Hey, NSA: Stay Away from my Market! Future Proofing App Markets against Powerful Attackers

Sascha Fahl
USECAP
FKIE, Fraunhofer
Bonn, Germany
sascha.fahl@fkie.fraunhofer.de

Sergej Dechand
USECAP
University of Bonn
Bonn, Germany
dechand@cs.uni-bonn.de

Henning Perl
USECAP
FKIE, Fraunhofer
Bonn, Germany
henning.perl@fkie.fraunhofer.de

Felix Fischer
USECAP
University of Hannover
Hannover, Germany
fischer@usecap.uni-hannover.de

Jaromir Smrcek
Zoner, Inc.
jaromir.smrcek@zoner.com

Matthew Smith
USECAP
University of Bonn
Bonn, Germany
smith@cs.uni-bonn.de

Abstract

Mobile devices are evolving as the dominant computing platform and consequently application repositories and app markets are becoming the prevalent paradigm for deploying software. Due to their central and trusted position in the software ecosystem, coerced, hacked or malicious app markets pose a serious threat to user security. Currently, there is little that hinders a nation state adversary (NSA) or other powerful attackers from using such central and trusted points of software distribution to deploy customized (malicious) versions of apps to specific users. Due to intransparencies in the current app installation paradigm, this kind of attack is extremely hard to detect.

In this paper, we evaluate the risks and drawbacks of current app deployment in the face of powerful attackers. We assess the app signing practices of 97% of all free Google Play apps and find that the current practices make targeted attacks unnecessarily easy and almost impossible to detect for users and app developers alike. We show that high profile Android apps employ intransparent and unaccountable strategies when they publish apps to (multiple) alternative markets. We then present and evaluate Application Transparency (AT), a new framework that can defend against "targeted-and-stealthy" attacks, mount by malicious markets.

We deployed AT in the wild and conducted an extensive field study in which we analyzed app installations on 253,819 real world Android devices that participate in a popular anti-virus app's telemetry program. We find that AT can effectively protect users against malicious targeted attack apps and furthermore adds transparency and accountability to the current intransparent signing and packaging strategies employed by many app developers.

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

http://dx.doi.org/10.1145/2660267.2660311.

Categories and Subject Descriptors

D.4.4 [**Software**]: Communications Management—*Network communication*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Data Sharing*

General Terms

Security, Human Factors

Keywords

Android, Apps, Signing, Application Transparency

1. INTRODUCTION

The process of installing software is a highly security relevant action – and in the face of powerful attackers including nation state adversaries, it is currently a leap of faith that the software being installed has not been tampered with. While in the past physical media offered some assurance as to the provenance of software and made it unlikely that the version of software being installed was tampered with in a targeted attack, the move to digital downloads and app markets offers a very convenient way for powerful adversaries to target specific users with customized malware. In the case of digital downloads from websites, users could in theory verify the software being installed by comparing checksums[1]. However, this procedure is hard to use for the average user and never received widespread adoption. Central software repositories, and more recently mobile app markets, require developers to sign software/apps prior to inclusion into the repositories. This offers more convenience and in many cases a higher level of security for users compared to direct downloads from webpages. However, there is no automated mechanism for users or developers to verify that an app being in-

[1]Assuming the channel used to obtain the checksum was not compromised as well.

stalled is actually the original and untampered app released by the developer.

Recent revelations have shown how infrastructure organizations can be pressured or attacked by nation state adversaries who want to gain access to specific targets. In May 2014, it became public that parcel services gave the National Security Agency access to routers that were intended to be shipped overseas so they could install backdoors. The packages were then resealed and shipped containing the backdoors [15]. In September 2011 the DigiNotar Certificate Authority was attacked by the Iranian government and issued fake SSL certificates to attack 300,000 Iranian Gmail users [23]. These two publicly known attacks illustrate the power that nation state adversaries can and will assert on infrastructure and service providers which distribute software and hardware to circumvent security measures.

The central app distribution process via markets or repositories is a similarly tempting target. An adversary in control of an app market or repository as well as an adversary who can coerce an app market or repository can easily distribute custom malware to specific users or groups of users. This is a particularly convenient attack vector since most app markets require personal registration and hence make it easy to identify and target specific users. App markets have already been the target of attacks with the aim to distribute malicious apps in the past [6, 20, 14, 28, 27, 25, 5, 26, 11]. Especially in the light of the limited capabilities of mobile anti-virus apps [24, 21], Android's app ecosystem is an attractive candidate for targeted attacks.

A signature verification process vulnerability discovered in 2013 [4, 3] unveiled that maliciously modified apps may be planted on around 75 % of all Android devices without users being any the wiser [1]. This attack allows attackers to tamper with app updates in addition to fresh installs. However, there are more subtle ways app markets can be used to open up attack vectors against specific users: On a regular basis, new security vulnerabilities are being discovered, (hopefully) fixed and updates are released. However, for an app market it is very easy to withhold updates from specific users, thus guaranteeing that they remain vulnerable to known security issues. While most app markets probably have their users' best interest in mind most of the time, attacks against the app markets themselves and the Snowden revelations necessitate the possibility to verify that app markets and software repositories behave correctly and treat all users equally.

In this paper, we conduct an extensive analysis and evaluation of the state of current Android app markets in terms of transparency and accountability during app installation and updating. We show that for both users and developers, the current model of app distribution is severely flawed and makes targeted (as well as non-targeted) attacks unnecessarily easy.

To overcome the resulting lack of transparency and accountability, we present a new framework called Application Transparency (AT). Our framework protects against "targeted-and-stealthy" app markets by making attacks against specific users or groups of users easily detectable. AT extends the concept of Certificate Transparency (CT) [19]. Unlike CT our system provides synchronous prevention instead of only retroactive notification. With respect to app installation and updates, AT guarantees that all users see the same app and version as everyone else and that developers can be certain that no tampered versions of their apps are distributed without their knowledge. Our approach can be applied to any repository or app market based software distribution system. We fully evaluated and implemented our solution for Google Play and use this as an example throughout the rest of the paper. We chose the Android app market for the deployment of our solution, as it is one of the largest and most vibrant app markets and has been the target of a number of real world attacks in the past.

In this paper we make the following contributions:

- We analyze 989,935 distinct free Android apps from Google Play (97% of all free apps in Google Play in April 2014) in terms of developers' app signing practices. We show that the current employment of signing keys is unnecessarily intransparent and insecure.
- For 45 high-profile apps including WhatsApp and Facebook, we evaluated the signing and packaging strategies across Google Play and 22 alternative markets. We show that current signing and packaging strategies make it almost impossible for users to assess whether they are running original apps or malware.
- We introduce and discuss a threat model for centralized software distributors.
- As an effective and easily deployable countermeasure, we introduce Application Transparency (AT). AT is a new framework to make software installations and updates transparent to end users and developers, protecting users against currently invisible (targeted) attacks.
- We provide an integration of the AT client side into the OS as part of the "Verify Apps" procedure for Android 4.4 as well as a standalone app.
- To analyze the effectiveness of AT, we deployed it to 253,819 real world Android devices that participate in the Zoner[2] anti-virus telemetry program and evaluate metadata of (third-party) apps installed on these devices. We show that AT can effectively protect users against malicious (targeted) attack apps and also adds transparency and accountability to the current intransparent signing and packaging strategies many app developers employ.
- We provide a public AT log for all Android apps that we analyzed during our research. Our log currently consists of 2,493,786 Android apps and is growing steadily.

