Fides: Unleashing the Full Potential of Remote Attestation

Bernd Prünster[1,2][a], Gerald Palfinger[1,2][b] and Christian Kollmann[3]

[1]*Institute of Applied Information Processing and Communications (IAIK), Graz University of Technology, Austria*
[2]*Secure Information Technology Center – Austria (A-SIT), Austria*
[3]*A-SIT Plus GmbH, Austria*
{*bernd.pruenster, gerald.palfinger*}*@iaik.tugraz.at, christian.kollmann@a-sit.at*

Abstract: In connected mobile app settings, back-ends have no means to reliably verify the integrity of clients. For this reason, services aimed at mobile users employ (unreliable) heuristics to establish trust. We tackle the issue of mobile client trust on the Android platform by harnessing features of current Android devices and show how it is now possible to remotely verify the integrity of mobile client applications at runtime. This makes it possible to perform sensitive operations on devices outside a service operator's control.

We present Fides, which improves the security properties of typical connected applications and foregoes heuristics for determining a device's state such as SafetyNet or root checks. At its core, our work is based on the advancements of Android's key attestation capabilities, which means that it does not impose a performance penalty. Our concept is widely applicable in the real world and does not remain a purely academic thought experiment. We demonstrate this by providing a light-weight, easy-to use library that is freely available as open source software. We have verified that Fides even outperforms the security measures integrated into critical applications like Google Pay.

1 INTRODUCTION

Establishing trust in remote clients is a common challenge. This work targets this issue in the mobile context and presents a solution to remotely establish trust in Android applications—something that has not been reliably possible before and is not adverted even by Google (Mayrhofer et al., 2019). We accomplish this by leveraging the operating system's key attestation features and mapping it to trusted computing concepts. As our main contribution, we demonstrate how Android devices can be considered a trusted environment. This makes it possible to perform sensitive tasks remotely and even outsource cryptographic operations onto devices outside a service operator's control— without compromising security. Consequently, this augments Android's security model with comprehensive trusted computing features. While remote attestation has been possible on Android for some years, its utility was rather limited: It was possible, for example, to verify that cryptographic keys created by an Android application were stored inside a trusted hardware module. However, no reliable statement about the

integrity of the application performing this task and the subsequent attestation could be made. As a consequence, applications performing critical tasks could be secretly modified to include malicious behaviour. Using runtime code injection frameworks like *Xposed*[1], neither simple root checks nor Google's own (rather sophisticated) *SafetyNet* (AOSP, 2018) are able to detect manipulations. In short, relying on such heuristics does not effectively combat changes to the operating system or to mobile applications.

Our contribution to this matter is twofold: We first provide an extensive analysis of Android's key attestation capabilities, discuss their implications for application security and show how this maps to trusted computing concepts. We then present *Fides*, a ready-to-use solution utilising Android's key attestation framework to remotely establish trust in mobile client applications running in unmanaged environments. In contrast to SafetyNet or other typical root checks, our concept does not rely on heuristics but rather provides a definitive statement about whether a device and a mobile application running on it are uncompromised. In other words: For the first time, it is now possible for service operators to verify the state of mobile clients running

---

[a] https://orcid.org/0000-0001-7902-0087
[b] https://orcid.org/0000-0001-6633-858X

[1]https://repo.xposed.info/

in unmanaged environments in a reliable manner. This even rules out modifications made by the legitimate owner of a device. Most importantly, our approach is widely applicable in the real world, ready-to-use in production environments and does not impose a performance penalty. In order to demonstrate the practicability of our approach, we have made Fides's source code freely available on GitHub.[2]

The remainder of this paper is structured as follows: Section 2 provides the background on Android's security model and its hardware-backed keystore implementation. Afterwards, Android's attestation capabilities ate discussed in Section 3. Section 4 then shows how hardware-backed attestation can be used to remotely establish trust in applications running in unmanaged environment and describes how Fides implements this procedure. Having elaborated on our approach, Section 5 discusses how our system differs from related work. Our work concludes with Section 6, providing examples where Fides improves upon existing approaches used in real applications.

