Symbolic modelling of remote attestation protocols for device and app integrity on Android

Abdulla Aldoseri
axa1170@bham.ac.uk
University of Birmingham, University of Bahrain
United Kingdom

Tom Chothia
t.p.chothia@bham.ac.uk
University of Birmingham
United Kingdom

José Moreira
jose.moreira.sanchez@valory.xyz
Valory AG, Zug
Switzerland

David Oswald
d.f.oswald@bham.ac.uk
University of Birmingham
United Kingdom

ABSTRACT

Ensuring the integrity of a remote app or device is one of the most challenging concerns for the Android ecosystem. Software-based solutions provide limited protection and can usually be circumvented by repacking the mobile app or rooting the device. Newer protocols use trusted hardware to provide stronger remote attestation guarantees, e.g., Google SafetyNet, Samsung Knox (V2 and V3 attestation), and Android Key Attestation. So far, the protocols used by these systems have received relatively little attention. In this paper, we formally model these platforms using the Tamarin Prover and verify their security properties in the symbolic model of cryptography, revealing two vulnerabilities: we found a relay attack against Samsung Knox V2 that allows a malicious app to masquerade as an honest app, and an error in the recommended use case for Android Key Attestation that means that old—possibly out of date—attestations can be replayed. We employed our findings and the modelled platforms to tackle one of the most challenging problems in Android security, namely code protection, proposing and formally modelling a code protection scheme that ensures source code protection for mobile apps using a hardware root of trust.

CCS CONCEPTS

• **Security and privacy** → *Formal security models*; *Security services*; **Formal methods and theory of security**.

KEYWORDS

Remote attestation, Android apps, App integrity, Device integrity, Root detection

ACM Reference Format:
Abdulla Aldoseri, Tom Chothia, José Moreira, and David Oswald. 2023. Symbolic modelling of remote attestation protocols for device and app integrity on Android. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS '23), July 10–14, 2023, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3579856.3582812

1 INTRODUCTION

One of the major issues confronting the mobile development industry today is ensuring the integrity of a mobile app and its device. Failing to achieve these security goals might introduce issues for the app's user and the developer. For instance, sensitive apps (e.g., banking and payment) often check the integrity of their code and the mobile operating device using various techniques to avoid information leakage [35], e.g., on a rooted device. Similarly, in mobile gaming, evading integrity checks may enable cheating. Nguyen Vu et al. analysed over 28,000 apps, including mobile banking apps. They conclude that most root detection techniques are bypassable [35], even though Android provides a variety of dedicated mechanisms to safeguard contents and data, such as Digital Right Management (DRM) and the Android Keystore [23]. Android's architecture implies that mobile apps only run in "normal" userspace (and not e.g., ARM TrustZone), thus easily allowing static and dynamic attacks (e.g., reverse engineering, repacking, and debugging) to circumvent integrity checks [6].

Several researchers have proposed software solutions to preserve the integrity of apps and their host devices [14, 32, 46, 47]. However, these software defences can usually be bypassed [35], creating a cat-and-mouse game. Hence, these measures increase the difficulty of app tampering, rather than stopping it. More recently, Samsung Knox [44] and Google SafetyNet [24] have introduced a hardware root of trust for attestation, promising much stronger app integrity guarantees. However, their security claims remain largely untested.

In this paper, we examine Samsung Knox, versions 2 and 3, Google SafetyNet, and Android Key Attestation, systematise the problem space of app attestation, and evaluate their corresponding methods symbolically. In order to prove (or disprove) the correctness of app/device integrity checks with hardware-backed remote attestation on Android, we formally model and verify them using the Tamarin Prover [13, 33] and its front-end SAPiC [30]. As not all the considered protocols are fully open, in some cases, we performed symbolic verification based on the publicly available documentation or source code samples.

The novelty of our framework is represented in our way of modelling devices and apps. Each device has a secure world that models the cryptographic operations of the platform, app fingerprinting, and measuring its integrity. Only apps installed on the device can access their secure world. Our framework includes two types of apps: "honest" and "arbitrary". Honest apps are explicitly modelled as a Tamarin process and have a fixed, known fingerprint. In contrast, arbitrary apps may be controlled by attackers. These apps can still access the secure world, but importantly cannot control their fingerprint. Therefore, they cannot trick the secure world into attesting them as an honest app.

The framework presented in this work verifies the overall security of attestation platforms. Additionally, it can check general security properties of the attestation platforms (e.g., if there is a way for any app to incorrectly attest using Samsung Knox V2), and check the security properties of particular apps that use an attestation platform (e.g., does the design of an app, which uses Google SafetyNet, keep a particular value secure). Therefore, developers can use this framework to evaluate the security of their apps.

The security assumptions made during the design of remote attestation platforms are often subtle and sometimes not stated. For example, all frameworks implicitly assume that the devices cannot be rooted at runtime via, for instance a bug in the Android kernel. Another important assumption is that unlocking the bootloader of an Android phone will wipe the apps installed on it [4]. We investigate and state these assumptions, and include them into our attacker model.

Our analysis of the platforms reveals that Samsung's Knox V2 attestation fails to satisfy the device integrity security property, allowing an arbitrary app on a tampered device to relay an attestation statement from a second unrooted device and pass this off as its own. While studying the security of Android Key Attestation, we found that the challenge phase is missing in the official recommended practice of the protocol. This does not guarantee the freshness of the attestation. This means that an app can return an arbitrarily old attestation statement to any challenge, which may not accurately reflect the current state of the device. While the only default fresh values in the attestation is system time and the public key, both cannot guarantee the freshness of the statement. This is because the system time is an attacker-controlled value and not part of the secure world, while the key is not a known challenge value by the developer. In summary, our main contributions are:

- We analyse remote attestation protocols and develop an attacker model that captures their assumptions.
- We perform symbolic verification of common remote attestation protocols, namely Samsung Knox attestation V2 and V3, Google Android Key Attestation, Google SafetyNet, using the Tamarin Prover, leading to a framework that can be used by others to check the security properties of apps that use attestation platforms. We show this by modelling attested key exchange protocol and code protection protocol.
- Based on these models, we show that the device integrity check in Knox V2 remote attestation is flawed. We also demonstrate a freshness issue in the recommended use of Android Key Attestation.
- We employ the models to present a real-world case study on code protection for Android apps, which stops attackers that cannot root the device at runtime, without rebooting.

We provide all our open source artefacts, symbolic models and a demo video for running code protection on Android at https://akaldoseri.github.io/modelling_android_ra/

Responsible disclosure. We reported the issue of Samsung attestation V2 to Samsung in August 2020. Samsung confirmed the issue theoretically and stated that it has been fixed in Samsung attestation V3, which we can confirm with our models. Samsung requested a proof-of-concept as part of their reporting process. We were unable to create this because the Knox SDK is only available under a non-disclosure agreement. In a follow-up discussion, Samsung told us that they were deprecating V2 in the latest OS and that only V3 can be used with Android 13 (released August 2022).

We reported the issue in Android Key Attestation to Google in November 2021, who have accepted the issues and responded that they have fixed the Key Attestation documentation, which will be available in a future release. The issue can be tracked on Android public tracker via https://issuetracker.google.com/205589624.

2 BACKGROUND

Android device architecture. Android devices consist of two worlds: the *normal world* (untrusted) and the *secure world* (trusted). The Android OS and mobile apps operate in the normal world. They have access to the majority of the device's resources (e.g., display, storage and sensors). The secure world offers a Trusted Execution Environment (TEE) for Android devices [6]. It provides secure cryptographic operations offered to trusted apps (Trustlets). This enables data sensitive services to operate securely in Android devices such as NFC payments and fingerprint authentication.

Tampering with Android applications. Because mobile apps are considered untrusted entities, they are installed in the normal world (Android OS). Android enforces app signature verification for each app prior to installing it. This policy ensures that an app must be signed by a key to be installable. The app signature is generated by signing the app code base, its resources and the public key certificate of the signing key. Both the signature and the certificate are attached to the app for verification at installation time [3]. The Android package installer performs signature verification and assures the app validity before installing it. It is advantageous to sign apps with the same certificate for data sharing and permissions access (e.g., system apps utilise this feature to access high-level permissions and restrict them from third-party apps).

The signature verification process does not authenticate the apps (*i.e.*, verifying the developer's ownership of the app). Thus it allows stripping the apps' signatures and certificates to change their content, re-sign them, and re-install them on devices. We refer to this practice as "application repacking" or "application tampering". Furthermore, Android stores mobile apps in a public directory on the data partition, making the repacking process a core challenge to application integrity checks and anti-repacking solutions. Because of this, adversaries not only can extract the apps' code, but they can also perform dynamic analysis and debug the apps. In contrast, trusted apps have tamper resistance protection. They can be extracted on a compromised OS, but only signed versions of them by OEM keys are eligible to be installed.

Verified boot and device tampering. The bootloader is locked on the majority of OEM Android device types, e.g., Samsung and Google's Pixel. This prevents users from flashing arbitrary custom bootloader software or executables into device partitions. Nevertheless, OEM vendors can flash verified executables, such as Android OS, that have been signed with OEM keys. The verified boot validates the authenticity and integrity of the flashed components. It operates when the device boots up. It establishes a chain of trust starting with a hardware-protection root of trust and progressing through the bootloader to verify device partitions including boot, system, vendor, and optionally OEM partitions [7].