2. BACKGROUND

This section provides background information on the app signing and installation process on Android as well as an introduction to Certificate Transparency.

2.1 App Signing on Android

Android apps are digitally signed by signing keys held by their developers [13]. Apps' digital signatures are intended to identify application authors and to establish trust between apps and Google Play. Google's recommendation is to use self-signed certificates with a key size of 2048 bits for RSA keys and a validity period of at least 25 years. They do

[2] `http://www.zonerantivirus.cz/clanek/android`

not accept apps signed with a certificate that expires before 2033. In case developers author multiple Android apps, they are encouraged to sign their apps with the same signing key. Furthermore, app updates are expected to be signed with the same key to enable the detection of malicious updates.

2.2 App Publishing in Google Play

To publish apps in Google Play, developers have to package an APK file, sign it and upload the APK file to Google Play's developer console. After adding a description and determining in which regions and for which devices the app should be available, the app goes through internal checks before being published. One such check is Google's Bouncer service which analyzes apps for malware [12]. Other checks include the verification of the signing key's expiration date. After uploading the APK file, a popup is shown to the developer promising to make the app available for download in the next couple of hours. Usually apps or updates are available within 60 minutes after upload. Since many mobile app companies offer apps for multiple platforms such as Android, iOS, Blackberry and Windows Mobile, app building and publishing is often outsourced to third party companies that take care of signing and publishing apps (cf. Section 4).

2.3 Android App Installation and Updates

Whenever a new app is installed on the user's discretion, Android's "Verify Apps" mechanism checks the app with Google's integrated anti-virus feature, verifies the APK file's digital signature with the developer's certificate and keeps the certificate for checking future app updates.

Updates for apps need to be signed with the original private key that was used when the app was built initially. Otherwise updates will be rejected and users are required to uninstall previous versions before being able to install the app's latest version. Thus, in case developers lose their private key and need to sign app updates with a new key, their users are shown an error message stating the updates cannot be installed.

2.4 Certificate Transparency

Certificate Transparency (CT) is a framework proposed by Google that aims to securely and provably log all X.509 signing activities of Certificate Authorities [19]. The goal is to prevent CAs from creating "attack" certificates which without CT could then be used to mount Man-In-The-Middle attacks against SSL connections with a very low risk of being caught. The basic concept of CT is to offer a publicly available tamper-proof append-only log that contains all CA-issued SSL certificates. Immediately after an SSL certificate is added to the log, the log responds with a Signed Certificate Timestamp (SCT) which represents a promise of the log to include the newly added certificate to the log within the log's Maximum Merge Delay (MMD) time. The log provides two different types of cryptographic proofs: (1) Users of the log can obtain Proofs-of-Presence (PoPs) that allow everyone to verify that a given SSL certificate is part of the log and (2) users of the log can obtain Proofs-of-Consistency (PoCos) to verify that a snapshot of the log is a successor of a previous snapshot. CT knows two different types of clients: (1) auditors are third parties that monitor CT logs for correct behavior, e. g. an auditor checks whether PoCos for a log are correct, (2) client software (such as a browser) which does not directly communicate with a CT log server but relies on the SCTs signed by a log. These clients must trust the correct behavior of a log, which is checked by the auditors. Web-browser users are clients of auditors and can check a CT log's correct behavior by asking an auditor for the results of its consistency checks. A new extension of the CT log is built every MMD and includes all existing and new certificates added within the last MMD. The underlying data structure of CT's log is a Merkle Tree (MT), an append-only tree in which every node is labelled with the hash (typically SHA-256 or SHA-512) of the labels of its children and possibly some additional metadata describing the node. MTs enable the efficient verification that certain data is present in the tree. A PoP requires $\log(n)$ hashes, n being the layer count of the MT, and contains one hash of each layer of the tree. Proving that one snapshot of a MT is the successor of another snapshot can be done in logarithmic time and space by providing one hash per layer of the tree. The value at the root of an MT is signed with the private key of the CT log provider and called the Signed Tree Hash (STH).

2.4.1 Limitations

CT has a number of limitations:

Asynchronous Proof Validation.

For performance and infrastructural reasons, CT does not conduct synchronous proof validation. Instead, the CT RFC [19] proposes that users gossip received SCTs to detect possible attacks as soon as possible. This mechanism does not protect users when establishing an SSL connection, but uncovers malicious logs (i. e. logs that issue SCTs but do not add the certificates to the log after the MMD) and attack certificates with very high probability over time. This however leaves a time window open for successfully attacking CT users.

Revocation.

While CT provides PoPs, with CT alone it is not possible to check whether a given SSL certificate is still current and has not been revoked. Revocation Transparency (RT) [18] is a CT extension that allows certificate revocation. RT proposes two different methods to store revocation information. The first one is a so called Sparse Merkle Tree (SMT) that is a regular MT in which most leaves are zero. A complete path in an SMT has length 256 and represents the SHA-256 hash of an SSL certificate. The path ends with a 0 or 1 leaf according to whether the certificate is revoked or not. The second proposed structure to store revocation information is a sorted list organized as a binary search tree. However, Ryan [22] shows that both proposed approaches are inefficient since proofs requiring linear space and time require data sizes measured in tens of hundred of gigabytes, which makes them impractical.

3. RELATED WORK

In addition to Google's Certificate Transparency framework, there are further academic proposals to provide transparency or tamper-proof logs.

Kim et al. [17] propose a system called Accountable Key Infrastructure (AKI) for SSL certificates. In AKI, so called Integrity Log Server Operators are expected to log publicly

available X.509 certificates and push them to an Integrity Tree which is a lexicographically ordered hash tree. Since an Integrity Tree is not an append-only data structure, all operations in the tree are digitally signed and validators are expected to monitor the correct operation of the Integrity Log Server's data structures and perform consistencychecks between different versions of the tree. The Sovereign Keys (SK) system [8] proposes to operate timeline servers that act similarly to timestamping servers [16]. SK runs a Trust-On-First-Use model, i. e. the first registration of an X.509 certificate binds the key to the domain name, subsequently preventing duplicate registrations for the same name.

Ryan [22] proposes to enhance CT with an effective revocation mechanism. He introduces a second lexicographically ordered tree called LexTree in addition to the append-only MT-based data structure called ChronTree. In conjunction, both trees can now provide proofs of presence, currency and absence of data in the log. Adding a second tree to the log requires an auditor to not only prove consistency between two different ChronTree versions but also between the Chron- and LexTree. Consistency-Proofs between Chron- and LexTrees are linear in time and space. Ryan proposes to apply the enhanced CT framework to email encryption.

Related work also addresses app installation on Android devices. Barrera et al. [2] analyzed 11,104 Android apps and extracted 4,141 signing keys. They found that 18% of the certificates sign more than one app. They also operate the `https://androidobservatory.org` service that provides meta information such as requested permissions, version code and information about the signing key for 31,368 Android apps as of February 2014 from Google Play and alternative markets. Zhou et al. [29] present DroidRanger, a tool to detect malicious apps in popular Android markets. They propose a permission-based and a heuristic-based scheme to detect malware in Android app markets and come to the conclusion that the evaluated markets are functional and relatively healthy. Zhou et al. [28] analyzed the occurrence of repackaged Android apps in six popular alternative app markets and found that 5% to 13% apps in those alternative app markets were repackaged. The problem of repackaging of Android apps was also investigated by Zhou et al. [27]. Vidas et al. [24] analyzed repackaging in alternative markets and found that some markets exclusively distribute repackaged versions of apps containing malware. Permission dialogs allow users to decide whether they agree to an app's permission requests. Currently, those dialogs are the only mechanism that protects users against too permission hungry apps. Felt et al [9] found that many developers over-privilege their apps and that users have problems to correctly understand Android's permission dialogs or even do not read them at all [10].

