Secure Storage on Android with Context-Aware Access Control

Faysal Boukayoua, Jorn Lapon, Bart De Decker, Vincent Naessens

Secure storage on Android with context-aware access control

Faysal Boukayoua[1], Jorn Lapon[1], Bart De Decker[2] and Vincent Naessens[1]
firstname.lastname@cs.kuleuven.be

[1] KU Leuven, Dept. of Computer Science, Ghent Technology Campus
[2] KU Leuven, Dept. of Computer Science, iMinds-DistriNet

Abstract. Android devices are increasingly used in corporate settings. Although openness and cost-effectiveness are key factors to opt for the platform, its level of data protection is often inadequate for corporate use. This paper presents a strategy for secure credential and data storage in Android. It is supplemented by a context-aware mechanism that restricts data availability according to predefined policies. Our approach protects stored data better than iOS in case of device theft. Contrary to other Android-based solutions, we do not depend on device brand, hardware specs, price range or platform version. No modifications to the operating system are required. The proposed concepts are validated by a context-aware file management prototype.

Keywords: secure storage, context-aware security, mobile devices, Android, interoperability

1 Introduction

For years mobile devices have mainly been used privately. More recently, their potential has become apparent to enterprises. For instance, a sales representative could retrieve product information on-site to convince prospective clients and, in case of success, sign an agreement. Similarly, home care nurses could use their tablet or smartphone to consult a patient's dietary prescriptions. Additionally, in confined setups like retirement communities, nurses may also be granted controlled access to the locks of serviced flats. The result is a myriad of sensitive data becoming present on these devices, which are at the same time prone to theft and loss. In the end, this increases the risks of data compromise.

Nevertheless, many companies are issuing smartphones or tablets to their employees. The iOS platform is often selected for its built-in security features, offering protection against malware and theft. This is mainly due to strong application vetting and tight hard- and software integration. For instance, data protection is based on a crypto engine with keys fused into the application processor, making offline attacks to retrieve data very hard - even for jailbroken devices. Moreover, iOS supports multiple availability levels for data and credentials (i.e. *data protection classes*). On the other hand, opting for an Android device also offers benefits. A broad range of prices and specifications is available. The hard- and software costs are typically lower than iOS devices. However, regarding data protection, Android does not impose hardware constraints on device crypto, which makes asset protection (i.e. *sensitive data and credentials*) more difficult. Optionally, the file system can be encrypted, using a passcode-derived key. As a result, offline brute force and dictionary attacks remain possible.

Contribution. This paper presents a secure asset storage mechanism for Android-based devices. It is supplemented by a context-aware mechanism that further limits asset exposure. The secure storage mechanism is backed by a secure element, which is readily or optionally available for most Android tablets and smartphones. On-device passcode attacks are made infeasible, as opposed to iOS. The context-aware module provides decision support by automating asset management tasks and only releasing data to external apps, according to predefined policies. Note that no modifications to the operating system are required. This work is validated through the development of a context-aware file-management system.

The rest of this paper is structured as follows. Section 2 presents related work. Our general approach is described in section 3. Thereafter, the secure storage strategy and the context-aware management module are presented in section 4. The prototype is described in section 6. Both components are evaluated in section 7. Finally, conclusions are drawn and future work is suggested in section 8.

2 Related work

Many security-sensitive applications, across mobile platforms, use custom, weak storage mechanisms [7, 10]. Yet platform-provided facilities offer strong crypto as well as MDM integration [1, 2, 4].

Device encryption in Android can optionally be enabled since version 3. Even so, the external storage is never encrypted. From Android 4 onwards, the KeyChain introduces per-app private key storage. This component is still prone to offline attacks on the user's passcode. Version 4.3 further improves the KeyChain by adding hardware backing, although not many devices currently support this. It relies on the ARM TrustZone extensions, an implementation of GlobalPlatform's Trusted Execution Environment (TEE). The same approach is introduced in Windows Phone 8 and iOS 4. The hardware-enforced OS isolation allows the three platforms to partly implement their credential storage as trustworthy code. Passcode attacks are thus throttled, since they are confined to the device. Nevertheless, the number of attempts can be unlimited. Windows Phone and iOS also apply the above approach in how they implement file system encryption, a feature not yet supported by Android.