2 BACKGROUND

This section provides an overview about the measures Android implements to improve system security, how it separates and vets applications, and which risks remain. In particular, we discuss hardware-based security mechanisms present on Android and their implications for mobile application security.

2.1 Android System Security

As Android has become the dominant mobile operating system, the number of attacks have increased.[3] To protect the operating system from these attacks, a myriad of technologies have been implemented to secure the platform. We refer to 'The Android Platform Security Model' by Mayrhofer et al. (2019) for a full overview of Android's security architecture.

Android uses the concept of sandboxing to prevent applications from accessing each others data–based on features from the Linux kernel–such as assigning a *user ID* (UID) to each application. Since Android 4.3, this separation is enhanced with the help of *mandatory access control* (MAC) in the form of *SELinux* (AOSP, 2019d). However, these mechanisms are all enforced by software and can thus be circumvented. To remedy this situation and keep secrets secure from

a compromised operating system, Android has introduced *Trusty* (AOSP, 2019e), a *trusted execution environment* (TEE). Although the TEE can be used to execute tasks such as managing keys and performing cryptographic operations, it cannot run third-party code.

To protect the system against persistent modifications, Android supports verified boot since version 4.4. Verified boot validates the integrity of each stage in the boot process with the help of trusted hardware (if available on the device). Unless the bootloader of a device has been unlocked to support booting arbitrary system images, this ensures that none of the boot stages have been tampered with.

2.2 Android Keystore Features

The *Android Keystore System* (AOSP, 2019a) offers an interface to application developers to generate and use cryptographic keys—both secret keys and private/public key-pairs. Fig. 1 depicts an overview of the different types of storage back-ends used by the keystore. In case of a software-only backend, the application processor is used to execute cryptographic operations, while keys are stored on the main flash memory. Sabt and Traoré (2016) have shown that no security guarantees hold in practice on software-backed keystores. On devices with trusted hardware, the hardware can assist in securing key material. For this purpose, a TEE is usually used to provide trusted storage which is protected from attacks by both hardware and software, but is usually implemented as part of the application processor. With the advent of Android 9.0, the operating system also supports so-called *StrongBox hardware security modules* (HSMs), which include their own CPU and memory, secure storage, a true random-number generator, and further protections against attacks (AOSP, 2019a). This component must be "certified against the Secure IC Protection Profile BSI-CC-PP-0084-2014 or evaluated by a nationally accredited testing laboratory incorporating High attack potential vulnerability assessment according to the Common Criteria Application of Attack Potential to Smartcards" (Google Inc., 2019, p. 128). In essence, assurance levels of the hardware module conform to those of smart card evaluations and offer comprehensive protection of key material. Google "strongly recommends" future devices to support StrongBox (Google Inc., 2019, p. 128).

In addition to the protection against a compromised operating system, the hardware is able to enforce access control to private (or symmetric) key material. Authorisation to use the key may be bound to the algorithmic parameters (the operation, padding schemes,

---

[2]https://github.com/a-sit-plus/android-attestation-demo
[3]https://www.gdatasoftware.com/blog/2018/11/31255-cyber-attacks-on-android-devices-on-the-rise
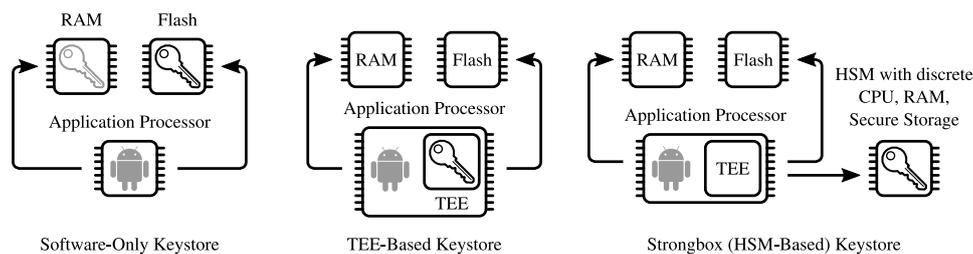
Figure 1: Different keystore back-ends; the key symbol depicts where the secret key is loaded during cryptographic operations.

digests, padding modes), temporal validity, or user authentication. In addition, the system can enforce user authentication on every key access using biometrics (most commonly using a fingerprint), regardless of any recently carried-out authentication procedure. Since Android 7.0, biometric inputs have to be verified inside the TEE or on a chip with a secure channel to the TEE (Google Inc., 2017).