4. APP SIGNING PRACTICES

In the following section we evaluate app signing and packaging strategies currently employed in Google Play and other alternative markets. Therefore, we analyze the signing practices of 989,935 distinct free Android apps from Google Play (97% of all free apps in Play as of April 2014)[3].

For these apps, we extracted the signing keys to evaluate Google Play's current app signing practices. We extracted 380,345 distinct certificates that were used to sign all apps in our corpus.

380,285 (99.98%) of the signing keys are self-signed, 5 apps were signed by one single certificate issued by a dedicated Android CA from Symantec. We found three certificates signed by CAs from major telco providers such as Sony Ericsson, Cisco and Samsung. The remaining 47 certificates were signed by developers' custom CAs.

Table 1(a) summarizes the algorithm suites employed by the certificates we analyzed. While 369,278 (97%) certificates apply current security best practices and secure algorithm suites, 2860 certificates employ MD5 and 23 certificates use MD2 as their signature algorithm and hence unnecessarily weaken their signature security.

51.6% of the certificates follow Google's security guidelines and have a key size of 2048 bits (cf. Table 1(c)). 0.1% of the certificates employ larger keysizes and 48.15% employ 1024 bit keys. However, 277 certificates use 512 bit RSA keys and hence undermine their apps' security and Android's security guidelines.

Google recommends a validity period of 25 years or longer for signing keys and requires that apps published in Google Play must be signed by a certificate with a validity period ending after 2033. Table 1(d) illustrates the distribution of validity periods in our app corpus.

While 1% of the certificates have a shorter validity period than recommended, 70 % are within the recommended validity period (25 – 50 years). 12.3% of the certificates have a rather optimistic validity period between 100 and 1000 years and 2033 of the certificates are valid for a 1000 years or longer. The longest validity period we found is a certificate that is valid until the year 10,049. We found 9 certificates that expired before 2033; 4 of them issued in 2013. The most recent of these certificates was issued in August 2013. Hence, Google Play's enforcement of its own guidelines seems to be rather laissez-faire.

Table 1(b) illustrates the distribution of signing keys and the number of apps signed by a specific key. 92.28 % of all keys are used to sign one unique app. However, we found 483 signing keys in our app corpus that signed up to 25,190 apps. Interestingly, 0.1 % of the keys we analyzed signed 113,842 (11.5 %) of the apps in our corpus.

Pathological Cases.

Four of the extracted keys signed 64,701 apps in our corpus (cf. Table 2). Hence, about 7 % of all apps in Google Play are signed by these four widely employed keys. While this alone is suboptimal for app security, the fact that the current Android ecosystem has no mechanism to revoke signing keys and their corresponding apps makes things even worse.

These four signing keys belong to service providers that allow their customers to create mobile apps without much coding effort, like e. g. the Qbiki Networks key, which signs apps for the Seattle Clouds[4] service provider. Their customers can download templates for apps, modify these templates according to their needs and then let the service provider build and publish their apps to app markets such as Google Play or Apple's App Store. During the build-process, all apps are signed with the same private key. The app publishing procedure creates two serious issues for both app owners

[3]Due to geographical restrictions, we were not able to download the remaining free apps.

[4]`http://seattleclouds.com/`

Table 1: Results of our Signing Key Analysis

(a) Deployed Signing Algorithms

Algorithm	Certificates	Affected Apps
sha1WithRSA	215004	584901
sha256WithRSA	154274	284607
dsaWithSHA1	8154	20600
md5WithRSA	2860	16669
sha512WithRSA	28	33
md2WithRSA	23	82
sha1WithRSA	1	6
dsa	1	1

(b) Apps Signed per Certificate

Apps Per Certificate	Certificates
< 5	390254
$5 - 10$	10967
$11 - 20$	3756
$21 - 50$	2292
$51 - 100$	673
$101 - 1000$	463
$1001 - 10000$	16
> 10000	4

(c) Deployed Key Sizes

Key Size	Certificates	Affected Apps
16384	3	3
8192	35	65
4096	599	1289
2048	196305	398322
1024	183126	506477
512	277	743

(d) Validity Periods of Certificates

Validity Period	Affected Certificates
< 25 years	3839
25 years	101691
$26 - 50$ years	164763
$51 - 100$ years	61065
$101 - 1000$ years	46954
> 1000 years	2033

Table 2: Certificates that were used to sign more than 10,000 apps

Certificate	Apps Count
L=Seattle/O=Qbiki Networks/OU=iOS/Android Development/CN=Andrew Vasiliu	25,190
C=CA/ST=MB/O=Andromo.com L=Winnipeg/OU=Development/CN=Andromo App	10,803
C=RU/ST=NSO/L=Novosibirsk/O=BestToolbars/OU=Desktopify/CN=Anton	15,269
C=US/ST=California/L=Chico/O=Bizness Apps/OU=Bizness Apps/CN=Andrew Gazdecki	13,439

and their users: (1) The app-building provider has access to all apps' source codes and (2) the signing and publishing process is in its control. Hence, app-building providers can modify apps and push these modifications to official app markets under the radar of the actual developers and users. Given the install counts provided by Google Play, up to 125 million devices have apps installed that were signed by this single signing key.

4.1 Alternative Markets

Many apps are not published exclusively via Goole Play, but can also be found in alternative Markets such as Amazon's Appstore or the F-Droid market. Overall, there are more than 30 "official" alternative markets available[5]. In the following, we evaluate app signing and packaging strategies for 45 popular Android apps[6].[7].

Results.

In our sample set, we could not find a single app of which the same version was available across all the markets we surveyed. In all cases, the Google Play version was the most current (but not necessarily as new as to be the most current version according to official developer announcements), which underpins the assumption that Google Play is the most important app distributor for most developers. 19 of the 45 apps we analyzed were signed with the same key across all markets offering the app. However, although the same signing keys were used, all apps distributed APK files with different SHA-256 values. In 26 cases, at least two different signing keys were employed. An interesting case is the Amazon market that is supposed to be a benign alternative market and offers 24 of the 45 popular Play apps we tested. 15 of those apps were signed with different keys than their counterparts in Google Play. A popular example is WhatsApp. The officially announced most current version[8] is 2.11.169. Google Play offers version 2.11.152 and we found 6 different "most current" versions of the app in 12 alternative markets.

4.2 Discussion

Analyzing the current app signing practices for 97% of Google Play's free Android apps and evaluating the update and packaging strategies employed by popular Android app providers unveils an unnecessary intransparency and inaccountability for app users. The fact that single app signing keys are entrusted with more than 25,000 Android apps constitutes a serious problem with verifying the authenticity of Android apps. While Google intended to create a path of authenticity between app developers and their apps by enforcing (self-signed) certificates to sign apps, the current trend towards outsourcing app signing and distribution to a few big players in the market undermines this intention. The fact that these service providers use one single key to sign thousands of apps without an effective revocation mech-

[5]Cf. `http://www.onepf.org/appstores/`

[6]We chose the 45 most popular Android apps across Google Play and 33 alternative markets, as listed by `https://play.google.com/store/apps`.

