RWTH Aachen

Department of Computer Science
Technical Report

Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms

André Egners, Björn Marschollek, Ulrike Meyer

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2012-07

RWTH Aachen · Department of Computer Science · May 2012

Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms

André Egners, Björn Marschollek, Ulrike Meyer

Research Group IT Security
UMIC Research Center
RWTH Aachen, Germany
Email: {egners, meyer}@umic.rwth-aachen.de
bjoern.marschollek@rwth-aachen.de

Abstract. In the past research on smart phone operating system security has been scattered over blog posts and other non-archival publications. Over the last 5 years with advent of Android, iOS and Windows Phone 7, an increasing amount of research has also been published in the academic sphere on individual security mechanisms of the three platfroms. However, for a non-expert it is hard to get an overview over this research area. In this paper, we close this gap and provide a structured easy to access overview on the security features and prior research of the three most popular smartphone platforms: Android, iOS, and Windows Phone 7. In particular, we discuss and compare how each of these platforms uses sandboxing and memory protection, provides code signing, protects service connections, provides application shop security, and handles permissions.

1 Introduction

As a result of the broad acceptance among users, Android, iOS and Windows Phone 7 have presented themselves as a interesting new attack target of all sorts. Nowadays, devices are considered always on, and they carry a multitude of sensitive and personal information. All platforms attract increasing attention from malware programmers trying to make a profit, e.g., by premium SMS or identity theft. Since the operating systems are relatively new, information about their security features as well as vulnerabilities is mainly scattered around the Internet. For instance, a lot of work has previously been published in blog posts or is available as talks at more practical IT Security conferences without proceedings such as DEFCON, Black Hat, and the Chaos Communication Congress. In roughly the last five years in creasing amount of work is also published in the academic sphere.

In this paper, we provide an overview over the most prominent smartphone operating systems iOS, Android, and Windows Phone 7. In particular, we discuss the advantages and disadvantages of how each of these platforms uses sandboxing and memory protection, provides code signing, protects service connections, provides application shop security, and handles permissions. We conclude with a high-level risk analysis providing a condensed and structured initial access to the state-of-the-art in smart phone operating system security.

2 Threats & Protection Mechanism

This section provides an overview on attack vectors and threats for smartphone operating systems and introduces the protection mechanisms applied in current smartphone operating systems. Many of the attack vectors and threats are well-known from desktop computers. However, some of the threats and attack vectors are smartphone-OS-specific. We do not consider threats and attacks on the baseband level of the devices, but rather focus on the application level, i.e., the respective operating systems.

2.1 Attack Vectors

This section is going to introduce the most important attack vectors for smartphone platforms. There is a wide range of ways attackers can use to attack phones.

Applications: By opening up for third-party applications, smartphone platforms expose themselves to the most severe source of attacks of all: Running code that has not been developed by a person or company the platform vendor trusts. Depending on the platform, this can be native or interpreted code, or even both. Bad-natured developers are targeting smartphones with viruses, worms, spyware, and other malware of all kinds. While this threat is well-known from desktop computers, its a new threat for mobile devices, which until recently shipped with and ran pre-installed applications only.

SMS, MMS and email: Sending a malicious SMS to a vulnerable phone only requires knowledge of the victim's mobile phone number (e.g., guessing a valid phone number is easy). SMS messages with binary content will not even be displayed to the user on most mobile phones, but be handled directly instead. Recent attacks [31] have for instance shown that it is possible to knock a device off the cellular network or to perform even stronger denial-of-service attacks by making the phone reboot. In addition, SMS and email can contain links to malicious websites that, e.g., exploit browser vulnerabilities. Email may also carry harmful attachments that may exploit vulnerabilities of the operating system itself or an installed viewer application.

The Internet: One of the primary sources of attacks against desktop computers is the Internet and the same holds true for smartphones. Attackers can be able to communicate with infected devices directly; either by using an application to download content (e.g., instructions) from a server or by using a push notification service offered, e.g., by iOS, Android, and Windows Phone 7. Vulnerable web or media file browsers can be a serious source of attacks as well, as vulnerable libraries tend to be borrowed form the desktop world (e.g., the WebKit engine[1]). Infected content may exploit a browser vulnerability or harm the system by offering manipulated files being opened by the system automatically without the user's consent (e.g. PDF files on iOS).

Wi-Fi, Bluetooth, and other PAN Radio Technologies: From a hardware perspective, almost all of today's smartphones feature a Wi-Fi module and are able to create (ad-hoc) networks for data exchange and communication with each other. However, most software vendors do not yet implement this feature or limit it to providing a network that other Wi-Fi clients can connect to in order to share the smartphone's Internet connection. Bluetooth has proven to be insecure in various versions, and attacks have been published that, e.g., allow for SMS manipulation, phone book manipulation, and phone call initiation [6].

[1] `http://www.webkit.org/`

Compromised Desktop Computers: Another attack vector is a compromised desktop computer. Users voluntarily connect their devices to a computer, be it to charge their battery or to perform backup and synchronization actions. As it should be easy to detect the connection to a smartphone, it might be possible to attack the device via an infected desktop computer.

Jailbroken Devices: All smartphone platforms employ numerous security mechanisms to protect the device and its owner. However, these mechanisms cut down the freedom of the users in some aspects. Users may for example want to run software that is considered malicious or is simply not approved by the operating system vendor. In order to do so, the user has to *jailbreak* the device, i.e. to disable at least some of the security features offered by the operating system. An attacker may specifically target jailbroken phones.

2.2 Threat Categories

We categorized the most relevant threats according to their target into *owner* threats (OT), *platform* threats (PT), threats to *other users* (OUT), and *mobile network operator* threats (MNOT).

[OT-1] Premium Services: Smartphones are not only a valuable target because of the data they store, but also because of the fact that they are phones. They can establish phone calls and send SMS or MMS messages to expensive service numbers in foreign countries. If an attacker manages to install a dialer or similar malware on a device, this can induce costs. Dialers and SMS trojans have been a common form of mobile phone malware for a long time [36].

[OT-2] Identity Spoofing: Using a mobile phone or a smartphone as a proof of identity has become increasingly popular in the past years. A way to recover a lost Google account password is to have Google send an SMS with a confirmation code to the owner's phone number. Some banks employ similar two-factor authentication by sending codes to a specified mobile number (known as MTAN) in order to verify the identity of the customer before accepting a transaction. A mobile version of the Zeus trojan has been found, which was able to intercept SMS messages sent by the bank and disrupt the two-factor authentication. It required an attacker to gain control of both the victim's credentials and the mobile phone [36].

[OT-3] Denial of Service: Especially in time-critical appliances, users increasingly depend on their phones. They store important contact information and ensure constant reachability. Not being available via phone, email, or SMS may cause delayed response to certain events due to a lack of knowledge. For instance, stock-exchange brokers may be unable to sell shares well-timed and thus lose money.

[OT-4] Threats to Privacy: Most smartphone attacks target spying on the user and collecting data, which compromises the user's privacy. Because smartphones provide a mass of information about the user, attackers are able to obtain a detailed user profile. This includes the contact details calendar information of the user, the contact details of his friends and business partners, which applications are installed on the phone, the data these applications collect and store, as well as the content of SMS, MMS, and email messages. A user can be tracked by using the different location features of the device. In combination with unique

identifiers that are available on many platforms, location reports can be associated to a particular phone. Even worse, if the platform allows developers to read the phone number associated with the inserted SIM card, all harvested data can be associated with the user himself.

[PT-1] Threats to Integrity: In this context, *platform integrity* does not refer to the unmodified state of the platform in the cryptographic sense, but to an operating system that has not been tampered with, is not compromised, and still runs all of its security mechanisms in the way they were intended by the vendor. If a platform is compromised by disabling some or all of the security features, the device is often referred to as *jailbroken*. On most platforms, the circumvention of the code signing requisition is entailed. However, other security mechanisms may be suspended, e.g., manipulating the application permission checks on Android, that users may not be aware of, even if they are intentionally jailbreaking their devices.

[PT-2] Account Hijacking: The three operating system vendors, Google, Apple, and Microsoft, offer a free cloud account with their system. These accounts allow shopping in the respective online shops and are used to personalize associated devices. Software licenses are stored for the account the purchase was made with. If a device has, e.g., been restored to factory settings, it is easily possible to reinstall all applications associated to this account. All vendors ask for contact information of the account owner, e.g., Microsoft and Google offer services like email and calendar. Consequently, appointments and details of the user's contacts can be stored in the account. Both companies offer to synchronize this data to the accounts in order to be able to restore them to the phone at a later time. This turns an account for the cloud services into a valuable target for attackers.

[OUT-1] Affecting Contacts in the Address Book: Today, most operating systems store contacts in an address book and assign multiple entries to one contact. An entry may contain the cellphone number, landline number, postal address, email address as well as notes and other data like the date of birth. This information is not only a valuable target for address databases of spammers, but can also be used by malware to spread to other devices.

[MNOT-1] Carrier or an Enterprise Network: Malware-infected devices may perform attacks on mobile communication networks and devices thereon. This could inflict severe damage or outages on the networks for instance by trying to spread to other devices. The creation of mobile botnets is an interesting angle since distribution of malware is more easy in the age of apps.

Georgia Weidman created a proof of concept for a recent Android version [44]. In [45], Xian et al. describe how it would be possible to construct an Android-based Botnet. Besides the proof-of-concept construction, the authors also elaborate on mitigation techniques. Also attacks on core components of mobile communication network, e.g., the HLR (Home Location Register) have been shown by Traynor et al. in simulations [42].

2.3 Common Protection Mechanism

The three operating systems iOS, Android, and Windows Phone 7 all employ some form of sandboxing and isolation, code signing, specify permission models for applications, support service connections that require special protection, and offer application downloads via market places. In the following we briefly discuss these features.