The keystore system binds any key to the app that requested its generation. Therefore, applications can not access each other's keys. To prevent attackers from simply installing an unauthorised system image to circumvent these defences, all cryptographic material becomes inaccessible upon unlocking a devices bootloader. However, as shown by Cooijmans, Ruiter and Poll (2014), an attacker may be able to use keys if they gain root access to the Android system.

Android 7.0 introduced support for version binding of key material (AOSP, 2019c). With this feature, the keys are bound to the version of the operating system and patch level and will become inaccessible if the system is rolled back to an older software version.

2.3 Android Application Security

With the open nature and customisability of Android comes a risk of malicious applications. A plethora of different analysis approaches have been proposed to detect these unwanted applications, utilising concepts such as static and dynamic analysis; we refer to Sadeghi et al. (2017) for an extensive overview on this subject.

One way malware vendors try to circumvent security measures is to repackage benign applications to include malicious code and advertise these apps alongside their original versions. This works because Android allows users to install applications from sources other than Google's *Play Store*. Although each application needs to be signed by its vendor and this signature is checked upon app installation and updates, this only prevents updating an uncompromised application to a repackaged one. Multiple mechanisms to detect and hamper such repackaging attacks have been proposed. Desnos and Gueguen (2011) detect repackaging attempts by evaluating the similarity of applications,

Huang et al. (2013) provide an evaluation of detection mechanisms on obfuscated programs, and Zhou, Zhang and Jiang (2013) and Ren, Chen and Liu (2014) propose different watermarking techniques to improve the detection of repackaged applications in.

With *Google Play Protect*[4], a similar approach is now deployed on the majority of Android devices. Still, these methods are not designed to guard developers against users modifying applications in their devices. For example, Ziegler et al. (2018) have recently shown that a mobile cryptocurrency miner can be modified to produce virtually unlimited tokens by repackaging the mining application to include malicious code.

To prevent such attacks, the application provider needs a reliable attestation mechanism to establish trust in devices running in unmanaged environments. Google aims to provide this attestation through their SafetyNet service using various software and hardware information to validate the integrity of the device and the invoking application (AOSP, 2018). In practice, however, SafetyNet can be circumvented by certain rooting providers, such as *Magisk*[5]. Although Magisk requires an unlocked bootloader, it does not alter the system partition to provide root access. We have verified that patching a device's boot image using Magisk to obtain root access does not trip SafetyNet on a Google Pixel 2 running Android 9.0 with the latest security patches (April 2019 as of this writing). In conjunction with the Xposed framework, this allows for altering the control flow of programs at runtime and, therefore, breaks any assumptions about the integrity of applications.

Summing up, software-based attestation approaches do not provide adequate protection and cannot be relied upon when it comes to outsourcing sensitive data and critical computations to mobile clients. The following section therefore presents an analysis of Android's hardware-based attestation mechanisms and discusses device support.

---

[4]https://www.android.com/play-protect/
[5]https://github.com/topjohnwu/Magisk

3 THE STATE OF ANDROID KEY ATTESTATION

To remedy the shortcomings of software-based attestation approaches, recent Android versions rely on their hardware-backed keystore implementation to provide attestation capabilitie as follows: An application creates a non-exportable public-private key pair using Android's hardware-backed keystore implementation. The keystore API then enables obtaining an attestation result which proves that this key was indeed created using the dedicated cryptographic hardware. On a technical level, this attestation result is implemented as an X.509 certificate extension: A certificate containing the previously generated public key is created and signed inside the cryptographic hardware module using an attestation signing key. We refer to this certificate as the *attestation certificate*. This signing key, as well as a chain of certificates (each signed by Google), are preloaded onto the hardware module during the manufacturing process. Google mandates that at least 100 000 devices need to be provisioned before changing the attestation key—to avoid tracking individual devices (Google Inc., 2019). Furthermore, the root certificate of the chain is published by Google.