[7]The complete list of apps and alternative markets we checked is available as additional material at `http://application-transparency.org/files/altmarkets.pdf`

[8]As of February 20th, 2014

anism at hand jeopardizes both app developers – since one stolen private key may result in harmful updates for thousands of apps – as well as app users, since one successful attacker could plant malicious apps into hundreds of millions of devices. In addition to the app signing bug uncovered in 2013 that still affects a huge portion of Android devices, the fact that Google accepts signing keys with 512 bits for apps and that such apps are actually deployed to Google Play shows that Google's line of defense is fragmentary.

Additionally, we found that even the most popular apps are signed with different keys for the same version across multiple markets, making it hard for their users to reliably determine the authenticity of apps. Not having a straightforward tool at hand for verifying apps' authenticity becomes even more serious in the light of the fact that some alternative markets turn huge portions of their apps into malware and then act as super-distribution points for malicious Android apps [29]. Hence, users of alternative markets have no chance to differentiate between harmless and harmful apps – even if they compare apps' checksums or certificates.

5. THREAT MODEL

In the following, we present and discuss our threat model we call "targeted-and-stealthy". Here we assume that reputable app markets are not interested in distributing malware to all of their customers, but might be interested or coerced to attacking specific target users. Secondly we assume that the app market attacker does not want these attacks to become public knowledge. Currently, users who search, download and install apps from centralized app markets and repositories need to more or less blindly trust these distributors. The power to deploy (targeted) attacks held by software distributors in the app market ecosystem is comparable to Certificate Authorities in the world of SSL certificates [23]. Due to their central and trusted position in the software ecosystem, coerced, hacked or malicious app markets have the ability to easily deploy targeted and stealthy attacks against specific users or groups of users. Since apps are signed with developer keys, mass-distribution of tampered (malicious) apps cannot be considered a stealthy attack. App developers could simply verify the authenticity of their own apps and unveil the rogue market's malicious behavior. However, if tampered apps are only distributed to specific targets, it is easily possible to run "targeted-and-stealthy" attacks, especially in the light of current app signing and packaging strategies employed by developers (cf. Section 4).In our threat model we distinguish between app markets and app market clients installed on users' devices. While app markets maintain repositories, app market clients send install requests to markets and perform the actual installation. In our threat model, app markets as technical and legal entities can be compromised or coerced and thus are our threat actor. In contrast, app market clients are installed on the users' devices and for this work we consider them to work correctly. The current status quo for app market clients is that the installation of new apps must be triggered by users, but updates to installed apps are installed silently without user interaction. This type of silent (update) install can be triggered by the app market by offering a newer version of an already installed app. Hence, app markets cannot conduct silent installs but may trigger silent updates. In this work we do not list kill switches as a threat since these are an OS feature and not an app market feature and thus are out of scope of this paper. In general, malicious code in the OS or the app market client is not considered in this paper[9]. In summary, our threat model covers the following attacks app markets can mount with a low risk of discovery:

Tampered/Malicious Apps: App markets can offer tampered (malicious) apps to specific users or groups of users.

Tampered/Malicious Updates: App markets can offer tampered (malicious) updates to specific users or groups of users.

Withheld Apps: App markets can withhold applications from specific users or groups of users, for instance to prevent the installation of security software or to perform censorship.

Withheld Updates: App markets can withhold updates of apps (e. g. security patches) from specific users or groups of users, for instance to keep them vulnerable to security issues discovered in older app versions.

Removed Apps: In case that malware is discovered and actually removed from app markets for the masses, the markets can still offer the malware to specific users or groups of users.

The above attack vectors are serious security issues for app market customers. While no targeted-and-stealthy attacks mount by app markets – either voluntarily or under the pressure of a nation state adversary – have become public knowledge, similar attacks in similar ecosystems have been seen in the past [23, 15]. Thus we suggest preemptive action to prevent such attacks from becoming reality.

6. APPLICATION TRANSPARENCY

To counter the lack of transparency in current app market installation processes, we propose a new framework called Application Transparency (AT). While Certificate Transparency focuses on SSL certificates, we propose to transfer the core idea of transparency to the world of software binaries in general and provide a proof-of-concept implementation and evaluation for Android applications. We build on the data structures and algorithms proposed by CT [19] and LT [22] and leverage the unique properties of app markets to improve previous work and solve the outstanding deployment issues.

The Application Transparency framework addresses the security issues presented in our threat model (cf. Section 5). For this, AT provides three different kinds of cryptographic proofs that allows users to verify the authenticity of apps provided by app markets. The Proof-of-Presence (PoP) is a cryptographic proof that provides information about the presence of an app in a market. This proof allows users to make sure that a provided app is publicly available and not a targeted-and-stealthy attack by the app market. Hence, app markets cannot (be coerced to) send targeted (malicious) apps to specific users, without these being easily detectable by checking the public logs. Proofs-of-Presence also

[9]This is a similar assumption to stating that web browsers are considered to work correctly when performing SSL certificate validation.

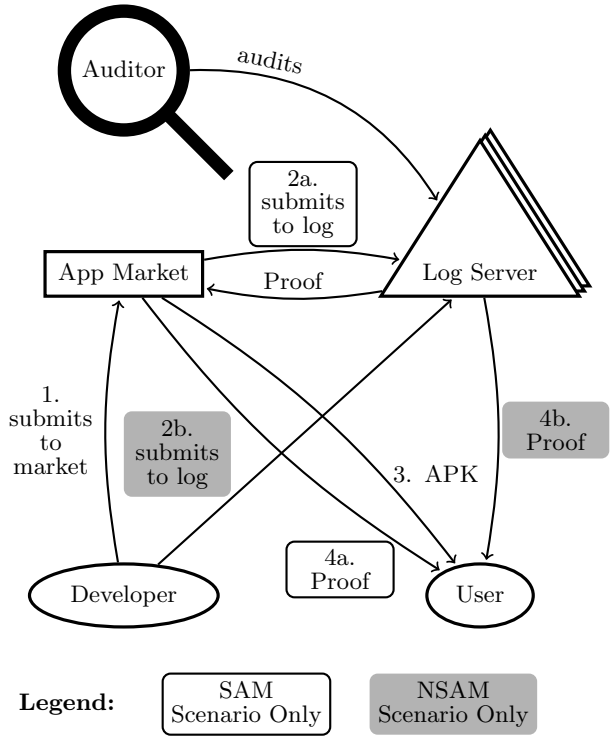

Figure 1: All Parties involved in Application Transparency

cover the prevention of silent installations of targeted-and-stealthy malicious updates. The Proof-of-Currency (PoC) is an extended PoP version and provides cryptographic information that allows users to verify the currentness of an app's version. Hence, app markets cannot (be coerced to) withhold certain apps' updates from specific users. PoCs can be utilized as a revocation mechanism. Whenever an app's version should be removed from the log, a new (can be null) hash value is added to the log. The Proof-of-Absence (PoA) provides cryptographic information that allows users to verify app absence messages presented by app markets. Hence, app markets cannot (be coerced to) withhold certain apps from specific users.

AT involves multiple actors as illustrated in Figure 1:

App Developers.

App developers create Android apps and submit them to markets – either to Google Play or one of the many alternatives. In case the app market(s) support AT, the app release procedure for developers does not change. However, if AT is not supported by the respective market, developers have to take one extra step, namely push their app to an AT log. More information on the two cases can be found in Section 7.

