ShareIff: A Sticky Policy Middleware for Self-Destructing Messages in Android Applications

António Goulão, Nuno O. Duarte, Nuno Santos
INESC-ID / Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal
Email: {antonio.m.goulao,nuno.duarte,nuno.m.santos}@tecnico.ulisboa.pt

***Abstract*—Self-destructing messaging applications have garnered immense popularity due to the arrival of Snapchat. However, Snapchat's history has shown that building such services on modern mobile platforms is very challenging. In fact, either caused by programming errors or due to the limitations of existing mobile operating systems, in Snapchat and other similar applications it is possible to recover supposedly deleted messages against the senders' expectations, therefore leaving millions of users potentially vulnerable to privacy breaches. This paper presents ShareIff, a middleware for Android that provides an API for secure sharing and display of self-destructing messages. Using this middleware, Snapchat or any similar application, is able to encrypt the message on the sender's endpoint and send it to the recipient such that the message can be decrypted and securely displayed only on the recipient's device for the amount of time specified by the sender. ShareIff provides this property by relying on specialized cryptographic protocols and operating system mechanisms. ShareIff offers application developers a simple programming abstraction and adds marginal overheads to system and app.**

I. Introduction

Although the research community has explored the notion of self-destructing digital messages [1], this concept has become truly widespread by Snapchat. Snapchat is a mobile application that allows users to share photos that "self-destruct" a few seconds after being viewed by the recipients. A sender can take a picture (e.g., a photo from the camera) and share it with a friend specifying a visibility timeout, which by default is 10 seconds. When the recipient opens the picture – named *snap* – Snapchat displays it on the recipient's screen for the duration of the visibility timeout. As soon as the timeout expires, the picture vanishes from the screen and cannot be read again by the recipient. Since 2011, the year of Snapchat's first release, the service has been upgraded to support other kinds of content sharing (e.g., videos, and text) and richer sharing policies (e.g., replayed for three times before permanent deletion). As of 2014, Snapchat had grown incredibly popular, reaching a user base of 100 million users and a content exchange rate of 700 million photos and videos per day [2].

However, in spite of all its success, Snapchat has been troubled by a history of security flaws, enabling supposedly deleted snaps to be recovered by receivers or even by untrusted third-parties. For example, in a high-profile data breach, named "the snappering", over 100K snaps were leaked by an API exploit [3]. In many cases, security flaws come from application programming errors, ill-designed cryptographic protocols, or insecure RESTful APIs. In other cases, insecurity is due to existing limitations of Android or other mobile operating systems, which, for example, allow users to take screen shots while snaps are being displayed. Security flaws have motivated a formal complaint to the US Federal Trade Commission [4], forcing Snapchat to modify its privacy policy to clearly state that: "We can't guarantee that messages and corresponding metadata will be deleted within a specific timeframe" [5].

Nevertheless, Snapchat remains a highly popular application, which makes us wonder whether security is important at all for its users. In 2014, researchers from the University of Washington conducted a user survey [6] to help understand how people use Snapchat. This study shows that for most respondents, security is not the main concern: they enjoy the fact that messages "disappear", but may not need messages to disappear *securely*. However, while it appears that security is unimportant, up to 25% of the respondents admitted to having sent sensitive content: sexual content, legally questionable material, private documents, and insulting or offensive messages. Moreover, a non-negligible fraction (38.6%) reported that, in response to learning that message destruction is not secure, their behavior would have been different, e.g., by avoiding using Snapchat, sending different content, or sending messages to different people. For such users, a security breach could bring serious and irreversible consequences. Given the large size of the Snapchat user base, millions of users could be affected. Therefore, this study suggests that there is room for more secure self-destructing messaging applications.

To support the development of secure self-destructing mobile applications, we built ShareIff. ShareIff is a middleware for Android platforms. It provides an API that allows mobile applications to encrypt a message on the sender's endpoint and send it to a recipient such that the message can be decrypted and securely displayed only on the recipient's device for the amount of time specified by the sender. ShareIff ensures that the message (1) cannot be recovered while traveling from the sender to the receiver,

(2) cannot be digitally captured while it is being displayed on the receiver's device, and (3) is effectively deleted after the visibility timeout elapses.

To enforce remote message self-destruction, ShareIff adopts a *sticky policy architecture*. At the sender side, the message is enclosed inside a cryptographic envelope along with a sticky policy, which specifies the visibility timeout to be enforced at the recipient endpoint. To prevent interception or modification of the message in transit, the envelope can be decrypted only with a key that is maintained by a protected ShareIff service residing in the recipient's device. While displaying the message, ShareIff keeps the raw bytes of the message inaccessible to the application and to potentially dangerous OS functions that could cause data leakage (e.g., screenshot capture). To guarantee that the message cannot be recovered after deletion, ShareIff ensures correct timer countdown, sanitizes internal buffers, and wipes envelope keys. ShareIff offers its services to application programmers through a simple primitive named `TrustView`, which is inspired by a programming abstraction familiar to Android user interface designers.

Since ShareIff relies on the security of cryptographic algorithms and on the integrity of the recipient's operating system, the operating system must be trusted. Bootstrapping trust in the OS can be achieved by implementing trusted boot using trusted computing hardware [7]. Note that ShareIff is not intended to prevent analog side-channel attacks, i.e., take a photo of the screen using an external camera while the message is on display.

We implemented ShareIff and tested it on Nexus devices. Our evaluation shows that ShareIff adds negligible performance overheads to the system and most of its cost comes from encrypting and decrypting snaps. To validate the programming effort of ShareIff, we implemented a simple photo sharing application based on trustviews. Our experience is that ShareIff is simple to use, requiring just a few lines of code in order to incorporate into the application.

II. Motivation and Goals

A. Background on Snapchat

Snapchat follows a typical client-server architecture. Servers are responsible for managing the user base and for forwarding snaps between clients. Clients consist of instances of the Snapchat mobile application running on users' devices. From this application, users can send or receive snaps to / from their friends. Clients communicate with the servers over HTTP through a REST API.