3.1 Suitable Devices

All devices shipping with Android 8.0 or later that include a fingerprint reader are required to feature hardware-backed management of cryptographic material using a TEE or HSM and must provide comprehensive remote attestation capabilities. In addition, even some entry-level devices without biometric sensors such as the *Nokia 1*[6] match these specifications. According to a 2018 whitepaper, the FIDO Alliance expects 'almost all coming Android mobile devices (8.0 or later)' (FIDO Alliance, 2018) to support hardware-based key attestation and even attests current Android devices to fulfil all *Level 2 FIDO Authenticator Certification* requirements, arguing that current Android devices can be considered to have effective defences in place against large-scale attacks and operating system compromise. Therefore, we consider such devices robust and secure enough to outsource sensitive computations to.

For the remainder of this paper, we assume a device of this class is used. Android 8.0+ in general has a market share of >38% (AOSP, 2019b), although only some devices that originally shipped with an OS version prior to 8.0 support all the features we require. Without any openly available statistics on the number of suitable devices currently active, we rely on a

conservatively estimated lower bound of 100 million according to Prünster, Fasllija and Mocher (2019).

Before going into detail about the actual attestation workflow, the next section summarises the information contained in an attestation result.

3.2 Attested Information

The information that can be extracted from the attestation certificate provides a comprehensive assessment of device, system, and application integrity. The following attestation properties are relevant for our approach:

- Security level for the attestation; takes a value of either software, TEE, or StrongBox

- Basic cryptographic properties of the attested key itself, such as its public key (in case of a public-private key pair)

- Whether user authentication (and the timeout after a successful authentication) is required for all operations involving the private (or secret) key

- Whether the key is rollback-resistant, meaning that it will become inaccessible upon a system or bootloader downgrade

- Whether the device/bootloader is *locked* and will only accept signed bootloader images

- Boot state according to the Verified Boot feature:
  - *Verified:* Vendor-signed bootloader and system
  - *SelfSigned:* Third-party-signed bootloader and system
  - *Unverified:* Freely modifiable boot chain
  - (*Failed:* never used as part of an attestation result)

- OS version and the security patch level, the patch level of the vendor image (since Android 9.0), and the boot image (since Android 9.0)

- Application information: package name, version, and digest of the package's signature certificate

- The chain of certificates starting with the certificate for the key attestation key, leading up to the published root certificate (see Fig. 2)

- The challenge value provided when requesting an attestation

By evaluating this information in a certain manner, it is possible to establish trust in a remotely deployed application running on an unmanaged Android device, to the point where it can be considered a trusted environment. Although some values of the attestation information are only software-enforced, the following section argues how this aligns with trusted computing concepts such that all values can be trusted.
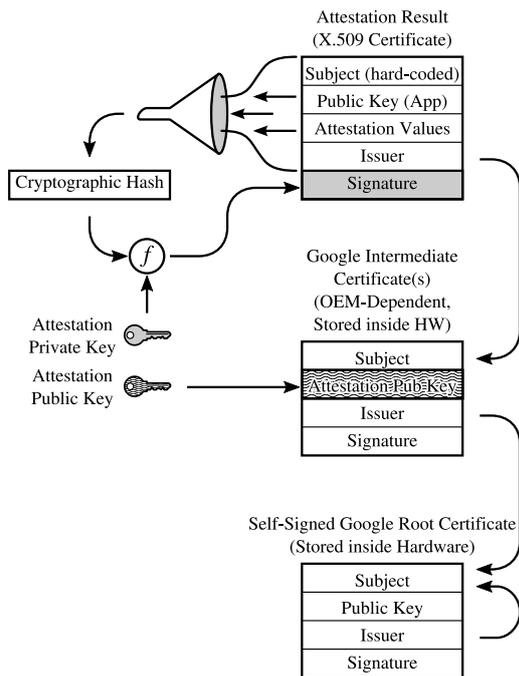

Attestation Result
(X.509 Certificate)

Subject (hard-coded)
Public Key (App)
Attestation Values
Issuer
Signature

Cryptographic Hash

f

Attestation
Private Key

Attestation
Public Key

Google Intermediate
Certificate(s)
(OEM-Dependent,
Stored inside HW)

Subject
Attestation Pub-Key
Issuer
Signature

Self-Signed Google Root Certificate
(Stored inside Hardware)

Subject
Public Key
Issuer
Signature

Figure 2: Attestation chain of trust.