App Markets.

App markets accept apps from developers and distribute them to mobile users. App markets supporting our AT framework will additionally distribute corresponding AT proofs (cf. Section 6) to their users.

Mobile Users.

Mobile users install apps from app markets utilizing app market clients such as the Google Play app on Android. Along with the apps, AT proofs are sent to the users' devices. In case AT proof validation is supported by the integrated app market client, users do not have to take any extra steps. In case AT proof validation is not handled by the integrated app market client, users who want to benefit from AT can use the standalone AT app.

Log Providers.

Log providers operate a pair of two logs that constitute the AT log: A ChronTree and a FixTree (cf. Section 6.1). There are two interfaces, one for accepting new apps from app markets or developers and a second to provide AT proofs. The AT log is rebuilt every MMD to be able to provide the most current proofs. Candidates for log providers are app markets, anti-virus companies or organizations such as the EFF. By cryptographically verifying Proofs-of-Consistency over time and from different locations, the correct behavior of an AT log provider can be ensured. Hence, users do not have to blindly trust the provider to work correctly – even a nation state adversary could run an AT log without compromising any of AT's security properties. Should a nation state adversary such as the NSA submit an app to their own log, a Proof-of-Currency would be available immediately. Due to the log's append only structure, the app could not be removed again. Even if the NSA revoked the app, it would remain in the log and while a Proof-of-Currency would no longer be available, the corresponding Proof-of-Presence would remain. Hence, the fact that the app had been submitted at one point could not be expunged again.

Auditors.

AT auditors monitor the correct behavior of log providers and inform the public in case a log provider turns malicious. auditors monitor AT logs by consistently verifying Proofs-of-Consistency. AT log providers cannot be their own auditors, but can audit other AT log providers. Anti-virus companies could act as auditors, as well as for example the EFF. Moreover, app development companies/app generators as well as Android users can act as auditors.

Table 3 illustrates the different actors and how they benefit from participating in the AT ecosystem.

6.1 Log Structure

The AT log consists of a tuple of two interdependent Merkle trees (ChronTree and FixTree).

As proposed by LT [22], we use the ChronTree which is an append-only and therefore chronologically ordered Merkle Tree. It is extended by appending leaves from left to right, creating a balanced tree. After appending an app to the tree, which is done in constant time, the root hash needs to be re-built. Hence, insertion is $O(\log(n))$. The tree provides Proofs-of-Presence (PoPs) for inserted apps and Proofs-of-Consistency (PoCos) which are able to show that any two versions of the tree are consistent. This is given if one tree is a subset of any later version of that tree. PoC and PoCo checks can be done in $O(\log(n))$. Unfortunately, in a ChronTree PoCs and PoAs demand $O(n)$ since the tree has to show that an app is outdated or revoked.

To make PoCs $O(\log(n))$ and to enable PoAs, we utilize a second lexicographically ordered Merkle Tree [22, 17]. The

Table 3: Application Transparency Actors and Their Benefits

Actor	Benefits
Mobile User	Targeted Attack Prevention - can be sure that they are treated equally to all other users.
App Developer	Protect their apps against manipulations by app markets.
App Market	Are able to provide cryptographic proofs for their correct behavior.
Log Provider	Provide transparency for the app market ecosystem.
Auditor	Monitor correct behavior of log providers and help to establish a foundation of trust in the AT ecosystem.

FixTree[10] is organized as a binary search tree that stores nodes in a way that an in-order traversal yields the certificates stored in the tree in lexicographic order of the certificates' subjects. Insertion, PoC and PoA are $O(\log(n))$ in the average case.

In contrast to the ChronTree, the FixTree alone is not able to provide consistency proofs in $O(\log(n))$ since it does not have the necessary append-only property. Therefore we use a ChronTree/FixTree pair utilizing the advantages of both trees to finally achieve PoPs, PoCs, PoAs and PoCos in $O(\log(n))$. This is done by inserting an app into the FixTree first, and subsequently adding a tuple consisting of the app and the freshly built root hash of the FixTree to the ChronTree. One drawback of using a binary search tree as the underlying data structure of the LexTree is that insertions can unbalance and degrade the tree. We therefore employ a Merkle Tree built up on a 2-3 tree as the underlying data structure. We call this tree FixTree since it guarantees a balanced tree for inserts in $O(\log(n))$ and equally sized proofs of $O(\log(n))$ for all data in the tree. In contrast to the LexTree proposal, the data (i. e. the package information) is contained only in the leaf nodes. Each leaf node for an app has the form $(package_{<ext>}, (h_{v1}, h_{v2}, ...))$ where *package* is the unique packagename (e. g. *com.google.android.gm*) of an app and $< ext >$ identifies a device-, language- and region-based app version[11] – e. g. some apps are only available on certain device types, in certain languages and are limited to specific geographic regions such as the U.S.. The values $(h_{v1}, h_{v2}, ...)$ are lists of SHA-256 checksums for the corresponding $package_{<ext>}$ identifier consisting of different chronological versions of an app. Hence, the checksum lists store all app versions that are available for different devices, languages and regions. Whenever an app is outdated – since a newer version is available – an updated hash for the app's $package_{<ext>}$ identifier is appended to the corresponding checksum list. The size of the list of chronological versions of an app the FixTree keeps is bound by a constant N. In other words, the FixTree keeps only $N-1$ chronological versions of an app. The chronologically ordered list of SHA-256 hashes for an app's $package_{<ext>}$ identifier is utilized as a revocation mechanism. Whenever an app's version should be revoked, a new (maybe null) checksum value for the $package_{<ext>}$ identifier is added to the log. Hence, the revoked version of an app is not totally removed from the log, but marked as not being the most current version any longer. In order to still be able to search efficiently in $O(\log(n))$, the non-leaf nodes hold the (lexicographic) minimum of the leftmost subtree and the maximum of the rightmost subtree.

Both the ChronTree and the FixTree are re-built every MMD, which we propose to be around 30 minutes. An MMD of 30 minutes provides a buffer for capturing the app publication state of the Play Market. Based on the apps we crawled from Google Play, we found that around 12,000 new apps or app updates are published in Google Play every day. Updating the trees every 30 minutes results in 48 updates of the tree every day. Every log update has to append an average of 250 new apps to the current trees. We measured the time our implementation requires to append new apps to the trees. Our tests showed that 48 updates a day are easily feasible from both a computing performance point of view and with respect to the app publishing parameters Google Play offers.

[10]The FixTree is an extension of the LexTree [22] concept.

[11]If required, more information can be added to a package name's extension.

Construction of PoC.

Each rebuild of the FixTree potentially changes all PoCs for apps in the FixTree. In order to make retrieving PoCs as efficient as possible, the Merkle audit paths for each leaf are built after every merge and kept in a key-value store to make sure the FixTree does not need to be traversed for every request.

Construction of PoA.

Given a hash h that is not included in the FixTree, a PoA is constructed as follows (cf. Figure 2):

1. Search for h in the FixTree to find the closest leaf l smaller than h.
2. Execute an in-order traversal starting from l to the next leaf r to find the closest leaf greater than h (solid path in Figure 2).
3. Supply Merkle audit paths for l and r.

Given the Merkle audit paths for l and r, a verifier can then validate that h is absent in the FixTree as follows:

1. Check that $l < h < r$.
2. For the audit path for l, check that each node in the path is either a right child or the parent is also included in r's audit path.
3. For r's audit path: check that each node is a left child, respectively.