The latest Android devices support unlocking the bootloader through the `UnlockOEM` system option [4]. This allows flashing executables into them, regardless of their signature, which allows full control over their normal world (Android OS), their data, apps, and resources. We refer to this process as "device tampering", as it voids the integrity of OEM devices. This issue poses a crucial challenge to identify the status of these devices, because tampered devices can behave as non-tampered devices. Therefore, to ensure detecting such a tampering, OEM devices keep track of the status of these devices and their bootloader. For instance, most Google-based devices keep track of the status of their bootloader and verified boot by storing them in tamper-resistant storage in the secure world [8]. Additionally, unlocking/locking the bootloader of devices, completely factory resets them [4]. This prevents adversaries from accessing the devices' data after unlocking/locking their bootloader. Samsung additionally uses a special one-time programmable warranty bit that is set when their devices have been tampered . This disables data-sensitive apps and services [39].

Remote attestation. Remote attestation (RA) is a mechanism that provides an authentic, timely report for a third-party entity (known as a *challenger*) about the validity of the attested platform. [17]. It is based on a typical challenge-response protocol. The protocol begins with a mobile app that requests a measurement report from a trusted entity (e.g., a trustlet). The report includes measurements and information about the device's status and the requester app info, if applicable. In certain implementations, device information is saved in tamper-resistant storage in secure hardware (e.g., Android Key Attestation). The report is signed with a per-device private key and sent to a remote server to be verified for its authenticity and integrity via the requester app. Finally, the server decides to either trust the device and continue to communicate with it or not [24, 41, 43].

On Android, there are several generic and vendor-specific implementations for remote attestation (e.g., Samsung Knox remote attestation [41, 43], Google SafetyNet [24] and Android Key Attestation [8]). Older versions of SafetyNet were software-based. Therefore, they were bypassable in a compromised OS [15, 29]. However, the recent versions rely on a hardware root of trust similar to Samsung Knox and Android Key Attestation.

Modelling with Tamarin and SAPiC. We have used the Tamarin Prover [13, 33] and its front-end SAPiC [30] for modelling. Tamarin is a state-of-the-art tool used to verify the security properties of protocols in the symbolic modelling of cryptography (the Dolev-Yao model [20]). SAPiC allows modelling protocols in (a dialect of) the applied pi-calculus [1], and converts the processes specification into multiset rewrite rules (MSRs) that can be processed by Tamarin. Everything that SAPiC does can be expressed in MSRs in Tamarin. However, SAPiC provides a convenient encoding (e.g., for locks, reliable channels, and state handling). Security properties are expressed as first-order logic formulas, Tamarin then determines if the protocol expressed as MSRs satisfies such properties.

Dolev-Yao adversaries have absolute control over the public channel: they are free to intercept, forward, delay, drop, rearrange the order or modify (via public function symbols) any message in that channel. The public channel represents communication over insecure networks. This is achieved by the SAPiC constructs Out(m) and In(m) for sending and receiving messages to/from the public channel. There are several alternatives to model communications over private channels. We have opted to use the SAPiC construct that allows giving access to Tamarin MSRs directly: $[\ ] \,-\![\ ]\!\rightarrow [\mathrm{SEND}(\mathit{secret})]$ for sending a message and $[\mathrm{SEND}(\mathit{secret})] \,-\![\ ]\!\rightarrow [\ ]$ for receiving it. For further details about Tamarin and the SAPiC syntax, please see appendix B or refer to [13, 30].

3 RELATED WORK

Several solutions have been proposed to ensure the integrity of applications and Android devices. We categorised them into software-based and hardware-based techniques.

Software-based device and app integrity techniques. Berlato and Ceccato performed a survey for anti-debugging and anti-tampering techniques for mobile apps. The techniques presented are mainly software-based, which include emulator detection, dynamic analysis framework detection, and debugger detection [14]. Additional device tampering techniques are introduced in [47]. Other solutions rely on app hardening to hinder tampering, Android Studio uses ProGuard and R8 (in recent Gradle versions) to obfuscate mobile apps by renaming parts of the app's source code. Such an operation complicates the apps modification, making reverse engineering them a difficult task [5]. DexGuard is based on ProGuard with some enhancements which include code encryption [25]. Yuxue Piao et al. find that DexGuard's code encryption is hard-coded within the app [49]. AppIS [46] aims to protect apps against modification by inserting "guards" at random locations throughout the code and checking the hash of certain regions of the apps. Removing the guards one by one and repacking the mobile app is difficult, since the guards' locations are dynamically changed in each run. Finally, OWASP suggests using a combination of different (imperfect) software-based approaches to frustrate adversary attempts to tamper with apps [37].

The major drawbacks of these software based methods are related to the core design issues of Android (app tampering and device tampering), which are discussed in section 2. The proposed solutions are based on querying a fundamentally untrusted host (the Android OS) about its status and assessing the result within the app. However, an untrusted host might be compromised and provide an inaccurate status, e.g., by overriding relevant methods with frameworks such as Xposed, Cydia Substrate, or Frida. Furthermore, as mentioned in section 2, in-app checks may be removed from the app prior to their execution.

Hardware-based device and app integrity techniques. Hardware-based techniques offer more assurance than software based techniques. However, because they rely on attestation, they can only inform developers about the status of devices at the time of the request, which may be altered if the devices are compromised later. Namely, attestation only guarantees that at some point between the attestation request and the verification of the attestation report, the device was in a trustworthy state. In such a circumstance, continuous detection is required. As discussed in section 2, these solutions (*i.e.*, Google SafetyNet, Samsung Knox remote attestation and Android Key Attestation) rely on trusted components in the secure world (trusted apps) to assess the integrity of the devices and apps that they run. Kozyrakis and Census lab analyse the security of SafetyNet (software-based) versions and showed how to bypass its protections [15, 29]. SafetyNet has introduced hardware measurements to overcome these issues. Ibrahim et al. [26] evaluate the misuse of SafetyNet on mobile apps. But the study did not find any weaknesses in SafetyNet's core implementation. Samsung Knox remote attestation V2 and V3 are similar systems that ensure device integrity of Samsung devices and, for V3 only, the apps. Finally, Android Key Attestation is an Android technique that uses the Android Keymaster to attest a hardware key, in addition to providing application and device integrity.

Modelling remote attestation. Fotiadis et al. present a symbolic abstraction for TPM-based remote attestation protocols to verify the integrity of network attached devices (e.g., routers) [22]. Their work only looks at the attestation of devices and does not consider individual apps, or attacker apps running on the same device as honest apps.

De Oliveira Nunes et al. discussed a design and verification for a hybrid (hardware/software) remote attestation protocol for embedded devices named VRASED [36]. They conclude that software-based approaches (e.g., Viper [31] and Pioneer [45]) are not suitable for such a case. Based on that work, de Oliveira Nunes et al. extend VRASED to include additional functionality, e.g., software update. Similarly, for embedded devices, they chose to focus on a more specific problem in remote attestation, which is time-of-check to time-of-use issues [19].

Jacomme et al. [27] present a reporting capability for SAPiC to create and verify a cryptographic report in an isolated execution environment IEE (e.g., Intel SGX, ARM TrustZone). Processes modelling IEEs are given a unique identifier (called a *location*). These processes can produce reports bounded to the named location. Then, an external party can verify that a given report was produced inside a given location. They show the feasibility of their approach through several case studies, including attested computation and the OTP protocol.

Our work uses this reporting mechanism but includes a framework to allow the attestation of apps outside of the trusted computing base, the hardware measurements and rooted devices that allow the attacker to give false information to the trusted hardware. Our framework models the installation of multiple applications in the same device, including both honest app and arbitrary/attacker apps. These extensions make it possible for us to model how leading attestation platforms work and analyse their security.

Unlike our work, Jacomme et al.'s [27] framework is aimed at modelling IEEs where the code runs inside the trusted hardware. Such hardware-based solutions (e.g., Intel SGX, ARM TrustZone) are promising, however, both Intel SGX and ARM TrustZone are not fully accessible to third-party Android applications. VRASED [36] considers the architecture of low level IoT devices (*i.e.*, the MSP430 microcontroller). While, we consider Android architecture without any modification to its system, hardware level services and features.

Code protection. In section 7, we demonstrate the effectiveness of the modelled protocols' framework by solving a real world problem namely app's code protection. The problem is challenging because Android OS lacks protection for apps' source code. Over the years, several studies proposed source code protection solutions. This section discusses them while pointing out their drawbacks.

First, Faruki et al. [21] conducted a survey that illustrates different code protection techniques used by malwares. The techniques are varied like obfuscation, encryption, stenography and packaging using either custom tools or off-the-shelf tools like Progaurd, DexGuard and APKProtect packer. Among the obfuscation techniques is DexPro, which is a byte level obfuscator for Android apps that obfuscates the program control flow by inserting opaque predicates before the return instruction of function calls [50]. Such complexity makes it harder for an attacker to trace protected calls. Kim et al. [28] enhanced Android packers tools Bangcle, Ijiami and Liapp, to protect multidexing apps. They do this by decrypting dex files of the app at runtime, performing a kind of dynamic code loading, which is widely used in packer tools.

The dynamic code loading approach relies on excluding partial parts of the application's source code from the application and safeguarding them in a particular place (e.g., remote server or encrypted locally). Then, retrieving these parts at application runtime. Tanner et al. proposed a repacking protection architecture that verifies the application integrity at runtime and decrypts encrypted bytecode sections of an app using a derived key at runtime [48]. Utilising remote features, Google dynamic delivery is based on dynamic code loading. It allows certain app functionality to be downloaded conditionally or on-demand by splitting a mobile app into a base module and feature modules [2]. The base is the application's core, and it may conditionally request feature modules from Google servers.

None of these solutions change the Android OS architecture guarantees. Android OS only permits mobile applications to operate in the untrusted section of the device, allowing static and dynamic analysis techniques (e.g., reverse engineering, applications repacking and debugging) to uncover their source code [21, 34]. Overall these proposed solutions are software-based similar to the techniques addressed at the beginning of section 3. They share similar limitations as they increase the difficulty of apps' reverse-engineering but do not prevent it. Therefore, in section 7, we propose a hardware-based code protection protocol that overcomes these limitations and provides trust level protection using trusted hardware.