4 Fides – REMOTE ATTESTATION AT WORK

Our solution targets mobile applications that are connected to a back-end—the classic client-server model. The following section introduces such an abstract system model, that is kept as simple as possible for wide applicability. We also introduce our security model and discuss the underlying assumptions. We then illustrate the attestation process that leads to a trusted state based on this model.

4.1 System and Security Model

Up until now we have assumed that an unlocked bootloader is required to modify the OS and/or inject code into otherwise unmodified applications. In reality, however, exploitable vulnerabilities in the OS or bootloader can arise. Still, the correctness of any target platform is typically assumed by developers. We therefore assume the hardware to be trusted and the system to operate as intended, just as any application developer needs to rely on the target platform's functionality. Consequently, we do not consider the requirement to trust hardware manufacturers and the OS vendor a limitation, as this has been a common trust model for decades. We consequently assume Google's root certificate of the attestation chain (see Fig. 2) to be trusted.

We do not, however, trust the user. In fact, we assume a malicious user that is willing and able to unlock their device and employ freely available tools such as Magisk and may even try to repackage the client application for personal gain.

Our system model can be described using a client-server use-case where the client is to be deployed on hardware that falls into the class of suitable devices. We assume the back-end to be controlled and thus trusted by the service operator. We base the remainder of this section on the following scenario:

- A service operator publishes a mobile client application targeting Android 8.0+. This process may or may not happen through Google's Play Store.

- As every Android application is required to be signed by its creator, the service operator signs the client and records the digest of the signature certificate.

- Client-server connections are assumed to mandate *Transport Layer Security* (TLS) using mutual authentication.

The following section retraces the attestation steps performed by modern Android smartphones to illustrate how it is practically possible to verify that an unmodified client application is running on top of a trusted operating system on real hardware.

4.2 Attestation Workflow

The following enumeration explains how an attestation result is constructed based on Fig. 3 and elaborates on the implications of this process.

1. When powering-on an Android device, the trusted hardware module verifies the bootloader signature, and logs whether an unmodified bootloader image will be used to boot the system.

2. The bootloader subsequently verifies the signature of the operating system image. Upon doing so, the bootloader communicates the OS version and patch level to the hardware.

3. The operating system verifies an application's signature upon installation of the application, and records the metadata.

4. The application creates a public-private key pair using the keystore API and requests it to be attested.

5. The operating system passes the software-based values to the hardware, which combines them with the hardware-enforced values to create an attestation result.

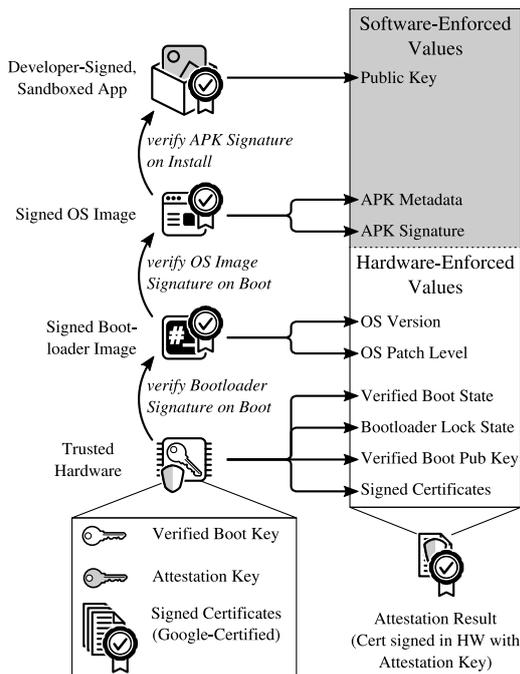6. The hardware signs this attestation result using the attestation key.

Figure 3: High-level structure of an attestation result.