One specific case is constructing the proofs for an element that is smaller or greater than every element in the FixTree. In those cases, the PoA consist of only one audit path of the leftmost or rightmost node respectively.

7. DEPLOYMENT

We propose to operate multiple logs – ideally every app market should operate its own log to avoid a single point of failure but still provide transparency for mobile users.

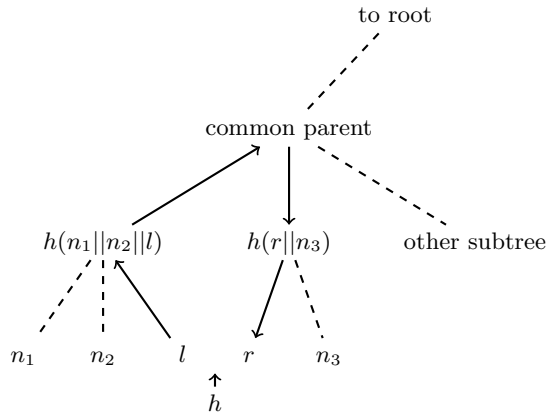

Figure 2: Generating a Proof of Absence for h with the adjacent nodes l and r

In this section, we show the applicability of AT for Android apps by analyzing various deployment aspects. We provide an open source implementation of our Application Transparency framework, a working log with 2,145,973 Android apps[12] and a standalone app for proof verification. The deployment is viable without any changes to the app market. However, we also discuss a second deployment option in which the app market participates in the transparency process. We classify these two deployment approaches as (1) the Supportive App Market (SAM) scenario, where the market actively submits its apps to (its own) AT log(s), and (2) the Non-Supportive App Market (NSAM) scenario where apps are submitted to AT log(s) by developers or others. We apply the AT framework to the installation of new apps and to the process of updating installed Android apps.

7.1 Adding Apps to the Log

Adding an app to one or multiple AT log providers is essential to benefit from AT's properties. Immediately after receiving an app, the log provider issues a Signed Application Timestamp (SAT) acting as a promise to add the app to the next snapshot of the AT log and as a proof of the point in time when the app was sent to the log provider. After the MMD passed, all apps that received a SAT in the last MMD interval are included into the next snapshot of the log and all types of proofs for the apps can be requested from the log. The FixTree's leaf node structure (cf. Section 6.1) allows for the inclusion of multiple versions of an app in a single AT log. Chronological updates of app versions can simply be appended to the app version's corresponding list of SHA-256 values. Every (non)-developer is allowed to submit apps to an AT log. While this does not prevent fake submissions, the log's properties make those submissions transparent and public knowledge. Hence, (malicious) fake submissions are publicly detectable and AT's revocation mechanism (cf. Section 6) allows for the transparent removal of unwanted apps from the log.

SAM Scenario: In case an app market is supportive and submits all its apps to one or multiple log servers, the inclusion of new apps or updates of existing apps is straightforward: Google Play, for example, runs malware detection and other administrative tasks before a submitted app is made available. Currently, this process usually takes between 60 minutes and two hours. Hence, Google Play could easily submit apps to one or multiple logs without noticeably extending the existing publishing period of an app.

NSAM Scenario: In case an app market is not supportive, there are two alternative mechanisms to make apps available to one or multiple AT log providers. Developers who intend to make their app available in an AT log submit their apps to the log(s) before making them publicly available. To make pushing an app to the AT logs as easy-as-possible for app developers, we propose to include this step into the app building process. Therefore, we extended the `Manifest.xml` file of Android apps [13] and now allow developers to configure one or multiple AT log providers to which the app should be pushed. Based on this configuration, we implemented app pushing into the conventional app building process to make app publishing transparent and easy-to-use for developers. After packaging the app's APK file with all its compiled code and required resources, the app file's checksum is submitted to the configured AT log(s). The logs then generate a SAT for the app and queue the app for inclusion into the log's next snapshot [14]. The SAT is sent back to the developer. Next, the SAT is inserted into the app's APK file and finally signed with the developer's signing key. This procedure is transparent to developers and does not unnecessarily draw out the conventional app building process. To support developers and to provide a starting point for the Application Transparency system for Android apps, crawling app markets is a pragmatic approach.

7.2 Proof-Verification

The verification of cryptographic proofs is a vital part of Application Transparency. In AT, a PoP and PoC or a PoA is a set of two auditpaths that both have the length $\log(n)$ with n being the number of apps in the log. The auditpath length in our current log is 21 and the set of required proofs consist of $2 * 21 * 32 = 1344$ bytes of auditpath data. The average size of an Android app in our sample set is 4,936,800 bytes. This gives an average Proof-To-Payload ratio of $1344/4936800 = 0.027\,\%$. Proofs are verified during app installations or update requests. Hence, we suggest synchronous security checks. Subsequently, proof-delivery for the SAM and NSAM scenarios are discussed.

7.2.1 SAM Scenario

In case app markets support Application Transparency, delivering proofs synchronously is straightforward. On every MMD, app markets need to fetch current proofs for their apps, cache them for the MMD period and deliver them to their users for every app installation or update. Proofs are then sent directly with the APK file. There are no privacy issues or out-of-band connections which could slow down the app installation process.

7.2.2 NSAM Scenario

In case app markets do not support Application Transparency, users need to fetch proofs directly from their preferred log(s). As illustrated above, the Proof-To-Payload

[12]This includes all apps we downloaded from Google Play as well as apps installed on real world devices gathered by our AT deployment (c.f. Section 8).

[13]As long as the Manifest.xml file only contains valid XML, Android will not reject apps.

[14]No longer than the log's MMD

ratio is rather small. Hence, synchronously fetching the proof does not noticeably slow down the installation/updating process. To speed things up, our implementation triggers proof-fetching as soon the packagename and version code of an app are available, which happens before the download is complete.

Additionally, the proof verification process can be split into two more scenarios:

7.2.3 Android OS Integration

In case proof verification is integrated into the Android OS, it is straightforward. As soon as the proof is available (either delivered from Google Play or fetched directly from an AT log provider), proof verification can be performed within Android's default app installation/update routine. We implemented proof verification into Android's "Verify Apps" routine that is executed on installation and update. Our implementation verifies a given set of AT proofs for an app and communicates the verification result to the user.

7.2.4 Standalone App

In case proof verification is not integrated into the Android OS, a standalone app can perform proof verification. Similarly to conventional anti-virus apps on Android, our app is triggered whenever a new app or an update for an existing app is installed. Our app then fetches the proof for the given app (update), verifies the proof and communicates the verification result to the user. To protect the users' privacy, we added a TOR opt-in feature to our proof-verification app that allows users to hide their identity from AT log providers.

7.3 Benefits over CT

While CT is used for the transparency of issued certificates, in AT we offer binary protection of the distributed software packages. Due to the different attack models, AT introduces new requirements resulting in the proposed modification of the proof concepts, but also an easier proof deployment. Our proposed system takes advantage of the AT requirements and offers following additional properties in contrast to CT:

Easier Deployment.