4 AN ATTACKER MODEL FOR REMOTE ATTESTATION PLATFORMS

The security assumptions made during the design of remote attestation platforms are often subtle and sometimes not stated. In this section, we analyse the design of the attestation systems and identify these assumptions. Then, we state our attacker model, which we use in the formal modelling section.

Design assumptions. We considered the following assumptions to model remote attestation protocols. These assumptions are based on the behaviour of Android OS and the remote attestation protocols' public documentation. First, in terms of tampering with devices e.g., rooting, it is assumed that tampering can only occur by unlocking the bootloader when the device is powered off [4]. We are not considering tampering to occur at run time as this could result in leaking the data of a previously locked device. Additionally, changing the bootloader's status (e.g., unlock/lock) will factory reset the device [4]. Compromising a Samsung-based device will set a special Knox warranty bit and trigger appropriate reactions. It will block access to any keys stored in TrustZone and functions that rely on Knox security [39, 40].

Second, we assume that application tampering is feasible, which includes supplying malicious apps and repacking apps. Third, we assume that the hardware root of trust that manages the attestation is secured from all software-based attacks and cannot be broken by hardware-based attacks. Finally, in terms of network communication that involves attested apps, the official documentation states that the protocols should run over a secure channel like HTTPS [24, 44]. However, this will not prevent a local adversary from obtaining messages before sending/receiving them through the network.

We note that these are strong assumptions, which may not always hold. For instance, a vulnerability in the Android kernel would allow rooting a device without restarting, and implementation attacks (e.g., fault injection [16]) may be able to extract keys from the trusted hardware. However, these are the assumptions on which the attestation platforms are built, so we include them in our model.

The effect of extracting keys from the secure world would completely remove any protection an attestation platform offers. Being able to root the device while it is running would let an attacker attest a non-tampered device, and then root the device so, while less powerful, this would still probably lead to a complete compromise of a single instance of an app. Unlocking the device without a factory reset would allow an attacker to learn long term secrets stored on the device, however, a well-designed app that always attested the device and did not store secrets in long term memory might still be secure. In terms of Samsung devices, Samsung tamper detection relies on Real-time Kernel Protection (RKP), which monitors the integrity of the kernel [42]. However, vulnerabilities that trick the kernel into changing its own memory are beyond RKP protection [11]. In such a case, tampering will not be detected and may cause a similar impact to compromising the Android kernel addressed above.

Attacker model. Since these protocols ensure the integrity of devices and applications, we consider a threat model that consists of two adversaries: physical adversary and network adversary. The physical adversary has access to devices and can tamper with them (i.e., unlock their bootloader and root them), tamper with apps (*i.e.*, has access to mobile apps) including installing honest apps or their own repacked apps as addressed in section 2, and communicate with the attestation endpoints (e.g., developers servers). For the network adversary, we consider a Delov-Yao adversary that can intercept messages in the network [20]. Honest apps can be retrieved from secure channels (*i.e.*, representing a trust source like an honest app' developer or an app marketplace). On the other hand, an adversary can intercept network traffic in a tampered device or tampered app before sending it to developers' servers. The adversary aims to bypass either the app integrity check or device integrity or both to continue communicating with the developer server to retrieve sensitive information that should only be obtained by verified, attested devices and apps.

5 MODELLING REMOTE ATTESTATION PLATFORMS

We consider all Android device-specific remote attestation protocols that rely on hardware-level protection namely: Google SafetyNet (Software and hardware), Samsung Knox remote attestation (V2 and V3), and Android Key Attestation. Google's software-based SafetyNet is included to ensure our models can detect known software-based issues.

We chose the Tamarin Prover and its process calculus SAPiC to model the protocols. Its syntax allows us to model protocols as sets of rules and processes. It supports both falsification and unbounded verification of security properties expressed as lemmas [30].

5.1 Modelling SafetyNet and Samsung Knox attestation

Most of the attestation protocols take a similar flow. Five parties are involved in the attestation process, an app, an attestation's client, an attestation's trusted app, a developer server and an attestation server. Figure 1 shows a generic attestation flow that represents Samsung Knox and SafetyNet, while Key Attestation is illustrated in Fig. 2. Their technical details and differences are addressed in their formal models.

The setup phase, highlighted in dotted boxes in Fig. 1, represents the keys setup and the creation of an app that utilises a remote attestation protocol.

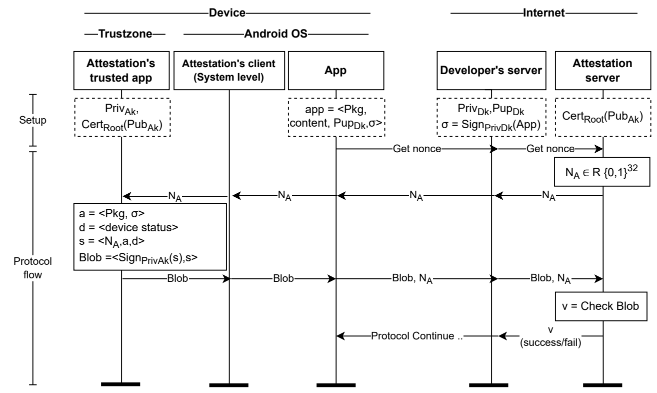

Figure 1: Generic flow of remote attestation protocols for SafetyNet, Knox V2, and Knox V3 based on [24, 41, 43]

Setup phase. Each vendor sets up its private key in the secure world and the attestation server. Samsung's and Google's documentation do not provide much detail on how such a set-up is performed, therefore, we abstract the process based on our best understanding of the available source code and documentation [24, 39, 43], which goes as follows:

Each device contains an attestation key based on a key pair certificate (`PrivAk` and `CertRoot(PubAk)`). The certificate is signed with a root certificate that is available on the attestation server. Hence, data signed by the attestation's trusted app can be verified by the attestation server.

The developer creates an `app`, which has source code and resources, which are denoted by `content`, and package name `Pkg`. The developer signs the application (including it source code and its resources) using its developer key `PrivDk`. The signing process attaches the public certificate of the developer key `PupDk` to the app to create the application signature `o`. The Android package installer can verify the application signature when installing it on the device.

Protocol flow. The protocol runs as follows:

- Mobile apps request a nonce `Na` from the developer server.
- `Na` can either be generated by the developer server or the attestation server. Its purpose is to ensure the freshness of the attestation reports' content (*i.e.*, device integrity and application integrity).
- `Na` is transferred to the app, which passes it to the attestation client on the device (Google Play service in SafetyNet, Attestation agent in Knox). This makes a request to the attestation trusted app in the TrustZone to initiate the attestation.
- The attestation trusted app creates a report `Blob` containing the `Na`, device integrity information `deviceStatus d` and, app integrity information `a` (*i.e.*, package name `Pkg` and the application's signature `o`). Then, it signs them with an attestation key `PrivAk`. `PrivAk` in Samsung is signed by the Samsung root key. The `Blob` in Knox V2 lacks app information.
- The signed `Blob` is sent to the attestation server through the app and the developer server, which verifies its integrity. In Knox V3, the signed `Blob` is sent directly from the attestation client to the attestation server in response to a unique id. The unique id instead is forwarded to the attestation server.
- The attestation server verifies the `Blob`, or the unique id in Knox V3. Then, replies to the developer server with a verdict `V` indicating the verification result, app status and device status.
- Finally, the developer server can verify the signature of the app `o` and the `Na` using the received verdict.

5.2 Modelling Android Key Attestation

Key Attestation attests key pairs in addition to device and app integrity. It relies on certificate chain verification rather than an attestation server for verification. Therefore, the developer undertakes the verification process, and it is crucial to validate the entire certificate chain.

The Key Attestation protocol flow is depicted in Fig. 2, based on Google's documentation [8]. It starts with the key setup phase and app installation. Application installation and its notation are identical to the previous model.

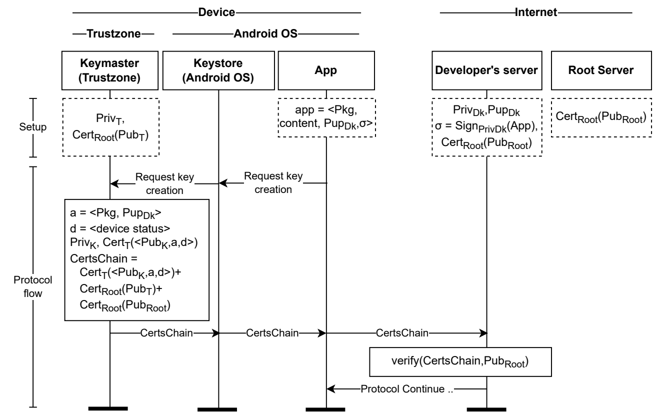
Device
Internet
Trustzone
Android OS
Keymaster (Trustzone)
Keystore (Android OS)
App
Developer's server
Root Server
Setup
$Priv_T$, $Cert_{Root}(Pub_T)$
app = <Pkg, content, Pup_{Dk},σ>
$Priv_{Dk}$,Pup_{Dk} σ = $Sign_{PrivDk}$(App), $Cert_{Root}(Pub_{Root})$
$Cert_{Root}(Pub_{Root})$
Request key creation
Request key creation
Protocol flow
a = <Pkg, Pup_{Dk}>
d = <device status>
$Priv_K$, $Cert_T$(<Pub_K,a,d>)
CertsChain = $Cert_T$(<Pub_K,a,d>)+ $Cert_{Root}(Pub_T)$+ $Cert_{Root}(Pub_{Root})$
CertsChain
CertsChain
CertsChain
verify(CertsChain,Pub_{Root})
Protocol Continue ..