Based on this workflow and the semantics of the attestation values described in Section 3.2, it becomes possible to establish trust in a mobile client application. We illustrate this by applying this knowledge to our system model as follows:

- *Boot State:* On devices with a *locked* bootloader and a *Verified* boot state, we can safely assume that only a vendor-supplied bootloader can be used to subsequently boot only vendor-signed operating system images (cf. Section 3.2). A system booted into a *locked*, *Verified* state is therefore guaranteed to uphold all aspects of the Android security model. This rules out modifications to the base system such as rooting.

- *System Integrity:* Since the system's integrity is verified by the hardware, the system software is itself elevated to a trusted state. Consequently, software-enforced attestation values provided by the operating system can be considered trustworthy.

- *Application Integrity:* By comparing the digest of the client application's signature certificate contained in the attestation result with the digest of the signature certificate value recorded upon signing it, the service operator can indisputably verify the client's integrity. This rules out undetected repackaging of client applications.

If the back-end in our scenario evaluates an attestation result accordingly, the integrity of clients can be reliably determined. As a result, services can choose

to grant only fully-verified clients access and thus exclude compromised devices. This aligns with established trusted computing concepts, specifically trusted hardware bootstrapping a trusted system capable of remote attestation. While it may seem obvious in hindsight, utilising Android's key attestation features like this and the consequences this entails, have never been explored before. Our system's architecture, which does precisely that, is introduced in the following section.

4.3 Architecture

Fides consists of two components: a client-side library and a server-side library. The major advantage of Fides over other solutions is that it can be transparently plugged into existing client-server applications as part of connectivity establishment, also enforcing TLS with mutual authentication. Provisioning of the server-side certificate can be done in whichever way the service operator sees fit. On the client, however, Fides mandates the attestation result to be used as the client certificate to authenticate to the back-end. This effectively ensures that an authenticated channel is established between client and server based on the cryptographic material that has been attested. The following section outlines how Fides executes the previously described attestation process to establish trust in a mobile client.

4.4 Establishing Trust in Mobile Clients

Each time a client connects to the back-end, its level of trust needs to be established. When a client first connects to the back-end, Fides accomplishes this as shown in Fig. 4:

1. The client announces its presence to the back-end over a TLS connection (server-authenticated only).

2. The back-end generates a random challenge.

3. The challenge is sent to the client.

4. The client generates a key, feeds the challenge to the hardware, obtains an attestation certificate, and resets the TLS connection.

5. The client establishes a new TLS session (mutually authenticated) to the back-end, using the newly generated key and the attestation certificate as credentials.

6. The back-end verifies the attestation result, contained in the client's TLS certificate.

7. The back-end matches the attestation result against a policy (see below).

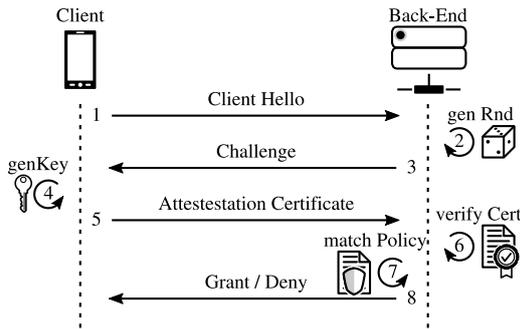8. The back-end informs the client whether access is granted or not.

Figure 4: Initial connection between client and back-end.

Subsequent connections only require steps 5–8 as part of regular connectivity establishment between client and server. Thus, Fides does not require invasive changes to either client or server logic.

A client is considered trustworthy if its attestation result proves that (a) the client application itself has not been modified, (b) that it is running on an unmodified base system (c) whose integrity is verified by a trusted hardware module. In reality however, trust is not binary, but is often modelled in a more granular manner. Fides accomplishes this by providing the possibility to match attestation results against policies, which specify a trust level based on attestation values.

The most important property is the security level of the attestation and keymaster implementation, which takes a value of software, TEE, or HSM. As argued before, a software-based attestation result is not considered trustworthy. An initial level of trust can thus be assigned based on whether a TEE or an HSM is present on a device. Further nuances of the level of trust can be defined by interpreting the version of the operating system and the security patch level.

