validated and stored by the app server. On the client side, connection to Apple servers is required. On the server side, attestation validity is verified against Apple's root certificates. Occasional connections to Apple servers are required only if the app platform wants to use the security metric about the total number of generated keys over the past 30 days on the device. Attestation tokens have an embedded receipt token that the app server can send to Apple servers to get a new receipt containing the updated metric, which is found in the Risk Metric (field 17) of the receipt response. Note that whenever a receipt's Expiration Time (field 21) is reached before the app server checks it, then the ability to check receipts for that device any further might be lost. We infer that this is to prevent attackers from using old receipts leaked from exfiltrated data. Also, the app server cannot check the metric before the receipt's Not Before time (field 19). We infer that this is to prevent cooperation among app platforms to match and identify users. This metric is expected to stay within relatively low values, to account for all apps re-installs, device restores, and factory resets.

III. Prior Work

To address vulnerabilities in ephemeral messaging implementations, a possible strategy is to deploy a middleware that encrypts, decrypts and displays images on top of messaging apps, and then somehow authenticates or distributes keys among end users [24]–[27]. One way to do so is to deploy a new trusted third party that generates and distributes those keys. However, this recursively brings security and privacy concerns related to that new entity into play. Alternatively, if users are supposed to manually validate keys out of band, this is inconvenient, and they may end up not doing that verification, thereby undermining the security of the system. In the case of sexting within dating apps, this also defeats the purpose since they are mainly used to connect with strangers.

As an example of the above, JPEG encryption algorithms [25]–[27] produce a JPEG-compliant ciphertext from a JPEG file, and are typically robust to common image transformations (e.g. compression) on the ciphertext. Although prior research has focused more on the use case of semi-honest messaging apps (while providing all expected functionality, they covertly exploit image data), JPEG encryption can be a way to tackle buggy implementations of ephemeral messaging features, since pictures are encrypted before being accessed by the app execution space. However, those proposals often do not go into the details of how keys are generated and distributed. They generally require end users to exchange and validate keys a priori, or to deploy a new trusted third party.

In other proposals, the external trusted server distributes temporary image decryption keys. Specifically, a Key-Policy Attribute Based Encryption system is proposed in [24] to enforce access constraints based on expiration time, by providing a fresh key every time the user requests to access an image. As with JPEG encryption, this requires end users to authenticate. In [28], instead of authenticating, recipients are required to solve a CAPTCHA and to provide a hash of the ciphertext, but this only mitigates large-scale attacks where bots are used to collect images massively.

Other strategies leverage hardware-backed pre-installed device keys [29], [30]. Then, a PKI (Public Key Infrastructure) or a WOT (Web of Trust) can be used to authenticate remote public keys. However, since the sender's middleware has direct access to the recipient's public key or user ID, new keys/IDs would have to be generated for every image exchange so as to provide user-to-middleware privacy. Additionally, if the security of the solution relies solely on the secrecy of devices' private keys, there is no solid ground for suspicious activity detection. For instance, massive decryption requests over a short time span might indicate that an attacker has compromised a device's private key and is using a bot. Finally, such approaches do not offer a basis for advanced, privacy-preserving policies, such as requiring recipients to prove ownership of a phone number from a specific country so as to hinder the activity of attackers operating at distant locations.

As an example of a proposal based on pre-installed keys, ShareIff [29] is a middleware service running on the mobile device, between the app and the MOS, in charge of encrypting and decrypting images using device keys from the hardware keystore. It allows the sender to attach a *self-destruction policy* to images, which is a kind of sticky policy [31]. Because the recipient's public key is signed by the manufacturer, the sender can check its authenticity. Similarly, Rushmore's approach [30] uses TEE (Trusted Execution Environment) to perform the middleware operations. Because both in ShareIff and in Rushmore the remote public key has to be authenticated, we infer that a PKI of all manufacturers or a WOT needs to be deployed, which raises user-to-middleware privacy concerns and limits the security of the approach to the secrecy of the device's private key.

IV. Proposed Architecture

Our base scenario consists of a sender who wants to share a private picture with a recipient over an existing mobile messaging app. Without loss of generality, we say that the sender has a phone with MOS-A, and the recipient has MOS-B (they could also be of the same MOS type). For the sake of the example, let us assume that MOS-A is Android and MOS-B is iOS. Our architecture is then composed of:

- **ImageMarker** This is a system function provided by MOS-A that lets the user select the pictures from the phone gallery that will be protected by our system. Thereafter we will refer to them as 'private pictures'. Private pictures can only be accessed through the PrivatePictureAPI, meaning that the messaging app never has access to their contents.
- **PrivatePictureAPI** This API is to be supported in an updated version of MOS-A and MOS-B. It has two methods: PickPicture and ShowPicture. Messaging apps that want to be compliant with our architecture must use this API for selecting pictures to be shared and for showing received pictures. The API bridges those requests to the PA-A app.
- **PA-A platform** PA-A stands for Privacy Agent A. We will assume that the whole PA-A platform is controlled by MOS-A, although in the general case it is just a third party trusted by the sender. This platform does not replace current app integrity APIs; it uses them to perform its functions. It is composed of:
  - **PA-A app** This is a regular Android and iOS app. It encrypts and decrypts private pictures, provides remote attestation to the server, and enforces policies. Note that on the left side of the diagram of Figure 5, the PA-A app is an Android app controlled by Google, whereas on the right side it is an iOS app controlled by Google.
  - **PA-A server** The server verifies the remote attestation provided by the app, enforces policies, and provides the image decryption key to the recipient's PA-A app.