Application Permission Models: All platforms in question employ a permission model for applications that restricts the access of applications to the content and resources available on the smartphone. This for example includes accessing the address book or being able to establish an Internet connection. The decision which permissions an application gets is left to the user and has to be set during installation.

Sandboxing and Isolation: In order to enforce the permission model, all platforms follow the principle of least privilege. All applications are run in a sandbox that separates running processes from each other and encapsulates access to system resources.

Code Signing: Another central protection mechanism of all three platforms is code signing, i.e., signing an application with a digital signature for the package or its contents. This signature proofs the identity of the creator of an application as well as package integrity. All platforms check the signature on each launch of the application.

Service Connections and Remote Functionality: Performing actions remotely is one central security feature of all smartphone platforms as they progress into becoming *cloud terminals*. Vendors may invalidate or delete applications, wipe the entire device in case it is stolen and more. Such a connection exists for all three platforms, but neither Apple, nor Google or Microsoft extensively document the exact purpose and protection of this connection.

App Store, Android Market, and Windows Phone Marketplace: The software shops play a very central role on all platforms, as they are the primary or even exclusive source of third-party applications for the systems. A client application on the devices is used to browse the shop, display details of offered applications, process the actual purchase, and download. Billing information is typically stored along with the user's account. Account credentials and the purchase is done using the deposited payment method, e.g. via credit card.

3 Related Work

Besides the comprehensive Android developer documents [3] [2] and various other articles on the web [37], [35], [33], [26], [24], [22], [19], [20] are discussing a broad range of security issues of the Android operating system. Also studies with a general focus on Android exist, e.g., by Enck et al. [16], [17] and Shabtai et al. [39]. Other researchers explicitly focus on a single security mechanism, e.g., the permissions used by Android apps [18], [41], [38], [13], [1], the construction of Android botnets [46], and inter-application communication [10].

Research on iOS is not as broadly available as for Android mostly due to its closed nature. However, on the web there are articles discussing security issues, e.g., covering security parctises [9] and malware [11], [21], [30] Besides the official developer documents [4] and review guidelines of the App Store [5], other research exists on the sandbox [7] and research by Charlie Miller [28], [29]. A very recent security evaluation of the iOS 4 operating system has been published by Dion A. Dai Zovi [14].

The available research for the newest of the platform, Windows Phone 7, is most scarce. Again, developer documents [25] are available. Explicit details on the system architecture can be found in [12], as well as its security architecture [27]. As jailbreaking is also important for Windows Phone 7, details on the first known jailbreak, ChevronWP7, can be found in the article by Tom Warren [43].

4 Sandboxing & Memory Protection

Each of the three systems employs both sandboxing, i.e., confining apps into respective separate logic domain which offer an additional level of protection for the host system, and memory protection, i.e., low level security such as address space layout randomization (ASLR) making it considerably harder to reliably exploit vulnerabilities.

4.1 Android

Application-level security on is achieved by a sandboxing mechanism using UNIX user IDs to enforce restrictions. Each application has a unique user assigned at installation time, i.e., `app_` as prefix followed by a consecutive number. The package contents are extracted into a directory below `/data/`. This directory is named the same as the application's bundle identifier. Only the newly created user that is associated with the application is allowed to read and modify data inside this application directory. The same technique is used when the application process is launched inside its own instance of the Dalvik VM [15] with the associated user name and the according rights. Hereby, the application is isolated on the system and jailed in a sandbox.

The only way for applications to read each other's data is to be assigned the same user ID, i.e., it can be requested by an entry in the manifest file. However, the system is only going to acknowledge the request if the requesting package is signed by the same certificate as the application whose user ID is requested. The benefit of shared user IDs is that two applications with the same user ID can read and write each other's data directly in terms of file system permissions.

Besides stack randomization Android (up to current versions) does not employ any other memory safeguard, such as address space layout randomization (ASLR) or non-executable memory pages by using the execute-never (XN) bit of the ARM CPU.

4.2 iOS

Every application on iOS is executed in its own sandbox. For this purpose, Apple uses a hosted hypervisor, i.e., a virtualization technique that uses device drivers of the system. The sandbox is realized as an extension for the TrustedBSD framework called *Seatbelt* [7]. It allows profiles to precisely define the permissions of applications. The same mechanism was introduced in desktop Mac OS X in version 10.5. Each application is installed in a dedicated directory that is named using a globally unique identifier (GUID). Below this directory, there are the directories `Documents`, `Library`, and `tmp` and the `.app` package including resources, binaries, and the code signature. The application is allowed to write its own directory, however, the package is read-only.

In terms of memory protection iOS uses the security features of ARM processors, the so-called *XN (execute never) bit* controlling whether a memory page is executable. iOS makes heavy use of XN and protects every page on the stack, as well as on the heap by marking them non-executable. No page can have the permissions read, write and execute (RWX) at the same time; only `RW_` and `R_X` are possible [29]. This makes sure that no application can write code to memory and then execute it later.

A frequently used attack type to circumvent non-executable memory are *return-to-libc* attacks which use buffer overflows to jump to, e.g., the C library which provides useful routines for attackers. iOS 4.3 introduced address space layout randomization (ASLR) which randomizes the memory addresses of libraries each time they are loaded. On 64-bit systems, this makes return-to-libc attacks very unlikely, but on 32-bit systems, this does not significantly improve the situation [40]. Yet, real multitasking is still impossible for third-party applications and reserved for the system itself.

4.3 Windows Phone 7

Windows Phone 7 isolates running applications such that they cannot influence each other and cannot interfere. The operating system guarantees that there always are enough resources available for the application to run by full priority to the foreground application. Isolation also ensures confidentiality and integrity of the application data as the data cannot be modified or read by other installed applications. Using the sandbox, the system also prevents applications from accessing native APIs.

The system provides a host process for the application to be launched in. Before each execution, the code is integrity checked by the runtime. Only if the check confirms the validity of the code signature, the software will be allowed to run and is supervised by the *Execution Manager*. It monitors the application's usage of system resources and may terminate the process if it considers the application to be misbehaving or unresponsive.

The Windows Phone 7 security architecture uses the basic principles of isolation, least privilege, and introduces *chambers* as a concept [27]. Each chamber provides a security and an isolation boundary inside which processes can run. Different security policies define the chambers and create a security level hierarchy. There exist four different types of chambers (and thus basic security levels). The most permissive chamber is the *Trusted Computing Base* (TCB) chamber. It allows unrestricted low-level access to almost all resources. The kernel and device drivers run in TCB chambers. The exclusive capability of the TCB chamber is the possibility to modify the security policy. This separates it from the *Elevated Rights Chamber* (ERC), which is intended for user-level device drivers and services that provide phone-wide functionality or shared resources that are going to be used by other applications. Consequently, applications providing non-global functionality, run in the third chamber, *Standard Rights Chamber* (SRC), which is the default for pre-installed system applications. Finally, there is the *Least Privileged Chamber* that every third-party application will run in. These chambers are configured individually by the system at installation of the application.

Applications are typically executed as managed code only, the exception are phone vendors which are allow to program native code. Every application runs bytecode in a virtual machine provided by the Common Language Runtime (CLR) which translates the code into Common Intermediate Language (CIL). When the user starts an application, a just-in-time compiler creates native code.

4.4 Summary

All of the sandboxing mechanisms have proven to be only partially secure. It is impossible for apps to break out of the sandboxes on their own, however, all platforms allow app interaction with daemons, libraries or frameworks that are running natively, and often even privileged which can ultimately lead to the exploitation of vulnerabilities which in turn lead to privilege escalations. In terms of memory protection, Android has the most room for improvement. Neither does Android protect its stack or heap, i.e., both are executable, nor does it use techniques such as ASLR. The stack used by Android is randomized, but no additional heap base randomization is employed. In contrast du Android, Apple's iOS heavily uses the XN bit and does not allow app memory pages to be writable and executable at the same time. Since iOS 4.3, ASLR is enabled, however, due to the lack of entropy on mobile devices, it has already been defeated [2]. Windows Phone 7 mostly runs managed code, but for some hardware vendors it is also possible to run native code. However, if unsigned code execution is possible, it also runs native code outside of the Common Intermediate Language.

5 Application Code Signing

Signing an application creates a digital signature for the package contents. This helps to prove the identity of the author of an application package. All platforms perform signature checks, either prior to launching or installing the app.

5.1 Android

While the benefits of code signing are obvious, Android takes advantage of only a few of them. If code signing used, the system only allows the execution of code from signed application packages. However, Android does not employ a central trusted certification authority signing all developer public keys, but instead allows self-signing.

This results in developers being able to generate many different key pairs and the corresponding certificates to sign their applications. So, the most important use of Android code signing is for the developer to be able to update existing applications, i.e., updates must be signed with the original signature key. Moreover, it allows to prove that two or more applications originate from the same author. This enables secure data sharing between the author's applications. For example, he can request the same user ID for applications signed with the same certificate or grant special permissions only to his own applications.

The lack of a central certification authority makes it easy to develop and distribute malicious applications. Installations are not only possible via the official Google Market, but also by copying the application package to the SD card or by having the user download the application directly via a browser. To protect

[2] `http://www.webcitation.org/65cMyk1N9`

the user, installation from non-market sources is disabled by default, but this setting can be easily overridden in the system settings. A malicious developer does not need to register with Google and there is no chance to link a public key to a developer. If the malicious developer wants to distribute his applications via the official Market, he can do so anonymously using a prepaid credit card to pay the registration fee of $ 25.

A major flaw of the code signing process is the fact that Android does signature checks only on code bundled with the application packet but not on all code that is executed. Thus, any application is able to download a binary from the Internet and execute it within its standard permissions.

5.2 iOS

Apple requires developers for their mobile platform to register and pay a fee of $ 99 per year. In return, Apple's certificate authority issues a certificate for the public key of the iOS application developer who can now sign his applications. The Apple-signed certificate is embedded into the application bundle such that the device is able to verify the code signature and check if the public key has been signed by Apple. In fact, signing is not only an option, but a requirement. Prior to executing code, iOS checks if the signature is valid, if the certificate has not expired yet, and if the public key was signed by Apple. Only if all these criteria are met, the application is allowed to run on the device.