5   RELATED WORK

Primarily utilised on the desktop, Intel's *Software Guard Extensions* (SGX) have been extensively discussed in literature: Compared to Android's TEE implementation, Intel's SGX can be used to run arbitrary applications in a separate, trusted part of the CPU (inside a so called *secure enclave*). SGX also features remote attestation capabilities, although these require contacting an Intel-run service. Initially, SGX was designed to run only critical parts of an application in the secure environment (Hoekstra et al., 2013). Since then, various efforts have been made to allow unmodified legacy applications to work in the secure enclave. *Haven* (Baumann, Peinado and Hunt, 2014) uses a Windows-based library operating system inside the enclave to provide the functionality needed for full applic-

ation binaries. *SCONE* (Arnautov et al., 2016) is an approach allowing Docker containers to run inside SGX. *Graphene-SGX* (Tsai, Porter and Vij, 2017) is a more modular library OS based on Linux. *PANOPLY* (Shinde et al., 2017) introduces micro-containers, which makes it possible to split up applications into different containers, reducing the size of each unit.

These efforts show that there is a real demand for running complex, or even unmodified applications in a trusted environment, although no unified solution exists. Achieving this goal is notoriously difficult because (compared to modern mobile platforms like Android) no comprehensive security concept that shields a system from modifications is in place. Consequently, significantly more effort is required to reach a state where an application can be executed on a desktop system in a trusted state due to the fundamental design differences and security concepts compared to mobile environments. Our approach, on the other hand, utilises the security concept of modern Android versions to its fullest and thus enables unmodified applications to harness the platform's complete software and hardware feature set. With Android being the dominant mobile end-user platform, Fides covers a wide area of practically relevant applications and is ready to use on a vast amount of consumer devices. Moreover, the certificate-based attestation process makes it possible to re-use any PKI-library of choice, compared to the SGX remote attestation workflow based on elaborate protocols, always mandating a connection to an Intel-run service.

Prünster, Fasllija and Mocher (2019) also work with Android's key attestation to increase the security properties of decentralised peer-to-peer networks. Their approach is tailored to defeat Sybil and eclipse attacks. Nevertheless, the authors also argue that an Android system can be trusted if certain properties are attested by hardware. In effect, their approach reduces Android phones to "not much more than a universally trusted smartcard" (Prünster, Fasllija and Mocher, 2019). We, however, aim to provide a more universal solution to the problem of trusting mobile clients and therefore provide a general extension to the current Android security model that is not bound to a specific system architecture.

6   CONCLUSIONS

This work presented Fides, which solves the issue of reliably establishing trust in Android clients running in unmanaged environments. The proposed scheme provides a real improvement over existing solutions based on heuristics such as SafetyNet or root checks. We have verified that a variety of applications pro-

cessing critical data (such as *PayPal* and even *Google Pay*) do not notice system modifications performed through Magisk (when enabling *Magisk Hide* and repackaging Magisk Manager), while Fides reliably detects an unverified boot chain. As a consequence of the verifiably unaltered system state, software-based security features like *SELinux* can also be expected to be working as intended.

In summary, it is now possible for service operators to establish trust in mobile clients and reliably deny access to compromised instances, even in cases where Google's own protection mechanisms fail to do so. In case exploits are an issue for certain sensitive applications, Fides can be configured to only trust fully patched devices (a property which cannot be spoofed in software, since it is attested by the hardware). It is therefore easily possible to trade off compatibility for increased security. Our freely available libraries can easily be integrated into the connectivity establishment workflows of existing services, as it only relies on TLS.

REFERENCES

AOSP (17th Apr. 2018). *Protecting against Security Threats with SafetyNet*. URL: https://developer.android.com/training/safetynet/ (visited on 11/01/2019).

AOSP (23rd Jan. 2019a). *Android keystore system*. URL: https://developer.android.com/training/articles/keystore (visited on 20/02/2019).

AOSP (7th May 2019b). *Distribution dashboard*. URL: https://developer.android.com/about/dashboards/ (visited on 10/05/2019).

AOSP (2019c). *Android Keystore - Version Binding*. URL: https://source.android.com/security/keystore/version-binding (visited on 20/02/2019).

AOSP (2019d). *Security-Enhanced Linux in Android*. URL: https://source.android.com/security/selinux/ (visited on 19/02/2019).

AOSP (2019e). *Trusty TEE*. URL: https://source.android.com/security/trusty/ (visited on 11/01/2019).