Consider two users: Alice and Bob. To send a snap to her friend, Alice—the sender—selects a picture, picks Bob's contact from her friend list, and defines the visibility time of the snap (e.g., 10 seconds). Alice clicks the send button, and the local client uploads the snap to its servers, triggering a notification to Bob's client. When Bob—the recipient—opens the snap, the local client downloads the snap from the servers and displays it on the screen. Once the visibility timeout expires, the image vanishes from the screen and is deleted from the local client. The snap is immediately removed from the servers after being downloaded. Until a snap is not opened by a recipient, it remains in Snapchat's servers for 30 days, whereupon it is permanently deleted.

For security enforcement, the Snapchat client relies on Android's *application sandboxing* and *permissions*. Android has a Linux based kernel that enforces UID-based application sandboxing by forcing apps to run inside individual processes. Applications may write persistent data to their private directories and leverage application sandboxing to prevent unauthorized access by other applications. Android permissions regulate app access to system resources.

B. Security limitations of Snapchat

Unfortunately, Snapchat has had security limitations that may result in unauthorized access to users' snaps, i.e., (1) a snap can be retrieved by someone that is not in the recipient list, or (2) a snap can be recovered by the recipient violating the visualization time specified by the sender.

L1. The "analog hole": Consists of side-channel attacks aimed at obtaining an analog copy of a snap while it is displayed on the receiver's device. To mount such attacks and capture the screen output, the receiver requires auxiliary external hardware, e.g., an external camera. Snapchat can neither prevent nor detect such attacks.

L2. Built-in screenshot function: An easy way for a user to make a persistent copy of a snap is by using Android's built-in screenshot service, e.g., by holding down the "volume down" and the "power" buttons for 1-2 seconds. Since Android applications cannot block screenshot capture, Snapchat cannot prevent the user from obtaining a digital copy of the snap and save it on the phone's persistent memory.

L3. Persistent buffers: After downloading a snap from the servers, Snapchat keeps a local copy of the snap on the phone's persistent memory. In Snapchat's first version, such a copy was saved in unencrypted format outside its application sandbox, i.e., in a public space. This flaw allowed for a rogue application to easily access it and create a persistent copy of the snap. Moreover, Snapchat did not delete the snap's local copy after the time limit expired. On later versions, Snapchat fixed this problem by keeping snap copies on its private area and in encrypted format.

L4. Flawed use of encryption: The first version of Snapchat did not use encryption at all. As a result, a snap could be freely accessed by a network eavesdropper or by someone with privileged access to the server. In an effort to secure the snaps while in transit between sender and recipients, Snapchat's later versions started using AES for end-to-end encryption of snaps between sender and receiver endpoints. However, the implementation was seriously flawed: the

symmetric key for encrypting and decrypting the content was unique—for every piece of content, for every user—and was provisioned directly in the Snapchat's application binary. Recent versions of Snapchat retain the keys hardcoded in the application binary and have been reversed engineered [8].

L5. Insecure APIs: By reverse engineering the application binary, researchers managed to decode the API protocols [9], allowing for an ecosystem of tools and third-party apps to appear. By installing such apps, it was straightforward for a recipient to save a snap and replay it any number of times. Although the latest Snapchat API version has been reinforced to mitigate third-party apps, the fundamental problem remains: the security of the service depends upon the obscurity of the API.

L6. Replay attacks: To open a snap on a recipient's device the Snapchat client must perform a sequence of steps: download the content, decrypt it, render it on the screen, set a timer, etc. However, Snapchat cannot guarantee the atomicity of this sequence: by cleverly interrupting or stalling this process, a recipient can visualize a snap multiple times or for a longer amount of time than permitted.

L7. Rooting the device: By rooting the device, a user can execute applications with superuser privileges, allowing such applications to access the Snapchat's sandbox and create permanent copies of the downloaded snaps. Although the rooting operation involves some degree of sophistication, it is not uncommon for Android users to root their devices. Against such attacks, Snapchat is helpless.

C. Goals, threat model, and assumptions

From the security limitations presented above, we see that while some of them are fundamental (namely L1), others are caused by deficiencies in the design or implementation of the Snapchat application (L3, L4, L5, L6) or by existing limitations in the Android operating system (L2, L7). Our goal is to enable the development of secure self-destructing message sharing applications. In particular, our focus in on mitigating the security limitations due to deficient application programming or to OS-related issues. We aim to provide a general solution for self-destructing message delivery, which can be used not just by Snapchat, but also by alternative applications, such as Cyber Dust [10] or Confide [11]. Note that it is not our goal to mitigate side-channel breaches through the "analog hole" (L1).

To attain our goals our approach is to build a middleware based on a sticky policy architecture. Essentially, the middleware aims to provide a simple primitive to applications that enables senders to encode self-destructing messages into cryptographic *envelopes*. Each envelope includes a sticky policy which defines the visualization timeout. The middleware must ensure that the envelope can only be opened at the receiver's endpoint and that the enclosed message (1) cannot be recovered while traveling from the sender to the receiver, (2) cannot be digitally captured while it is being displayed on the receivers device, and (3) is effectively deleted after the visibility timeout elapses. The middleware must provide simple programming abstractions to application developers.

We contemplate the following classes of attacks:

a. Network attacks: An attacker may intercept the communication, collect envelope packages, modify them, and inject new ones. An attacker may attempt to impersonate legitimate receivers in order to launch MITM attacks.

b. API exploits by third-party applications: Third party applications may attempt to retrieve envelope data from the application server API or from the public interface of the application sandbox on the recipient's device.

c. Remote application exploits: The application may have bugs that can be exploited by an attacker on the recipient's endpoint and result in unwanted message recovery.

d. Malicious user operations: A message recipient may try to use OS services, such as taking screenshots or attaching a debugger, in order to create digital copies of messages.

e. Forensic analysis of persistent memory: A forensic analyst with physical access to the device may be able to recover relevant material from devices' persistent memory (e.g., key material, encrypted or unencrypted envelopes, etc).