As depicted in Figure 5, these are the steps that must be followed to share a picture with our proposed architecture:

1) SETUP PHASE (only done once)
   (a) The sender uses ImageMarker to select which pictures of the local phone gallery must be treated as private. This would be typically implemented as an MOS function that places them in a special album (as in iOS Hidden album [32]), or that adds a flag in the metadata. ImageMarker also allows to assign policies to each private picture, such as the view once policy.
   (b) Both sender and recipient install and enroll with the PA-A app.
2) SENDING PHASE
   (a) When pressing the 'share picture' button, the messaging app calls the PickPicture method. The user then selects a picture from the phone gallery.
   (b1) If the selected picture is not marked as private, the PrivatePictureAPI works like a regular image picker.
   (b2) Otherwise, the PA-A app gets the picture file from the PrivatePictureAPI, and wraps it by encrypting it with a random key. That key is then encapsulated with *PA-A server's public key*. The privacy policies and two random ID values (UNSEND-ID and PICTURE-ID) are also encrypted with that key, and attached separately. The resulting file (containing the encrypted private picture, the encrypted policies, the encrypted IDs, the encapsulated

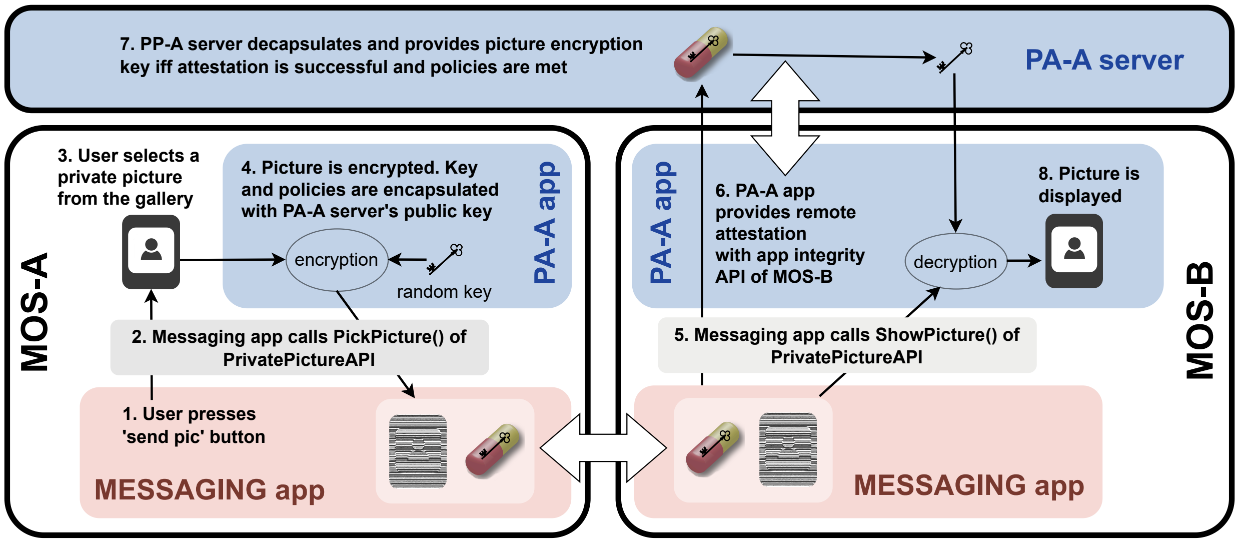

7. PP-A server decapsulates and provides picture encryption key iff attestation is successful and policies are met
PA-A server
3. User selects a private picture from the gallery
4. Picture is encrypted. Key and policies are encapsulated with PA-A server's public key
encryption
random key
PA-A app
MOS-A
2. Messaging app calls PickPicture() of PrivatePictureAPI
1. User presses 'send pic' button
MESSAGING app
PA-A app
6. PA-A app provides remote attestation with app integrity API of MOS-B
8. Picture is displayed
decryption
5. Messaging app calls ShowPicture() of PrivatePictureAPI
MOS-B
MESSAGING app

Fig. 5. Overview of the proposed architecture.

key, and a 'PA-A' tag) is passed onto the messaging app. There is no key exchange between sender and recipient, and no server is contacted to complete this step.

(c) The messaging app processes the picture as usual and transmits it to the recipient.

3) RECEPTION PHASE

(a) When the recipient taps on the picture, the messaging app calls the ShowPicture method.

(b1) If the picture is not a wrapped private picture, the API returns *false*, meaning that the app is in charge of displaying it by using its legacy method for showing images. The process terminates here.

(b2) Otherwise, the 'PA-A' tag is detected, and the received file is passed onto the PA-A app. The Attestation Phase is then started. The API will return *true* after the completion of that phase.

4) ATTESTATION PHASE

(a) The PA-A app sends an attestation request to the PA-A server, which responds with a nonce.

(b) The PA-A app uses the app integrity API of MOS-B to generate an attestation token from the nonce. In our example, since MOS-B is iOS, DeviceCheck is used. This token is sent to the server, along with the encapsulated key, the encrypted policies, and the encrypted IDs.

(c1) If the attestation token is invalid or does not meet a minimum security level, the server returns an error to the client and the process terminates here.

(c2) Otherwise, the server decapsulates the key using its *PA-A server's private key*. The IDs are stored in the server's database. The key, the policies, and PICTURE-ID are sent to the app client, which uses the key to decrypt the picture. Each privacy policy is enforced by the server or by the client, depending on the case. For instance, if the picture has a view once policy and the PICTURE-ID already exists in the database, the server denies access.

(d) The picture is displayed. After a timeout or when closing the picture, control is passed back to the messaging app.