Figure 2: Android Key Attestation protocol based on [8]

Setup phase. Based on the available documentation and analysing exported certificates from Android devices [8, 10], we found that the Keymaster is a trusted app available in the secure world that can generate key certificates. It has a unique per-device certificate key `PrivT` and `CertRoot(PubT)`, which is signed with a Google root certificate `CertRoot(PubRoot)` to form a certificate chain. Any key generated by `PrivT` will have the full certificate chain of the Keymaster and Google root certificate. Thus, ensuring that the key created in the trusted hardware can be publicly verified using the Google root public certificate `CertRoot(PubRoot)`.

Protocol flow. The protocol's flow proceeds as follows:

- The mobile app requests a key pair certificate from the Android Keystore `PrivK,CertT(<PubK,a,d>)`.
- The Android Keystore requests that the Android Keymaster generates a key pair certificate. The certificate is signed by a TEE's key or a Strong box's secure element key (Google Pixel phones only) `PrivT`. The certificate generated in a form of a certificate chain `CertsChain` that includes the certificate itself `CertT(<PubK, a,d>)`, TEE certificate `CertRoot(PrivT)` and Google root certificate `CertRoot(PubRoot)`. Device `d` and app status information `a` is added to the attribute section of the generated certificate. Unlike previous protocols, the app signature is not included in the app status information, but, instead, the app's public certificate is.
- The mobile app can retrieve the chain certificate `CertT(<PubK,a ,d>)` without the private key. The private key never leaves the secure hardware but can be used for cryptographic operations via the Keymaster and the Android Keystore. Then, the app sends the certificate chain `CertsChain` to the developer server for verification.
- Finally, the developer server verifies the certificate chain `CertsChain`. This include verifying `CertT(<PubK,a,d>)` and `CertRoot(PrivT)`, then the root certificate `CertRoot(PubRoot)` using a public version to ensure they are identical. The information about the device `d` and the app `a` can then be extracted from the certificate's attribute section to verify the app certificate and identify the device status to either continue communicating with the app or not.

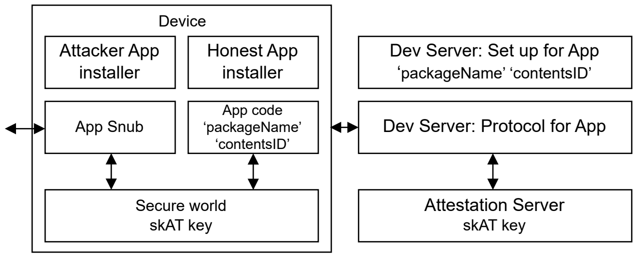
Device
Attacker App installer
Honest App installer
App Snub
App code 'packageName' 'contentsID'
Secure world skAT key
Dev Server: Set up for App 'packageName' 'contentsID'
Dev Server: Protocol for App
Attestation Server skAT key

Figure 3: Overview of modelling framework: each box represents a process in our SAPiC model

5.3 Modelling Android OS and the adversary

We model a minimalist Android OS functionality that includes the app life cycle, and the required cryptographic operations. An overview of our modelling framework is given in Fig. 3. Each box in this figure represents a process in our SAPiC model. The "App code" and "Dev Server: Protocol for App" boxes model a protocol that uses the "Attestation Server" process. The "SecureWorld" process models the trusted hardware on the device, and works with the app installation and "Dev Server Set Up" processes, which ensure the secure world can correctly measure the device and apps. The "Attacker App Installer" and snub allow for arbitrary attacker apps to run on the same device as the honest app. In this section, we describe the app processes that use this framework.

Modelling Android Applications. An Android application consists of a package name, source code, resources and a public key certificate of the app's developer. A package name is an identifier to distinguish an app within a device. The source code and resources shape the content and logic of the app, and we model these in Tamarin as constants (e.g., 'App1PackageName' and 'App1Content' in the example below). The user of our framework must ensure that the same values are used in the Dev Server and app installation process. The app code functionality is modelled as a SAPiC process.

The Dev Server Set Up process signs the package name, contents string and developer certificate with its signing key `advk` to obtain the application signature `appSignature`. This is packed together with the signed components to make the `app`. Then, the app is outputted on a public channel, and we use a SAPiC private fact "App_Published" to allow authentic installation(e.g., via Android market place or an honest developer).

```
let DevServer =
  //creating and publishing the application
  new ~advk; // Developer server signing key
  new ~devId; // Developer Id

  // Signing the app
  let appSignature = sign(~advk,
    ⟨'App1PackageName','App1Content',pk(~advk)⟩) in
  //packing the app
  let app = ⟨'App1PackageName','App1Content',pk(~advk),
    appSignature⟩ in
  event App_Created('App1PackageName',appSignature);
  // Send the app publicly
  out(app);
  // Send the app privately (e.g., to Android marketplace)
  [ ] -[ ]-> [!App_Published(~devId,app)];
```

For installing apps onto a device, we consider two forms: honest installation and arbitrary installation. Honest App installation refers to installing apps that are explicitly modelled in the Tamarin code. This could be, for instance, an app from a known developer that we want to analyse.

The "Device" process creates a new device ID, sets up a secure world process for this device ID and then starts the app installation processes. The device ID is a model-only reference number to distinguish multiple devices within each run. The "Honest App Installtion" process uses the `App_published` channel to receive the app, checks its signature (using pattern matching) and starts the app running.

```
let HonestAppInstalltion =
  [!App_Published(devId,⟨'App1PackageName','App1Content',
  pk(advk), sign(advk, ⟨'App1PackageName','App1Content',
  pk(advk)⟩)⟩)] -[ ]-> [ ];
  !AppCode
```

The "app code" process now has the device ID, package name and contents string of the app, and these will be used when the process calls to the secure world.

Arbitrary installation (Attacker apps): Following our attacker model, we want to allow arbitrary attacker apps to run on the device and make calls to the secure world. However, the secure world on an un-tampered device should still be able to correctly measure these devices (i.e., an attacker app cannot run its functionality and be measured with the same app contents string as an honest app). Therefore, the attacker app installation process lets the attacker pick any package name and signing key, but uses a fresh name to represent the app contents:

```
let ArbitraryAppInstallation =
 in(⟨packagename, devKey⟩);
 new ~content;  //content is different from any honest app
 let appSignature = sign(devKey, ⟨packagename,~content,
   pk(devKey)⟩) in
 let app = ⟨packagename,~content,pk(devKey),appSignature⟩ in
 out(app);
 !AttackerProcess
```

The attacker process receives any input and forwards it to the secure world, and then broadcasts the reply. This allows the attacker to use the secure world in any way they wish.

Modelling network. The protocols' documentation recommends using a secure channel (e.g., TLS) when communicating with developer servers for attestation. MSR facts can be used to model sending messages through a secure channel as follows: [ ] -[ ]-> [Blob_TLS_Ch(*blob*)] for sending the attestation report `blob`, and [Blob_TLS_Ch(*blob*)] -[ ]-> [ ] for receiving the attestation report `blob`. However, with such approach, the network adversary cannot send/receive messages to/from the developer servers through the secure channel. Therefore, we added two processes to allow the adversary to communicate with the developer servers via a secure channel as follows:

```
let Attacker1 = [Nonce_TLS_Ch(devId,nonce)]-[ ]→[Out(nonce)]
let Attacker2 = [In(blob)]-[ ]→[Blob_TLS_Ch(blob)];
```

Here, the adversary can receive a nonce from the developer server over the secure channel and output it to the public channel. Similarly, it can send a blob to the developer server over the secure channel. The attacker utilises `In(blob)` and `Out(nonce)` to receive and send the messages to the public channel. This allows the attacker to craft its own nonce and attestation report `blob`, or use a generated one from a tampered device or a tampered app.

5.4 Security measurement

The secure world measures the status of the device and creates an attestation report `blob`; in Knox, Samsung relies on a warranty bit (a one-time not programmable bit), which indicates the status of the device. The bit is set when tampering is detected by the runtime kernel protection (RKP) and the TrustZone-based Integrity Management Architecture (TIMA), which periodically monitors the kernel and certain device components [11, 42]. SafetyNet uses Compatibility Test Suite (CTS) profile match measurements to detect tampered devices. CTS is a set of unit tests that test the compatibility of various Android classes and components of an Android device including signature checking for public Android APIs [9]. Key Attestation relies on the bootloader status (e.g., locked or unlocked) and the verified boot status that are stored in secure, tamper-resistant, hardware storage[8]. As stated in our attacker model, we assume these approaches work correctly to measure the device and app.

As seen in the following code, we generalise these measured properties and abstract them into two Tamarin terms: `hardwareMeasurement` and `softwareMeasurement`. On one hand, `hardwareMeasurement` represents a hardware-based measurement that is obtained from a trusted source and stored in tamper-resistant storage (e.g., the Knox warranty bit and Bootloader status). On the other hand, `softwareMeasurement` refers to a measurement that is performed in the normal world within the operating system (e.g., software-based SafetyNet), and therefore it can be controlled by the attacker on a tampered device only. Hence, when it is used, we model this as a public input in Tamarin.

The attacker's device rooting attempt is modelled by a public input (`in(status)`) at the start of the secure world process. After this, the secure world will run in either unlocked or locked mode. We do not model the unlocking of a locked device, because unlocking, according to the specification, factory resets the device. Therefore, this is modelled as the creation of a new rooted device. In software-based attestation systems, `softwareMeasurement` will be used for measurement. This value is supplied by an adversary-controlled input `customSoftwareMeasurement` when the device has been tampered with.