We assume that the OS's kernel and middleware services are trusted and are therefore part of the trusted computing base. To prevent attacks based on device rooting (L7), it is possible to employ hardening techniques coupled with trusted computing hardware in order to assure trusted boot state [12]. Such techniques can be deployed by device manufacturers when manufacturing ShareIff-enabled devices. We also assume that ShareIff devices can be shipped with a cryptographic key pair that can be used to uniquely identify the device. This key pair is unique per device. We also rely on the fact that the content of the device's volatile memory is lost when a device is powered down.

III. Design

We present ShareIff, a sticky policy middleware to support self-destruct messages in Android applications.

A. Architecture

Figure 1 represents the architecture of the ShareIff middleware on an Android device. ShareIff consists of three main components: API, manager, and renderer components. The ShareIff API provides applications an interface to create envelopes on the sender (close operation), render them on the receiver (open operation), and manage ShareIff-specific keys. The ShareIff manager and the ShareIff renderer are components that reside in the OS domain and are isolated from user applications. The ShareIff manager holds sensitive cryptographic keys and performs encryption and decryption

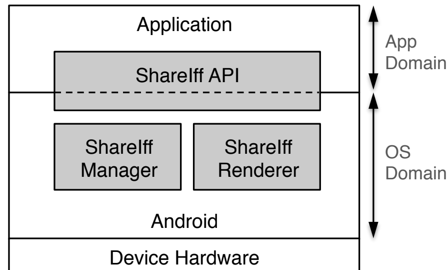

Figure 1. Architecture of the ShareIff middleware.

securely from the app. The ShareIff renderer displays internally decrypted envelopes on the local screen.

At a high-level the typical workflow performed by application programmers when using ShareIff is as follows:

1. Share public keys with other users. Every ShareIff device contains a set of cryptographic keys upon which envelope close and open operations are built. In order for a user to encrypt envelopes and let them be visualized by her friends, the application must obtain the public keys stored by ShareIff on her device and share such keys with their friends. The public keys are obtained through an API call.

2. Close envelope on the sender endpoint. Once in possession of the ShareIff public keys of their friends, a user can send a self-destructing message securely to a friend by generating an envelope. The application invokes an API call to "close" an envelope containing the message content and a sticky policy that specifies the visualization timeout. The resulting envelope can then be forwarded by the application to the receiver, e.g., through an application-specific server.

3. Open envelope on the receiver endpoint. When the application running on the recipient's device retrieves the envelope, the enclosed messaged can be shown on the screen by invoking the "open envelope" API call. This call forwards the envelope to ShareIff's protected components residing in the OS. ShareIff decrypts the envelope using the keys maintained by the manager and renders the recovered message (typically an image) on screen for the time duration specified in the envelope's sticky policy. While the message is on display all potentially dangerous channels are blocked (e.g., screenshots are disabled). ShareIff provides that the message vanishes once the maximum time is enforced, that it is securely erased, and that the message cannot be opened twice, therefore avoiding message replays. Next, we dive into design details of ShareIff.

B. The Trustview API abstraction

Among all ShareIff API calls, the open envelope operation is the most disruptive for application programmers since it affects the way they typically manipulate images or text messages on their applications. In fact, typical applications have full control of the images to display and their placement on screen. However, on the recipient's endpoint, ShareIff applications have no access to the image enclosed in the envelop since the envelope is decrypted and rendered exclusively by the OS-protected ShareIff components.

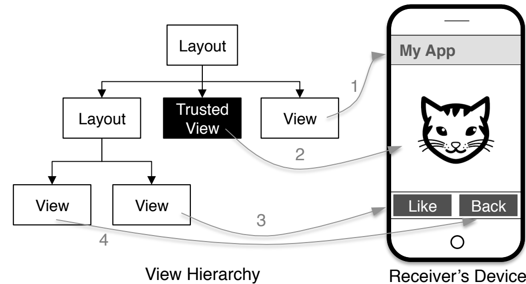

Figure 2. Example of a trustview in a screen layout.

To lessen the impact of such restrictions to application programmers, ShareIff provides a simple abstraction that allows for the manipulation of enveloped images in the application. This abstraction, named `TrustView`, is inspired on Android's `View` class, which application developers are familiar with. In particular, images are typically rendered inside `ImageView` objects, which represents a screen area in which images are displayed.

Trustviews borrow the concept of view in order to represent GUI elements where self-destrucing messages can be securely rendered and deleted once the associated timeout expires. Such GUI elements are represented by `TrustView` objects and can be declared like a regular view, i.e., in the XML layout file or programmatically in the application code. Furthermore, trustviews can be incorporated into a typical view hierarchy, co-existing with regular views, just like in the example shown in Figure 2.

In contrast with regular `ImageView` objects, trustviews have hardened method interface. First, a programmer can only load content into a trustview so long as it is encoded inside an envelope. Second, the programmer cannot read or change the raw bytes of the trustview content. Furthermore, the programmer cannot change the visibility timeout associated with the content and specified in the envelope's sticky policy. The programmer is restricted to a few method calls to: create or destroy a trusted view, configure its layout properties, and open envelopes within it.

C. Security protocols

This section describes the security protocols that are triggered by client applications through the ShareIff API. We use the following notation to describe cryptographic operations. For asymmetric cryptography, K^- and K^+ denote private and public key, respectively. To refer to symmetric keys and asymmetric keypairs, we drop the superscript. Notation $\langle x \rangle_K$ indicates that a piece of data x is encrypted with key K and $\{x\}_K$ that x is signed with K. The symbol $||$ represents concatenation. Function *hmac*(m, k) yields the HMAC of message m with key k.