5) OPTIONAL OPERATIONS

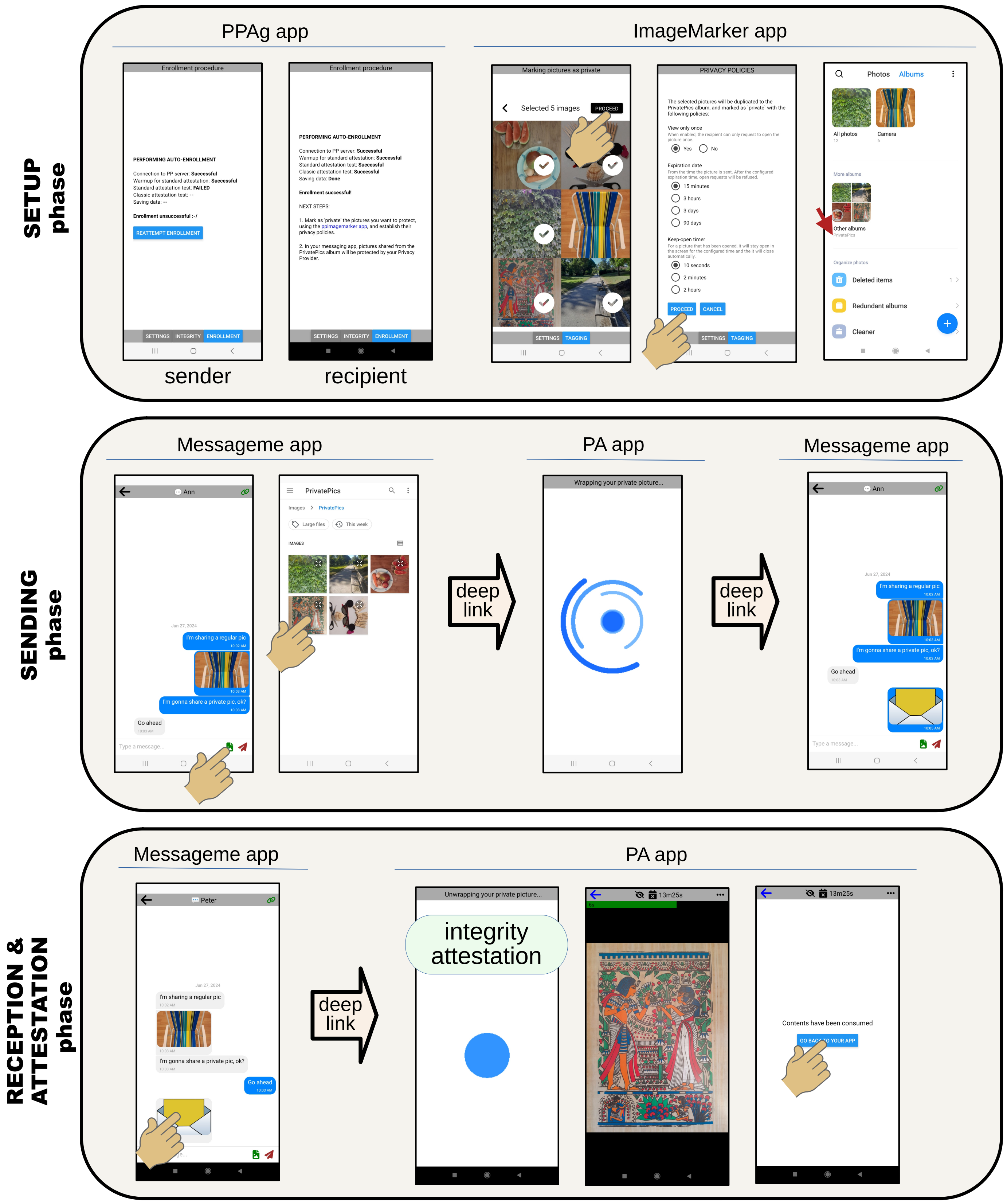
(a) If desired, the sender's messaging app may invoke a special method of the PrivatePictureAPI to unsend (or, delete) a private picture. The PA-A app then uses the UNSEND-ID to request the server to block that picture. This UNSEND-ID is never known to the recipient.

(b) If desired, the recipient can report the contents from within the PA-A app, if deemed in breach with the community guidelines. To do so, the PA-A app first sends a reporting request to the server, which adds a 'reported' flag in the corresponding PICTURE-ID in its database. This will allow authorized content moderators of the messaging app to view the picture. Then, the ShowPicture method raises an exception within the messaging app for it to proceed with the reporting request.

V. Prototype Implementation

We first built Messageme[2], which is our playground, no-frills messaging platform (app + server). It represents a messaging app that existed before our architecture was defined. We then developed the components of our architecture and made the necessary adaptations to Messageme[3]. As for Optional Operations, only the reporting (step 5.b) was partially implemented. All apps were built for both iOS and Android with React Native, as it is a popular multiplatform framework. We only aimed to cover iOS and Android because, combined, they represent more than 99% of the worldwide market share of MOSes [33]. Figure 6 has screenshots of the Android tests.

[2]Available at https://github.com/sam-maverick/messageme
[3]Available at https://github.com/sam-maverick/enhanced-messageme



PERFORMING AUTO-ENROLLMENT

Connection to PP server: **Successful**
Warmup for standard attestation: **Successful**
Standard attestation test: **FAILED**
Classic attestation test: **--**
Saving data: **--**

Enrollment unsuccessful :-/

REATTEMPT ENROLLMENT

PERFORMING AUTO-ENROLLMENT

Connection to PP server: **Successful**
Warmup for standard attestation: **Successful**
Standard attestation test: **Successful**
Classic attestation test: **Successful**
Saving data: **Done**

Enrollment successful!

NEXT STEPS:

1. Mark as 'private' the pictures you want to protect, using the ppimagemarker app, and establish their privacy policies.