Android has two types of persistent memory. The internal storage is sandboxed and provides each app with exclusive access to its files. The external storage, on the other hand, is accessible to every application with the corresponding Manifest permission(s). It is used by many popular apps to keep permanent and temporary data and copies of IPC-obtained data [12, 17]. Relying on the user to meticulously manage this data, is not only user-unfriendly, it would likely lead to the increased exposure of sensitive information. This is a key motivator for the context-aware management module in section 5.

In the light of an integrated user experience, Android offers developers numerous IPC capabilities. However, this leads to heightened security risks [8], which are made worse by the BYOD trend in corporate mobile computing. In anticipation, Samsung has been equipping Android with the KNOX extension [3]. It divides the OS into *application containers*, f.i. in a private and a work-related one. IPC and data sharing are only allowed within the same container. This separation is hardware-enforced, also relying on the TrustZone extensions. Moreover, the use of hardware-based certificates is supported, notably CAC cards (used by the US Department of Defense). Apps can access them through a PKCS interface. Other uses include setting up VPN connections and unlocking the screen.

Context-aware resource protection is a well-discussed topic in scientific literature. ConUCON [5] presents an Android-based model for resource usage and access control. It provides both specification and enforcement. However, being merely OS-based, information is left vulnerable in case of device theft or loss.

Saint [13] provides Android apps with mechanisms to restrict their interfaces towards other applications. Its decision making is context-aware. As Saint modifies the OS, it succeeds in providing far-reaching enforcement. Secure storage is not considered.

The approach by Feth and Jung [11] uses TaintDroid's [9] capability to analyse data flows throughout the device. Here as well, the approach is purely OS-based.

Bubbles [16], by Tiwari et al, allows users to assign resources like data, applications, cameras and network connections, to one or more *bubbles.* The authors describe these as representations of different social contexts, separated by digital boundaries. A combination of bubble-assigned resources can only be used within that particular bubble. Administration is user-centric, and therefore less suitable for corporate use.

Riva et al propose a user authentication model based on a contextual learning component [14]. It implements different forms of authentication, each of which is assigned a *confidence level.* Depending on how sensitive a resource is deemed, a higher confidence level is required to access it. Contrary to this work, the focus lies on making authentication more usable, rather than on resource protection.

In a nutshell, we observe that iOS and Windows Phone devices tend to outperform many of their Android counterparts regarding secure storage. Much can be attributed to the device requirements imposed by platform vendors. The choice that organisations are left with, is either to adopt a different platform or to be limited to a specific subset of Android devices. This restriction is stringent, bearing in mind that Android accounts for a 78.9% marketshare (source: Strategy Analytics, 2013 Q4). At the same time, it goes without saying that approaches that modify the platform, are less likely to be adopted. In this work we explicitly aim not to change the OS. From thereon, we explore the data protection level that can be offered.

3 General approach

Prior to introducing the architecture, we list the requirements it must satisfy and the assumptions it is subject to.

3.1 Security requirements

S1 Assets on the mobile device can be managed and selectively disclosed according to specified policies.

S2 Following device theft or loss, computational attacks to retrieve the assets, are infeasible. Physical attacks are more difficult than against the state of the art (see section 2).

3.2 Usability requirements

The approach should not impose an insurmountable burden on the user, f.i. by requiring a long, complex passcode.

3.3 Interoperability requirements

I1 Our solution must be deployable on a device base that is representative of existing Android versions, hardware specifications and price ranges.

I2 No platform modifications are allowed.

I3 Standard platform building blocks are preferred over custom-made components.

3.4 Assumptions

A1 The security controls of the OS are enforced correctly during legitimate use.

A2 The user does not weaken platform security, f.i. by rooting the device.

Given that installation channels other than the trusted repositories are disabled by default, it is reasonable to assume A1. An internal corporate store only contains trusted applications, while Google Play has several mechanisms to fend off malware outbreaks: automated scanning, manual removal from the Store and over-the-air updates. Assumption A2 implies that eligible users are not considered potential adversaries.