1. API primitives: The ShareIff API provides a set of security primitives to perform three main functions: close envelopes, open envelopes, and manage internal cryptographic keys. Such keys are maintained by the ShareIff manager in each device and consist of three keypairs: *platform key* (KP), *screening key* (KS), and *receiver key* (KR). The purpose of these keys is clarified in the following sections. The private part of these keypairs is maintained secretly by ShareIff. The core API primitives of ShareIff are:

- CLOSEENV($M, T, C_{KP}, C_{KS}, [C_{KR}]$) $\rightarrow E_{[4|5]}$: Generates an envelope at the sender side. It takes a message M to be sent, a maximum visualization time T, a certificate of the platform key C_{KP}, a certificate of the screening key C_{KS}, and (optionally) a certificate of the receiver key C_{KR}. These certificates must be obtained at the receiver side through local key management primitives. The return is an envelope, which can be of format E_4 or E_5 (discussed below) depending on whether or not C_{KR} is provided as input.
- OPENENV(E, v) $\rightarrow$ `OK` | `Fail`: Renders the content of a received envelope E on the trustview v. If the call is authorized, ShareIff decrypts the message, shows the resulting image according to the enclosed timeframe restriction, and returns `OK`. Otherwise, the message is not shown and the call aborts.
- GETPKEY() $\rightarrow C_{KP}$: Returns the certificate of the local platform key KP. The certificate must be transmitted to potential senders.
- NEWSKEY(C) $\rightarrow C_{KS}$: Creates a new screening key on the local ShareIff service. The created key has a maximum envelope opening count C. Returns a certificate of the screening key (covered below). The certificate must be transmitted to potential senders.
- NEWRKEY() $\rightarrow C_{KR}$: Creates a new receiver key on the local ShareIff service. Returns a certificate of the newly created key KR (discussed below). This certificate is only required by senders that require strong authentication of the message recipient.

The end goal of these primitives is to generate an envelope E (returned by CLOSEENV) that can be securely decrypted and rendered in the receiver's device (when invoking OPENENV). Envelope generation is performed cryptographically and constitutes the main challenge since several security properties must be satisfied. Next, we present the security protocols involved in envelope generation, starting from the simplest version (E_1) to the most complete (E_5).

2. Provide end-to-end privacy: We start by ensuring end-to-end privacy between the sender and the receiver of a self-destructing message. To this end, the envelope is given by:

$$E_1 \rightarrow \langle M, T \rangle_{KE}$$

To generate E_1, ShareIff encrypts both the message M and a sticky policy T with an *envelope key* (KE). The envelope key is a symmetric key which aims to protect the confidentiality of the image data. For each envelope, there is a unique randomly-generated KE, which is generated at the sender side by CLOSEENV. The sticky policy includes the visualization timeout. In order for the recipient to decrypt the envelope, the key needs to be securely transmitted to the recipient's device. To prevent MITM attacks, the receiver's client must be properly authenticated.

3. Bind to remote trustview environments: The first step towards a proper authentication of the receiver's endpoint is to bind the envelope to remote trustview environments. In other words, we need to guarantee that an envelope can be decrypted and rendered only inside a legitimate trustview environment protected by the OS. Without such guarantee, the envelope can potentially be decrypted on the user space, allowing an application to save a permanent copy of the message, thereby violating the sticky policy.

$$E_2 \rightarrow \langle M, T \rangle_{KE}, \langle KE \rangle_{KP^+}$$

$$C_{KP} \rightarrow \{KP^+, \texttt{ShareIff-KP}\}_{KM^-}$$

To bind the envelope to remote trustview environments, in E_2, we encrypt the envelope key KE with the public key of a certified *platform key* (KP). A platform key is a unique keypair that device manufacturers can deploy when bundling their ShareIff-enabled Android devices (one keypair per device). Such a key can be stored in encrypted format and released to the ShareIff manager upon correct validation of integrity of the Android software during a verified boot sequence [13]. The public part of KP can be certified by the device manufacturer using its key KM so that the sender can validate whether that key corresponds to a legitimate Shareiff-enabled device. Since in such a device the private part of KP never leaves the Shareiff manager, by encrypting the envelope key with KP^+, senders can be assured that the envelope can be decrypted only inside a trusted ShareIff manager service, which in turn restricts enveloped images to be rendered inside trustviews only.

4. Authenticate the remote application: A second step to properly authenticate the receiver's endpoint consists of authenticating the application. In other words, we must ensure not only that the envelope is bound to be opened on trustviews only, but also that such trustviews belong to a trusted application. Otherwise, envelopes' sensitive messages could be recovered inside trustviews attached to an illegitimate third-party application. To authenticate the remote application, we extend E_2 as follows:

$$E_3 \rightarrow \langle M, T \rangle_{KE}, id_A, hmac(\text{m}, KE), \langle KE \rangle_{KP^+}$$

$$\text{m} \rightarrow \langle M, T \rangle_{KE} \mid\mid id_A$$

In this version, the envelope contains identity information (id_A) about the targeted remote application. This identity can be obtained by simply calculating the hash of the

application's distribution package, which in turn is usually signed by the respective application programmer. When the ShareIff manager receives the envelope at the recipient endpoint, it first checks the identity of the local application that invoked the OPENENV primitive and verifies if that application ID matches the ID specified in the envelope. If not, authentication fails and envelope decryption will be denied. Envelope format E_3 also has a HMAC of the payload keyed with KE to guarantee the integrity of the envelope.

5. Prevent replay attacks: Before presenting the last step for complete remote peer authentication, we need to address the potential threat of replay attacks, in which a given envelope can be (partially or totally) visualized multiple times: E_3 is vulnerable to such attacks. On the one hand, an application can successfully invoke the OPENENV to open an envelope multiple times. Plus, as long as the platform key (KP) lives in the system, envelopes can be decrypted. This is because KP is used to decrypt the envelope keys. As a result, a leak of the platform key (for example, obtained through forensic extraction from persistent memory) may allow for the decryption of all past envelopes that have been sent to the device identified by that key.