2. In your messaging app, pictures shared from the PrivatePics album will be protected by your Privacy Provider.

The selected pictures will be duplicated to the PrivatePics album, and marked as `private` with the following policies:

View only once
When enabled, the recipient can only request to open the picture once.
[✓ Yes] No

Expiration date
From the time the picture is sent. After the configured expiration time, open requests will be refused.
[✓ 15 minutes] 3 hours 3 days 90 days

Keep-open timer
For a picture that has been opened, it will stay open in the screen for the configured time and the it will close automatically.
[✓ 10 seconds] 2 minutes 2 hours

PROCEED CANCEL

Jun 27, 2024
I'm sharing a regular pic 10:02 AM
I'm gonna share a private pic, ok? 10:03 AM
Go ahead 10:03 AM

Contents have been consumed

Fig. 6. Screenshots of the PA app and the modified Messageme app in Android.

```
diff -Bbr messageme/client/src/ enhanced-messageme/emclient/src/
[App.js] 10a11,14
  > // We need this to let PrivatePictureAPI register the Linking.addEventListener at boot time [...]
  > import * as PrivatePictureAPI from '../os_update/PrivatePictureAPI.js';
[visuals/PrivateChat.jsx] 39a40
  > import * as PrivatePictureAPI from '../../os_update/PrivatePictureAPI.js';
[visuals/PrivateChat.jsx] 423c424
  <         let result = await ImagePicker.launchImageLibraryAsync({
  ---
  >         let result = await PrivatePictureAPI.PickPicture({
diff -Bbr messageme/client/node_modules/react-native-lightbox-v2/ enhanced-messageme/emclient/node_modules/react-native-lightbox-v2/
[dist/Lightbox.js] 4a5
  > import * as PrivatePictureAPI from '../../../os_update/PrivatePictureAPI';
[dist/Lightbox.js] 34c35,38
  <     const open = () => {
  ---
  >     const open = async() => {
  >         let resultShowPicture = await PrivatePictureAPI.ShowPicture(children.props.source.uri);
  >         if (resultShowPicture)
  >             return; // If it was a private picture, return here. Otherwise, continue processing.
```

Fig. 7. Modifications made to the code of the Messageme app to comply with the PrivatePictureAPI.
NOTES: Output presentation has been compacted. Some code comments and added logging facilities have been omitted.

A. *ImageMarker*

We implemented ImageMarker as an app. Assuming that the source images are in JPEG format, the app lets the user select a picture and assign privacy policies. The app adds a UserComment EXIF metadata field containing the privacy policies, plus a flag indicating that the picture is private. We implemented the following privacy policies: View Once (Yes/No), Expiration Date (time span relative to the moment the picture is wrapped in step 2.b2), and Keep-Open Timer (whenever the recipient opens the picture, it will automatically close after timer expiration).

B. *PrivatePictureAPI*

A first approach for implementing this component would be to edit and recompile the MOS code itself. However, this has two important limitations. Firstly, this is not allowed by Apple's software license, and it is not technically feasible, realistically. Secondly, modifying Android's source requires rooting the phone, which implies that the Play Integrity API will not work. In the face of such limitations, and to test the entire workflow with a full user experience, we simulated the MOS modifications by embedding a React Native module into the app, located in the `os_update` directory. We then introduced the necessary modifications in Messageme to use this new API, as detailed in the `diff` of Figure 7.

C. *PA-A platform*

We developed the PA-A platform (app + server) as described in §IV. Enrollment is performed automatically upon installation. In Android, the enrollment tests the app integrity attestation, just for convenience. In the case of iOS, attestation is performed as described in [13].

The communication between the PrivatePictureAPI and the PA-A app is done via deep links [34] structured in a predefined format. Execution flow is controlled by mutexes. Whenever the PrivatePictureAPI needs to pass a file to the PA-A app, this is done with a FileProvider in Android, whereas in iOS, file contents are simply encoded as a URL parameter. In Android, temporary file access permissions are granted to the PA-A app, and a custom PrivacyProvider permission ensures that only authorized PA apps interact with the API.

For the encryption of private picture files, we use AES-256-CBC, whereas for the key encapsulation we use 4096-bit RSA, since those are precisely the only algorithms available in the `react-native-quick-crypto` module. For the communication between the app and the server, we use HTTPS with certificate pinning over a private CA, so as to mitigate Person-in-the-Middle attacks on the own network environment.

For better integration, the wrapping of the private picture (step 2.b2) is done by using a static royalty-free PNG host picture of 39 KB [35] to which we add two metadata chunks. One chunk contains the name of the PA platform (in our case, 'PA-A'), and the other contains the rest of the payload, including the encrypted private picture. To detect whether a file is a wrapped private picture, the PrivatePictureAPI just checks that the image is in PNG format and that those two chunks are present. We chose PNG because it allows for metadata fields of up to 1 GiB [36]. Standard EXIF in JPEG files only allows for 64 KB fields [37]. Alternatively, we could have used the ISO/IEC 19566-4:2020 standard, which allows for encryption in JPEG files, but it does not have widespread support yet.

For the attestation part, in Android we use the Standard Play Integrity API with a control over the `timestampMillis` field to verify the freshness of the attestation. If the last API warmup was performed more than 30 seconds ago, our client issues a new warmup to ensure this freshness. We also check that the `deviceActivityLevel` does not exceed a parametrized value, as it could be a sign of an active attack. We selected the Standard variant because the Classic does not seem to offer any advantage. In iOS, we check that the `Counter` field stays within consistent values. To avoid emulators, virtual devices, and non-genuine hardware, in Android we verify that the attestation contains the `MEETS_STRONG_INTEGRITY` flag. In iOS, attestations are hardware-backed by design.

VI. Experiments

In this section, we present the results of the experiments of our prototype, conducted in a controlled lab environment in a LAN. We verify that it exhibits the expected behavior, and we measure the added delay. We also analyze the parts of our implementation that could be optimized in a real-world implementation. For the lab setup, we used the following material:

- Xiaomi Redmi Note 8T with Android 11, 4-GB of RAM, 4-core CPU @ 2.0-GHz & 4-core CPU @ 1.8-GHz, and 64-GB of storage
- Samsung Galaxy A04S with Android 13, 3-GB of RAM, 4-core CPU @ 2.0-GHz & 4-core CPU @ 2.0-GHz, and 32-GB of storage. Rooted with Magisk [7].
- iPhone 8 with iOS 16, 2-GB of RAM, 2-core CPU @ 2.39-GHz & 4-core CPU @ 1.42-GHz, and 64-GB of storage.
- iPhone 14 with iOS 17, 6-GB of RAM, 2-core CPU @ 3.23-GHz & 4-core CPU @ 2.02-GHz, and 256-GB of storage.
- Kali Linux server with 16-GiB of RAM and one CPU with 4 dual-thread cores @ 2.70-GHz.