3.5 Architecture

To address the Android shortcomings listed in section 2, we propose 2 complementary approaches. First, a context-aware management module provides soft security by managing assets semi-automatically. It thereby relieves the user from this task. In addition, it selectively discloses them to trusted apps under predefined contextual conditions. The second, hard security approach introduces secure storage that is backed by tamper-resistant hardware.

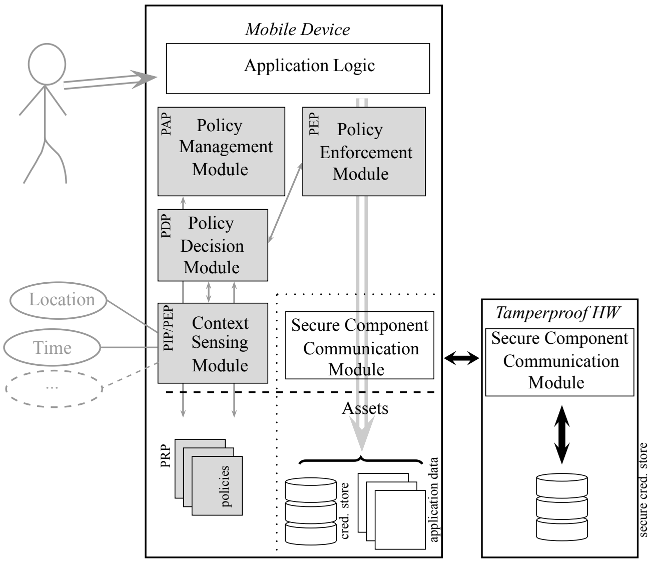

Fig. 1. Overview of the modules for both the soft (grey) and strong (white) security approaches.

The architecture is depicted in figure 1. Five components make up the context-aware module. The *Policy Management Module* allows the user or corporate administrator to manage its policies. A policy event may be triggered when an application tries to access an asset. As a result, the *Policy Enforcement Module* inquires the *Policy Decision Module*. It thus obtains and enforces an access decision, along with actions that must be carried out before or afterwards, if any. The *Context Sensing Module* serves as a policy information point, as it is responsible for retrieving policies and sensory information and making them known to the PDP. The *Policy Retrieval Point* exists in the form of a database on the device, in conjunction with a remote policy server with which it synchronises. The context may change, potentially triggering additional operations. For instance, when a tablet leaves the company premises, certain assets might be erased. In such cases, the Context Sensing Module is also a Policy Enforcement Module.

The second approach improves secure storage on the mobile device. App-controlled encryption is provided, backed by a secure element. The latter ensures that only predefined users and apps can access protected assets: it only exposes cryptographic key material upon successful user and app authentication. The component implementing this –the *Secure Component Communication Module*– resides on both the device and the secure element. Since communication between the app and the secure element takes place over a secure channel, a contactless smartcard can be used without concessions regarding confidentiality and data authentication.

4 Secure asset storage

The first part of our approach is application-controlled secure storage. The important concerns are access control, data authentication and confidentiality. We propose using a secure element for hardware backing. Different types are available: a SIM card, a secure microSD card, a contactless smartcard or an embedded secure element. This makes our solution deployable on nearly every Android device. The tamper-resistance property also provides better security guarantees than solutions based on a Trusted Execution Environment (see section 2).

Cryptographic keys on the secure element can be used to encrypt assets, to authenticate to a server or for digital signatures. For the encryption of large amounts of data, symmetric keys are the method of choice. Different strategies are conceivable. A first one is for all crypto operations to take place on the secure element. This provides the highest level of protection, but is not feasible due to the computational and bandwidth constraints of such tamper-resistant hardware. In a second option, the secure element maintains a key pair for each application, while wrapped symmetric keys are stored on the persistent memory of the mobile device. Upon an app's request, the secure element unwraps and returns these keys. Alternatively, symmetric keys can be stored and retrieved directly. While the latter approach imposes less overhead, the former allows confidential asset sharing on untrusted servers, using a PKI-based approach.