Certificate Transparency relies on the certificate issuer or owner to push a certificate to the CT log. This list encompasses multiple hundred CAs. As of today, only a few CAs have committed to pushing future certificates to the CT log [7]. However, the long term success of Certificate Transparency crucially depends on the participation of all important CAs which comprises a list of more than 100 companies. In AT, app markets can decide whether they wish to provide transparency and do not require the participation of additional app markets. The participation of GooglePlay, providing more than 90% of all installed apps[15], would cover most app installations in the wild. However, even if no app market participated in the AT ecosystem, app developers could start a bottom up approach and register their apps with an AT log to protect their customers. As long as an app is not present in any AT log, a PoA would be spread, while as soon as the app is present in a log, the PoA would be replaced with a PoP, thus giving users transparency on a per app basis. In CT, as long as no PoP is spread for a website, users need to trust the log provider. Hence, CT's security features take full effect only when all SSL certificates are pushed to one or multiple logs, while AT also offers benefits to early adopters.

Synchronous Proof Validation.

While AT offers synchronous proof verification, the CT project decided against it. Synchronous validations either requires every web-server administrator to always provide the current version of the proof for their website and deliver it in-band during an SSL handshake, or it requires users to fetch the proof out-of-band. Both options have enormous disadvantages that discouraged the CT project from performing synchronous proof verification. App installation, on the other hand, is a relatively rare event and one which takes more time, so the additional overhead of synchronously verifying the proofs does not extend the installation process notably for the users.

Additional Proof Features.

In addition to the Proof-of-Presence (similar to CT' certificate proof) that enables app markets to cryptographically prove that a presented app has actually been submitted to a log, our system introduces new crucial features to prevent the following attacks:

Withholding Apps Every query to an app market [16] which does not return a positive result and the corresponding PoP has to return a PoA – either provided by the app market (SAM) or the AT log provider (NSAM) – showing that the requested app has indeed never been submitted to that market and the corresponding log. In case of the targeted withholding of an app, a PoA can not be provided, since the app is actually present in the log (and the market) – i. e. a PoP exists – but was withheld only from the attacked user. The non-existence of a PoA and the withholding of the corresponding PoP makes the attack immediately detectable.

Withholding Updates Every update query requires a PoC – either provided by the app market (SAM) or from the AT log provider (NSAM) – guaranteeing that the offered application is the most current version and is also issued to all other app market users.

8. EVALUATION

To evaluate AT, we implemented both the ChronTree and FixTree, a standalone Android app that can verify AT proofs for new apps and updates and an integration into Android's OS-level "Verify Apps" feature. We also deployed the system in the NSAM scenario by integrating AT into the telemetry feature of Zoner's anti-virus app for Android.

Gathered Telemetry Data.

Over a period of four months from January 2014 to April 2014, we gathered the following meta information from 253,819

[15]The number is based on the meta information we gathered with Zoner's telemetry program.

[16]A query is considered to be the full package name of an app plus its extension value (cf. Section 6.1 for the AT log structure).

devices that participated in the telemetry program and who gave their consent to anonymously analyze the data for our research:

Pseudonym We assigned a 256-bit random pseudonym to each device to protect the users' privacy. The pseudonym did not reveal any private information.

DeviceInfo We collected manufacturer- and device model information as well as the installed Android version.

DeviceFlags We gathered three different flags for every device: (1) Whether developer options were enabled, (2) whether app installs from untrusted sources were allowed and (3) whether USB debugging was enabled.

PackageInfo For every (pre-)installed app we gathered the package name and version code.

PackageHashes For every (pre-)installed app we gathered SHA256 checksums of the packages and their corresponding signing keys.

AV-Result For every (pre-)installed app we collected the AV detection result.

Results.

The telemetry program gathered the above meta information. Whenever we found only one checksum value for a <packagename, version> tuple across all devices, we treated these findings as harmless, since everybody had the same binary installed and thus no tampered binaries were installed for specific targets. However, when multiple checksums were present across devices, we treated these findings as checksum conflicts. A checksum conflict can have different root causes: (1) App developers compile different versions of an app for different app markets (cf. Section 4.1), (2) apps got repackaged to (benignly) add or remove certain features from the original app or (3) apps are turned into malware to mount (targeted) attacks. In all three cases, users would benefit from AT's transparency features. While most checksum conflicts we found fall into categories (1) and (2), in combination with anti-virus software AT can help to assign apps to categories (1) and (2) and to distinguish apps that fall into category (3).

We collected information for 912,393 different Android apps[17]. While 824,203 apps had no checksum conflicts, we found 88,190 apps (10.7 %) with checksum conflicts as shown in Table 4(a). Here we distinguish between three different types of apps: (1) System apps signed with the same key as the Android SDK, (2) vendor apps which were pre-installed by the vendors and (3) third-party apps installed by the user.

Since Google Play is the most important and most widely used app market for Android, we chose this market as our transparency baseline, i. e. we compared the apps' checksums with the values we found in Google Play. Although Google Play could theoretically have provided us with some tampered apps when we crawled the market, for our further analysis we assumed that Google Play played fair with us. Table 4(b) shows the anti-virus results for the conflicts for apps distributed by the Google Play store.

The checksum conflicts result from the usage of different keys for different markets, app customization, or in the open source case, different distributors. Another explanation for conflicting checksums for apps signed by the original signing key would be targeted attacks deployed by the original app developer. Although this is highly unlikely in most cases and we could not find supporting evidence, this circumstance reveals the lack of intransparency of current mobile app distribution: Many of these cases cannot be fully assessed without costly analysis of every suspicious application for malicious behavior on a per app basis.

For the 5,445 apps for which we found conflicting checksums signed by multiple keys, our AV app yielded (partly) positive malware detection results, i. e. for either all versions or only some versions, the AV app tagged apps as either adware or other malware. 4,657 of these apps were tagged as adware – for 760 apps the official Google Play store version was tagged as adware and the non-official versions were detected as non-adware. These apps pointed to tools that remove ad libraries from existing apps and recompile the original apps. 3,897 apps were detected as non-adware in the original version, but were found to be adware whenever signed by a different key. Hence, in these cases installing apps from alternative markets or off-site resulted in catching adware on a device instead of installing the original app version. 788 apps were detected as rootkits. In all cases, the official Google Play store app versions were detected as non-malware while off-site installs were malicious. These results confirm previous findings [24]. In these cases AT would have protected the users from accidentally installing malware on their devices by informing them that they were dealing with non-publicly known versions of an app.

8.1 Discussion

Analyzing AV telemetry meta-data of 253,819 real world Android devices shed light on the current status of apps' intransparency in the wild, demonstrated the need for an effective tool that allows to verify apps' authenticity and confirmed the smooth deployability of AT. For 89.3% of the apps we analyzed in the wild, we did not find suspicious patterns. Hence, most of the installs in the wild are already transparent although no "everything's-logged" cryptographic proof such as provided by AT is available. These installations directly benefit from an AT deployment and give users and developers the certainty that they were not subject to a targeted-and-stealthy attack by the app market.

However, also the remaining 10.7% installations for which we found conflicting checksums would benefit from widespread AT deployment. A huge portion of apps with conflicting checksums resulted from the signing and packaging strategies employed by many app developers (cf. Section 4.1). In theses cases, having AT at hand would allow users to rely on the authenticity of apps, as well as allowing developers to make sure that users of AT do not accidentally install tampered versions of their apps. A rather small fraction of apps we found in the wild were tagged as malware by our AV app. These cases would benefit from a widespread AT deployment as well: We assume malware apps would not be submitted to publicly available AT logs and hence would not be valid installation candidates for users. If malware were submitted to a log, in contrast to current malware detection, tagging a malicious app as malware once and then (transparently) removing the app from the AT logs by using AT's revocation mechanism (cf. Section 6.1) would immediately protect all users of the AT infrastructure. Finally, we found very few conflictingc checksums for which we currently cannot be certain whether our findings are harmless or actually

[17]Each <packagename, version> tuple was treated as a single entity.