A. Correctness

We tested all cross-platform combinations, that is: from Android to Android, from Android to iOS, from iOS to iOS, and from iOS to Android. As expected, a wrapped private picture received by the Samsung device cannot be decrypted, since rooted devices cannot provide valid attestations to the server (unless some vulnerability is exploited). This shows that an unmodified messaging app running on a rooted phone (which is precisely the attack scenario we presented in §I) will be denied access to the decryption key. The same behavior is exhibited when receiving a wrapped private picture to the non-rooted Xiaomi device with a modified version of the PA app, since it lacks the appropriate digital signature provided by Google Play or App Store. This shows that recipients who run modified versions of the PA app to override ephemeral messaging restrictions will also be denied access.

B. Latency

Messaging apps typically downsample images automatically, for efficiency. For instance, we took six sample pictures with the Samsung phone, which got downsampled to a median of 128 KB when shared over Snapchat. For our tests, we used the picture from our sample equal or above to this median. This is a 161-KB picture, to which we will refer as REFPIC. Table I illustrates the average values, over nine executions, of the added latency of our system when sharing REFPIC from the Samsung to the Xiaomi device.

For non-private pictures, the added latency is negligible, since the PrivatePictureAPI just checks that the selected picture is not marked as private in the sending phase (or not a wrapped private picture in the reception phase), which we implemented efficiently. As expected, the latency for private pictures is much higher since there are switchovers between the messaging and the PA-A apps, cryptographic operations, and file manipulations. Receiving a private picture takes even longer since, unlike sending, it involves server operations and app integrity API calls.

TABLE I
Average latency introduced by our architecture

	Non-private picture	Private picture
PickPicture (sending)	26 ms	3799 ms
ShowPicture (receiving)	21 ms	4501 ms

C. Latency sources

Using log breakpoints, we did try-and-error until we narrowed down a set of commands that make up to at least 95% of the total execution time, grouping them by function performed. This allowed us to identify optimizable delay sources, when sharing REFPIC as a private picture from the Samsung to the Xiaomi device. For this analysis, we considered the instance with median latency out of nine executions, which corresponds to 3755 ms for sending and 4299 ms for receiving. One of the goals of this analysis is to show that, in a real-world implementation, the latency could be significantly reduced. Figure 8 gives a visual overview of the results.

When sending REFPIC, the switching via the deep link from the Messageme app to the PA-A app, plus vice versa, introduced a delay of 1904 ms. The processing within the PA-A app could be significantly reduced when the user selects pictures from the gallery (as opposed to taking them on the fly with the camera), since the phone could pre-encrypt pictures in the background when idle. This would save about 1019 ms, mainly corresponding to the symmetric encryption and the CRC32 calculation in PNG. Additional 551 ms correspond to Base64 encoding conversions, which could be saved by developing a new module for manipulating files as binary streams, since the `react-native-fs` only supports UTF8 (which we ruled out since it is not binary-safe) and Base64.

When receiving REFPIC, the switching from the Messageme app to the PA-A app introduced an estimated delay of 952 ms[4]. As before, latencies derived from Base64 conversions, which took 1550 ms, could likely be optimized. We observe that the recipient consumes a longer time with those conversions than the sender. This is mainly because the operations performed by the recipient are not perfectly symmetrical to the sender's. For instance, the recipient has to parse a PNG file to check if it is a wrapped private picture, which uses the `png-metadata` module, whose methods get the data in binary format. In contrast, the sender has to parse a JPEG file to check if it is marked as private, which uses the `piexifjs` module, whose methods get the data in Base64. Finally, in our tests, the PA-A client always did a Warmup, which implies that Android's Play Integrity operations, both within the app and the server, involve contacting Google servers.

[4]We do not include the switchover delay from PA-A to Messageme since we consider that the process finishes when the picture is shown on the screen.