The threats we take into account, are the following.

Eavesdropping. The communication between the app and the secure element cannot necessarily be trusted (e.g. when using a contactless smartcard). To address this, the secure element exposes a –certified– public key, designated to set up a secure channel.

Illegitimate access. Authorised apps have exclusive access to their keys. Before being granted access, two conditions must be fulfilled. First, the user must authenticate using a personal passcode. Since the number of entry attempts is limited, this can safely be a PIN. Second, the app must prove knowledge of a secret that it obtained during an initial pairing phase.

Denial of service against PIN entry. The aforementioned PIN limit precludes brute-force attacks. However, illegitimate applications could perform trials until the allowed attempts are exhausted. To prevent this, the secure element first verifies the application secret and subsequently authenticates the user. Only then, a PIN attempt is counted. The application secret is a 128-bit pseudorandom string, making resilient enough against unlimited trials from unauthorised apps.

4.1 Protocols

This section elaborates on the protocols, used by our approach.

Pairing. In this phase, the application registers to the secure element. Upon success, the latter creates an app-specific credential store. Both parties now also share the same app secret $K_{\mathsf{SE,A}}$. Furthermore, the application obtains the secure element's public key pk_{SE}. The most simple approach is for the user to express his consent using a long PUK code. Alternatively, a third party can mediate between the app and the secure element.

Access to assets. Once the pairing phase is complete, the application can access its credential store upon user consent. This is depicted in protocol 1. First, a session key is established using the secure element's public key [step 1]. The key agreement protocol used, is *dhOneFlow*, from the NIST 800-56A specification [6]. Next, the app requests the user's passcode and transfers it together with its secret $K_{\mathsf{SE,A}}$ (N_{SE} represents a nonce) [steps 3-4]. Subsequently, the secure element verifies $K_{\mathsf{SE,A}}$ and the user's passcode [steps 5-6]. The secure channel is now set up and the app can retrieve its credentials.

Protocol 1 (setupSecureChannel)

```
(1) SE ⇆ A : $K_{sess} \leftarrow \mathsf{authKeyAgreement}(sk_{\mathsf{SE}}; pk_{\mathsf{SE}})$
             % further communication is encrypted with key $K_{sess}$
(2) SE → A : $N_{SE}$
(3) U → A  : $passcode \leftarrow \mathsf{enterPasscode}()$
(4) SE ← A : $h(K_{\mathsf{SE,A}} || N_{SE} + 1), passcode$
(5) SE     : if (not $\mathsf{verify}_{\mathsf{A}}(K_{\mathsf{SE,A}})$) abort
(6) SE     : if (not $\mathsf{verify}_{\mathsf{U}}(passcode)$) abort
```

5 Context-aware asset management

The rationale behind the second part of our approach is to selectively constrain the availability and the presence of assets towards apps. Note that the context-aware component provides soft, *user-assistive security.* It assumes the user is already authenticated, a concern addressed in section 4. If this is not the case, the app's symmetric key is not released by the secure element and the stored assets cannot be decrypted.

Each asset is accompanied by a policy, which specifies the apps that can access it and under what contextual conditions. Apart from access control for apps, actions executed before or after such a request or following the triggering of an event, are also part of the policy. Typical examples are asset download, synchronisation and removal. Metrics that qualify as context include time, location, application white- or blacklists and the presence of nearby wireless networks. The party acting as policy administrator can vary, depending on the application scenario: an individual user, a corporate administrator or a third party service provider. The policies described here, are related to both *access control* and *device self-management.* Prominent languages in these domains are XACML and Ponder2, respectively. Regardless of the overlapping expressiveness in each other's domains, we have opted for XACML. There are advantages to using a standardised policy language: portability across platforms, tool support, extensibility and familiarity among security administrators.