Code signing provides authentication, because the platform is able to check if an application originates from the claimed developer, and integrity, because it can be verified that the code has not changed after creating the signature. Additionally, it prevents repudiation by the developer, as a valid signature can only be generated with access to the corresponding private key. The iOS kernel enforces the code signature check on every call of the system function `execve()`. It inspects the binary checking for `LC_CODE_SIGNATURE` segments. If it does not find a signature for the address range, the signature from the binary will be loaded and the pages will be verified.

An additional use case of code signing is to prevent libraries being directly from the memory, because the signing is linked to the memory page permissions. They can still be loaded from the disk, but must be signed in order to have executable memory pages [29].

5.3 Windows Phone 7

Windows Phone only allows installing an app package, if it carries a valid Microsoft signature. The installation is performed according to the information given in the apps's manifest file. This is the only way for an app to influence the installation process; it cannot run code during installation. The manifest file will be inspected in the review process that is done by Microsoft prior to the admission of an app to the their *Marketplace*. Also, apps cannot influence update or un-installation processes. The decision whether any of these actions are performed, is entirely left to the user.

The licensing mechanism of the platfrom also supports trial versions with a limited lifetime using licenses with an expiration data. Moreover, the Marketplace frequently checks for license revocations. If a license turns invalid or an apps's license is revoked, the Marketplace can initiate the deinstallation of the app.

Windows Phone 7 does not require the developer to sign his code, but rather leaves this task to Microsoft. After reviewing an app on submission and deciding to whether approval is given, the app package is will be signed by the Microsoft generating a valid trusted signature.

5.4 Summary

The code signing mechanisms are implemented very differently on the three platforms. Apple and Microsoft use a certification authority, while Android's solution is based on self-signed certificates. Android thus only enables a developer to prove that he is the author of particular applications and may therefore update the application or share data between two or more of his applications. Android's signing process does not protect the platform against malicious code. iOS enforces signature checks at every system call that starts a new process and hence outscores Windows Phone in this discipline.

6 Service Connections

Android, iOS, and Windows Phone 7 each use a persistent connection to its operating system vendor Google, Apple, and Microsoft. This connection is typically used for management purposes, e.g., synchronized user accounts, installation and removal of apps, as well as to locate lost or stolen devices.

6.1 Android

The service connection of Android is based on the GTalkService which is a TCP connection using the XMPP protocol with SSL encryption [32]. The payload is encoded in Google's Protocol Buffer[3] format, which is used by the official Google Market app to communicate with the Google servers. A so called *secret code* can be dialled in the telephony application to start the a GTalkService connection monitor. Secret codes can be defined in the app's manifest file by registering for the `android.provider.Telephony.SECRET_CODE` intent (see [3, 2]).

Google's *cloud-to-device* messaging service is also using this connection to deliver its contents. Downloading and installings apps is not initiated by the Market application on the user's device, but rather remotely by Google's servers by sending an INSTALL_ASSET intent (see [3, 2]) to the device. Security researcher Jon Oberheide discovered [33] that this intent is invoked via the GTalkService connection, just as the REMOVE_ASSET intent used to remotely remove a particular applications. This enables Google to remotely install and uninstall applications at will. The `REMOVE_ASSET` intent is sent to the device indicating the app to be removed. After successful removal, a message is left in the notification bar of the device. Google may have the option to remove applications silently, however, there has been no indications for the existence of this option, yet. Handling the `REMOVE_ASSET` intent works similar to the `INSTALL_ASSET` intent. A malware app may want to prevent the system from being able to remove it. The software would

[3] `http://code.google.com/p/protobuf/`

have to prevent the system from receiving the `REMOVE_ASSET` intent, which is sent over the SSL-encrypted connection. So, prevention would only possible, if the malware was able to break SSL.

Google has been the first vendor using this type of remote wiping. In June 2010, Google remotely removed two proof-of-concept applications by Jon Oberheide [33]: *Rootstrap* was an application that was able to download and install a rootkit for Android and *Twilight Eclipse Preview* which was an alleged preview of an upcoming cinema movie, but actually a cover for its hidden Rootstrap functionality. Besides remote installation and removal of apps the GTalkService connection is also used to perform important and highly security-relevant operations, e.g., wipe the user's data. In his analysis, Oberheide points out that (apart from the SSL encryption of the entire connection) there is no cryptography used in the install command. Thus, an installation request could be spoofed if the attacker manages to manipulate the SSL connection.

Prior versions of the Android Market ($\leq$ Android 1.6) can be convinced to disable the SSL certificate check for the communication with the Market servers. It requires an attacker to disable system security by setting the system property *ro.secure* to 0 and *vending.disable_ssl_cert_check* to `TRUE`. This enables eavesdropping on the connection using an SSL man-in-the-middle software such as Moxie Marlinspike's *sslsniff*[4] in combination with a self-signed CA certificate. However, newer Market versions address the issue by forcing the SSL certificate check for installation requests. As of the version shipped with Android 2.2, the Android Market app ignores this flag and does not trust any self-signed certificates.

6.2 iOS

Since iOS 3.1, Apple provides several security features for its devices in case it gets lost or is stolen. The web-based *MobileMe* service offers to display the current location of the device on a map, lets the user play a sound and display a message, invokes the passcode lock immediately, or completely wipes the device and deletes all user data. All these services are remotely available and do not require physical access. However, for devices with 2G/3G capabilities, the remote commands will only succeed if the SIM card is inserted and the phone is connected to the network for identification purposes. This functionality is realized using the persistent SSL-encrypted service connection Apple keeps to every individual device, e.g., using Wi-Fi or the cellular data network.

Another feature that uses this connection is the *Apple Push Notification Service* (APNS), which is a unidirectional communication from the service to the device. It is used by Apple and third-party developers to send notifications to applications, e.g., to inform the user of a new message in a social network. This technique is important to deliver information to applications that are not currently running. Devices are identified to the APNS using so-called device tokens that are sent to the provider by the device as soon as the user enables to app to receive push notifications. This token is then sent by the provider to the push gateway, along with the actual payload.

To prevent the unauthorized use of iOS devices, Apple requires an *activation* via iTunes. When the device is first used, it must be connected to a computer

[4] `http://www.thoughtcrime.org/software/sslsniff/`

running Apple's iTunes software. In non-activated state, iPhones can only call emergency numbers. Access to any other system component is denied until successful activation. Apple requires users to register a free account (*Apple ID*). Without such an account, the activation cannot be done and the device refuses to operate beyond the emergency call screen. The activation status is monitored by the *lockdownd* daemon. Also, activation can typically only be done with a supported carrier SIM card. SIM-locked phones will not be activated with a SIM card of an unsupported carrier. In the activation process, iTunes generates an activation token that depends on several hardware and SIM features. This token is sent to Apple's activation server, which is going to validate it, generate a *wildcard ticket*, and send this ticket back after digitally signing it and encrypting it with the Tiny Encryption Algorithm [23]. The activated device stores this ticket, also containing the token, named *wildcard_record.plist* ticket. Circumventing the activation process is possible. One option is to patch the running `lockdownd` in RAM. A major downside of this technique is that push notifications and the YouTube service are broken, because both require a valid device token, which is not available if the activation protocol has not been performed correctly. The better way to permanently *hacktivate* devices is using the Subscriber Artificial Module (SAM) software method, where iTunes is used to perform the activation protocol just in the way it is supposed to, but an official SIM card is simulated using the SAM. iTunes then generates a valid device token. Activation is not to be confused with SIM unlocking. Activated phones are still SIM locked, as activation does not affect the baseband, which enforces the SIM lock. Activated, but locked phones can use all device functions except for SIM-related ones such as making phone calls or sending short messages.

6.3 Windows Phone 7

Microsoft's platform offers several remote security functions via its persitent service connection holds to Microsoft usign cellular or Wi-Fi networks. After logging into the Windows Phone website[5], it is possible to have the phone displayed on a map or to ring it. In addition, the device can be locked remotely, i.e., a four-digit code can be entered that is required to unlock the phone again. Without the code, only the enclosed message can be read. If the device cannot be found, the user has the option to whipe his device.

Windows Phone 7 also supports push notifications via the service connection, in particular messaging applications which are currently suspended. This is an important feature, as Windows Phone 7 does not allow multitasking and there is always only one active application in the foreground. Three types of push notifications are supported: *Raw* notifications are those received when the target application is currently running. They will be handled by the application itself. *Toast* notifications are simple text messages that are displayed to the user at the top of the screen while the application is either in an active or suspended state. *Tile* notifications can change the tile of the application, i.e., the icon on the home screen.

Each app requesting to receive push notifications must register with the Microsoft server by requesting a unique URI that will be used by the sending

[5] `http://windowsphone.live.com/`

service to identify the device. This registration is only needed once and can be done silently at the first application launch. The service sends the URI and the corresponding payload via HTTP to the Microsoft Push Service which will then take care of the delivery of the message. The push service is not providing reliability, i.e., devices that do not have an always on connection might never receive the message, or the delivery may be delayed.

The raw notifications are very interesting, because they deliver raw bytes to an application that is currently running. For instance, a weather application can use push notifications to inform the user about new weather conditions at a user-favored place. Instead of informing the user that there is an update and that he has to reload it manually, the new data can be sent along with the notification. This can also be used to send any kind of data to the phone, however, the payload size is restricted to 1 KB per raw message.

6.4 Summary

We observed that all systems have carefully implemented the service connection to the vendor. Theses connections are encrypted and the certificates are checked for validity, which results in proper authentication of the remote station. All systems support the installation of additional root CA certificates, but none allows the use of user-installed certificates for the service connection. This prevents man-in-the-middle attacks using self-signed CA certificates.

