An Antivirus API for Android Malware Recognition

Rafael Fedler, Marcel Kulicke, Julian Schütte
{fedler,kulicke,schuette}@aisec.fraunhofer.de

September 11, 2014

Abstract

On the Android platform, antivirus software suffers from significant deficiencies. Due to platform limitations, it cannot access or monitor an Android device's full file system, but only small fractions of it. Currently, antivirus software relies on checking almost exclusively package names and package files. It is unable to monitor dynamic behavior by installed apps on the file system. This includes the downloading of malicious files after installation, and other file system alterations. That has grave consequences for device security, as any app – even without openly malicious code in its package file – can still execute malicious files without any danger of being detected by antivirus software. Given privilege escalation code, malware can also alter arbitrary file system portions unnoticed. Product tests, however, suggest a high level of protection offered by these antivirus products. Such protection may not be reached in practice, leaving device users in an illusive feeling of security.
In this paper, we present a proposal for an antivirus interface to be added to the Android API. It can enable Android antivirus software to deploy techniques for malware recognition similar to those of desktop antivirus systems system-wide. Antivirus software would be capable of performing signature and heuristics based malware recognition tests on arbitrary file system objects. The proposed measures comply with Android's security architecture and user data privacy is maintained. Through our approach, antivirus software on the Android platform would reach a level of effectiveness significantly higher than currently, and comparable to that of desktop antivirus software.

1 Introduction

The currently most extensive test of Android antivirus software reports detection rates above 90% for approximately 25% of all Android antivirus software, and more than 50% of all products showed detection rates of 65% and above [4]. As own tests have shown, these detection rates are only achieved by two means: (1) *Package name comparison*, i.e., antivirus software scans an Android device's

package database for the package names of installed apps and compares them with its database of known malicious apps. (2) *Tests of package files.* However, most portions of an Android file system, including the working directories of apps, are not accessible by antivirus software. Antivirus software cannot scan working directories' contents. The two approaches are naturally prone to easy evasion by malware developers. With Approach 1 it can often suffice to change the package name of malicious apps to bypass antivirus software at least for a short time. If malware is not spread widely, antivirus software vendors will less likely be able to add the package name to their databases for a longer period. Such malware will then not be detected by Android antivirus software at all. Approach 2 can be easily circumvented by not including any openly malicious functionality in the initially installed package file of an app itself, but only into components downloaded and executed at runtime. Due to these reasons, tests of antivirus software may report high detection rates for most products in theory, but in practice this does not result in a high level of protection.

The deficiencies of antivirus solutions on Android and their lack of any effective detection methods for dynamic behavior, i.e., heuristics or signature based approaches on arbitrary file system objects, result from platform limitations. As any other software installed on Android-based devices – commonly referred to as apps – antivirus software is also confined in its own file system sandbox [3]. It cannot access directories of any other app, and thus cannot scan other apps' files for malicious patterns or behavior. Any app which is not included in the antivirus apps' databases will not be detected when downloading malicious files to its own directory, e.g., privilege escalation exploits or other malicious components. Thus, we consider the level of protection offered by antivirus software on the Android platform insufficiently low.

To improve the capabilities of antivirus software on the Android platform, we propose an addition to the Android API. Specifically, we present an interface for antivirus software to perform various operations on file system objects of an Android device. It allows for both signature-based and heuristic detection methods on arbitrary file system objects, as well as almost full file system scanning and monitoring. The interface is secured by Android's own mechanisms in such a way that only legitimate antivirus software can access it. User data privacy is also preserved. Neither file system objects' names and full paths, nor file contents may be read by antivirus software. Our approach would increase the level of protection that can be offered by Android antivirus software to a level comparable to that of desktop malware recognition solutions, which is a major improvement over the capabilities demonstrated by current Android antivirus solutions.

The rest of the paper is organized as follows. We introduce the background theory in Section 2. This includes Android system basics and a discussion of Android malware techniques. To illustrate the problematic situation of malware recognition on Android, the issues of antivirus software on the Android platform are explained in Section 3. In Section 4 we introduce our concept for an antivirus interface as part of the Android API. We will detail possible operations, resulting feasible recognition techniques, user data privacy, and how the

interface is secured against unauthorized access. We will discuss our approach in Section 5 before concluding the paper in Section 6.

2 Background

After providing the necessary background information related to the Android operating system, we will give a short insight into Android malware behavior. More specifically, we will briefly explain one typical technique deployed by malware for propagation, infection, and primary functionality. This will help understand the gravity of the deficiencies of current antivirus software.

2.1 Android Platform

In the following, we will introduce the basics of the Android platform's measures for file system access control and for managing the list of installed packages. The whole Android security architecture will not be discussed in detail, as a multitude of publications on this topic can be found [10, 22, 11].