To further illustrate the intended approach, two example policies are demonstrated below. To get around XACML's extensive syntax, they are shown in pseudocode. The first one constrains the availability of door credentials on a home nurse's tablet. Door credential Y is removed if it has not been used for over one hour or if the device is away from the residence of patience X.

```
If location offsite "Residence of patient X"
OR unused > 1hr
Then Remove "DoorCredential Y"
```

In the second policy, access to a contract on a sales representative's tablet, is restricted. The contracting app is only granted read and write access during working hours.

```
If 8:00 <= time <= 18:00
Then App.Contract has R/W-access
to "Contract Z"
```

Lastly, access to privileged code invocation can also be controlled. The exact implementation of the above concepts is platform-specific and is therefore described along with the prototype in section 6.

6 Prototype: context-aware file management

The proof-of-concept implements a corporate file management system, consisting of three parts: a corporate file server, an administration component and a mobile component on a phone or tablet.

6.1 File server

The file server is considered to be legacy infrastructure. State of the art file management solutions offer a myriad of functionality. However, to validate the interoperability of our approach, we limit ourselves to a simple FTP server with confidentiality and mutual authentication over a TLS layer (FTPS). Apache Commons Net 3.2 is used for this purpose.

6.2 Administration component

The MDM server resides on the same machine as the file server. In more complex deployments, they can be hosted separately. The administration component allows to create and modify policies and to push them to the file server and to affected mobile devices. A notification is sent to each of these devices. Consequently, their mobile component connects to the file server and retrieves the policies involved. Push notifications serve to relieve mobile devices from listening to incoming connections. The push service we use, is Google Cloud Messaging.

6.3 Mobile component

The mobile component –a file management app– uses the mechanisms described in section 4, to protect its files.

File synchronisation and secure storage The secure element is a Giesecke & Devrient Mobile Security Card. It runs Java Card 2.2.2 and offers tamper-resistant storage of key material. A symmetric key belonging to the mobile component, is housed in it. This key is created when the pairing phase is successfully completed (see section 4.1). The Mobile Security Card can be easily addressed using the `MSC SmartcardService` from the SEEK4Android project, installable in the same way as any app.

Managed files must be confidential and authentic, while policies are not considered confidential. To fulfill both, AES-GCM (AES in Galois Counter Mode) is used. Not only do authentication and encryption take place in a single step, GCM takes better advantage of parallellism than f.i. the more frequently-used CBC mode. To quantify the performance of our approach, we executed 100 encryptions and decryptions of 10KB, 100KB, 500KB, 1MB, 5MB and 10MB files from the internal app storage to the external storage and vice versa. We compared our results to Android's file system encryption, which uses Linux's `dm-crypt`. To obtain meaningful test results, we switched to AES-CBC, the algorithm used by `dm-crypt`. Table 1 lists the mean values of four setups: unencrypted I/O, Android's file system encryption and two versions of the proposed secure storage mechanism: a Java-based (Bouncy Castle v1.47) and a C-based one (PolarSSL v1.3.2) that is accessed through the Java Native Interface. The tests were run on a Samsung Galaxy Tab 2, with Android 4.1.2. Note that dm-crypt is nearly as fast as having no encryption at all. The Java-based implementation, on the other hand, is prohibitively slow: between 7 and 27 times. This has led us to create a native implementation. A performance gain can clearly be observed. For 10MB, the C implementation with a 20KB buffer encrypts and decrypts only 1.5 and 3 times slower than dm-crypt, which operates at the kernel-level. The C-based implementation can be further optimised by not only executing the cipher operations natively, but the I/O as well.

Confidentiality and authenticity are also a concern when files are in transit between the file server and the mobile component. We address this using a TLS layer with mutual authentication. For prototyping purposes, trust is established by exchanging and trusting public keys offline. In large-scale setups, this is typically realised using a public key infrastructure. The app's private authentication key is stored in the secure element.