7 Software Distribution

7.1 Android

The installation of applications to Android devices is done by the *PackageManager* service (`pm`). As a service, it does not use any user interface and performs every single installation and uninstallation action, keeps track of what applications are installed and of the state of these applications. Moreover, it does not take care of any permissions, but instead assumes that they have been approved by the user. This makes it necessary that the PackageManager checks if the application that calls `pm` with an installation request was granted the system permission `INSTALL_PACKAGES`, which has protection level 2. As such, the requester must be signed with the same certificate as the application that defines the permission, which is in this case, is the Android system itself. So, only vendor-approved applications can install other applications. On a standard Android system, the only application holding the `INSTALL_PACKAGES` permission is the *PackageInstaller* which belongs to the core system

The Android Market divides the available software into applications and games, and subdivides them into 26 and 8 detailed categories, respectively. The user can browse the categories and is displayed the name of the author of the application and its price. The Market application presents a description and screenshots. If the user decides to install the application, he taps the price tag. The button then turns into an OK button and the text label above then says "Accept permissions", while the permissions requested by the application are unveiled.

The fact that one and the same button is used to show the details of an application and to approve security-relevant permissions is a major design flaw. Due to Android's multitasking capabilities, some components of applications are allowed to run in the background and can take up resources. If a device is almost out of memory and starts reacting sluggishly, the user might tap a second time thinking that his first tap has not been recognized or he barely missed the button. When the system finally becomes responsive again, it interprets the second tap and installs the application.

The Android Market application on the phone communicates with the Market using Google's Protocol Buffer format over HTTPS. The Protocol Buffers project is open-source and defines a serialization format and an interface description language. Despite the fact that details on the Market API are not disclosed, it was possible to reverse-engineer a large part of it. Developers on Google Code, created a project called *android-market-api*[6]. It provides a third-party implementation of a subset of the Market API, such as loading comments for an application in the Market.

However, the implementation does not provide the means to download application packages. This can be done by a simple HTTP GET request and needs the following data: The authentication token with the Android Market of a legitimate user; the user ID that belongs to the same account as the auth token; the asset identifier of the package that is supposed to be downloaded and a valid Android device ID.

One option to obtain the identifier and token is to take a look at the traffic a device produces when installing an application from the Market. After receiving the `INSTALL_ASSET` intent, the device performs a GET request. The user ID can be read from one request and stays the same for all following ones. Unfortunately, there is no other way to obtain the user ID yet, because it seems to be linked to the account name in an undisclosed way. The authentication token and the device ID can also be read from the Android device, using the permissions `GET_ACCOUNTS` and `USE_CREDENTIALS`.

Besides the Android Market, there are several other application shops for Android that follow a similar approach, i.e., offering applications via a client application. The platform makes this possible by allowing third-party applications to install other packages with the user's consent. The alternative shops must equalize their disadvantage that the number of applications they offer is significantly smaller than the official Market's. For example, many accept other payment methods like PayPal or Amazon Payments. The most important alternative shops are GetJar, AndroidPIT, and the Amazon Appstore. After all, the alternative software shops act inside the granted permissions by the user and they are prevented by the system to install applications on their own without asking the user. Installations are eventually handled by the Android system service PackageManager. Hence, they are not relevant from a security perspective

Application packages that have been copied to the device can be installed by using a file browser and web browser. When tapping an `.apk` file in the file browser, the PackageInstaller activity is launched and the user is asked to grant the permissions that the application requires. If the URL of an application package is entered, the browser automatically downloads the file. As soon as the

[6] `http://code.google.com/p/android-market-api/`

downloads completes, the user can tap the file in the notification area and the PackageInstaller is launched. By default, however, the installation of non-market sources is disabled and must be explicitly enabled by the user in the system settings.

Another option is to use Android Debug Bridge (ADB) shipped with the Android SDK. The `adb` command allows the user to install an `.apk` from the computer to a USB-connected device by running `adb install /path/to/application.apk`. To use this feature, the user must enable the debugging mode via USB in the system settings and connect his device via a USB cable.

Web-Based Market: In February, 2011, Google introduced a new additional web-based interface for its Market application to be usable from a desktop computer as well. Users can now access the market via a browser from a desktop computer and buy and install applications remotely.

Google installs software that has been bought via the mobile Market application using the `INSTALL_ASSET` intent to Android handsets, which means that the installation is triggered remotely by Google, not locally by the Market application. This is also true for applications purchased from the web. As soon as the user accepts the permissions requested, his handset starts downloading and installing immediately. No further interaction with and thus not even physical access to the device is required. It suffices to possess a victim's Google account credentials to remotely install software on his device. This makes Google credentials an even more valuable goal for attackers, because they can also access the victim's private data stored on his mobile phone by remotely installing malware. The web-based Market demonstrates that application installations are completely independent from the Market application; unlike for Apple's and Microsoft's markets, where the installation is a task that is exclusively performed by the respective market application.

Only a few days after the official launch of the web-based Market, Oberheide discovered a serious cross-site scripting (XSS) vulnerability in the Market website [34]. In the publishing interface, developers are asked for an application description. This text field allowed an attacker to enter JavaScript code that was executed on a customer's computer while browsing the application page in the web Market. Oberheide also demonstrated an exploit that achieved arbitrary code execution on the user's device.

To install an application from the web-based Market, the browser sends a HTTP POST request to `http://market.android.com/install`. The body of the request must contain four variables: The bundle identifier of the application that is to be installed, the field `xhr` (which is always 1), the device identifier of the target device and the authentication token for the Market. The latter two could be read from a cookie that was set by the Market on the desktop computer after logging in. The bundle identifier is very easy to obtain as the developer chooses it himself and even if the attacker is not the developer of the malware, the identifier is displayed in the web-based market. So the necessary HTTP POST request for installing a malicious application could easily be performed using AJAX. This does not directly give an attacker code execution, because the application is only installed but not launched directly. However, the attacker could register his malware for the `PACKAGE_ADDED` broadcast intent, which is sent whenever a new application is installed, and then just install another arbitrary application using the same technique. This causes the intent to be fired and the first application to launch. Thus, the goal of code execution on a device could be achieved without ever having access to the device itself or any user credentials. The victim just needed to browse the web Market and open the page of the malicious application. Google fixed the web-Market vulnerability by the end of February, 2011, by disallowing the cookie to be read from a script and fixing the XSS vulnerability.

Quality Management: Compared to the other two operating systems with a similar Market application, iOS and Windows Phone 7, the Android Market employs a very liberal policy. There is no further validation or review of the application, which makes it very easy to offer malware to a wide range of customers.

The Android Market's security model relies entirely on the community to identify and report malicious applications. This implies that there is a certain number of users required to *sacrifice* themselves and try out new applications. Users can flag an application as inappropriate using one of the explanatory statements "sexual content", "graphic violence", "hateful or abusive content" or "other objection". However, malware applications typically hide their malicious routines behind useful functionality; many users will not notice the secret features and never flag the application as inappropriate. One of the first malware applications found, *Droid09*, was phishing for bank accounts. In February 2011, Google removed a group of more than 50 malware-infected applications [26]. An approval process similar to the one Apple and Microsoft enforce for applications on their App Store or Marketplace, respectively, could have most likely prevented the admission to the official Android Market. Static code analysis would have uncovered the hidden functions as well.

One very important security feature is the so-called *safe mode* the device can boot to. Most devices enter this mode by holding a particular key pressed while booting. In safe mode, no third-party applications are allowed to run. This allows to clean an infected system without being interfered by protection mechanisms of the malicious software. However, there is malware that *roots* the phone (or especially target rooted phones) and copies itself to the system partition. This will reinstall the malware at the next boot.

Copy Protection: Until mid-2010, the Android Market offered the option to enable "copy protection" when posting an application. The difference between protected and unprotected applications was that unprotected packages were saved in `/data/app`, while the protected applications went to `/data/app-private`. From the first directory, the debug shell is able to read and pull the packages to a desktop computer, transfer them to another phone and run the application there. The `/data/app-private` directory cannot be read by the debug shell. However, on rooted phones the user can read any file. In fact, apart from this artificial separation, no further protection has been implemented. Even worse, once the applications have been pulled from a rooted phone, they are copied to `/data/app` on any other phone automatically at installation; where they can easily be read and distributed by everyone.

In July 2010, Google added the possibility for developers to query the Market server and see if the current user has actually purchased the application that is running [8]. This rudimentary licensing mechanism requires a data connection which may be unavailable at times, or blocked on purpose by user with rooted phones.

7.2 iOS

The very first iPhone generation did not allow any third-party applications to run natively on the device. Instead, developers were encouraged at the Worldwide Developers Conference 2007 to write *web apps*; applications that ran on the web, but mimicked the iOS UI elements and behaved like they were running natively on the device. An advantage of this policy is security, because no third-party code is run on the device. However, due to slow network speed (the classic iPhone did not support 3G networks), user experience suffered.

With the release of the iPhone 3G and iOS version 2.0, Apple opened the device for third-party applications. Developers could pay an annual $ 99 registration fee and distribute their applications via the new online software shop *App Store*. The client application for the App Store allows to purchase, download and install directly from the device. The App Store can also be accessed via the iTunes software from a desktop computer. All software that is bought for an iOS device is connected to the user's Apple ID which enables sharing it across devices. The App Store is the only Apple-legitimate source to install third-party software on the device.

Review Policy: Apple employs a strict policy of which applications are allowed in its App Store. Its App Store Review Guidelines [5] clearly state that Apple will "reject Apps for any content or behavior that we believe is over the line". Applications which Apple does not approve will be rejected. Common reasons for a rejection are that an application is malfunctioning, unstable, uses private API calls, or behaves otherwise maliciously. For the end user, this is a huge quality improvement and security advantage, because Apple performs a set of analysis techniques to unveil unwanted or malicious behavior and prevent large parts of malware to be released to the devices, albeit the review process is not bullet-proof.