Arnautov, Sergei et al. (2016). 'SCONE: Secure Linux Containers with Intel SGX'. In: *OSDI 2016*. USENIX Association, pp. 689–703.

Baumann, Andrew, Marcus Peinado and Galen C. Hunt (2014). 'Shielding Applications from an Untrusted Cloud with Haven'. In: *OSDI 2014*. USENIX Association, pp. 267–283.

Cooijmans, Tim, Joeri de Ruiter and Erik Poll (2014). 'Analysis of Secure Key Storage Solutions on Android'. In: *Security and Privacy in Smartphones & Mobile Devices – SPSM@CCS*. ACM, pp. 11–20.

Desnos, Anthony and Geoffroy Gueguen (2011). 'Android: From Reversing to Decompilation'. In: *Proc. of Black Hat Abu Dhabi*, pp. 1–24.

FIDO Alliance (June 2018). *Hardware-backed Keystore Authenticators (HKA) on Android 8.0 or Later Mobile Devices*. URL: https://fidoalliance.org/wp-content/uploads/Hardware-backed_Keystore_White_Paper_June2018.pdf (visited on 14/01/2019).

Google Inc. (18th Apr. 2017). *Android 7.0 Compatibility Definition*. URL: https://source.android.com/compatibility/7.0/android-7.0-cdd.pdf (visited on 20/02/2019).

Google Inc. (8th Feb. 2019). *Android 9.0 Compatibility Definition*. URL: https://source.android.com/compatibility/9/android-9-cdd.pdf (visited on 20/02/2019).

Hoekstra, Matthew et al. (2013). 'Using innovative instructions to create trustworthy software solutions'. In: *Workshop on Hardware and Architectural Support for Security and Privacy – HASP*. ACM, p. 11.

Huang, Heqing et al. (2013). 'A Framework for Evaluating Mobile App Repackaging Detection Algorithms'. In: *Trust and Trustworthy Computing*. Ed. by Michael Huth et al. Berlin, Heidelberg: Springer, pp. 169–186.

Mayrhofer, René et al. (2019). 'The Android Platform Security Model'. In: *CoRR* abs/1904.05572. arXiv: 1904.05572. URL: http://arxiv.org/abs/1904.05572.

Prünster, Bernd, Edona Fasllija and Dominik Mocher (July 2019). 'Master of Puppets: Trusting Silicon in the Fight for Practical Security in Fully Decentralised Peer-to-Peer Networks'. In: *16th International Conference on Security and Cryptography*. SciTePress. URL: https://graz.pure.elsevier.com/en/publications/master-of-puppets-trusting-silicon-in-the-fight-for-practical-sec. In press.

Ren, Chuangang, Kai Chen and Peng Liu (2014). 'Droidmarking: Resilient SoftwareWatermarking for Impeding Android Application Repackaging'. In: *29th ACM/IEEE international conference on Automated software engineering*, pp. 635–646.

Sabt, Mohamed and Jacques Traoré (2016). 'Breaking into the keystore: A practical forgery attack against Android keystore'. In: *European Symposium on Research in Computer Security*. Springer, pp. 531–548.

Sadeghi, Alireza et al. (2017). 'A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software'. In: *IEEE Trans. Software Eng.* 43, pp. 492–530.

Shinde, Shweta et al. (2017). 'Panoply: Low-TCB Linux Applications With SGX Enclaves'. In: *Network and Distributed System Security Symposium – NDSS 2017*. The Internet Society.

Tsai, Chia-che, Donald E. Porter and Mona Vij (2017). 'Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX'. In: *USENIX Annual Technical Conference 2017*. USENIX Association, pp. 645–658.

Zhou, Wu, Xinwen Zhang and Xuxian Jiang (2013). 'AppInk: Watermarking Android Apps for Repackaging Deterrence'. In: *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security - ASIA CCS '13*. New York, USA: ACM Press.

Ziegler, Dominik et al. (July 2018). 'Spoof-of-Work: Evaluating Device Authorisation in Mobile Mining Processes'. In: *15th International Conference on Security and Cryptography*. Vol. 2: SECRYPT. Portugal: SciTePress, pp. 380–387.