Setting up a secure channel between the mobile component and the secure element involves an authenticated key agreement step. 192-bit ECDH (Elliptic Curve Diffie-Hellman) is chosen over RSA, as it is less computationally intensive. This is particularly a point of attention for resource-constrained secure elements. A performance test of 100 runs shows that the average time to set up the secure channel and retrieve the app encryption key is 1667ms. Note that this step only takes place when the app is started: the key remains available as long as the it

Table 1. Secure storage performance: comparative test results (milliseconds)

File size	**10KB**	**100KB**	**500KB**	**1MB**	**5MB**	**10MB**
No encryption						
Outgoing stream	1.17	5.06	24.14	48.56	239.83	638.28
Incoming stream	2.13	17.73	86.23	171.53	871.87	1784.20
Android file system encryption (dm-crypt)						
Outgoing encryption	1.18	5.58	23.95	49.78	250.13	903.48
Incoming decryption	2.61	17.69	95.65	181.22	937.48	1962.45
Java-based AES (Bouncy Castle)						
Outgoing encryption	19.46	140.98	644.34	1309.11	6940.81	13801.72
Incoming decryption	18.78	147.93	688.69	1438.58	7222.27	14607.33
C-based AES through JNI (1K buffer)						
Outgoing encryption	3.84	37.90	194.22	402.09	1977.56	4138.84
Incoming decryption	6.93	55.89	272.75	539.62	2678.56	5333.50
C-based AES through JNI (20K buffer)						
Outgoing encryption	15.50	45.95	143.60	277.94	1384.46	2799.56
Incoming decryption	15.32	45.98	158.44	308.13	1491.25	2957.24

is running. A usability-security tradeoff exists in how frequently a user is asked to enter his PIN.

Our secure storage API consists of two layers. The first one is specific to the storage of cryptographic keys. It enables the developer to interchangeably use different technologies. As a validation case, we created an implementation with the Mobile Security Card as well as with the Android KeyChain. The biggest challenge in the adding the latter, was how to deal with its asynchronous invocation. The second layer encrypts and decrypts the stored items and exposes them to the application. It is realised as an intuitive key-value store.

Contextual sensing, decision and enforcement The Context Sensing Module is built on top of a generic codebase, which can be extended to contain various types of context. The internal policy representation is hierarchical and event-driven. Monitoring change events is necessary, as policy decisions do not only occur in relation to an access request, but also as a result of context change. This proof of concept incorporates the following contextual metrics: location, absolute and recurring time intervals and application black- and whitelists. Multiple implementations of the same contextual variable are supported. For instance, time can be retrieved from the mobile device or through a trusted time server, for more critical uses.

The Context Sensing Module interfaces with a `ContentProvider` that is extended to act as a Policy Decision Module and a Policy Enforcement Module. A `ContentProvider` is a flexible Android building block that offers dynamically assignable and revocable permissions. It can essentially supply any structured data type, including files. Two-way context-aware synchronisation is implemented between local files and their remote counterparts.

To denote the type and id of a requested file, an external app constructs and sends an `Intent` with a local URI to the `ContentProvider`. This allows per-file decisions and enforcement. Note that nothing prevents receiving apps from saving a copy of acquired files. Determining which apps to trust, is seen as part of the administrator's task. This trust preference is specified as an application white- or blacklist in the policy. The beneficial result of this approach is that neither Android nor the requesting app need modification: secure storage and policy enforcement are handled transparently by the `ContentProvider`.

Automatic download, synchronisation and deletion tasks are executed by a subclassed Android `Service`. It also interacts with the Context Sensing Module. The tasks are executed in two different manners: relative to a user's workflow (e.g. deleting a credential one hour after last use), or completely automated (e.g. periodic cleanup of assets not used for more than a given time span).

More advanced functionality can be realised if we allow modifications to external applications. `ContentProvider`'s API provides a `call` function for the invocation of custom methods it exposes. Similarly, RPC interfaces can be offered by extending the above Android `Service` implementation. External applications can bind to it and invoke its exposed set of methods. The `Service` enforces controlled access in much the same way as the `ContentProvider`. This approach allows support for credential-based operations without releasing the secrets (e.g. signing and proof generation).

7 Evaluation

This section evaluates the added value of the context-aware management module (CAM) and the secure storage module (SST). It mainly focuses on the added security that CAM and SST offer in comparison to major mobile platforms. Moreover, we argue that the interoperability and the usability of our approach, positively contribute to its adoption. Table 2 summarises the main benefits.