2.1.1 Android File System Security

On traditional desktop operating systems, processes started by one user all inherit that user's user ID (UID). As file system access controls are based on the UID, every process can access all files by every other process with the same UID, also including all personal files by the user who started the respective processes. To introduce additional security and data privacy, and to prohibit access to any file system objects by other apps, the Android operating system breaks with the aforementioned traditional UID assignment scheme. Instead of assigning one identical UID to every process, every app is assigned its own unique UID at installation time by default [3]. The ownership of every app's working directory (located in `/data/data/[package name]`) and all files contained in this working directory is set to the respective app's UID. Access to files is by default only permitted to the app's UID, and thus only to the app itself. Most file system objects outside of apps' working directories are owned by system UIDs and thus cannot be accessed as well. Ultimately, this UID assignment scheme and connected file system access permissions establish a file system sandbox for apps [22]. They are confined in their own directory, separating each app from every other, and from operating system's files and folders.
Android antivirus software, which is also just usual apps, is limited drastically by this file system-based sandboxing. As a consequence, antivirus apps cannot scan the file system on demand or monitor file system changes. Most importantly, antivirus apps cannot scan the working directories of other apps and are oblivious to any files other apps might download at runtime. Apps can do this to add functionality after installation in the form of native code, which will thus not show in package files.
The aforesaid does not hold true for external storage like memory cards. Such

memory is usually formatted with a file system which does not support the Linux kernel-derived access control mechanisms and is readable by any app [2]. However, recognition techniques limited to external memory are hardly of any use, as malware would usually operate on system memory, where the working directories are located, and not on external memory.

2.1.2 Package Database

The Android operating system keeps track of installed apps in a package database (located in `/data/system/packages.xml`) [8]. This database stores a set of data for every app. It includes the *code path* where an app's package file with its bytecode is stored, the app's *package name*, it's *UID*, and other entries.
In contrast to many other Android operating system resources, the package database is publicly readable [14]. As antivirus apps cannot scan the file system for malicious files, they resort to comparing the package names of all apps listed in the package database with the antivirus' database of known malware package names.

2.1.3 Package File Access

While the package file directory (`/data/app`) is not readable by arbitrary apps, and thus listing of package files in it is prohibited, the package database is world-readable. Furthermore, the package files – while they cannot be listed through file system access – are also readable by any app. This in combination gives read access to package files: Antivirus software can extract the path to package files from the package database and then open package files directly, instead of listing the contents of the package file directory. This way, common antivirus detection techniques can at least be applied to the static app package file.

2.2 Android Malware Techniques

While different malware deploys various techniques to achieve propagation, infection and operation, one technique of propagation and infection is of particular relevance when discussion antivirus software on Android and its (in)effectiveness: the dynamic fetching and execution of code [23]. We assume it will be increasingly used by malware for various reasons. (1) Native code enables malware to escalate privileges to root access, as all currently available standalone root exploits are given as native code. (2) Carrying additional malicious code along with a malware app's package file instead of dynamically downloading it after installation can lead to detection by Google's Bouncer service during publishing in the Android Market ("Google Play") [20]. (3) In general, it is advantageous for being undetected as malware to download any malicious payload after installation, so it cannot be found in an app's package file. The code for fetching and execution can be written in a completely neutral manner so that no malicious intent is visible.
This technique can already be witnessed by malware in the wild, as for example

demonstrated by RootSmart/BSmart [16]. Also, this technique demonstrates the deficiencies of antivirus software for Android. A seemingly completely harmless app which is not (yet) included in an antivirus app's database can carry out arbitrary operations on a system by firstly fetching executable code from the Internet. Such newly added code cannot be noticed by antivirus software. Especially when the malware is not very widely spread, it is unlikely that the antivirus database will be updated quickly. The worst case would be a targeted attack which will go unnoticed for a very long time. Furthermore, when root exploits are used by malware, detection with a delay of a few days is too late to uninstall the app before the malicious payload can assume full device control.

3 Android Antivirus Software

In this section, we will summarize the aforementioned deficiencies of antivirus software on Android. We will also discuss why product tests based solely on the detection rate are misleading regarding the real level of protection offered.

3.1 Limitations

The root cause of Android antivirus software ineffectiveness is the inability to directly access the file system and its contents. While this is a big benefit in all other cases as it ensures data security and privacy of installed apps, especially against access by other apps, it is a major hindrance for antivirus software. Android antivirus cannot deploy recognition techniques based on heuristics or signatures to arbitrary file system objects, and especially not to app working directories' contents. Thus, dynamically downloaded code will not be found. This dynamically fetched code may also be the only component which openly demonstrates malicious behavior, keeping the app which downloaded the malicious payload free of any suspicion and detection. Also, apps which do not demonstrate malicious behavior with a big portion of users, or even worse, with only some users or one user in case of targeted attacks, will never or only very late be included in Android antivirus databases. Delayed recognition may often be without any use to the device's security and its user, because malware may already have assumed full device control by privilege escalation exploits. Such malware must be detected before, during, or shortly after installation.