During the review process, Apple performs a detailed analysis of the binaries submitted by developers. This analysis involves both automatic and manual tests for malicious behavior of any kind, private API calls and poorly programmed software that does not function or does not comply with Apple's human-interface guidelines. Moreover, trial versions and software that Apple simply does not want on the App Store will be rejected [5]. Apple's review process is known to be rather strict and hence created large customer confidence in the store. However, it does not protect from applications that serve a legitimate purpose, but use hidden malicious features, e.g., posting phone details to the Internet.

Copy Protection: Apple uses its *FairPlay* digital rights management to protect applications. Each application is encrypted using a *master key*, which is stored within the package in an encrypted fashion. *User keys* can be used to decipher the master key and eventually the application bundle. When a customer purchases an application, a new user key is generated to encrypt the master key, which makes user keys unique per user and application. Every device using an application must hold the user key in order to decrypt the respective application's master key. The keys are transferred to devices on synchronization with iTunes. Likewise, the user keys are stored in an encrypted manner, such that they can only be read by Apple software. In practice, users are allowed to install a purchased application to any number of iOS devices he owns.

However, FairPlay has been proven to be defective and can be circumvented. The application binary is the only part that is encrypted. Therefore, the program code in memory is unprotected, enabling attackers to use a debugger to suspend the program and create a dump of the memory to obtain the unencrypted binary code. Finally, the encrypted part of the binary is replaced with the plain counterpart and the application's manifest file. This file contains declarative information on the application and is modified such that the system does not try to decrypt the binary and signature checks are turned off, as the signature is invalid after the modification. There have been applications similar to Apple's official App Store that allowed downloading pirated applications for free. Of course, this only works if the device allows the execution of unsigned code, as the installer application would never be approved for the App Store and the unencrypted applications additionally carry an invalid signature. In particular the device must be *jailbroken* in order to turn off signature checks.

7.3 Windows Phone 7

Microsoft's implementation of an online software shop is called *Windows Phone Marketplace*. It acts as a gatekeeper for third-party software, being the one central source of third-party applications for their devices. The Marketplace is the only official way to obtain software; side-loading applications is not supported. Developers must submit their application package to Microsoft for review, before the application is published. In this process, Microsoft performs both a manual and an automatic analysis of the binary in order to exclude malware and find poorly implemented applications. Additionally, there are tests that confirm if the application is indeed written in the Common Intermediate Language (CIL), a completely stack-based, object-oriented and type-safe assembly language. The type-safety makes buffer overflows impossible and thus takes away a severe danger. This is, however, a constraint to the .NET framework. Standard C# allows unsafe methods by adding the `unsafe` keyword to the method signature, leaving the CIL and producing native code. Applications that use unsafe code will be rejected by Microsoft. Hence, the user is supposed to only see well-written, high-performance applications, and gain the confidence that Marketplace applications are of good quality.

If the application complies with the rules and Microsoft decides to approve the application, it is going to be digitally signed and an appropriate license will be issued. This license also provides a copy protection functionality to ensure that a legally purchased application cannot be transferred to another phone. In order to be able to post applications to the Marketplace, developers must register with Microsoft and pay an annual $ 99 registration fee. In return, Microsoft will sign their applications and issue licenses.

The Marketplace can conveniently be accessed from an application on the phone and via the Zune software for Windows PCs. Browsing the Marketplace requires a Windows Live ID to process the payment and store licenses for purchased applications in order to be able to download them again in case of a phone restore or after an uninstallation.

Besides its gatekeeper function, the Marketplace offers non-repudiation. Developers must prove their identity before releasing applications to the Marketplace. Depending on the key that is used for the code signature, applications can always be tracked back to the original developer.

Additionally, the Marketplace also provides a protection of intellectual property. As software piracy is rather significant, also on mobile devices, there is the need for such a mechanism. Each application is not only signed by Microsoft, but it also has an according license that is needed in order to run. This license is saved with the Windows Live ID and queried at every application launch. Thus, even if attackers manage to sideload applications, there is still the need to circumvent the licensing mechanisms.

7.4 Summary

Despite iOS encrypting all binaries, they reside in memory in the clear which enables attackers to attach a debugger and dump the binary. These binaries can then be run on jailbroken phones. Android and Windows Phone 7 score with the option for developers to contact the vendor's servers from an application in order to verify if the account associated with the phone indeed owns a license for the application. For Android, this feature is only available for applications targeting Android 2.2 and higher. Applications that want to target older version of Android as well cannot use this feature.

While similar statements hold true for the Windows Phone 7 Marketplace, the Android Market follows an opposite concept. Google permits every application on the Market as long as the developer is registered with Google. There is no review whatsoever and the whole security concept relies on the user community to identify and flag malicious applications. This has caused the Android Market to be flooded with an overabundance of malicious applications of all kinds. Leaving the task to the community seems to be irresponsible, but at least, Google has proven to maintain very short reaction times if a really dangerous application is spotted on their Market.

8 Permission Model

8.1 Android

In contrast to iOS, Android employs a very fine-grained application permission model. As of API version 11, 116 different permissions are predefined [2]. Developers are free to add custom permissions to protect their applications. An application that uses an SQLite database to store addresses may want to allow others to access the datasets, but only if the user agrees. An application has to explicitly request every single permission it is going to use in its manifest file. This model is supposed to make it impossible to hide undesired activities for applications. While it may sound reasonable for a Tetris game to request Internet access to maintain an online highscore list, it would be suspicious if it also requested the permission to read the address book.

If an application component attempts to perform an action that is protected by the need for a particular permission, Android raises a *SecurityException* and the action will fail. Depending on the component type, permissions can be enforced at the time the application calls a protected function in the system; when starting a new activity to disallow the start of third-party applications, when accessing a content provider, when binding or starting a service, and when sending or receiving broadcasts to exclude applications without permission from receiving or sending. Developers can, at any time, check if the calling process has more fine-grained permissions that have been defined by calling the context's `checkCallingPermission()` method.

URI Permissions: Content providers are often protected by permissions, but may want to pass a URI to another application for data transfer. For example, the mail application protects the emails from being read without permission; however, a third-party image viewer requests to display an image attachment. The image viewer is handed a URI to the data with the `Intent.FLAG_GRANT_READ_URI_PERMISSION` flag set by the caller.

Protection Levels: Permissions are categorized in four protection levels, 0 to 3: Category-zero permissions are *normal* permissions with a low-risk factor that typically only affect the application scope. They are automatically granted by the system without explicit approval, optionally the user sees the permission request prior to the installation. Only if the user gives his consent to all requested permissions, the application is installed. *Dangerous* permissions, belonging in category 1, are higher-risk permissions that for instance allow costly services like initiating a phone call, access to the device's sensors or user data. Protection level 2 holds permissions which are granted only if the installation candidate is signed with the private key corresponding to the same certificate as the application that defined the permission. These *signature* permissions are useful for developers to make sure that third-party applications cannot be granted this permission, even if the user would consent. Permissions on the highest level 3 can only be granted by the system to applications that are contained in the system image, or to ones that are at least signed with the same certificate as the system image. Representatives of this category are the permission to install new system applications or to change security settings.

Granularity: Even though there exists a fine-grained permission model, its application suffers from some weaknesses. The most important one is that the user is only able to grant or deny all permissions at once. There is no chance to grant or deny particular permissions. This forces the user to refrain from installing an application that might be useful, but requests too many permissions. Moreover, permissions that have been granted can only be revoked by uninstalling the affected application. This is a strong plea to the user's discipline. He might want to test an application and not care about the permissions dialog.

8.2 iOS

The permission model that is used by iOS is implemented in an implicit fashion. By default, the model restricts access to sensitive frameworks. However, it does not protect as many functions as the other systems do. For instance, every application is automatically allowed to access the Internet or to use the camera. Developers do not need to care about permissions that have to be requested, because the system displays the security prompt at the first use of a protected framework automatically. If the user allows access to the framework, iOS remembers this decision and does not ask again. If, however, the user decides to deny the access two consecutive times, access is permanently denied for this application. Protected frameworks are for example the AddressBook framework, CoreLocation and the push notification service. As far as the location discovery and push notifications are concerned, the user can, at any time, change his mind and grant or revoke the permission afterwards in the system settings. Interestingly enough, the access to the address book cannot be revoked once it is granted.

8.3 Windows Phone 7

Applications must request permission for accessing protected APIs or resources (e.g., location services or network access) during the deployment process. Capabilities are declared in a so-called application manifest file. The creation of the capability list is done by a detection utility shipped with the developer tools. This makes sure that an application requests exactly the capabilities it needs in order to run. The requested capabilities are displayed to the user in the Marketplace application on the phone and he is asked to explicitly approve them prior to the installation process. Minimal capabilities are automatically granted, which can only affect the application's own scope. For example, writing an isolated storage file is allowed without special request. The permission checks are enforced at runtime.

The use of the capabilities detection utility ensures that all and not too many capabilities are declared by the application. However, granting the permissions is still the user's responsibility. The automatic generation of the capabilities list does not add additional security benefits, because if an application tried to perform an action that uses a protected resource without declaring the appropriate capability, the execution would fail at runtime.

8.4 Summary

With respect to the permission models, none of the platforms provides a satisfying solution. Non of the models is robust in its application (i.e. applications without a particular permission cannot gain this permission otherwise), fine-grained enough to restrict applications to the least privileges they need, and flexible enough to allow the user to revoke particular permissions at the same time. For instance, Apple fails horribly regarding the granularity by only protecting address book, location and push notifications. The permission models promise application-level security to the user, but do not realize it in an acceptable manner.

9 Jailbreaking

This section covers a selection of popular jailbreaking techniques for Android, iOS, and Windows Phone 7.