Table 2. Comparison of security and usability properties of Android, iOS, the context-aware management module (CAM), and the secure storage module (SST).

	Android	**iOS**	**CAM**	**SST**	**SST + CAM**
Security					
-Factors	passcode	passcode, device	n/a	passcode, SE	passcode, SE
-Attack barriers	`PBKDF2`	`PBKDF2` (Δt)	n/a	attempt limit	attempt limit
-Local asset mgmt	app/user	app/user	semi-auto	app/user	semi-auto
-Data revocation	wipe	wipe	n/a	implicit	implicit
Prerequisites	*Internet*	*Internet*		*none*	*none*
Usability					
-Passcode complexity	high	medium	high	low	low
-Hardware reqs?	no	yes: on board	no	yes	yes
-Context mgmt gran.	none	coarse	fine	none	fine

For the security analysis, we assume that the sandboxing mechanism works correctly on both Android and iOS. More specifically: malware without root privileges cannot steal data that is stored in an app's context. We also assume this malware cannot intercept entered passwords. Similar assumptions are taken in [15] and are reasonable if a user does not root or jailbreak his device. However, a skilled adversary can access the –encrypted– file system on a stolen device and launch dictionary and brute-force attacks on the passcode.

Android's file system encryption uses `dm-crypt`. It wraps the file system key with a passcode-derived one using the `PBKDF2` algorithm. A adept attacker can extract the contents of the encrypted file system and perform offline passcode attempts, thus exploiting additional computing power. iOS' file encryption keys are derived from a passcode as well as a hardware-backed secret (i.e. the device `UID`). This precludes offline passcode attacks. If configured by the user or the mobile device management admin (MDM), Android and iOS can automatically wipe themselves after a specified number of failed attempts. However, this circumventable if the device is rooted or jailbroken. As a result, iOS' key derivation function and its hardware backing only slow down passcode attacks (the imposed time delay is denoted by Δt in table 2).

SST surpasses the security offered by both platforms. It is based on a PIN and a tamper-resistant secure element (SE). The attempt limit cannot be circumvented without successfully tampering with the SE. This protection holds, regardless if the device is rooted or not. The remaining possibility is to attack the cryptosystem, a computationally infeasible option. Also note that an Internet connection is needed to initiate a wipe on a default Android or iOS device. The secure element in our approach is blocked after a number of failed PIN attempts, de facto wiping the device. Initiating this requires no network connectivity. Furthermore, unauthorised applications cannot exhaust the PIN entry limit, since an attempt is only counted after successful verification of the app secret. An attacker with physical access to the device, cannot decrypt the data for lack of the PIN. A blocked secure element can be conveniently reactivated by the organisation when the device is back in the hands of the eligible user.

Although CAM is not resistant to physical and rooting attacks, it does minimise the presence of temporary and residual data during its uncompromised use. This means that less sensitive information is harvestable in general. It also provides an effective countermeasure against information-hungry greyware, a significant problem in today's app ecosystem.

Although SST and CAM increase protection level of corporate assets, they do not impose high usability barriers. On the contrary. Passcode complexity is reduced: the attempt limit makes a 4-digit PIN acceptable. Adding to that, asset management is partly automated, which relieves the user from doing this manually.

As for interoperability, the secure element in our solution is –readily or optionally– available in different forms: a SIM card, a secure microSD card, a contactless smartcard or an embedded SE. This makes deployment possible on nearly every Android device. This is in contrast to Android 4.3 and Samsung KNOX, where an organisation would be limited to a handful of devices with hardware support or to Samsung's high-end range, respectively. The operating system is never altered, lowering the adoption barrier even further. If a `ContentProvider`-based approach is taken, as described in section 6, our approach is interoperable with many third-party apps without having to modify them.

8 Conclusions and future work

This paper has presented a secure asset storage strategy for the Android platform. It is backed by a secure element that verifies the eligibility of the user and the app before key material can be accessed. This approach is supplemented by a semi-automated context-aware, management module. It selectively constrains the availability and the presence of assets according to predefined policies. Our approach protects stored data better than iOS in case of device theft and requires no modifications to the operating system. The secure element-based approach ensures that nearly every Android phone can be equipped, contrary to Samsung KNOX or Android 4.3. The proposed concepts have been validated by a context-aware file management prototype.

