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

António José Monteiro de Oliveira Goulão
antonio.m.goulao@tecnico.ulisboa.pt
Instituto Superior Técnico, Universidade de Lisboa

Abstract

Snapchat is a mobile application that garnered immense popularity due to an innovative photo sharing service that allows the delivery of self-destructing messages. However, Snapchat’s history has shown that building such a service 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 the system and application.

1 Introduction

Although the research community has explored the notion of self-destructing digital messages [20], this concept has become truly widespread by Snapchat [6]. 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 [27].

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 [26]. 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 [37], 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” [47].

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 [44] 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 changed, 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 [22]. 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 very simple to use, requiring just a few lines of code in order to incorporate into the application.

2 Motivation and Goals

2.1 Background on Snapchat

Figure 1 represents Snapchat’s 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.

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 that a new snap is awaiting to be read. 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.

Just like any other mobile application, the Snapchat client relies on the security mechanisms provided by Android: 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. Additionally, Android provides a permission model which it leverages to regulate app access to OS
resources. This model is also used to protect apps’ resources from other app unauthorized accesses.

2.2 Security limitations of Snapchat

Unfortunately, Snapchat has had security limitations that may result in unauthorized access to users’ snaps. Unauthorized access occurs in two cases: (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 constraints 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, for example an external camera, and there is normally loss of quality. 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. A screenshot can be taken, 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. To alleviate this problem, Snapchat leverages an Android function to notify the sender if the receiver has taken a snapshot of the picture. However, this mechanism can easily be thwarted if the receiver disconnects its client from the network before taking the screenshot [25]. This move prevents the receiver’s client from forwarding the notification to the Snapchat server.

L3. Persistent buffers: After downloading a snap from the server, 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 [32]. This flaw allowed for a rogue application to easily access it and create a persistent copy of the snap. The user could also plug the device into a computer over a USB cable and extract the snap from the device using a debugger. Access to snaps was further facilitated by the fact that Snapchat did not delete the snap’s local copy after the time limit expired. On later versions, Snapchat fixed this problem by making sure to keep any snap copies on its private area and in encrypted format. (Ironically, however, it is possible to decrypt the content due to a flaw in the use of encryption.)

L4. Flawed use of encryption: The first version of Snapchat did not use encryption at all. The snaps travelled the network unencrypted and were stored in Snapchat servers in plaintext. 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. Firstly, AES used ECB, an insecure block encryption mode. Secondly, and most critically, the symmetric key for encrypting and decrypting the content is unique—for every piece of content, for every user—and is provisioned directly in the Snapchat’s application binary. By reverse engineering the binary, researchers were able to recover the encryption key, which is now public domain [15]. Recent versions of Snapchat fixed the encryption mode to CBC and use two encryption keys. However, both keys remain hardcoded in the application binary and have been reversed engineered [21]. As a result, Snapchat remains insecure.

L5. Insecure APIs: Whenever a snap is rendered on the receiver’s device, the responsibility for limiting the exposure time of the image is of the Snapchat mobile application. Therefore, a third-party application that can impersonate the authentic Snapchat application and successfully download the snap from the server can easily bypass the client-side protections and store a permanent copy of the snap. To prevent such attacks, Snapchat servers expose a private REST API that includes defense mechanisms against third-party applications. However, in early versions of Snapchat such mechanisms were very fragile. They relied upon obscure security protocols encoded in Snapchat’s mobile application. By reverse engineering the application binary, researchers managed to decode the API protocols [14], allowing for an ecosystem of tools [31] and third-party apps to appear [41]. 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 [46], the fundamental problem remains: the security of the service depends upon the obscurity of the API.

L6. Replay attacks: As described above, 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. As a result, by cleverly interrupting or stalling this process, a recipient can visualize a snap multiple times or for a longer amount of time than permitted. Such attacks work particularly well for videos, which can be replayed on the recipient’s device for an unbounded number of times by disconnecting the device from the network immediately after the video starts playing.
L7. Rooting the device: In order to properly implement visualization restrictions on the recipients’ devices, Snapchat requires that the underlying operating system and hardware platform operate correctly and that the user cannot gain arbitrary control over the software of the device, namely the OS. Such conditions are expected to be found on a typical commodity Android device. However, 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.

2.3 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 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 [4] or Confide [3]. Note, however, that it is not our goal to mitigate potential 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:

- **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.

- **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.