9.1 Jailbreaking Android

There have been several exploits that can be used to root current Android devices. The *Android Exploit Crew*[7] around Sebastian Krahmer has published the source code of several implementations. With some minor modifications, these exploits can be used in applications that are distributed via the Market. A simple way would be to set the setuid flag of the standard Android shell `sh`, such that the shell is always executed with `root` permissions. Hence, every application can use the shell to gain full access to the system. Applications with `root` access can read any data and post it to any service they want. Be it SMS messages, emails, address book entries and such. Even Google authentication tokens can be read.

If the rooting functionality is hidden as an appealing game or useful application, the user may not even notice that his device is rooted. As Android applications can be easily reverse-engineered, an existing application can be taken from the Market and the rooting functionality can be embedded. This is what the so-far most sophisticated Android malware, *DroidDream*, did. It took existing applications, decompiled them, equipped them with an exploit and silently rooted the device [19, 20].

Exploiding the Linux Device Manager udev: One important rooting method is an exploit called *exploid*. It exploits a vulnerability in the Linux device manager `udev`, described in CVE 2009-1185. The `dev` daemon monitors and evaluates *hotplug* events, that is connected or disconnected devices. The daemon is notified by the kernel about these events using a `NETLINK` message. Versions prior to 1.4.1 did not verify that the message indeed originates from the kernel and accepted messages from user space. This enables attackers to gain `root` privileges by sending `NETLINK` messages to `udev`.

Android does not use a new `udev` executable, but major parts of its code have been moved to the `init` process which runs with root privileges. By exploiting the vulnerability, unprivileged users can gain root access. It tricks the `init` process to perform attacker-defined actions by copying itself to a directory that is writable for an unprivileged user, e.g., anywhere on the SD card. It tries the typically world-writable directories `/sqlite_stmt_journals` and `/data/local/tmp`; one or both may not exist, depending on the device used, so the last resort is the current working directory. Then, it sends a `NETLINK_KOBJECT_UEVENT` message that is manipulated such that its own copy is executed as soon as the next hotplug event is triggered. Inside the exploit the root user id is checked. If this is the case, it copies the original `sh` to `/system/bin/rootshell` and sets the setuid bit, such that the shell always runs with `root` permissions. This vulnerability has been patched by Android 2.2.

[7] `http://c-skills.blogspot.com/`

Zygote Jailbreak zimperlich: The `zygote` jailbreak *zimperlich* follows a similar approach, but exploits a different vulnerability. On Android, there is a maximum number of user processes that are allowed and this limit can be queried by `ulimit -a` (In case of the HTC Desire Z running Android 2.2, for example, this limit is 2983).

The exploit forks processes until the maximum number of allowed processes is reached. As the exploit is triggered by a Dalvik application, the `zygote` is responsible for setting the correct Linux user ID for the newly forked process. If, however, the limit of user processes is reached, the `setuid()` system call performed by the `zygote` fails. Its original intent is to limit the privileges of the new process by setting the user id to the (unprivileged) one associated to the application. However, the `zygote` process does not check the return value of the `setuid()` call. If the call fails, the newly forked process will remain with `root` privileges that originate from its parent process, the `zygote`.

Once the privileges have been escalated, the exploit creates a copy of the `sh` shell in `/system/bin/rootshell` with the setuid flag set. This vulnerability has also been patched by Android versions greater than 2.2.

9.2 Jailbreaking iOS

The main goal of jailbraking an iOS device is to allow the execution of unsigned code. In order to do so, code signatures have to be circumvented. These checks are implemented in the kernel which also enforces them, as soon as the `execve()` system call is executed on a binary. In addition, all applications run as the unprivileged user `mobile`, i.e., even if code execution of unsigned binaries is possible, the attacker must escalate his privileges.

The kernel is loaded from the system partition, which is mounted read-only. It is compressed with all kernel extensions and this archive is encrypted and signed. However, the kernel cache, which contains the kernel and its linked extensions from the last boot, can be changed by `root` user. And still, even if the manipulation succeeds, the kernel cache is signed as well and a modification would cause the signature check by iBoot to fail. Thus, the device would not boot.

A chain of trust prevents manipulations of the kernel, iBoot and the low-level bootloader (LLB), unless there is an according vulnerability. Everything except the bootROM is signature checked. So, the most promising angle to jailbreak is to attack the bootROM itself and sequentially patch out all signature checks in the chain of trust. The great advantage in bootROM exploits is that they cannot be fixed by Apple, as a hardware revision is needed in order to eliminate the vulnerability. Therefore, a jailbreak is always possible in future iOS versions, even if the vulnerabilities have been removed from the LLB, iBoot and the kernel. The signature checks just have to be patched out again.

However, besides this permanent jailbreaking technique, there are others which are reversible through a firmware update. Depending on which component of the boot sequence exhibits vulnerabilities, every stage is a potential entry point for jailbreakers. The general idea for a userland jailbreak is to exploit a vulnerability that allows code execution, deploy the jailbreak payload, gain `root` access by using a privilege escalation exploit, and finally patch the boot sequence components. Each of these exploited vulnerabilities is fixable by Apple with a software update, though. It can still be relevant to attackers that want to gain (temporary) `root` access to the device via an application, if they manage to pass the App Store review process.

Pwnage Tool and redsn0w: The Pwnage Tool is a jailbreak tool developed by the iPhone DevTeam[8] that modifies the original firmware that is provided by Apple such that the device will be jailbroken once this modified firmware version has been installed. It uses an original Apple IPSW firmware file to integrate the jailbreak payload. The manipulated version meets the requirements described in the previous section to jailbreak the iPhone (patching out signature checks, for example) and may contain custom software. Additionally, the intended firmware update for the baseband can optionally be denied by replacing the new baseband firmware version with an older one that can be exploited. This is very important for the carrier unlock, because there is no option to downgrade the baseband yet.

The newest versions of the PwnageTool use the so-called *limera1n* exploit that profits from an insecure handling of USB commands in the DFU mode causing a heap overflow. The DFU mode is used to install firmware and is directly entered from the bootROM (on key press or via USB command). It starts a stripped-down version of iBoot, called *iBSS* and *iBEC*. Thanks to the limera1n exploit, signature checks on the kernel and the update RAMdisk can be circumvented. Next, the kernel is loaded and the RAMdisk determines the update steps to be performed. On completion, the iPhone reboots and the boot sequence corresponds to the broken chain of trust with the modified kernel.

redsn0w is another jailbreaking tool by the iPhone Dev Team. It uses the same technique as the Pwnage Tool, but offers less customization options. It requires an iPhone in DFU mode, uses limera1n, and sends its jailbroken kernel and update RAMdisk. Apart from the jailbreak, no further customization can be done.

Star/Jailbreakme.com: The most famous userland jailbreak is *Star* that is used by the website `jailbreakme.com`. At the time of writing it is able to jailbreak iOS versions 3.1.2 through 4.0.1. It exploits a vulnerability (CVE 2010-1797) in the Compact Font Format (CFF) font parser that allowed unsigned code execution and additionally used a privilege escalation exploit for the `IOSurface` kernel extension. The userland process MobileSafari was used as injection vector, as it automatically opens PDF files. The exploit redirects the user to a PDF file that includes a malformed font in order to exploit the vulnerability in the CFF library `libCGFreeType.A.dylib` using a simple stack buffer overflow attack. A long payload for a buffer in `cff_decoder_parse_charstrings()` allows the attacker to control the program counter. After gaining code execution, the payloads for obtaining `root` access and the post-install instructions can be deployed. `IOSurface` was exploited using the technique of return-oriented programming and resulted in `root` access. The Star source code has been published[9] by the author. The vulnerabilities have been fixed in iOS 4.0.2, however, there are still many devices in use that are not upgraded to recent iOS versions, e.g., 1st-gen iPhones.

This serious attack demonstrates how to gain `root` access and arbitrary code execution by exploiting a userland application. Similar unpatched vulnerabilities could be used to drive-by infect devices and gain full access.

[8] `http://blog.iphone-dev.org/`
[9] `https://github.com/comex/star`

greenpois0n: *Greenpois0n* is the name used for a jailbreak and a toolkit. The jailbreak uses the same limera1n exploit as the Pwnage Tool in combination with a userland exploit. The toolkit is open-source software[10] consisting of five modules: *Syringe* is the injector module that helps to boot devices into a jailbroken state. *Cyanide* assists in deploying iBoot payloads, and *Dioxin* is the module for developing userland jailbreaks *Anthrax* helps to build RAMdisk jailbreaks in the redsn0w style, and *Arsenic* allows deeper changes in original firmware packages, for instance to exchange the baseband firmware.

9.3 Jailbreaking Windows Phone7

With its release in October 2010, Windows Phone 7 is the newest smartphone operating system under consideration here. So far, there is only one known vulnerability. However, this vulnerability enabled attackers to install applications that have not passed Microsoft's review process. These applications were never signed by Microsoft and never appeared on the Marketplace. The *ChevronWP7* software for desktop PCs allows users to sideload applications that are not available in the Marketplace. This is usually a feature that is exclusive to registered and paying developers, because they must be able to test their applications on their phones. The ChevronWP7 employs a fake Microsoft server and fools the device into thinking that it is legitimately registered as a developer phone.

ChevronWP7 only bypasses the code signing and licensing mechanism of Windows Phone 7, because the system then accepts and runs non-Marketplace applications. It does not touch any other protection mechanism of the platform. Microsoft has acknowledged the problem and fixed it in the *NoDo* version of Windows Phone 7. After all, the vulnerability was not too serious, because no security mechanism of the system was compromised. This functionality can also be legitimately achieved by developers for the yearly fee of $ 99. Moreover, Windows Phone 7 calls home to Microsoft every two weeks in order to find unlocked developer devices that must be locked again, because the developer is not a legal subscriber in the developer program anymore. For this purpose, a device identifier is sent to Microsoft and compared to the ones stored on Microsoft's servers. ChevronWP7-unlocked phones will not appear on this whitelist and the devices will be re-locked [43].

10 Comparison

This section contrasts the protection mechanisms of the platforms. Table 1 shows the results of an evaluation of the solutions of the three operating systems, using the ratings ++, +, ◦, − and −−, from best to worst.