In the next version of the protocol, we aim to prevent replay attacks such that an envelope can be opened a single time only and cannot be visible to the local user beyond the time frame indicated in the sticky policy. Thus, even if the application interrupts the visualization of the message before the maximum time frame, the receiver must not be able to replay the message. After finishing the visualization time, the message must not be able to be recovered at all. For this purpose, we modify the protocol as follows:

$$E_4 \rightarrow \langle M, T\rangle_{KE},\ id_A,\ hmac(\text{m},\ KE),\ \langle KE\rangle_{KS^+}$$

$$C_{KS} \rightarrow \{KS^+, \texttt{ShareIff-KS}, C\}_{KP^-}$$

In this version of the protocol, instead of encrypting the envelope key with the public part of the KP, the sender encrypts it with a *screening key* (KS). The screening key is in fact a keypair which is created by the ShareIff manager and aims to protect the envelope key against replay attacks. To preserve the original link to the platform key, the public part of the screening key must be signed by the KP and the respective certificate sent to the sender. The sender can then verify the screening key certificate and generate the envelope as before but now using the KS to encrypt the envelope key.

There are two additional features about screening keys that mitigate the problem of the replay attacks. First, screening keys are ephemeral. ShareIff preserves the private keys KI^- in volatile memory only and are never written to persistent memory. As a result, if the device is shut down or rebooted, KI^- keys are forcefully destroyed. Therefore, even if the KP is forensically extracted, it will not be possible to recover KI^- required to decrypt the envelope key.

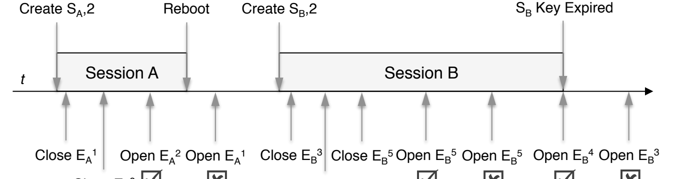

Figure 3. Examples of screening sessions.

Second, screening keys "have memory" of all envelopes that they have previously open. Whenever an envelope is requested to be open by an application, ShareIff maintains a volatile log associated with the screening key that records a hash of the envelope that was opened. If the same envelope is requested to be reopened, ShareIff checks the log and refuses to decrypt and render the image if an entry is found on the log. This method defeats repeated invocation attempts to open the same envelope.

To prevent screening key logs from growing indefinitely, ShareIff sets a limit to number of different envelopes that can be open. In other words, screening keys expire after a pre-defined number of envelope open operations has been reached. This pre-defined number corresponds to the log capacity, which is the parameter C of the NEWSKEY operation. The log capacity C is also included in the certificate returned by ShareIff upon creation of the screening key. Once the log is full, the key is destroyed by the system. Application programmers are free to define a value for C that satisfies the requirements of their applications.

Adopting these techniques introduces some changes to the semantics of trustview's basic primitives. From the application point of view, this semantics can be captured by the notion of *screening sessions*. A screening session corresponds to the lifespan of a screening key KS. The session is opened when the key is created and closed if one of two events occurs: the device reboots, or the key expires. Envelopes can only be recovered when a session is opened.

We illustrate this notion in Figure 3, which represents the lifespan of two sessions—A and B—initialized with a log size of 2 entries. This means that each session can open at most two envelopes. Five envelopes have been closed in total: E_A^1 and E_A^2 for session A, and E_B^3, E_B^4, and E_B^5 for session B. As shown by envelopes E_A^1 and E_A^2, an envelope can be opened only while its respective session is active (in this case session A). E_B^3 illustrates that an envelope can be opened once. Rebooting the device leads session A to termination. Opening a number of envelopes equal to the maximum log size ends the session. This is what happens to session B after opening two envelopes: E_B^4, and E_B^5.

An immediate consequence is that some envelopes may never be able to be opened, either because the target device has rebooted and screening key was lost or because the number of opened envelopes has expired. ShareIff apps must keep track of such situations and if necessary regenerate the envelope on the sender and re-send it to the recipient.

6. Authenticate the receiver: The last step to authenticate the remote endpoint is to authenticate the message recipient itself. Normally, in a typical messaging application, every user that signs up is assigned a unique name within the application domain. If someone wants to send a user a message, the recipient is identified based on this user name. For stronger security, ShareIff allows for recipients to be authenticated directly using cryptographic mechanisms bundled into the envelope as follows:

$$E_5 \rightarrow \langle M,T \rangle_{KE},\ id_A,\ hmac(\text{m},\ KE),\ \langle\langle KE \rangle_{KS^+} \rangle_{KR^+}$$
$$C_{KR} \rightarrow \{KR^+, \texttt{ShareIff-KR}\}_{KP^-}$$

Essentially, the idea is simply to encrypt the encrypted KS key with another layer of encryption using a *recipient key* (KR). The recipient key is a unique key associated to the local user. It can be generated by the application using the primitive GENRKEY. This call returns a certificate of KR^+, but the respective private key never leaves the ShareIff manager. Thus, by encrypting the envelope key with the recipient's public, only the legitimate receiver can decrypt the envelope. This ensures strong recipient authentication. Since this last step is not crucial for the correctness of the self-destructing service, we leave it as an optional feature.

D. Policy enforcement architecture

Figure 4 gives us an overall perspective of the ShareIff policy enforcement architecture. When the recipient application receives an envelope containing the image to be displayed, it issues an OS system call (step 1) to the ShareIff Manager Service, so that it can process the image in question. This OS system call is transparently invoked by the OPENENV. At that point, this service extracts the envelope's contents, validating its signature and other metadata contained in the envelope, and decrypts the image (step 2). In possession of the image raw information, the service delivers this same information to ShareIffUI (step 3), a component created specifically to handle the rendering and posterior destruction of the image.