Table 4: Telemetry Program Results

(a) App Checksum Conflicts across all Devices

App-Type	Amount
System Apps	8,632
Pre-install Vendor Apps	15,999
Third-party Apps (Google Play)	46,481
Third-party Apps (only other Markets)	17,078

(b) Google Play Apps with Checksum Conflicts

Signature Keys	Result	Amount
Same Key	`Trojan.AndroidOS.Stealer.A`	2
	No Information	21,623
Various Keys	Permission Remover Service	2
	Adware	4,657
	Rootkits	788
	Negative	5,241
	No Information	4,265

malicious. While this is a limitation of our evaluation, it urgently illustrates the problem of current app deployment: There are cases in the wild that cannot be reliably assessed with current tools. On the one hand these checksums look suspicious, but on the other hand the limited capabilities of current malware detection mechanisms make a final decision whether malicious apps were found or not a gambling game. A widespread AT deployment would effectively disclose attackers that try to invisibly smuggle malicious apps into markets. Consequently AV providers could focus their malware detection efforts on the apps present in public AT logs which eases the development of new, more effective off-device detection mechanisms.

9. CONCLUSION

The installation of software is a security critical task and the current centralized app market paradigm present in the appified world is boon and bane together and offers powerful attackers a lot of very convenient ways to plant malicious software on users' devices. We illustrated parallels between app markets and similar ecosystems that have already been exploited by nation state adversaries and revealed the urgency to equip app markets and other software repositories with an effective countermeasure. By analyzing signing and packaging strategies of 97% of the Android apps in Google Play, we illustrate that current signing practices directly threaten mobile security and indirectly make it almost impossible for app users to verify apps' authenticity with current tools. We found evidence that a handful of signing keys, used to sign more than 7% of all apps, are not under control of the actual developers and enable a handful of app distribution providers to stealthily create malicious updates. Additionally, we found that pushing different app versions, signed by different signing keys – which leads to different checksums across multiple markets – is a common practice and makes apps' authenticity verification even more difficult.

We then presented the AT framework: an effective mechanism to protect users and app developers against "targeted-and-stealthy" app market attacks. We show that the central software distribution paradigm makes the deployment of AT easier compared to other transparency approaches such as Certificate Transparency (CT). We discuss two deployment paths and show that AT is easier to deploy than CT even if app markets do not support AT. However, AT also provides better security compared to CT due to synchronous proof validation and the availability of proofs of currency and absence.

In an extensive field study we analyzed app metadata from 253,819 real world Android devices that participated in Zoner' anti-virus telemetry program. We found that 90% of all apps would directly benefit from AT by being able to present "everything's-logged" cryptographic proofs. However, also the remaining less transparent cases would benefit significantly from AT. A huge fraction of the currently conflicting cases could be clarified by applying AT: The harmless conflicting apps would become apparent by submitting them to AT logservers. Notably, we also found cases when AT would have prevented users from (unwittingly) installing malicious apps.

10. REFERENCES

[1] Android. Android dashboard. `http://developer.android.com/about/dashboards/index.html`, Jan. 2014.

[2] Barrera, D., Clark, J., McCarney, D., and van Oorschot, P. C. Understanding and improving app installation security mechanisms through empirical analysis of android. SPSM '12.

[3] BlackHat, B. Android master key - bluebox black hat talk. `https://media.blackhat.com/us-13/US-13-Forristal-Android-One-Root-to-Own-Them/-All-Slides.pdf`, Jan. 2014.

[4] Bluebox. Android master key - bluebox. `http://bluebox.com/corporate-blog/bluebox-uncovers-android-master-key/`, Jan. 2014.

[5] Chia, P. H., Yamamoto, Y., and Asokan, N. Is this app safe?: A large scale study on application permissions and risk signals. WWW '12.

[6] d'Heureuse, N., Huici, F., Arumaithurai, M., Ahmed, M., Papagiannaki, K., and Niccolini, S. What's app?: A wide-scale measurement study of smart phone markets. *SIGMOBILE Mob. Comput. Commun. Rev.* (Nov. 2012).

[7] DigiCert. Digicert - certificate transparency. `http://www.digicert.com/news/2013-09-24-certificate-transparency.htm`, 2013.

[8] Eckersley, P. Sovereign key cryptography for internet domains. `https://git.eff.org/?p=sovereign-keys.git;a=blob;f=sovereign-key-design.txt;hb=HEAD`, 2011.

[9] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. CCS '11.

[10] Felt, A. P., Ha, E., Egelman, S., Haney, A., Chin, E., and Wagner, D. Android permissions:

User attention, comprehension, and behavior. SOUPS '12.

[11] Gibler, C., Stevens, R., Crussell, J., Chen, H., Zang, H., and Choi, H. Adrob: Examining the landscape and impact of android application plagiarism. MobiSys '13.

[12] Google. Android and security. http://googlemobile.blogspot.de/2012/02/android-and-security.html, Feb. 2012.

[13] Google. Android app signing. http://developer.android.com/tools/publishing/app-signing.html, May 2014.

[14] Grace, M., Zhou, Y., Zhang, Q., Zou, S., and Jiang, X. Riskranker: Scalable and accurate zero-day android malware detection. MobiSys '12.

[15] Greenwald, G. *No Place to Hide: Edward Snowden, the NSA and the Surveillance State*. Penguin Books Limited, 2014.

[16] Haber, S., and Stornetta, W. S. How to time-stamp a digital document. CRYPTO '90.

[17] Kim, T. H.-J., Huang, L.-S., Perring, A., Jackson, C., and Gligor, V. Accountable key infrastructure (aki): A proposal for a public-key validation infrastructure. WWW '13.

[18] Laurie, B., and Kasper, E. Revocation transparency, 2013.

[19] Laurie, B., Langley, A., and Kasper, E. RFC 6962: Certificate transparency, June 2013.

[20] Petsas, T., Papadogiannakis, A., Polychronakis, M., Markatos, E. P., and Karagiannis, T. Rise of the planet of the apps: A systematic study of the mobile app ecosystem. IMC '13.

[21] Rastogi, V., Chen, Y., and Jiang, X. Droidchameleon: Evaluating android anti-malware against transformation attacks. ASIA CCS '13.

[22] Ryan, M. D. Enhanced certificate transparency and end-to-end encrypted mail. NDSS 2014.

[23] Vasco.com. http://www.vasco.com/company/about_vasco/press_room/news_archive/2011/news_diginotar_reports_security_incident.aspx, 2011.

[24] Vidas, T., and Christin, N. Sweetening android lemon markets: Measuring and combating malware in application marketplaces. CODASPY '13.

[25] Wu, L., Grace, M., Zhou, Y., Wu, C., and Jiang, X. The impact of vendor customizations on android security. CCS '13.

[26] Yan, L. K., and Yin, H. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. USENIX Security'12.

[27] Zhou, W., Zhang, X., and Jiang, X. Appink: Watermarking android apps for repackaging deterrence. ASIA CCS '13.

[28] Zhou, W., Zhou, Y., Jiang, X., and Ning, P. Detecting repackaged smartphone applications in third-party android marketplaces.

[29] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium* (Feb. 2012).