- **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.

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

- **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 operating system’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 [8]. 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. We also rely on the fact that the content of the device’s volatile memory is lost when a device is powered down.

3 Design

We present ShareIff, a sticky policy middleware to support self-destruct messages in Android applications. We start by providing a high-level description of the ShareIff architecture and then dive into its design details.

3.1 Architecture

Figure 2 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 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 be able to decrypt envelopes and visualize them on her device, the application must obtain the public keys stored by ShareIff on her device and share such keys with their friends. Obtaining the public keys is performed 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 (see Figure 1).

3. **Open envelope on the receiver endpoint.** When the application running on the recipient’s device retrieves the envelope, the enclosed message 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.

### 3.2 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 envelope since the envelope is decrypted and rendered exclusively by the OS-protected ShareIff components.

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 familiarized with. Typically, an application’s GUI is arranged into screens that contain UI elements, such as buttons, text panels, etc. In Android, GUI elements and layout arrangements are defined by instances of subclasses of `View`. 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-destructing 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 3. The right hand side of the figure represents an application’s screen comprising: one regular view showing the application name (1), a trustview where a self-destructing message is shown (2), and two views for the buttons “Like” (3) and “Back” (4). Listing 1 sketches how this layout hierarchy can be represented in XML, and Listing 2 provides the Java code to open an envelope inside the trustview instance.
The following sections. The core API primitives are: platform key keypairs: the ShareIff manager in each device and consist of three internal cryptographic keys. Such keys are maintained by tions: close envelopes, open envelopes, and manage in-

As mentioned in Section 3.1, the ShareIff API provides 3.3 Security protocols

envelopes within it. 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.

3.3 Security protocols

As mentioned in Section 3.1, 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 core API primitives are:

CLOSEENV\((M, T, C_{KP}, C_{KS}, [C_{KR}]) \rightarrow E_{[45]} \) : 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’s endpoint through the local invocation of key management primitives. The return is an envelope, which can be of format \(E_4\) or \(E_5\) depending on whether or not \(C_{KR}\) is provided as input.

OPENENV\((E, v) \rightarrow \text{OK} | \text{Fail} \) : 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 time-frame restriction, and returns \(\text{OK}\). Otherwise, the message is not shown and the call aborts.

\[\text{GETPKEY}(\rightarrow C_{KP}) : \text{Returns the certificate of the local platform key } KP\] The certificate must be transmitted to potential senders.

\[\text{NEWSKEY}(C) \rightarrow C_{KS} : \text{Creates a new screening key on the local ShareIff service. The created key has a maximum envelope opening count } C\] Returns a certificate of the screening key. The certificate must be transmitted to potential senders.

\[\text{NEWRKEY}(\rightarrow C_{KR}) : \text{Creates a new receiver key on the local ShareIff service. Returns a certificate of the newly created key KR. This certificate is only required by senders that require strong authentication of the message recipient.}\]

The ultimate goal of these primitives is to generate an envelope \(E\) (returned by \text{CLOSEENV}) that can be securely decrypted and rendered in the receiver’s device (when invoking by \text{OPENENV}). The generation of such an envelope is performed cryptographically and constitutes the main challenge since several security properties must be satisfied. This envelope is encoded as follows:

\[E_S \rightarrow \langle M, T \rangle_{KE}, id_A, hmac(m, KE), \langle \langle KE \rangle_{KE} \rangle_{KP}\]

\[C_{KR} \rightarrow \langle KR^+, \text{ShareIff-KR} \rangle_{KP}\]

Due to space constraints, we omit the description of the envelope’s constituent elements.

3.4 Policy enforcement architecture

Figure 4 gives us an overall perspective of the ShareIff policy enforcement architecture. As can be seen in this figure, when the receiver 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 \text{OPENENV}.

At that point, this service extracts the envelope’s contents, validating its signature and other meta-data 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 account and mitigate data leak channels present in Android’s stock architecture. In step 4,
ShareIffUI addresses one of such vulnerabilities, namely the ability of an attacker to physically access the device using a debugger in order to retrieve the image in question. By disabling the communication between the OS and the Android Device Bridge (ADB) in the System Settings, ShareIffUI ensures that no physical attacker can plug the device to a computer and leverage ADB to either take screenshots (using the screencap command), or use a debugger to access variables that might contain the image’s raw information. ShareIffUI handles the screenshot issue in step 5. We created a flagging mechanism in Global Screenshot that allows ShareIffUI to temporarily disable the screenshot capability for as long as there is a privacy sensitive image being displayed by ShareIff.