ShareIffUI corresponds to the ShareIff renderer and was designed to prevent access to raw trustview images through potentially compromising paths allowed in Android's stock architecture. In step 4, ShareIffUI addresses one of such paths, namely the ability to physically access the device using a debugger in order to retrieve the raw image data. By disabling the communication between the OS and the Android Debug Bridge (ADB) in the System Settings, ShareIffUI prevents a physical attacker from plugging the device to a computer and using ADB to take screenshots (using the screencap command), or a debugger to access variables that might contain the image raw data. ShareIffUI handles the screenshot issue in step 5. We enhanced Global Screenshot so that ShareIffUI can temporarily disable the screenshot capability for as long as there is a privacy sensitive image being displayed by ShareIff.

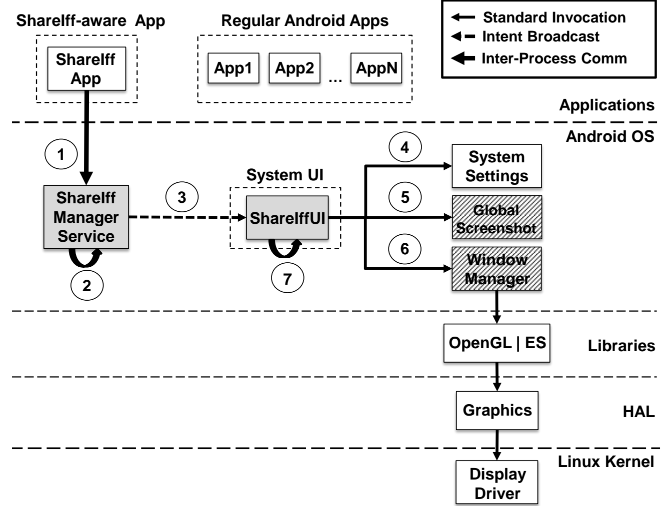

Figure 4. ShareIff overall architecture.

Then, Step 6 represents the actual rendering of the image, that is first invoked from the Window Manager, and traverses Android's layers, until reaching the OS, where the rendering on screen actually takes place, already in the device driver.

Finally, step 7, represents the image deletion mechanism. Basically, after issuing the render of the image, ShareIffUI launches a background thread that essentially works as a timer. After the timeout, ShareIffUI triggers the image removal through the Window Manager, signals the Global Screenshot to enable screenshots, and reestablishes the communication between the OS and ADB. To prevent replay attacks based on system clock manipulation, the timer used by ShareIffUI is independent of the system clock.

IV. IMPLEMENTATION

Our ShareIff prototype was developed on top of Android KitKat (4.4.1). We chose this version because of its maturity, but the process of porting ShareIff to a more recent Android version would be trivial, as the structures involved are transversal to the different versions. All the work involving end-to-end privacy and endpoint authentication follow the cryptographic protocols specified in Section III. For symmetric cryptography we use AES-256, and for digital signatures RSA-1024 and SHA-2. Regarding key management, and because we opted to make the keys ephemeral, they are not stored persistently, but rather in volatile data structures that are destroyed once the device is turned off.

The evolution in camera technology made so that current smartphones are equipped with powerful cameras, which produce high resolution pictures, and therefore big files. One technical challenge we faced was the transmission of pictures and envelopes containing these big sized pictures between apps and the system service responsible for encryption / decryption. Because apps and system services reside in different processes, standard system service method calls are done through Android's standard IPC mechanisnm (i.e.,

Resolutions (MP)	Resolutions (Pixels)	File Size
0.3MP	640x480	200KB
2MP	1600x1200	1.4MB
3.1MP	2048x1536	2.6MB
5MP	2592x1944	4.3MB
8MP	3264x2448	6.8MP
12MP	4096x3072	10.9MB

Table I
EXPERIMENT IMAGE RESOLUTIONS AND FILE SIZE.

Binder). Given that this mechanism has a fixed data buffer limit size of 1Mb, we opted to handle pictures and envelopes by storing them persistently and using URIs to referrence them. By doing so, we leveraged Android's application sandbox mechanism to protect this data from other apps' unauthorized accesses.

V. EVALUATION

The evaluation of the ShareIff prototype features a qualitative evaluation of its API, performance overhead measurements, and a security analysis of the system.

A. *Performance evaluation*

To study the performance of ShareIff, we measure the execution time of its API calls, through microbenchmarking. Our hardware testbed consisted of a Nexus 4 smartphone, featuring a quad-core 1.5 GHz CPU, 2 GB of RAM, 16 GB of storage, a 768 x 1280 display, and a camera with 8 MP, 3264 x 2448 pix. The device was flashed with Android 4.4.1 AOSP patched with ShareIff code. For each experiment, we report mean and standard deviation of 50 runs.

This performance study features the execution time measurements of two types of operations: (1) creation of an envelope, and (2) opening of an envelope. The first type of operation features the encryption of an image, and the creation of an envelope structure containing that image and a custom ShareIff policy. For testing purposes, our custom policy sets a 10 second maximum visualization time. The second type of operation encompasses the decryption of the image, as well as its rendering on a trusted image view. To evaluate the performance of this operation, we compare it with Android's original way of rendering an image.

Considering that the performance of these operations depend on the size of the images involved, we carried out measurements for images with different sizes. For a matter of consistency, we used the same image, and resized it to resolutions compatible with today's mobile device cameras. In our experiments, we selected camera resolutions supported by our Nexus 4 devices. When picking the resolutions we took under consideration Android's original `setImageBitmap` method's maximum resolution limits (i.e., 4096x4096 pixels). When discussing the results of these operations, we address the different images by their resolutions. The details on the resolution and file sizes of the images in question can be found on Table I.