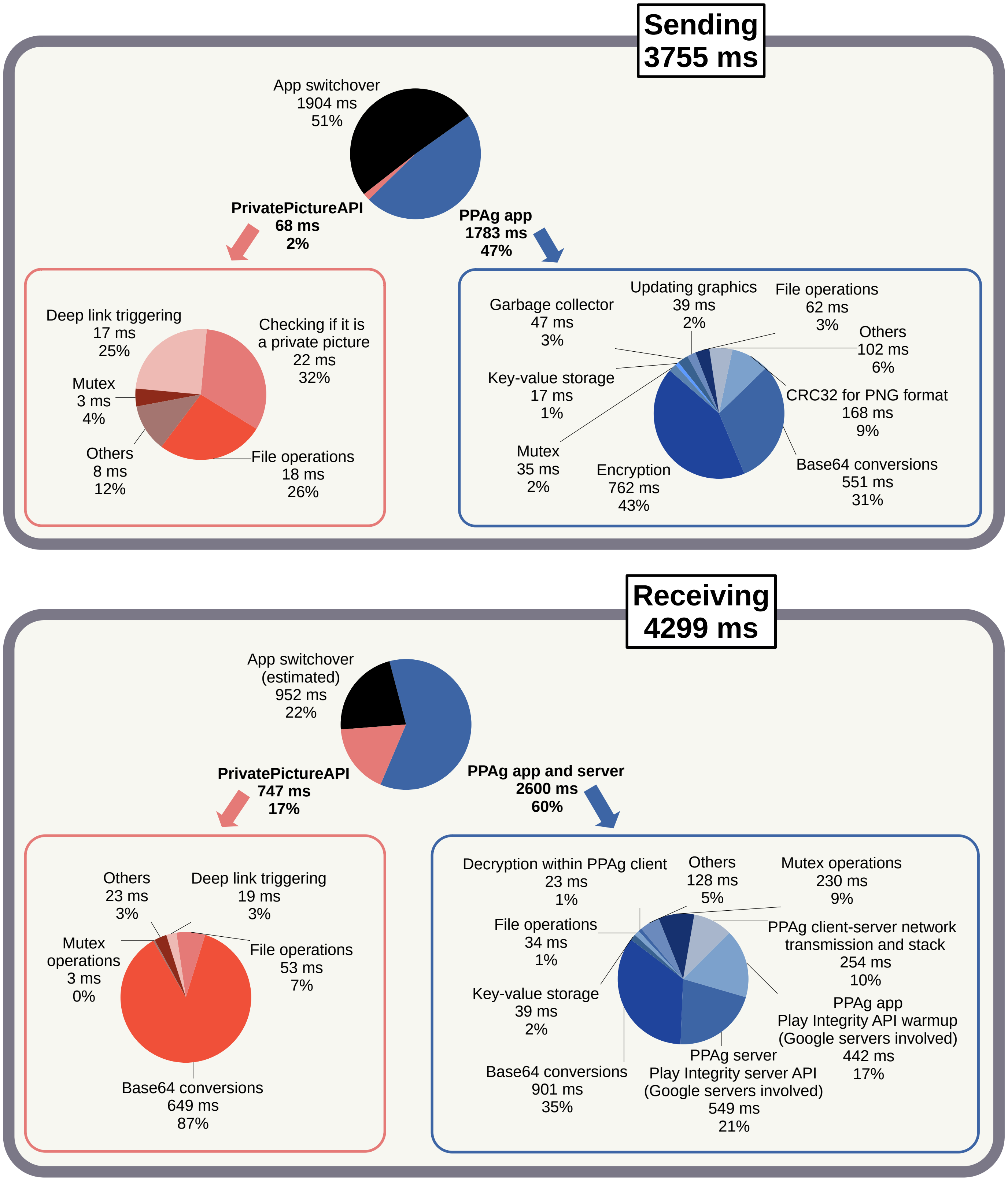


Fig. 8. Breakdown of the added latency when sharing REFPIC as private picture in Android, from the Samsung to the Xiaomi device, using our architecture.

VII. Analysis

In this section, we analyze and discuss privacy, security, usability, compatibility, and deployability properties of our architecture as compared to previous proposals based on a middleware for image encryption and decryption.

A. Privacy

Our architecture allows the PA-A server to enforce a variety of privacy policies. For instance, it can require the recipient to prove ownership of a subscriber line from the sender's country by checking a verification code over SMS for every decryption request or upon enrollment, so as to avoid interactions with bots and attackers operating from remote locations. Crucially, those policies are enforced without sacrificing privacy, since the PA-A server does not gain any information about sender's identity, picture contents, or what messaging app is used.

In solutions that use a PKI of manufacturers to validate keys, information about recipient's OS type (and even phone model, depending on the case) is revealed to the sender before the image exchange. In our model, the sender does not gain such knowledge. On the flip side, recipients gain knowledge about senders' OS type, but this knowledge is only gained after the image exchange (when they see either the 'PA-A' or the 'PA-B' tag in the image metadata).

PKI-based architectures would have to generate a new key-pair (signed by the manufacturer) for every private picture exchange, so as to preserve user-to-middleware privacy. Otherwise, the middleware would gain direct knowledge about the graph-pattern of who the sender is interacting with. In contrast, our middleware cannot easily construct a graph of such interactions. Even with elevated privileges at the MOS level, this knowledge is only indirect, since to obtain such information our middleware would have to analyze the execution logic of the app, or match user interactions by covertly uploading data to the server and looking for coincidences in time.

On the flip side, the PA-A server gains knowledge about how many private pictures have been accessed by a recipient's virtual identity[5], when they have been accessed, and with what privacy policies. Yet there is no information about the virtual identity of the sender or about the picture contents.

B. Security

The security of app integrity attestation does not have to rely only on the secrecy of the underlying device's private key. This allows for further non-cryptographic security protections, such as blocking a device that unwraps a high volume of private pictures over a very short time span, as it might suggest that the private key has been extracted from the hardware keystore and is being used by a bot.

We observe that neither iOS nor Android include the MOS version in attestation tokens. Our PA-A app provides this information separately, so that the server is able to reject attestations coming from devices running versions known to have vulnerabilities. However, an attacker who exploits an MOS vulnerability to gain elevated privileges can potentially hack this version verification mechanism.

[5]In iOS, the virtual identity is the public key generated in the enrollment, and the knowledge is gained by the PA-A server, which we assumed to be under MOS control. In Android, we assume that the Play Integrity API uses a public key known by Google servers, so that MOS servers gain knowledge.

C. Usability

Except for systems based on a PKI or a WOT, middleware proposals for image encryption typically require end users to manually exchange and validate keys a priori out of band. This does not have a good fit with users who prefer to chat anonymously. It is also inconvenient, and detrimental to security since users may end up skipping this validation step. We have lifted such requirement.

The recipient's middleware needs to be online every time it wants to open a private picture. This online requirement for the recipient has nevertheless the benefit that message unsending (step 5.a) takes immediate effect. As a caveat, when unsending, user-to-middleware privacy is affected unless an anonymity VPN is used by the sender.

In our solution there is repeated switching from the messaging app to the PA-A app and vice versa, which might initially cause confusion to end users. However, precisely because all private pictures are handled centrally through the PA-A app, this provides a more unified user experience of ephemeral features across messaging apps.

D. Compatibility

End-to-end encryption has become a fundamental feature of messaging apps. As with other middleware-based solutions, ours does not interfere with or undermine this feature. Our PA-A server does not receive the image ciphertext. So as long as our trusted code running on the mobile device stays legit, the messaging server, the MOS server, and the PA-A server cannot access the private picture contents, even if they collude.

Messaging apps typically allow to report received pictures in breach of community guidelines, which are handled by automated systems and/or human moderators for further examination. In automated systems, our proposal is only compatible with encrypted-domain algorithms and on-device classification. For human content moderation, popular messaging platforms typically outsource such task to third-party companies, which raises privacy concerns. In our system, the chain of custody is preserved because the receiving endpoint that inspects the picture contents must also pass the attestation as if it were a regular recipient. Additionally, the PA-A server will grant access to reported pictures only to endpoints authorized by the messaging platform. And as with conventional messaging platforms, our system allows for content reporting and moderation even after the contents have expired or the picture has been unsent. This reporting can be triggered while viewing the picture, which provides more usability.