To create an attestation report `blob` for Samsung Knox and SafetyNet, we use the extended `report` function for creating cryptographic measurement reports in an isolation execution environment [27].

```
let SecureWorldTA =
( //Did the attacker unlock the bootloader?
 in(status);
 !(
   // Apps call to the secworld
   [SecureWorld_Ch_In(sessionID, deviceId,~swId,nonce
   ,packagename,content,appSignature)]-[ ]→ [ ];
  new ~atId; // attestation id
  event Attesting_App(~atId,packagename,appSignature,deviceId);

  if(status = `unlockBootloader') then
   let hardwareMeasurement = `invalid' in
   //Software measurement is not used in HW attestation
   //in(customSoftwareMeasurement);
   // let softwareMeasurement = customSoftwareMeasurement in

   //On rooted devices nonce, packagename and appSig can be forged:
   in(⟨fnonce,packagename,appSignature⟩)
      //Creating a measurement report
   let report = report(⟨~atId,hardwareMeasurement,fnonce
        ,fpackagename,fappSignature⟩) in
      //Form a blob of measurement report and its signature
      // via the attestation key
      let blob = ⟨~atId, fnonce, hardwareMeasurement,fpackagename
        ,fappSignature, report,sign(report,~skAT)⟩ in
   event DeviceStatus(~atId,deviceId,hardwareMeasurement);
   // return attestation report to the app
   [ ]--[ ]->[ SecureWorld_Ch_Out(sessionID,blob) ]
  else
   let hardwareMeasurement = `valid' in
   let report = report(⟨~atId,nonce,hardwareMeasurement,
       packagename,appSignature⟩) in
   let blob = ⟨~atId,nonce,hardwareMeasurement,packagename,
       appSignature,report,sign(report,~skAT)⟩ in
   event DeviceStatus(~atId,deviceId,hardwareMeasurement);
   // return attestation report to the app
   [ ]--[ ]->[ SecureWorld_Ch_Out(sessionID,blob) ]
 )
 )@⟨'loc',pk(~skAT)⟩ //Indicating a trusted location.
```

This process creates a signed attestation report at a location (TEE) identified by the tuple including the secure world's public key `⟨'loc',pk(~skAT)⟩` using the `report()` function. The report contains `hardwareMeasurement`, `packageName` and `appSignature`. This report can be verified later by the attestation server using the particular secure worlds public key. Note that the technical details on how the TEE cryptographically protects the report are abstracted away in SAPiC. We using the appropriate Tamarin predicates [13], we allow the adversary to create reports in any other untrusted location.

Key Attestation certificates. Key Attestation relies on certificates as attestation reports instead of signed Blobs. Therefore, we modelled an abstraction of the Internet X.509 Public Key Infrastructure Certificate, based on RFC5280 [18]. We modelled `create_certificate/3` to create certificates given their information, their subject's public key and their issuer's private key. `verify_certificate/2` is used to verify certificates for a given subject certificate and an issuer certificate. We modelled the functions `get_public_key_certificate/1`, `get_signature_certificate/1`, `get_tbsInfo_certificate/1` to extract information from certificates, specifically the attribute section to retrieve bootloader status. We symbolically verify the PKI operations separately, considering different real-world scenarios including: self-signed certificates, forged certificate detection, chain certification verification, verification of certificates with extension data, and extraction of certificates' public keys.

5.5 Security properties

Below, we describe the security properties that we require for the protocols, expressed as first-order logic formulas. The syntax $\texttt{Event}(\ldots)@i$ denotes that $\texttt{Event}(\ldots)$ was executed at timepoint i. We assume a scenario where a developer verifies the integrity of an attested app running on a device. We aim to ensure the security properties below:

Verification of device and app integrity for honest apps. This property ensures that the validation of an attestation report by an honest app implies that the attestation was done correctly. The property states that for all 'valid' attestation verdicts (`Verdict_app` event), they have a valid report generated at a trusted execution environment, valid device integrity and valid application integrity. The app must have been created by an honest developer (via `App_Created` event), installed in a device (via `Application_Installed` event) and have been attested by the device, with a valid state (via `DeviceStatus` event).

$$\forall atId, blob, i.\ \texttt{Verdict\_app}(atId, blob, \text{'valid'}, \text{'valid'}, \text{'valid'})@i \Rightarrow$$
$$\exists deviceId, packageName, appSignature, a, b, c.$$
$$\texttt{App\_Created}(packageName, appSignature)@a \land$$
$$\texttt{Application\_Installed}(deviceId, packageName, appSignature)@b$$
$$\land\ \texttt{DeviceStatus}(atId, deviceId, \text{'valid'})@c \land c < i.$$

Satisfying this property ensures that honest apps can perform a successful attestation that yields a valid verdict. However, if this property fails, it means that the attestation report does not prove that the device really is valid.

Attestation report secrecy. This property ensures that the adversary cannot learn/craft an attestation report blob that results in a valid verdict by any method, including tampering apps and devices. It states that for all 'valid' attestation verdicts (`Verdict_app` event), then the adversary KU cannot learn their attestation report blob before it is validated.

$$\forall atId, blob, i.\ \texttt{Verdict\_app}(atId, blob, \text{'valid'}, \text{'valid'}, \text{'valid'})@i \Rightarrow$$
$$\neg(\exists k.\ KU(blob)@k \land k < i).$$

If this property fails and the attacker learns the attestation report before the verdict, then the attacker can use the attestation report to interact with the developer server and, for instance, obtain secret messages that should only be available for valid, attested apps.

Attestation report (blob) uniqueness. This property ensures that each attestation report (blob/certificate) that is accepted as genuine by the developer server is unique.

$$\forall \mathrm{n}, i, j.\ \texttt{BlobAccepted}(\mathrm{n})@i \land \texttt{BlobAccepted}(\mathrm{n})@j \Rightarrow i = j.$$

If this property fails then an attacker could reuse an old attestation report, successfully completing attestation to the development server when the device is not present or has changed its state.

Attestation report (blob) recentness. This property checks that the attestation report (blob/certificate) was generated in response to the development server's request. The development server's request for, and acceptance of, the attestation report are tagged with a `requestID` so that they can be matched. The property states that if an attestation report is accepted by the developer server, then it must have been requested by the developer server before it was created on the device.

$$\forall requestID, blob, i.\ \texttt{RequestedBlobAccepted}(requestID, blob)@i \Rightarrow$$
$$\exists j, k.\ \texttt{BlobRequested}(requestID)@j \land$$
$$\texttt{BlobCreated}(blob)@k \land (j < k) \land (k < i)).$$

Failure of this property would mean that attestation reports were not linked to the request for them, meaning that the developer might accept old, expired or compromised reports.

Reachability properties. To ensure that the model and all the processes finish, we add reachability properties identified with the "`Correctness`" prefix. These properties ensure that all possible branches of the model are executable. For example, the property ensuring that the process modelling an honest app finishes is encoded as:

$$\exists i.\ \texttt{checked}(\text{'honestAppFinished'})@i.$$

Failure of any of these lemmas would indicate an error in the model or the design of the protocol.

6 DISCUSSION AND RESULTS

Our symbolic verification results are illustrated in table 1 and demonstrate that software-based SafetyNet failed to satisfy the device and app integrity properties due to an adversary being able to tamper with the device's software measurement to obtain the attestation report as addressed in [15, 29]. This results in falsifying the attestation report secrecy property as the adversary can obtain the report and attest themselves to obtain secret messages from the developer . Surprisingly, Knox V2 failed to meet the same properties. The model shows traces of an adversary using an arbitrary app to make an attestation and obtain a valid verdict. Because Knox V2 does not include the app contents in the attestation "blob", an attacker (through the public network or on a rooted device) can relay a valid attestation `blob` from another app running on a non-tampered device and have the attestation server validate this. Technically, convincing a developer server that the phone is not rooted, when it in fact is. Because Knox V2 does not attest the app integrity, it cannot distinguish between honest apps and arbitrary apps. As described above, we disclosed this to Samsung who confirmed the issue, and that we were the first to report this.

SafetyNet (hardware-based) and Knox V3 satisfy all the security properties. Both of these ensure device integrity and app integrity. Ensuring only device integrity (as Knox V2 does) or only app integrity (as in SafetyNet (software-based)) will result in compromising integrity.

Our model of Android Key Attestation standard use case satisfies all security properties except the certificate/Blob recentness property. Unlike the other protocols, the developer server does not generate a nonce for the attestation report, but uniqueness of the

Table 1: Satisfied security properties by existing remote attestation protocols.

Protocol	Device and App integrity	Attestation Report Secrecy	Attestation Report Uniqueness	Attestation Report Recentness
Google's SafetyNet (SW)	✗	✗	✓	✓
Google's SafetyNet (HW)	✓	✓	✓	✓
Samsung Knox RA V2	✗	✗	✓	✓
Samsung Knox RA V3	✓	✓	✓	✓
Key Attestation	✓	✓	✓	✗

report is still guaranteed due to freshness of the subject key within the report.

Key Attestation uses the untrusted OS's device clock for certificate creation. We model this by allowing the secure world to receive the time on a public channel, and as this is the only source of freshness for the recommended use case, the attacker can send a time in the future to the secure world and so get a blob to attest to a key it can use much later when it is no longer fresh, and when the key or device may have been compromised.

Therefore, a developer following the standard use case will be unaware of when the key was created and when the attested certificate with its integrity information was created. We reported this issue to Google (as described below). They have stated that they will update their recommended use case for Key Attestation.