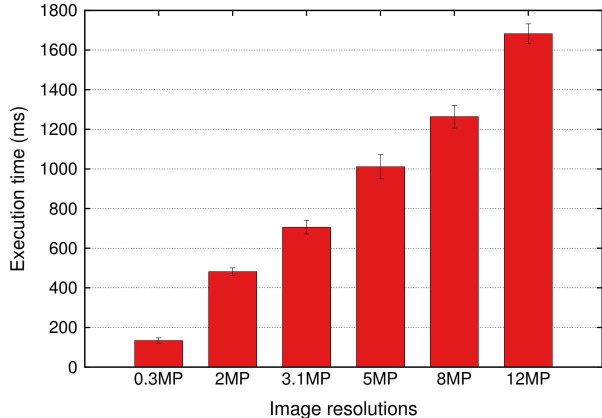

Figure 5. Execution time of envelope creation.

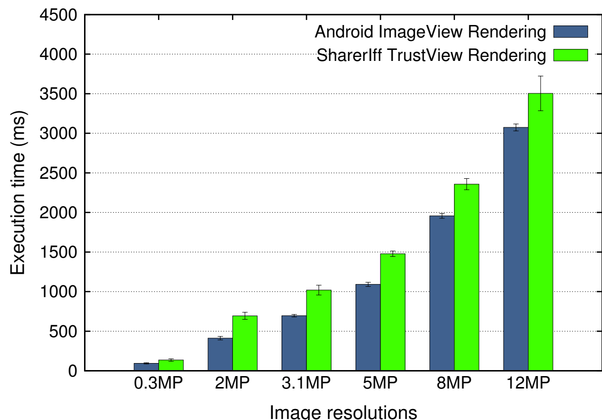

Figure 6. Android native and ShareIff's image rendering execution time comparison.

Figure 5 shows us the execution times of our envelope creation primitive, when handling different resolution images. In this figure we can observe a logarithmic pattern in the overhead introduced by the image encryption operation, the most time consuming component of this primitive. While there is an increase in this primitive's overhead, as the resolution of the image's increases, we can also observe that the growth rate of the overhead progressively decreases.

Figure 6 shows us a comparison between the image rendering execution times on native Android, and on ShareIff. As we can see, for lower resolution images the overhead of ShareIff's rendering can take close to twice as much when compared to Android's native rendering. However, while the overhead increases linearly, it does so at a slow growing rate, as we can observe in the higher resolution images of this rendering process.

It is important to stress that, in Android, we can measure the time spent on rendering an image by measuring the `setImageBitmap` method call's execution time. However, in order to reliably compare ShareIff's implementation with Android's, it is also important to consider the execution time of the `BitmapFactory.decodeByteArray` method

call as well. This is a computationally heavy method, that loads the data to be rendered from memory. Without accounting for this method, ShareIff's overhead compared to Android's native rendering execution would lead to great discrepancies. Therefore, our Android native rendering measurements account the execution times of both the decoding operation, and the actual rendering method call.

If we look at Figure 6 we can see that for 12MP images, the rendering time is of more than 3 seconds, which may compromise the user's experience. Even so, Android's native rendering for 12MP images is only half a second faster, which in comparison is not a great improvement. To preserve the user's experience, Android usually uses thumbnails to alleviate this time difference. For instance, before displaying a large image in the gallery, Android usually presents a lower resolution version of the same image (i.e., thumbnail), and then presents some sort of loading screen before showing the real image.

B. Case study

In order to test the effectiveness and benefits of the functionalities provided by ShareIff, we developed a use case application called Photobrake. This application provides a self-destruct messaging service similar to those of Snapchat and relies on a centralized server to forward snaps between clients. Building the Photobrake application allowed us to gather hands on experience about the programming complexity of ShareIff. Regarding both the creation of new envelopes and their respective decoding, ShareIff's primitives are very concise and intuitive. In less than 10 lines of code, it is possible to write the necessary Java code to create an envelope. Similarly, opening an envelope requires just a few lines of code to invoke a single trustview method.

However, the API methods involved in the management of the ShareIff-specific keys introduced an additional complexity when compared to a regular application like Snapchat. First, users must explicitly upload platform key certificates to the server the first time they register on the application. Second, Photobrake clients must necessarily create a new screening key every time the device reboots (remember that screening sessions are stored in volatile memory) and the respective certificate must be uploaded to the server. Third, the application must keep track of the log count associated with each screening session and if necessary generate a new screening key. Fourth, the application must keep track of snaps that could not be viewed due to a reboot of the recipient device. Nevertheless, we found that most of these steps could be performed relatively easy by extending the server API and adding a few records to the server database to keep track of the state of each key.

C. Security analysis

Except for the "analog hole" limitation (L1), the ShareIff middleware can effectively overcome the security limitations of Snapchat discussed in Section II-B. ShareIff thwarts limitation L2, by disabling the screenshot and ADB functions while snaps are displayed on screen. Limitations L3, L4, and L5 are overcome by ensuring secure end-to-end encryption of snaps' data between the sender and receiver and by the fact that raw snap data can only be accessible within the operating system. Replay attacks (L6) are defeated by employing volatile keys (screening keys) for encrypting the envelopes. Lastly, L7 is mitigated by employing verified boot mechanisms in order to detect rooted Android software.

If the phone gets stolen and the attacker turns off the phone, the volatile keys are automatically destroyed. By offering a rendering mechanism decoupled from the app, we can ensure that no untrusted application can overlay an image of its own covering the envelope's image. The image removal mechanism resides in the OS, which means no app can remove the image before the time is up (except through the back button), or let the image on screen for more time than the policy first stipulated.

In the current ShareIff design, because the creation of an envelope involves specifying the receiver's id through it public key, there is a dependency from the system service and an external service that translates a known user id into a certificate. It is from that certificate that the service extracts the receiver's public key to proceed with the encryption of the envelope contents.