E. Deployability

Messaging apps will need to migrate to the PrivatePicture-API, and to refrain from accessing or modifying private picture files directly (non-private pictures are not impacted, though).

Therefore, any techniques used by the messaging app that directly access private picture contents (e.g., nude or recapture detection algorithms, and perceptual hashing) or that modify the contents (e.g., digital watermarking, metadata removal, and image editing) will fail unless they operate on encrypted domain (e.g., JPEG encryption algorithms) or are performed with a system API (e.g., the SensitiveContentAnalysis API [38], [39] for on-device nude detection in iOS).

Messaging apps will also have to refrain from calling ShowPicture for rendering the chat history view. Instead, legacy methods must always be used for the chat history, which will show a static envelope image for private pictures. Otherwise, the ShowPicture method will produce undesired effects.

In systems based on a PKI of manufacturers, we infer that the public key of the devices is exchanged in-band through the messaging app, and then passed onto the middleware via some API. Similarly, in a WOT, some ID has to be passed. This requires substantial code modifications. In our system, though, the required code adaptations within the messaging app should be relatively simple since in our prototype they only required to modify a few lines of code as shown in the `diff` of Figure 7. Also, the messaging server code did not need to be modified at all. Besides, our PrivatePictureAPI implementation accounts for 562 lines of React Native code (excluding external modules). Yet, in a real-world implementation, our architecture will require updating the MOS, plus deploying a new online service.

If the specifications of app integrity APIs change over time or if attestation security enhancements are rolled out, this will not require adapting the messaging app (as long as PrivatePictureAPI specifications stay backward-compatible). This makes the required messaging app adaptations more stable over time.

An important limitation is the throttling of app integrity APIs. In Android Standard Play Integrity, there is a maximum of 10,000 requests in the basic tier per app platform per day (including both warmups and attestations), although this limit can be increased [40]. This means that, for a start, only 10,000 private pictures can be shared per day for all users globally, which is clearly insufficient. As a workaround, attestation could be performed only in the setup phase while enrolling users gradually, and a separate key in the hardware keystore (using existing key generation APIs) would then be generated at setup and used for the rest of operations. Yet this would reduce the security of the solution. In iOS DeviceCheck, Apple servers might throttle attestation traffic from a particular app if "too many instances" of the app make the attestKey call simultaneously [41], which obliges to enroll users gradually.

VIII. Conclusions

Despite significant technical challenges, we designed and tested an architecture to protect from attackers who attempt to circumvent ephemeral messaging features by tampering with the software stack on their own device. We showed that DeviceCheck API in iOS and Play Integrity API in Android provide a common ground that can be leveraged towards a more secure image sharing in messaging apps. This brings a combination of properties that we have not been able to observe in prior work, such as: no key exchanges among end users, and simplified integration with messaging platforms. Compatibility with end-to-end encryption is maintained, and moderation of user-reported messages is covered. We consider the added processing delay of a few seconds acceptable since it is likely optimizable and it will only affect private pictures.

One major issue of our proposal, though, is the throttling of the online services of app integrity APIs. Current limits make our architecture not ready for large-scale usage. However, this may be overcome with enough investment by Google and Apple on server resources, and/or with user segmentation (e.g., making it only available to more vulnerable communities). Another concern is that messaging apps will not be able to perform direct edition or analysis of private picture contents. Such facilities will have to be done on encrypted domain or with system-level APIs, in the case of private pictures. We also recall that our proposal requires establishing technical standards and updating the MOS. In spite of those inconveniences, our architecture is not only feasible but, we believe, the benefits —particularly towards a safer sexting— outweigh the efforts, given all that is at stake.

Acknowledgments

The project that gave rise to these results received the support of a fellowship from "la Caixa" Foundation (ID 100010434). The fellowship code is LCF/BQ/DI22/11940036. This work was also supported by FCT through the LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020).

References

[1] Ericsson, "Number of smartphone subscriptions worldwide from 2016 to 2021," https://www.ericsson.com/en/reports-and-papers/mobility-report/mobility-visualizer, 2022, accessed 3-29-2023.
[2] Reviews.org, "2022 Cell Phone Usage Statistics: How Obsessed Are We?" https://www.reviews.org/mobile/cell-phone-addiction/, 2022, accessed 3-29-2023.
[3] J. Jung, A. Umyarov, R. Bapna, and J. Ramaprasad, "Mobile as a channel: Evidence from online dating," in ICIS. AIS, 2014.
[4] Grindr, "Grindr Unwrapped 2023," https://unwrapped.grindr.com/, 2023, accessed 1-24-2024.
[5] Safe Communities Portugal, "Explicit images believed sent through Snapchat put online, with threats from hackers to upload more," https://www.safecommunitiesportugal.com/explicit-images-believed-sent-through-snapchat-put-online-with-threats-from-hackers-to-upload-more/, 2014, accessed 10-10-2024.
[6] Fox News, "Growing Snapchat 'sextortion' schemes target young boys, expert warns," https://www.foxnews.com/us/expert-warns-growing-social-media-sextortion-schemes-targeting-boys, 2023, accessed 10-10-2024.
[7] Magisk, "Magisk Documentation," https://topjohnwu.github.io/Magisk, 2018, accessed 9-15-2023.
[8] ECP | Platform for Information Society, "More than three-quarters of sexually explicit photos and messages sent via Snapchat," https://ecp.nl/meer-dan-driekwart-seksueel-getinte-fotos-en-berichten-verstuurd-snapchat/, 2017, accessed 11-5-2023.
[9] C. Geeng, J. Hutson, and F. Roesner, "Usable sexurity: Studying people's concerns and strategies when sexting," in SOUPS, 2020.
[10] LSPosed, "LSPosed Framework," https://github.com/LSPosed/LSPosed, 2021, accessed 9-30-2024.
[11] rhunk, "SnapEnhance," https://github.com/rhunk/SnapEnhance, 2023, accessed 9-30-2024.
[12] Android, "Play Integrity API," https://developer.android.com/google/play/integrity/overview, 2022, accessed 2-27-2023.