Performance overhead. In terms of performance, the Tamarin prover manages to prove all the lemmas in no more than 28 steps for SafetyNet and Knox for each lemma. Key Attestation took longer to prove (a maximum of 36 steps). The modelling of X.509 Public Key Certificate may have been the reason for the extra steps. Knox V2 achieves the shortest runtime of 4 minutes to verify all the lemmas. While Key Attestation was the longest with 7 minutes. The runtime was measured on a laptop with 8 cores intel i7, 16 GB RAM running Ubuntu 22.04.1 LTS.

Mitigation. As a mitigation, Knox V2 requires binding the attested app to the attestation process similar to Knox V3 by including the app package name and its signature in the measurement report. We conclude that device only remote attestation is not sufficient to ensure integrity, as an adversary could always tamper with the app and relay the attestation from a non-tampered device. Samsung confirmed the issue and stated that it was fixed in Knox V3. However, as part of their reporting policy, they asked for a proof of concept to prove the attack was practical. Because Knox licence is not publicly available and requires a specific partner licence agreement, which Samsung would not provide to us, we could not provide them with a proof of concept, therefore it remains an open issue.

For Key Attestation, when we model the protocol including a nonce from the developer server as an attribute in the generated certificate we find that the certificate freshness security property holds. With a nonce in the attestation, the developer can verify when it was created and only accept fresh attestations.

We note that this is not an issue with the Android Key Attestation protocol, rather it is an issue in the guidance given to developers on how to use the `KeyGenParameterSpec.Builder` class for key creation. I.e., unlike the Knox V2 vulnerability, which corresponds to breached security lemma for the Knox V2 system interacting with an arbitrary attacker process. The Key Attestation vulnerability is found when we look at a model of an app, which uses Key Attestation in the recommended way. To overcome the issue, the nonce needs to be set via `setAttestationChallenge` method in Key Attestation as advised by [38]. Google confirmed this issue and said they will address it in the future release of their documentation.

7 CASE STUDY: MODELLING A CODE PROTECTION PROTOCOL

Above, we used basic use cases to evaluate the security properties of remote attestation protocols. In this section, we show how our framework can be used to verify a more complex design that uses attestation.

Our example shows how to use attestation to provide code protection for apps. The difficulty of the problem is due to the design of the application installation life-cycle in the Android operating system (OS). Android does not protect applications' sourcecode. Leaking the source code is feasible via static and dynamic analysis techniques. Previous studies discussed in section 3 proposed several solutions that either need major changes to the Android OS or provide limited protections that may be circumvented using conventional techniques (e.g., repack a mobile application, compromise a device). No code protection solution for mobile applications has been proven to properly prevent code leakage yet, leaving the feasibility of such a goal to be questioned.

Threat model. For this issue, we consider a more aggressive threat model. In this model, we drop the use of secure channels (e.g., TLS) in the proposed protocol. Unlike the previous threat model, messages sent by honest apps and developers are sent through the public channel. This allows a Delov-Yao adversary to intercept, drop, modify and relay messages in the network [20] without relying on relay processes as before. Additionally, the physical adversary has access to devices and apps, and can tamper with them including installing their own repacked apps.

Candidate selection. To overcome the limitation of previously proposed code protection solutions, we rely on hardware remote attestation. Therefore, the suitable candidates based on our findings from section 6 are Knox V3, SafetyNet hardware-based and Key Attestation.

Key Attestation has potential since the attested key can only be used in the generated device. Moreover, having a per device key can identify and authenticate messages between an app and its developer server. This creates a form of continuous integrity check (*i.e.*, as long as the key is valid and in-use, the integrity of their apps and devices are preserved). Such an option is not available by default in Knox V3 and SafetyNet hardware-based, . Secondly, encrypting a secret with an attested key, binds the secret to the hardware root of trust. For all these reasons, we selected the Key Attestation protocol as a candidate for the code protection model.

Modelling code protection. It is difficult to have full protection for the app source code as it may require changes to the Android OS. Therefore, following previous studies, we choose to perform partial code protection in which an application is divided into base code and protected code. The purpose of the base code is to fetch and load protected code at runtime conditionally. This approach has been addressed before as dynamic code loading and used in packer apps [21] locally and remotely by Google Dynamic delivery [2]. The difference in our proposed model is that, unlike previous studies that are based on software-based techniques, we are relying on the hardware root of trust (namely hardware-based attestation) to ensure the integrity of applications and their devices (1). We are considering a more aggressive adversary that have a full control of devices, including installing apps and monitoring the network (2). Finally, we are not considering any custom changes to the Android OS. Our model considers Android OS policy with no changes to any system and hardware level services to ensure that it is applicable for practical implementation (3).

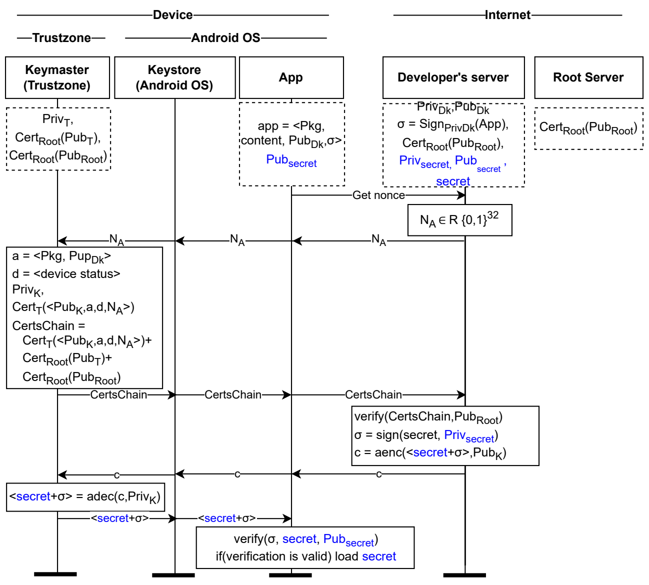
Device
Internet
Trustzone
Android OS
Keymaster (Trustzone)
Keystore (Android OS)
App
Developer's server
Root Server
Priv_T, Cert_Root(Pub_T), Cert_Root(Pub_Root)
app = <Pkg, content, Pub_Dk, σ> Pub_secret
Priv_Dk, Pub_Dk σ = Sign_PrivDk(App), Cert_Root(Pub_Root), Priv_secret, Pub_secret, secret
Cert_Root(Pub_Root)
Get nonce
N_A ∈ R {0,1}^32
N_A
N_A
N_A
a = <Pkg, Pup_Dk>
d = <device status>
Priv_K,
Cert_T(<Pub_K,a,d,N_A>)
CertsChain =
Cert_T(<Pub_K,a,d,N_A>)+
Cert_Root(Pub_T)+
Cert_Root(Pub_Root)
CertsChain
CertsChain
CertsChain
verify(CertsChain,Pub_Root)
σ = sign(secret, Priv_secret)
c = aenc(<secret+σ>,Pub_K)
c
c
c
<secret+σ> = adec(c,Priv_K)
<secret+σ>
<secret+σ>
verify(σ, secret, Pub_secret)
if(verification is valid) load secret

Figure 4: Proposed code protection protocol based on Android Key Attestation

Protocol Flow. Figure 4 illustrates the flow of the proposed code protection protocol. In the setup phase, first, the developers create secret source code. We model it as a fresh value (line 02). Then, the developers need to set up their apps to have a public verification key (`pk(secretSk)`/ `PubSecret`) as part of the application's content (line 07). This key will be used in later steps to verify the authenticity and integrity of the retrieved secret source code.

```
01|let DevServerKA =
02| new ~secret; //generate secret
03| event SecretGenerated(~secret);
// generate secret 's verification key
04| new ~secretSk; out(pk(~secretSk));
// creating and publishing the application
05| new ~advk; // Dev server signing
06| let packagename = `App1PackageName' in
07| let appContent = ⟨'App1Content', pk(~secretSk) ⟩ in
08| let appSignature = sign(
  |           ⟨packagename,appContent,pk(~advk)⟩,~advk)
09| let app = ⟨packagename,appContent,pk(~advk),appSignature⟩ in
10| out(app);
```

- The protocol starts with a request from the app to the developer server to get a nonce, which is passed from the developer server to the app, to the Android Keymaster to initiate the attestation.
- The Keymaster creates a public key certificate `PupK` containing the nonce, app's information (*i.e.*, package name and signature), and device status signed with the Keymaster certificate to form a chain certificate.
- The certificate chain is passed publicly to the developer server that verifies its content and the chain certificate using Google public root certificate. The validation of the app's information is crucial, as it not only attests the application for tampering, but also ensures that the verification key `PubSecret` within the app's content never changes. Also, it ensures the authenticity of the secure world generated key.
- After verifying the certificate and its content, the developer server signs the secret source code with `PrivSecret` to make the signature `σ`. Together, `σ` and the `secret` are encrypted with the attested key `PupK` to make the cipher `c` which is sent out publicly.
- Once an app receives the cipher `c`, it is passed to the Keymaster for decryption. Only the device that owns `PrivK` will be able to decrypt the cipher to obtain both the secret and its signature `σ`.
- Finally, the app verifies the integrity and the authenticity of the `secret` using `PubSecret` and `σ` prior to loading it at runtime.

The cipher `c` can be stored privately in app-storage. While the `s ecret` can be decrypted, loaded at run time and cleared whenever it is not being used. The decryption process can operate locally afterwards. No further attestation is required, because as long as the key is in use, the integrity of the device and the app should be preserved. Rooting the device by unlocking the bootloader should factory reset the device, deleting the generated keys and preventing access to the protected source code.

Security properties. For the security properties, we focus on code protection specific lemmas. We need to ensure that the following three lemmas are satisfied.