An interesting extension to this work, is to integrate the secure element in the Android KeyChain, so that API-level support is offered to any application using Android's standard credential storage. Additionally, this would allow the integration of our secure storage strategy into the Device Administration (i.e. MDM) API. A limitation to this approach is that the KeyChain has only been publicly available since Android 4.

Another promising track is trustworthy PIN entry on smartphones and tablets. An increasing number of mobile devices are equipped with a Trusted Execution Environment. Implementing the PIN input as a trusted application, would prevent malware with root access from intercepting it. In addition, trust indicators, such as a blinking LED when the TEE is active, would empower users to appraise the trustworthiness of a PIN input prompt. However, this extension must be traded off against deployability on a more narrow range of devices.

References

1. iOS security. `https://ssl.apple.com/iphone/business/docs/iOS_Security_Oct12.pdf`, October 2012.
2. Android Security Overview - Android Open Source. `https://source.android.com/tech/security/`, May 2013.
3. Samsung Knox. `http://www.samsung.com/global/business/business-images/resource/white-paper/2013/06/Samsung_KNOX_whitepaper_June-0.pdf`, June 2013.
4. Windows Phone 8 Security Guide. `http://go.microsoft.com/fwlink/?LinkId=266838`, September 2013.

5. Guangdong Bai, Liang Gu, Tao Feng, Yao Guo, and Xiangqun Chen. Context-aware usage control for android. In Sushil Jajodia and Jianying Zhou, editors, *Security and Privacy in Communication Networks*, volume 50 of *LNICST*, pages 326–343. Springer Berlin Heidelberg, 2010.
6. Elaine B. Barker, Don Johnson, and Miles E. Smid. NIST SP 800-56A. Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography. Technical report, Gaithersburg, MD, United States, 2013.
7. Andrey Belenko and Dmitry Sklyarov. "secure password managers" and "military-grade encryption" on smartphones: Oh, really? Technical report, Elcomsoft, Amsterdam, March 2012.
8. Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.
9. William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.
10. Sasha Fahl, Marian Harbach, Marten Oltrogge, Thomas Muders, and Matthew Smith. Hey, you, get off of my clipboard. In *Proceedings of the 17th International Conference on Financial Cryptography and Data Security*, Lecture Notes in Computer Science, Okinawa, April 2013. Springer.
11. Denis Feth and Christian Jung. Context-aware, data-driven policy enforcement for smart mobile devices in business environments. In AndreasU. Schmidt, Giovanni Russello, Ioannis Krontiris, and Shiguo Lian, editors, *Security and Privacy in Mobile Information and Communication Systems*, volume 107 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 69–80. Springer Berlin Heidelberg, 2012.
12. Michael J. May and Karthikeyan Bhargavan. Towards unified authorization for android. In *Engineering Secure Software and Systems*, pages 42–57. Springer, 2013.
13. Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. *Security and Communication Networks*, 5(6):658–673, 2012.
14. Oriana Riva, Chuan Qin, Karin Strauss, and Dimitrios Lymberopoulos. Progressive authentication: deciding when to authenticate on mobile phones. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 15–15, Berkeley, CA, USA, 2012. USENIX Association.
15. Peter Teufl, Thomas Zefferer, and Christof Stromberger. Mobile device encryption systems. In *Proceedings of the 28th IFIP TC-11 SEC 2013 International Information Security and Privacy Conference*, page 15, Auckland, New Zealand, July 2013.
16. Mohit Tiwari, Prashanth Mohan, Andrew Osheroff, Hilfi Alkaff, Elaine Shi, Eric Love, Dawn Song, and Krste Asanović. Context-centric security. In *Proceedings of the 7th USENIX conference on Hot Topics in Security*, HotSec'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.
17. Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Malicious android applications in the enterprise: What do they do and how do we fix it? In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 251–254, 2012.