All of the sandboxing mechanisms have proven to be only partially secure. It is impossible for applications to break out of the sandboxes on their own, however, all platforms allow app interaction with daemons, libraries or frameworks that are running natively, and often even privileged. This opens up to the exploitation of vulnerabilities that lead to privilege escalations.

In terms of memory protection, Android has the most room for improvement. Neither does it protect its stack or heap (both are executable), nor does it use

[10] `https://github.com/Chronic-Dev`

Protection Mechanism	Android	iOS	WP7
Sandboxing	∘	∘	+
Memory Protection	−−	+	+
Code Signing	−	++	+
Service Connection	++	++	++
Jailbreak Prevention	−−	−	+
(Application) Copy Protection	+	−	++
Application Shop Security	−−	++	++
Permission Model	−	−−	−

Table 1. Security Perspective on Protection Mechanisms

techniques such as ASLR in a reasonable way. The `randomize_va_space` option is set to `1`, which means that the stack is randomized, but there is no heap base randomization. Apple, in turn, makes heavy use of the XN bit of the processor and never allows any application memory page to be writable and executable at the same time. Since iOS 4.3, ASLR is enabled, however, due to the lack of entropy on mobile devices, it has already been defeated. Windows Phone 7 only runs managed code. However, if unsigned code execution is possible, it also runs native code outside of the Common Intermediate Language.

The code signing mechanisms are implemented very differently on the three platforms. Apple and Microsoft use a certification authority, while Android's solution is based on self-signed certificates. Android thus only enables a developer to prove that he is the author of particular applications and may therefore update the application or share data between two or more of his applications. Android's signing process does not protect the platform against malicious code. iOS enforces signature checks at every system call that starts a new process and hence outscores Windows Phone in this discipline.

We observed that all systems have carefully implemented the service connection to the vendor. Theses connections are encrypted and the certificates are checked for validity, which results in proper authentication of the remote station. All systems support the installation of additional root CA certificates, but none allows the use of user-installed certificates for the service connection. This prevents man-in-the-middle attacks using self-signed CA certificates.

One of the most severe problems of iOS is its copy protection mechanism. Despite iOS encrypting all binaries, they reside in memory in the clear as a whole. This enables attackers to attach a debugger and dump the binary. These binaries can then be run on jailbroken phones. Android and Windows Phone 7 score with the option for developers to contact the vendor's servers from an application in order to verify if the account associated with the phone indeed owns a license for the application. For Android, this feature is only available for applications targeting Android 2.2 and higher. Applications that want to target older version of Android as well cannot use this feature.

The most important discipline from a security perspective is preventing applications from gaining full access to the device, i.e., jailbreaking. Android and iOS have deficits due to the fact that applications are allowed to run natively, while Microsoft will not allow applications containing native code on its Marketplace.

Apple tries to compensate the risk of native code execution with the establishment of a review process that precedes every admission to the App Store. These review process techniques have proven to detect the largest part of malware and misbehaving applications. However, it is neither fully transparent, nor can one be sure that every malware incident is reported by Apple. Reported malware incidents on the App Store only involved minor privacy issues such as unique device identifiers being transmitted. The address book, which used to be readable by applications without user consent, is now protected and requires permissions.

While similar statements hold true for the Windows Phone 7 Marketplace, the Android Market follows an opposite concept. Google permits every application on the Market as long as the developer is registered with Google. There is no review whatsoever and the whole security concept relies on the user community to identify and flag malicious applications. This has caused the Android Market to be flooded with an overabundance of malicious applications of all kinds. Leaving the task to the community seems to be irresponsible, but at least, Google has proven to maintain very short reaction times if a really dangerous application is spotted on their Market.

With respect to the permission models, none of the platforms provides a satisfying solution. Non of the models is robust in its application (i.e. applications without a particular permission cannot gain this permission otherwise), fine-grained enough to restrict applications to the least privileges they need, and flexible enough to allow the user to revoke particular permissions at the same time. For instance, Apple fails horribly regarding the granularity by only protecting address book, location and push notifications. The permission models promise application-level security to the user, but do not realize it in an acceptable manner.

11 Conclusion

In this paper we have given a structured and comprehensive overview on the most prominent mobile operating systems Android, iOS, and Windows Phone 7. We presented a number of threats and attack vectors all platforms have in common, and detailed the most relevant security mechanisms of each platform. We have seen that today's smartphones feature operating systems that do not fall short on the ones of desktop computers, and as such, they also inherit many of their problems and vulnerabilities. As for desktop computers today, it is essential to maintain an up-to-date smartphone operating system. However, this responsibility mainly resides with the smartphone vendors, which tend to stop updating older devices. As a consequence these devices remain without the chance to fix existing vulnerabilities. Overall, the security mechanisms of the three platforms under discussion leave significant room for improvement.

Acknowledgments

Parts of the work presented in this article have been supported by the ASMONIA research project, partially funded by the German Federal Ministry of Education and Research (BMBF).

References

1. Permission re-delegation: Attacks and defenses. In *20th Usenix Security Symposium*, 2011.
2. The Developer's Guide. `http://developer.android.com/guide/`, 2011. Retrieved on 2011-05-09.
3. Android Developers. What is Android? Android Developers, `http://developer.android.com/guide/basics/what-is-android.html`, May 2011.
4. Apple Developer. *iOS Technology Overview*, October 2010.
5. Apple Developer. *App Store Review Guidelines for iOS apps*, 2011.
6. Seyed Morteza Babamir, Reyhane Nowrouzi, and Hadi Naseri. Mining Bluetooth Attacks in Smart Phones. *Communications in Computer and Information Science*, 87:241–253, 2010.
7. Dionysus Blazakis. The Apple Sandbox. Arlington, VA, January 2011.
8. Tim Bray. Licensing Service For Android Applications. Android Developers Blog, `http://android-developers.blogspot.com/2010/07/licensing-service-for-android.html`, July 2010. Retrieved on 2011-05-19, archived at `http://www.webcitation.org/5yn8Bv7KJ`.
9. Jason Chen. Aurora Feint iPhone App Delisted For Lousy Security Practices. Gizmodo.com, `http://gizmodo.com/5028459/aurora-feint-iphone-app-delisted-for-lousy-security-practices`, July 2008. Retrieved on 2011-05-20.
10. Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. pages 239–252, 2011.
11. Graham Cluley. First iPhone worm discovered - ikee changes wallpaper to Rick Astley photo. Sophos Naked Security Blog, `http://nakedsecurity.sophos.com/2009/11/08/iphone-worm-discovered-wallpaper-rick-astley-photo/`, November 2009. Retrieved on 2011-05-22, archived at `http://www.webcitation.org/5ys7LWTko`.
12. Istvan Cseri. Windows Phone Architecture: Deep Dive. April 2011.
13. Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege Escalation Attacks on Android. In *Proceedings of the 13th international Conference on Information Security*, ISC'10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag.
14. Dino A. Dai Zovi. Apple iOS 4 Security Evalution. Trail of Bits LLC, `http://www.trailofbits.com/research.html`, 2011.
15. David Ehringer. The Dalvik Virtual Machine Architecture. 2010.
16. W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *Security Privacy, IEEE*, 7(1):50 –57, jan.-feb. 2009.
17. William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.
18. Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.
19. Aaron Gingrich. Malware Monster: DroidDream Is An Android Nightmare, And We've Got More Details. Android Police, March 2011. Retrieved on 2011-05-19, archived at `http://www.webcitation.org/5ynIuuEgO`.
20. Aaron Gingrich. The Mother Of All Android Malware Has Arrived: Stolen Apps Released To The Market That Root Your Phone, Steal Your Data, And Open Backdoor. Android Police, March 2011. Retrieved on 2011-05-19, archived at `http://www.webcitation.org/5ynIYoxLb`.
21. Dan Goodin. Backdoor in top iPhone games stole user data, suit claims. The Register, `http://www.theregister.co.uk/2009/11/06/iphone_games_storm8_lawsuit/`, November 2009. Retrieved on 2011-05-23, archived at `http://www.webcitation.org/5ytg1Rclo`.
22. Dan Goodin. Researcher outs Android Exploit Code. The Register, `http://www.theregister.co.uk/2010/11/06/android_attack_code/`, November 2011. Retrieved on 2011-05-20, archived at `http://www.webcitation.org/5yoxrzjcq`.
23. Jerry Hauck, Jeffrey Bush, Michael Lambertus Brouwer, and Daryl Mun-Kid Low. Service Provider Activation with Subscriber Identity Module Policy. United States Patent Application Publication, No. US 2009/0061934, January 2008.
24. Xuxian Jiang. Android 2.3 (Gingerbread) Data Stealing Vulnerability. `http://www.csc.ncsu.edu/faculty/jiang/nexuss.html`, January 2011. Retrieved on 2011-05-20, archived at `http://www.webcitation.org/5youAnT9n`.