- Secret Validity / Correctness: The purpose of this property is to ensure that the loaded secret code must be generated by an honest developer previously. It states that for all the received secrets to a valid device (*non tampered device*) and an honest application, then, this secret must be generated by an honest developer at some time before.

$$\forall secret, sessionID, j, k.$$
$$\texttt{SecretReceivedatDevice}(sessionID, secret, \text{'valid'})@j \land$$
$$\texttt{SecretReceivedAppStatus}(sessionID, secret, \text{'valid'})@k \Rightarrow$$
$$\exists i.\ \texttt{SecretGenerated}(secret)@i \land (i < j) \land (j < k).$$

- Secret secrecy: This property ensures the confidentiality of the secret code against tampered devices, arbitrary attacker apps, and network adversaries. It states that, for all secrets generated by an honest developer, the secrets must not be known or obtained by these adversaries.

$$\forall secret, i.\ \texttt{SecretGenerated}(secret)@i \Rightarrow$$
$$\neg\big((\exists k.\ KU(secret)@k)$$
$$\lor\ (\exists sessionID, k.$$
$$\texttt{SecretReceivedAppStatus}(sessionID, secret, \text{'invalid'})@k)$$
$$\lor\ (\exists sessionID, k.$$
$$\texttt{SecretReceivedatDevice}(sessionID, secret, \text{'invalid'})@k)\big).$$

- Code injection: This property ensures that the code generated by an adversary cannot be loaded by an honest app running on a valid (non-tampered) device.

$$\forall secret, sessionID, i, k.\ \texttt{SecretReceivedatDevice}(sessionID,$$
$$secret, \text{'valid'})@i \land KU(secret)@k \Rightarrow$$
$$\neg(\exists j.\ \texttt{SecretReceivedAppStatus}(sessionID, secret, \text{'valid'})@j).$$

Discussion, results and limitations. The modelled protocol manages to verify all the security properties. Starting with secret validity and secrecy properties, which ensure that no adversary can learn the honest developer's generated secret code. The verification's source code key prevents adversaries-generated code from injecting apps in a non-tampered device. The properties are true even without using a secure channel between the app and the developer server, due to the use of Key Attestation. Considering these findings, leads us to model a variant of this protocol where instead of sending a secret code, the developer can send a secret decryption key to decrypt an encrypted protected code within the app. The variant achieves the same protection level. We implemented this protocol and ran it on a locked Samsung device (A73), loading code that ran 10 sorting algorithms. Running the app 100 times we found that it takes on average 2.74 seconds to complete the whole protocol. We will make the code open source, and it will be available on the website for this paper. In terms of limitations, while the protocol provides protection against strong adversaries that have not been considered by software-based techniques addressed in section 3 and the threat model of remote attestation protocols in section 4, it cannot hold when device tampering happens at runtime (e.g., via exploits in kernel or memory vulnerabilities). Developers can choose to attest only keys of recent Android OS versions or specific device brands. However, this makes the solution less practical. Ultimately, a continuous runtime integrity check is required to overcome this issue.

8 CASE STUDY II: BEYOND THE FRAMEWORK

The previous sections demonstrate several uses for our proposed modelling framework. Here, we show that our framework is general by modelling an existing protocol, namely attested key exchange that is described in [12, 27]. The protocol uses remote attestation to establish a shared key between a user and an IEE. The user sends their public key to the IEE and expects a fresh symmetric key in return, encrypted with the user's public key. We model the protocol between a developer server and the trusted app. We consider the same threat model as the code protection protocol.

The protocol illustrated in figure 5 at appendix A. It starts by creating an honest app that is shipped with its developer's public key. Then, the app starts the Key Attestation protocol, requests a nonce from the developer server, and requests creating a certificate chain from the Android Keystore. Because the Android Keystore does not export symmetric keys, the app will create the symmetric key and request to sign it with the generated certificate chain. Then, the Android Keystore encrypts the certificate chain, the symmetric key and its signature using the developer's public key. Finally, the encrypted content is sent to the developer server over a public channel for decryption and verification.

We model several security properties, including 'verification of device and app integrity' and 'shared key secrecy.' The first property ensures that all the attested, shared keys must be created by honest developers and attested by a valid device. On the other hand, the second property, ensures that an adversary cannot learn/supply a key to the developer that yields a valid verdict. All the security properties are proven successfully with our framework showing that this protocol would be secure when implemented as an Android app.

9 CONCLUSION

In this paper, we performed symbolic verification of several Android remote attestation protocols that ensure app and device integrity. Our modelling framework allows us to check general security properties of the attestation frameworks and also the properties of apps that use these frameworks. We have shown that Samsung Knox V2 attestation fails to ensure the integrity of devices because it fails to ensure app integrity. We conclude that ensuring device integrity alone is not enough; app integrity is required as well. Also, we have shown that the recommended practice of Android Key Attestation misses a challenge phase, which allows an adversary to attest old keys that might be compromised, without the developer's awareness. We have also discussed possible mitigation for both issues. Samsung Knox V2 requires a patch. While Key Attestation requires updating the recommended practice documentation to include a challenge phase. Finally, we presented two case studies to generalise the framework and solve a code protection problem in Android by utilising our findings and models.

ACKNOWLEDGMENTS

This research is partially funded by the NCSC through the TimeTrust project and by EPSRC under grants EP/R012598/1 and EP/V000454/1. The results feed into DsbDtech. Abdulla Aldoseri is supported by a stipend from the University of Bahrain.

REFERENCES

[1] Martín Abadi and Cédric Fournet. 2001. Mobile values, new names, and secure communication. In *ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL)*. ACM, London (UK), 104–115.

[2] Android. 2019. About Dynamic Delivery. https://developer.android.com/studio/projects/dynamic-delivery. (Accessed on 12/01/2019).

[3] Android. 2021. Application Signing | Android Open Source Project. https://source.android.com/security/apksigning. (Accessed on 11/24/2021).

[4] Android. 2021. Locking/Unlocking the Bootloader. https://source.android.com/devices/bootloader/locking_unlocking. (Accessed on 11/25/2021).

[5] Android. 2021. Shrink, obfuscate, and optimize your app. https://developer.android.com/studio/build/shrink-code. (Accessed on 11/26/2021).

[6] Android. 2021. Trusty TEE. https://source.android.com/security/trusty.

[7] Android. 2021. Verified Boot | Android Open Source Project. https://source.android.com/security/verifiedboot. (Accessed on 11/25/2021).

[8] Android. 2021. Verifying hardware-backed key pairs with Key Attestation. https://developer.android.com/training/articles/security-key-attestation.

[9] Android. 2022. Compatibility Test Suite | Android Open Source Project. https://source.android.com/compatibility/cts. (Accessed on 05/28/2022).

[10] Android. 2022. Key and ID Attestation. https://source.android.com/docs/security/keystore/attestation. (Accessed on 09/01/2022).

[11] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 90–102.

[12] Manuel Barbosa, Bernardo Portela, Guillaume Scerri, and Bogdan Warinschi. 2016. Foundations of hardware-based attested computation and application to SGX. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 245–260.

[13] David Basin, Cas Cremers, Jannik Dreier, Simon Meier, Ralf Sasse, and Benedikt Schmidt. 2021. Tamarin Prover (v. 1.6.1). https://tamarin-prover.github.io.

[14] Stefano Berlato and Mariano Ceccato. 2020. A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps. *Journal of Information Security and Applications* 52 (2020), 102463.

[15] CENSUS. 2017. Examining the value of SafetyNet Attestation as an Application Integrity Security Control. https://census-labs.com/news/2017/11/17/examining-the-value-of-safetynet-attestation-as-an-application-integrity-security-control/. (Accessed on 08/15/2019).

[16] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D Garcia. 2021. {VoltPillager}: Hardware-based fault injection attacks against Intel {SGX} Enclaves using the {SVID} voltage scaling interface. In *30th USENIX Security Symposium (USENIX Security 21)*. 699–716.

[17] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. 2011. Principles of remote attestation. *International Journal of Information Security* 10, 2 (2011), 63–81.

[18] R. Housley W. Polk D. Cooper S. Santesson S. Farrell S. Boeyen. 2008. rfc5280. https://datatracker.ietf.org/doc/html/rfc5280. (Accessed on 11/28/2021).

[19] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. 2021. On the TOCTOU problem in remote attestation. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2921–2936.

[20] Danny Dolev and Andrew Yao. 1983. On the security of public key protocols. *IEEE Transactions on information theory* 29, 2 (1983), 198–208.

[21] Parvez Faruki, Hossein Fereidooni, Vijay Laxmi, Mauro Conti, and Manoj Gaur. 2016. Android code protection via obfuscation techniques: past, present and future directions. *arXiv preprint arXiv:1611.10231* (2016).

[22] Georgios Fotiadis, José Moreira, Thanassis Giannetsos, Liqun Chen, Peter B Rønne, Mark D Ryan, and Peter YA Ryan. 2021. Root-of-Trust Abstractions for Symbolic Analysis: Application to Attestation Protocols. In *International Workshop on Security and Trust Management*. Springer, 163–184.

[23] Google. 2019. Android keystore system. https://developer.android.com/training/articles/keystore. (Accessed on 09/16/2019).

[24] Google. 2019. SafetyNet Attestation API. https://developer.android.com/training/safetynet/attestation. (Accessed on 07/23/2019).

[25] Guardsquare nv. 2019. DexGuard: Android obfuscation and runtime-self protection (RASP). https://www.guardsquare.com/en/products/dexguard.

[26] Muhammad Ibrahim, Abdullah Imran, and Antonio Bianchi. 2021. SafetyNOT: on the usage of the SafetyNet attestation API in Android. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. 150–162.

[27] Charlie Jacomme, Steve Kremer, and Guillaume Scerri. 2017. Symbolic models for isolated execution environments. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 530–545.