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.

5 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 memory, a 768 x 1280 display, and a camera with 8 MP, 3264 x 2448 pix. The device was flashed with a build of 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 is specified so that the image in question cannot be captured via screenshot while on the receiver’s device, and it also features a 10 second timeout, after which the image is deleted from the screen.

The second type of operation encompasses the decryption of the image, as well as its rendering on a trusted image view. Although this second operation can be broken down to these two phases, atomically, it can just be seen as the primitive responsible for rendering an image using our ShareIff approach. For this reason, we decided to compare this second operation with Android’s original way of rendering an image.

Considering that the performance of these operations
Table 1: Experiment image resolutions and file size.

<table>
<thead>
<tr>
<th>Resolutions (MP)</th>
<th>Resolutions (Pixels)</th>
<th>File Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3MP</td>
<td>640x480</td>
<td>200KB</td>
</tr>
<tr>
<td>2MP</td>
<td>1600x1200</td>
<td>1.4MB</td>
</tr>
<tr>
<td>3.1MP</td>
<td>2048x1536</td>
<td>2.6MB</td>
</tr>
<tr>
<td>5MP</td>
<td>2592x1944</td>
<td>4.3MB</td>
</tr>
<tr>
<td>8MP</td>
<td>3264x2448</td>
<td>6.8MB</td>
</tr>
<tr>
<td>12MP</td>
<td>4096x3072</td>
<td>10.9MB</td>
</tr>
</tbody>
</table>

Figure 5: Execution time of envelope creation.

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.

6 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 [11, 54], to application workflow control [35], but also to information flow control [19, 24, 48]. Another group of solutions protect users’ data by refining Android’s permission model. These systems provide solutions such as runtime permission revocation [11, 29], app sub-component permission assignment [51], as well as individual app resource usage constraining, either through OS configuration [36], or through app bytecode rewriting [9, 28, 52]. Other systems can also limit access to private data through security profile specification [17, 45], 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 [1], but depends on device manufacturers’ specific modules, which differ from device to device. Porscha [39] 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’s content policies target solely the protection of SMS, MMS, and email documents, rendering it useless to handle and protect images.

There is also a lot of work related with remote data management. A group of solutions aims at protecting / deleting sensitive data on stolen smartphones, through SMS [2, 7, 30], network [2, 5, 7], or even in the presence of an adversary that removes the device’s SIM card [53]. Another group of solutions improves users’ control over the data they publish in social networks [23, 33, 34, 42]. These systems employ a paradigm first introduced by Boneh and Lipton [12], 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. Curiously, CleanOS [50] uses this same mechanism in order to protect sensitive data on stolen smartphones. This paradigm and also the work introduced by Perlman [40] were later leveraged to provide expiration dates for users’ data. X-pire [10] provides such expiration capabilities, but has a bigger focus on securing both data and corresponding keys in compromised devices, by leveraging a TEE, and also to prevent the duplication of data before its expiration date, by leveraging steganographic techniques. On the other hand, there are
several systems that support the expiration date mechanism, but focusing on decentralized solutions. Vanish [20] leverages P2P distributed hash tables to store key shares, which are responsible for granting access to users’ encrypted data. EphPub [13] follows a similar approach, but saves the keys in the cache of DNS servers. The work from Reimann and Dürmuth [43] derives key shares from different websites instead.

A number of other systems target secure deletion of data once it is no longer needed by the application. Chow et al. [16] present a system aiming at reducing the lifetime of in-memory data. Lacuna [18] runs java applications in custom VMs and focuses on OS buffer erasure to eliminate app execution traces. PrivExec [38] provides private process execution to ensure that writes to the filesystem or swap cannot be recovered during or after application execution. Similarly to PrivExec, there are several other solutions focusing on persistent storage secure deletion in desktop computers [12, 40], and also on the cloud [49].

7 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. Even so, although not suffering severe repercussions, even Snapchat suffered from limitations of current mobile operating systems that compromised its message secure display and deletion principles. 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 TrustedView, that introduces marginal overheads to both system and application.

References


[24] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These Aren’t the Droids You’re Looking For: Retrofitting Android to Protect Data from Imperious Applications. In Proc. of CCS, 2011.