VI. Related work

There is already a vast amount of work targeting the protection of Android users' data and privacy. These systems leverage several solutions, ranging from data shadowing techniques [14] to information flow control [15]. Another group of solutions protect users' data by refining Android's permission model with additional techniques [14]. Other systems can also limit access to private data through security profile specification [16], effectively assuring data access isolation based on location and time of day (e.g., Work, Private). Although these systems provide solutions to control access to particular data on a user's device, they do not allow for a remote entity, more specifically that data's original owner, to specify usage constraints over that data.

Digital Rights Management (DRM) is another extensively used mechanism, that allows for data owners to constrain the way end-users and their applications handle private data. Android provides a framework that allows for developers to enable their apps to manage DRM-protected content [17], but depends on device manufacturers' specific modules, which differ from device to device. Porscha [18] is a content protection framework that allows data owners to express security policies to ensure that their data is sent to targeted phones, processed by endorsed applications, and handled in intended ways. However, Porscha involves trusted third-parties for content distribution, whereas ShareIff's message exchanging protocols work entirely between sender-receiver.

There is also a lot of work related with remote data management. Some solutions aim at protecting / deleting sensitive data on stolen smartphones [19], others to improve users' control over personal data published in social networks [20]. These systems employ a paradigm first introduced by [21], where data is encrypted with a symmetric key, and subsequent accesses to this data are further dependent on the availability of the key, which is managed by the user, usually through a cloud service. On the other hand, there are several systems that support the expiration date mechanism, but focusing on decentralized solutions. Vanish [1] leverages P2P distributed hash tables to store key shares, which are responsible for granting access to users' encrypted data.

A number of other systems target secure deletion of data once it is no longer needed by the application. Lacuna [22] runs java applications in custom VMs and focuses on OS buffer erasure to eliminate app execution traces. PrivExec [23] provides private process execution to ensure that writes to the filesystem or swap cannot be recovered during or after application execution.

VII. Conclusions

In this paper we presented ShareIff, a middleware for Android that provides an API for secure sharing and display of self-destructing messages. Many attempts have been made to create such messaging services, with Snapchat being perhaps the one who gained more popularity. In this work, we seek to tackle both the end-to-end security when exchanging messages, but also the secure handling of messages on the receiver's device. ShareIff offers apps the possibility to encrypt a message on the sender's endpoint and send it to the recipient such that the message can be decrypted and securely displayed only on the recipients device for the amount of time specified by the sender. ShareIff does so by offering application developers a programming abstraction called TrustView, that introduces marginal overheads to both system and application.

Acknowledgments: This work was partially supported by the EC through project H2020-645342 (reTHINK), and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 (INESC-ID).

References

[1] R. Geambasu, T. Kohno, A. A. Levy, and H. M. Levy, "Vanish: Increasing data privacy with self-destructing data," in *Proc. of USENIX Security*, 2009.

[2] B. Insider, "Snapchat Is A Lot Bigger Than People Realize And It Could Be Nearing 200 Million Active Users," 2015, http://www.businessinsider.com/snapchats-monthly-active-users-may-be-nearing-200-million/-2014-12.

[3] ——, "Hackers Access At Least 100,000 Snapchat Photos And Prepare To Leak Them, Including Underage Nude Pictures," 2014, http://www.businessinsider.com/snapchat-hacked-the-snappening-2014-10.

[4] U. S. of America Federal Trade Commission, "Complaint 132 3078," 2014, https://www.ftc.gov/system/files/documents/cases/140508snapchatcmpt.pdf.

[5] Snapchat, "Privacy Terms," 2015, https://www.snapchat.com/privacy.

[6] F. Roesner, B. T. Gill, and T. Kohno, "Sex, lies, or kittens? investigating the use of snapchat's self-destructing messages," in *Proc. of Financial Cryptography and Data Security*, 2014.

[7] T. C. Group, "TPM Main Specification Level 2 Version 1.2, Revision 130," 2006.

[8] GibsonSec, "Snapchat App Encryption Mode Reviewed," 2013, http://gibsonsec.org/snapchat.

[9] A. Caudill, "Snapchat API protocols decoded," 2012, http://adamcaudill.com/2012/06/16/snapchat-api-and-security.

[10] "Cyberdust," https://www.cyberdust.com.

[11] "Confide," https://getconfide.com.

[12] ARM, "ARM Security Technology – Building a Secure System using TrustZone Technology," ARM Technical White Paper, 2009.

[13] "Android Verifying Boot," https://source.android.com/security/verifiedboot/verified-boot.html.

[14] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "MockDroid: trading privacy for application functionality on smartphones," in *Proc. of HotMobile*, 2011.

[15] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proc. of OSDI*, 2010.

[16] M. Conti, V. T. N. Nguyen, and B. Crispo, "CRePE: Context-Related Policy Enforcement for Android," *Information Security*, vol. 6531, pp. 331–345, 2011.

[17] "Android DRM Framework," http://developer.android.com/reference/android/drm/package-summary.html.

[18] M. Ongtang, K. Butler, and P. Mcdaniel, "Porscha: Policy oriented secure content handling in Android," in *Proceedings of ACSAC*, 2010.

[19] K. Kuppusamy, G. Aghila, and R. Senthilraja, "A model for remote access and protection of smartphones using short message service," *arXiv preprint arXiv:1203.3431*, 2012.

[20] S. Guha, K. Tang, and P. Francis, "NOYB: privacy in online social networks," in *WOSP*, 2008.

[21] D. Boneh and R. J. Lipton, "A revocable backup system," in *Proc. of USENIX Security*, 1996.

[22] A. M. Dunn, M. Z. Lee, S. Jana, S. Kim, M. Silberstein, Y. Xu, V. Shmatikov, and E. Witchel, "Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels," in *Proc. of OSDI*, 2012.

[23] K. Onarlioglu, C. Mulliner, W. Robertson, and E. Kirda, "Privexec: Private execution as an operating system service," in *Proc. of IEEE SP*, 2013.