[28] Nak Young Kim, Jaewoo Shim, Seong-je Cho, Minkyu Park, and Sangchul Han. 2016. Android Application Protection against Static Reverse Engineering based on Multidexing. *J. Internet Serv. Inf. Secur.* 6, 4 (2016), 54–64.

[29] John Kozyrakis. 2015. SafetyNet: Google's tamper detection for Android. https://koz.io/inside-safetynet/. (Accessed on 08/05/2019).

[30] Steve Kremer and Robert Künnemann. 2016. Automated analysis of security protocols with global state. *Journal of Computer Security* 24, 5 (2016), 583–616.

[31] Yanlin Li, Jonathan M McCune, and Adrian Perrig. 2011. VIPER: Verifying the Integrity of PERipherals' Firmware. (2011).

[32] Kyeonghwan Lim, Younsik Jeong, Seong-je Cho, Minkyu Park, and Sangchul Han. 2016. An Android Application Protection Scheme against Dynamic Reverse Engineering Attacks. *JoWUA* 7, 3 (2016), 40–52.

[33] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN prover for the symbolic analysis of security protocols. In *International Conference on Computer Aided Verification (CAV) (LNCS, Vol. 8044)*. Springer, Saint Petersburg, Russia, 696–701.

[34] Gleb Naumovich and Nasir Memon. 2003. Preventing piracy, reverse engineering, and tampering. *computer* 36, 7 (2003), 64–71.

[35] Long Nguyen Vu, Ngoc-Tu Chau, Seongeun Kang, and Souhwan Jung. 2017. Android rooting: An arms race between evasion and detection. *Security and Communication Networks* 2017 (2017).

[36] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. 2019. VRASED: A Verified Hardware/Software Co-Design for Remote Attestation. In *28th USENIX Security Symposium (USENIX Security 19)*. 1429–1446.

[37] OWASP. 2017. Mobile Top 10 2016: M8: Code Tampering. .

[38] Shiv Sahni. 2019. Android Key Attestation. What the heck is Android Key… | by Shiv Sahni | InfoSec Write-ups. https://infosecwriteups.com/android-key-attestation-581da703ac16. (Accessed on 02/13/2022).

[39] Samsung. 2019. Knox: Device Health Attestation. https://docs.samsungknox.com/whitepapers/knox-platform/attestation.htm. (Accessed on 07/31/2019).

[40] Samsung. 2019. Knox: Root of Trust. https://docs.samsungknox.com/whitepapers/knox-platform/hardware-backed-root-of-trust.htm.

[41] Samsung. 2021. Enhanced Attestation (v3). https://docs.samsungknox.com/dev/knox-attestation/about-attestation.htm. (Accessed on 11/23/2021).

[42] Samsung. 2021. Real-time Kernel Protection. https://docs.samsungknox.com/admin/whitepaper/kpe/real-time-kernel-protection.htm.

[43] Samsung. 2021. Tutorial: Attestation (v2). https://docs.samsungknox.com/dev/knox-attestation/tutorial-v2.htm. (Accessed on 11/23/2021).

[44] Samsung. 2022. Knox Attestation. https://docs.samsungknox.com/dev/knox-attestation/about-attestation.htm. (Accessed on 08/05/2022).

[45] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, and Leendert van Doorn Pradeep Khosla. 2005. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. (2005).

[46] Lina Song, Zhanyong Tang, Zhen Li, Xiaoqing Gong, Xiaojiang Chen, Dingyi Fang, and Zheng Wang. 2017. AppIS: protect Android apps against runtime repackaging attacks. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, Shenzhen, China, 25–32.

[47] San-Tsai Sun, Andrea Cuadros, and Konstantin Beznosov. 2015. Android Rooting: Methods, Detection, and Evasion. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices* (Denver, Colorado, USA) *(SPSM '15)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2808117.2808126

[48] Simon Tanner, Ilian Vogels, and Roger Wattenhofer. 2019. Protecting android apps from repackaging using native code. In *International Symposium on Foundations and Practice of Security*. Springer, 189–204.

[49] Jin-Hyuk Jung Yuxue Piao and Jeong Hyun Yi. 2016. Server-based code obfuscation scheme for APK tamper detection. https://onlinelibrary.wiley.com/doi/pdfdirect/10.1002/sec.936. In *Security Comm. Networks*. (Accessed on 11/26/2021).

[50] Beibei Zhao, Zhanyong Tang, Zhen Li, Lina Song, Xiaoqing Gong, Dingyi Fang, Fangyuan Liu, and Zheng Wang. 2017. Dexpro: A bytecode level code protection system for android applications. In *International Symposium on Cyberspace Safety and Security*. Springer, 367–382.

APPENDIX

A ATTESTED KEY EXCHANGE PROTOCOL

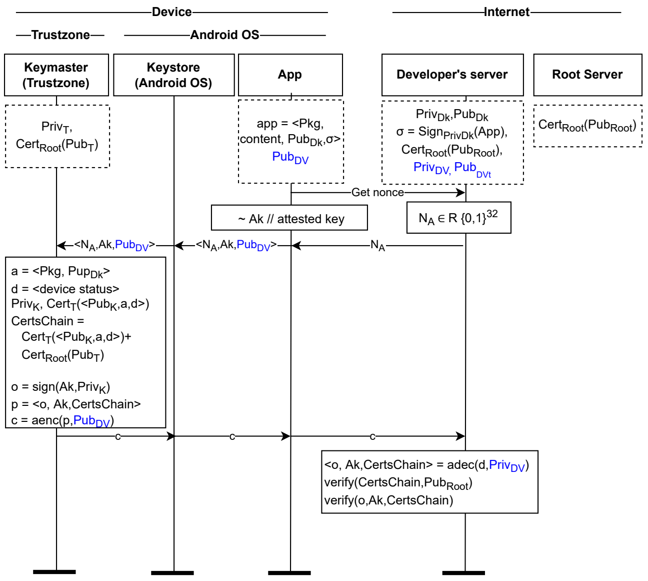
Device
Internet
Trustzone
Android OS
Keymaster (Trustzone)
Keystore (Android OS)
App
Developer's server
Root Server
$Priv_T$, $Cert_{Root}(Pub_T)$
app = <Pkg, content, Pub_{Dk},σ> Pub_{DV}
$Priv_{Dk}$,Pub_{Dk} σ = $Sign_{PrivDk}$(App), $Cert_{Root}(Pub_{Root})$, $Priv_{DV}$, Pub_{DVt}
$Cert_{Root}(Pub_{Root})$
Get nonce
~ Ak // attested key
$N_A \in R \{0,1\}^{32}$
<N_A,Ak,Pub_{DV}>
<N_A,Ak,Pub_{DV}>
N_A
a = <Pkg, Pup_{Dk}>
d = <device status>
$Priv_K$, $Cert_T$(<Pub_K,a,d>)
CertsChain = $Cert_T$(<Pub_K,a,d>)+ $Cert_{Root}(Pub_T)$
o = sign(Ak,$Priv_K$)
p = <o, Ak,CertsChain>
c = aenc(p,Pub_{DV})
c
c
c
<o, Ak,CertsChain> = adec(d,$Priv_{DV}$)
verify(CertsChain,Pub_{Root})
verify(o,Ak,CertsChain)

Figure 5: Modelling attested key exchange protocol from [12, 27] using the proposed remote attestation framework

B SAPIC SYNTAX

Fig. 6 describes the SAPiC syntax. The syntax allows to define a protocol as a process. It is then translated into a set of Tamarin MSRs that adhere to the semantics of the calculus, which is a dialect of the applied pi-calculus [1]. The calculus comprises an *order-sorted term algebra* with infinite sets of publicly known names PN, freshly generated names FN, and variables V. It also comprises a signature Σ, i.e., a set of function symbols, each with an arity. Messages are elements from a set of terms T over PN, FN, and V, built by applying the function symbols in Σ. Events in SAPiC are similar to Tamarin action facts, and they annotate specific parts of the process to be used to define security properties. We denote F the set of Tamarin action and state facts. As opposed to the applied pi-calculus [1], SAPiC's input construct $\mathtt{in}(M, N);\ P$ performs pattern matching instead of variable binding. See [13, 30] for the complete details.

$\langle M, N\rangle ::= x, y, z \in V$	variables
$\mid p \in PN$	public names
$\mid n \in FN$	fresh names
$\mid f(M_1, \ldots, M_k)$ s.t. $f \in \Sigma$ of arity k	function application
$\langle P, Q\rangle ::=$	processes
$\mid 0$	terminal (null) process
$\mid P \mid Q$	parallel execution of processes P and Q
$\mid\ !P$	replication of process P
$\mid$ new $\sim n; P$	binds n to a new fresh value in process P
$\mid$ out$(M, N); P$	outputs message N to channel M
$\mid$ in$(M, N); P$	inputs message N to channel M
$\mid$ if $Pred$ then P [else Q]	P if predicate $Pred$ holds; [else Q]
$\mid$ event $F; P$	executes event (action fact) F
$\mid P + Q$	non-deterministic choice
$\mid$ insert $M, N; P$	inserts N at memory cell M
$\mid$ delete $M; P$	deletes content of memory cell M
$\mid$ lookup M as x in P [else Q]	if M exists, bind it to x in P; else Q
$\mid$ lock $M; P$	gain exclusive access to cell M
$\mid$ unlock $M; P$	waive exclusive access to cell M
$\mid [L] -[A]\!\rightarrow [R]; P \quad (L, R, A \in F^*)$	provides access to Tamarin MSRs

Notation: $n \in FN, x \in V, M, N \in T, F \in F$.

Figure 6: SAPiC syntax.