[13] Apple, "Establishing your app's integrity," https://developer.apple.com/documentation/devicecheck/establishing_your_app_s_integrity, 2020, accessed 1-17-2023. This feature was introduced in iOS 14.

[14] M. Ibrahim, A. Imran, and A. Bianchi, "SafetyNOT: On the usage of the SafetyNet Attestation API in Android," in MobiSys. ACM, 2021.

[15] W. Beijnum, "Haly: Automated evaluation of hardening techniques in Android and iOS apps," Master's thesis, University of Twente, 2023, https://essay.utwente.nl/95578/1/van%20Beijnum_MA_EEMCS.pdf.

[16] Android, "Integrity verdicts," https://developer.android.com/google/play/integrity/verdicts, 2023, accessed 10-20-2024.

[17] ——, "Verified Boot," https://source.android.com/docs/security/features/verifiedboot, 2013, this feature was introduced in Android 4.4 (API level 19). Accessed 2-27-2023.

[18] ——, "Android Security 2015 Year In Review," https://source.android.com/static/docs/security/overview/reports/Google_Android_Security_2015_Report_Final.pdf, 2016, accessed 2-27-2023.

[19] Apple, "Apple Platform Security," https://support.apple.com/guide/security/welcome/web, 2023, accessed 1-11-2023.

[20] Android, "SafetyNet Attestation API," https://developer.android.com/training/safetynet/attestation, 2013, this feature was introduced in Android 4.4 (API level 19). Accessed 2-27-2023.

[21] ——, "Play Integrity API - Classic Requests," https://developer.android.com/google/play/integrity/classic, 2022, accessed 2-27-2023.

[22] ——, "Play Integrity API - Standard Requests," https://developer.android.com/google/play/integrity/standard, 2022, accessed 2-27-2023.

[23] ——, "Optional device information," https://developer.android.com/google/play/integrity/setup#optional_device_information, 2023, accessed 6-11-2024.

[24] J. Xiong, X. Liu, Z. Yao, J. Ma, Q. Li, K. Geng, and P. S. Chen, "A secure data self-destructing scheme in cloud computing," IEEE Transactions on Cloud Computing, vol. 2, no. 4, pp. 448–458, 2014, https://ieeexplore.ieee.org/iel7/6245519/7024233/06963363.pdf.

[25] M. Tierney, I. Spiro, C. Bregler, and L. Subramanian, "Cryptagram: Photo privacy for online social media," in COSN. ACM, 2013.

[26] M.-R. Ra, R. Govindan, and A. Ortega, "P3: Toward privacy-preserving photo sharing," in NSDI. USENIX, 2013, https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final165.pdf.

[27] W. Sun, J. Zhou, S. Zhu, and Y. Y. Tang, "Robust privacy-preserving image sharing over online social networks (OSNs)," ACM TOMM, vol. 14, no. 1, 2018.

[28] J. Backes, M. Backes, M. Dürmuth, S. Gerling, and S. Lorenz, "X-pire! – a digital expiration date for images in social networks," 2011, https://arxiv.org/pdf/1112.2649.

[29] A. Goulao, N. O. Duarte, and N. Santos, "ShareIff: A sticky policy middleware for self-destructing messages in Android applications," in SRDS. IEEE, 2016.

[30] C. M. Park, D. Kim, D. V. Sidhwani, A. Fuchs, A. Paul, S.-J. Lee, K. Dantu, and S. Y. Ko, "Rushmore: Securely displaying static and animated images using TrustZone," in MobiSys. ACM, 2021.

[31] D. Miorandi, A. Rizzardi, S. Sicari, and A. Coen-Porisini, "Sticky policies: A survey," IEEE TKDE, vol. 32, no. 12, 2019.

[32] Apple, "Hide photos on your iPhone, iPad, or Mac with the Hidden album," https://support.apple.com/en-us/HT205891, 2017, accessed 10-20-2023.

[33] Statcounter, "Mobile Operating System Market Share Worldwide, Apr 2022 - Apr 2023," https://gs.statcounter.com/os-market-share/mobile/worldwide, 2023, accessed 5-22-2023.

[34] Meta Platforms, "Linking · React Native," https://reactnative.dev/docs/linking, 2020, accessed 10-11-2024.

[35] OpenClipart-Vectors, "Alphabet Word Images E-mail," https://pixabay.com/vectors/alphabet-word-images-e-mail-email-1295488/, 2024, accessed 6-28-2024.

[36] World Wide Web Consortium, "Portable Network Graphics (PNG) Specification (Third Edition)," https://www.w3.org/TR/png/#5Chunk-layout, 2023, accessed 6-26-2024.

[37] Camera & Imaging Products Association, "Exchangeable image file format for digital still cameras: Exif Version 3.0," https://www.cipa.jp/std/documents/download_e.html?DC-008-Translation-2023-E, 2023, accessed 6-26-2024.

[38] Apple, "SensitiveContentAnalysis Framework," https://developer.apple.com/documentation/sensitivecontentanalysis, 2023, accessed 9-27-2023.

[39] Wired, "Apple Expands Its On-Device Nudity Detection to Combat CSAM," https://www.wired.com/story/apple-communication-safety-nude-detection/, 2023, accessed 9-27-2023.

[40] Android, "Increase your daily maximum number of requests," https://developer.android.com/google/play/integrity/setup#increase-daily-max, 2024, accessed 7-5-2024.

[41] Apple, "Onboard users gradually," https://developer.apple.com/documentation/devicecheck/preparing-to-use-the-app-attest-service#Onboard-users-gradually, 2024, accessed 7-5-2024.