25. Henry Lee and Eugene Chuvyrov. *Beginning Windows Phone 7 Development*. Apress, New York, 2010.
26. Kevin Mahaffey. Security Alert: DroidDream Malware Found in Official Android Market. `http://blog.mylookout.com/2011/03/security-alert-malware-found-in-official-android-market-droiddream`, March 2011. Retrieved on 2011-05-09, archived at `http://www.webcitation.org/5yYBUjtUS`.
27. Microsoft Corp. *Windows Phone 7 Security Model*, windows phone 7 guides for it professionals edition, December 2010.
28. C. Miller. Mobile attacks and defense. *Security Privacy, IEEE*, 9(4):68 –70, july-aug. 2011.
29. Charlie Miller and Vincenzo Iozzo. Fun and Games with Mac OS X and iPhone Payloads. April 2009.
30. Dan Moren. Third iPhone worm targets jailbroken iPhones in Europe, Australia. Macworld, `http://www.macworld.com/article/144039/2009/11/iphone_worm.html`, November 2009. Retrieved on 2011-05-22, archived at `http://www.webcitation.org/5ys9Xu18f`.
31. Collin Mulliner and Nico Golde. SMS-o-Death. 27th Chaos Communication Congress, Berlin, December 2010. Talk given on 2010-12-27.
32. Jon Oberheide. A Peek Inside the GTalkService Connection, June 2010. Retrieved on 2011-05-09, archived at `http://www.webcitation.org/5yYOFxfjH`.
33. Jon Oberheide. Remote Kill and Install on Google Android. `http://jon.oberheide.org/blog/2010/06/25/remote-kill-and-install-on-google-android/`, June 2010. Retrieved on 2011-05-09, archived at `http://www.webcitation.org/5yYCucC6N`.
34. Jon Oberheide. How I Almost Won Pwn2Own via XSS, March 2011. Retrieved on 2011-05-09, archived at `http://www.webcitation.org/5yXyUFBhy`.
35. Jon Oberheide and Zach Lanier. TEAM JOCH vs. Android: The Ultimate Showdown. Washington, DC, January 2011.
36. Fahmida Y. Rashid. Mobile Malware, Hacktivism Top List of Major Security Concerns, April 2011. Retrieved on 2011-05-14, archived at `http://www.webcitation.org/5yfcdXadP`.
37. Erick Schonfeld. Touching The Android: It's No iPhone, But It's Close. TechCrunch, `http://techcrunch.com/2008/09/23/touching-the-android-its-no-iphone-but-its-close/`, September 2008. Retrieved on 2011-05-19, archived at `http://www.webcitation.org/5ynWx29YF`.
38. Rene Mayrhofer Sebastian Höbarth. A framework for on-device privilege escalation exploit execution on Android. In *IWSSI*, 2011.
39. A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google android: A comprehensive security assessment. *Security Privacy, IEEE*, 8(2):35 –44, march-april 2010.
40. Hovav Shacham, Eu-jin Goh, Nagendra Modadugu, Ben Pfaff, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 298–307. ACM Press, 2004.
41. Wook Shin, Sanghoon Kwak, Shinsaku Kiyomoto, Kazuhide Fukushima, and Toshiaki Tanaka. A Small But Non-negligible Flaw in the Android Permission Scheme. In *Proceedings of the 2010 IEEE International Symposium on Policies for Distributed Systems and Networks*, POLICY '10, pages 107–110, Washington, DC, USA, 2010. IEEE Computer Society.
42. Patrick Traynor, Michael Lin, Machigar Ongtang, Vikhyath Rao, Trent Jaeger, Patrick McDaniel, and Thomas La Porta. On cellular botnets: measuring the impact of malicious devices on a cellular network core. pages 223–234, 2009.
43. Tom Warren. Windows Phone 7 calls home to re-lock ChevronWP7 unlocked devices, December 2010. Retrieved on 2011-05-19, archived at `http://www.webcitation.org/5yn7uK8Zd`.
44. Georgia Weidman. Transparent Botnet Control for Smartphones over SMS (ShmooCon'11. January 2011.
45. Cui Xiang, Fang Binxing, Yin Lihua, Liu Xiaoyi, and Zang Tianning. Andbot: towards advanced mobile botnets. pages 11–11, 2011.
46. Cui Xiang, Fang Binxing, Yin Lihua, Liu Xiaoyi, and Zang Tianning. Andbot: towards advanced mobile botnets. In *Proceedings of the 4th USENIX conference on Large-scale exploits and emergent threats*, pages 11–11. USENIX Association, 2011.

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2009-01 *	Fachgruppe Informatik: Jahresbericht 2009
2009-02	Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications
2009-03	Alexander Nyßen: Model-Based Construction of Embedded Real-Time Software - A Methodology for Small Devices
2009-05	George B. Mertzios, Ignasi Sau, Shmuel Zaks: A New Intersection Model and Improved Algorithms for Tolerance Graphs
2009-06	George B. Mertzios, Ignasi Sau, Shmuel Zaks: The Recognition of Tolerance and Bounded Tolerance Graphs is NP-complete
2009-07	Joachim Kneis, Alexander Langer, Peter Rossmanith: Derandomizing Non-uniform Color-Coding I
2009-08	Joachim Kneis, Alexander Langer: Satellites and Mirrors for Solving Independent Set on Sparse Graphs
2009-09	Michael Nett: Implementation of an Automated Proof for an Algorithm Solving the Maximum Independent Set Problem
2009-10	Felix Reidl, Fernando Sánchez Villaamil: Automatic Verification of the Correctness of the Upper Bound of a Maximum Independent Set Algorithm
2009-11	Kyriaki Ioannidou, George B. Mertzios, Stavros D. Nikolopoulos: The Longest Path Problem is Polynomial on Interval Graphs
2009-12	Martin Neuhäußer, Lijun Zhang: Time-Bounded Reachability in Continuous-Time Markov Decision Processes
2009-13	Martin Zimmermann: Time-optimal Winning Strategies for Poset Games
2009-14	Ralf Huuck, Gerwin Klein, Bastian Schlich (eds.): Doctoral Symposium on Systems Software Verification (DS SSV'09)
2009-15	Joost-Pieter Katoen, Daniel Klink, Martin Neuhäußer: Compositional Abstraction for Stochastic Systems
2009-16	George B. Mertzios, Derek G. Corneil: Vertex Splitting and the Recognition of Trapezoid Graphs
2009-17	Carsten Kern: Learning Communicating and Nondeterministic Automata
2009-18	Paul Hänsch, Michaela Slaats, Wolfgang Thomas: Parametrized Regular Infinite Games and Higher-Order Pushdown Strategies
2010-01 *	Fachgruppe Informatik: Jahresbericht 2010
2010-02	Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time

2010-03 Holger Krahn: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering

2010-04 René Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme

2010-05 Daniel Retkowitz: Softwareunterstützung für adaptive eHome-Systeme

2010-06 Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Computing maximum reachability probabilities in Markovian timed automata

2010-07 George B. Mertzios: A New Intersection Model for Multitolerance Graphs, Hierarchy, and Efficient Algorithms

2010-08 Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl: Automated Termination Analysis of Java Bytecode by Term Rewriting

2010-09 George B. Mertzios, Shmuel Zaks: The Structure of the Intersection of Tolerance and Cocomparability Graphs

2010-10 Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, René Thiemann: Automated Termination Analysis for Logic Programs with Cut

2010-11 Martin Zimmermann: Parametric LTL Games

2010-12 Thomas Ströder, Peter Schneider-Kamp, Jürgen Giesl: Dependency Triples for Improving Termination Analysis of Logic Programs with Cut

2010-13 Ashraf Armoush: Design Patterns for Safety-Critical Embedded Systems

2010-14 Michael Codish, Carsten Fuhs, Jürgen Giesl, Peter Schneider-Kamp: Lazy Abstraction for Size-Change Termination

2010-15 Marc Brockschmidt, Carsten Otto, Christian von Essen, Jürgen Giesl: Termination Graphs for Java Bytecode

2010-16 Christian Berger: Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles

2010-17 Hans Grönniger: Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten

2010-18 Ibrahim Armaç: Personalisierte eHomes: Mobilität, Privatsphäre und Sicherheit

2010-19 Felix Reidl: Experimental Evaluation of an Independent Set Algorithm

2010-20 Wladimir Fridman, Christof Löding, Martin Zimmermann: Degrees of Lookahead in Context-free Infinite Games

2011-01 * Fachgruppe Informatik: Jahresbericht 2011

2011-02 Marc Brockschmidt, Carsten Otto, Jürgen Giesl: Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting

2011-03 Lars Noschinski, Fabian Emmes, Jürgen Giesl: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems

2011-04 Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars

2011-07 Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe: An Operational Semantics for Activity Diagrams using SMV

2011-08 Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, Carsten Fuhs: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog

2011-09 Markus Beckers, Johannes Lotz, Viktor Mosenkis, Uwe Naumann (Editors): Fifth SIAM Workshop on Combinatorial Scientific Computing

2011-10 Markus Beckers, Viktor Mosenkis, Michael Maier, Uwe Naumann: Adjoint Subgradient Calculation for McCormick Relaxations

2011-11 Nils Jansen, Erika Ábrahám, Jens Katelaan, Ralf Wimmer, Joost-Pieter Katoen, Bernd Becker: Hierarchical Counterexamples for Discrete-Time Markov Chains

2011-12 Ingo Felscher, Wolfgang Thomas: On Compositional Failure Detection in Structured Transition Systems

2011-13 Michael Förster, Uwe Naumann, Jean Utke: Toward Adjoint OpenMP

2011-14 Daniel Neider, Roman Rabinovich, Martin Zimmermann: Solving Muller Games via Safety Games

2011-16 Niloofar Safiran, Uwe Naumann: Toward Adjoint OpenFOAM

2011-18 Kamal Barakat: Introducing Timers to pi-Calculus

2011-19 Marc Brockschmidt, Thomas Ströder, Carsten Otto, Jürgen Giesl: Automated Detection of Non-Termination and NullPointerExceptions for Java Bytecode

2011-24 Callum Corbett, Uwe Naumann, Alexander Mitsos: Demonstration of a Branch-and-Bound Algorithm for Global Optimization using McCormick Relaxations

2011-25 Callum Corbett, Michael Maier, Markus Beckers, Uwe Naumann, Amin Ghobeity, Alexander Mitsos: Compiler-Generated Subgradient Code for McCormick Relaxations

2011-26 Hongfei Fu: The Complexity of Deciding a Behavioural Pseudometric on Probabilistic Automata

2012-01 * Fachgruppe Informatik: Annual Report 2012

2012-02 Thomas Heer: Controlling Development Processes

2012-03 Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems

2012-04 Marcus Gelderie Strategy Machines and their Complexity

2012-05 Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs Automated Complexity Analysis for Prolog by Term Rewriting

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.