3.2 Product Tests

Typical antivirus product tests purely based on detection rate like those usually conducted with desktop antivirus products are only of limited significance for Android antivirus software. Due to the nature of these tests, in which a sample set is collected and then used as a testbed, only the antivirus vendors' retrospective knowledge of package files is tested. For many samples it is only tested weeks or months after their initial discovery. We do not consider such retrospective tests a valid representation of the real protection level offered in

real time, when new threats emerge. In reality, it is also of major importance how quickly a database was updated, determining how long users can install malware without being warned by their antivirus app. Also, tests do not take into account that no single antivirus app will be capable of detecting malicious activity inside the working directory of unknown malware, no matter how high the theoretical detection rate. It is also ignored that changing package names will lead to malware – identical except for the changed package name – failing to be detected, if only package name comparison is applied. Thus, we consider the detection rate figures reported by tests, e.g. by [4], misleading. They suggest nearly complete protection for some products, even though these products can be easily evaded by malware developers.

4 An Antivirus Interface

To address the weaknesses of current Android antivirus software and to increase its effectiveness significantly, we propose a new interface for antivirus apps on the Android platform. It allows for the execution of a number of operations, which enable antivirus software to carry out detection techniques on arbitrary files. User data privacy is kept in tact through aliasing of paths and by permitting only indirect access through certain operations to files, instead of reading file contents directly.

4.1 File System Navigation

File system access, to be provided via directory listings, is not granted directly but through API methods, which are to be offered by a corresponding class and methods in the Android framework. If a directory listing is requested by an antivirus app, where the directory can be specified as a path provided as a string or as an **alias**, the antivirus app will receive back a list of the directory contents. However, to ensure user data privacy, this list will not contain absolute or relative paths, but aliases. These are string representations of file handles. They again can be used to navigate through directories, or to perform operations on files for malware recognition. To allow traversing of the whole file system, e.g. through breadth-first or depth-first search, this list of aliases has to be ordered and aliases need to be calculated deterministically. Order is achieved by prepending an **entry number**. The alias can be a hash function, e.g. MD5 or a function from the SHA family, of the absolute path of a file system object. The list also needs to contain **flags** denoting whether an alias represents a file or a directory. This enables antivirus app to distinguish between directories which should be traversed, and files which should be scanned for malicious contents. We propose the following tuple format as entries for a directory listing:
$\langle entry\ no. \rangle\ \langle directory/file\ flag \rangle\ \langle alias \rangle$
To illustrate the concept, Listing 1 shows the output of a regular directory listing via the "ls" command, while Listing 2 shows the output of a directory listing as performed by our proposed method. Each line of Listing 2 corresponds to the

same line of Listing 1 and contains the tuple described above.

Listing 1: Usual directory listing

```
user@computer:/topleveldir$ ls
dir1
dir2
file1
file2
file3
```

Listing 2: Directory listing using aliases; output of the proposed API method for directory listings

```
1 d b72b7253c45f9d22044c86bf4d7e4902
2 d 515dc267bbd0af019d22e766af0cb7e4
3 f f8bb5cc06b4ed23683b276ca05153e82
4 f 7b2de0a0f16d100dfbf2d84603840ee2
5 f 9fa5ba9abe67916142cb6bc0eee7658b
```

If the antivirus app wants to continue traversing the file system, it can pass one of the directory aliases to the method for directory listings. To allow scanning from predefined or user-defined paths on downwards, the method for directory listing will also accept paths instead of aliases. If the antivirus app wants to perform operations on a file, it can pass a file alias to the corresponding methods for file operations. Lastly, the directory listing provided by the antivirus interface can also contain an entry for the parent directory, i.e. for "..", which is denoted by a "p" flag. The resulting output is shown in Listing 3.

Listing 3: Directory listing using aliases including an entry for parent directory

```
0 p a55822426a5330c04625a41d264c190b
1 d b72b7253c45f9d22044c86bf4d7e4902
2 d 515dc267bbd0af019d22e766af0cb7e4
3 f f8bb5cc06b4ed23683b276ca05153e82
4 f 7b2de0a0f16d100dfbf2d84603840ee2
5 f 9fa5ba9abe67916142cb6bc0eee7658b
```

File system access can be either granted from the root of the file system, i.e. for the whole file system, or only from the app code paths and working directories (`/data/app`, `/data/data` and `/system/app`) downwards. The latter case would allow for access to the code files and working directories of installed apps. The advantage would be overall reduced costs, i.e. execution time and energy, for scanning the accessible part of the file system. Mostly, but not always, access to these directories will be sufficient for malware recognition.
The implementation of the navigation by aliases and especially the whole traversal of the file system using aliases will require the API interface either storing or calculating the aliases of (direct) parents in the file system. Alternatively, a trapdoor function for computing aliases may be used to allow the interface to recover paths from aliases. These serve the purpose to map aliases passed by the antivirus software back to file system paths.

4.2 File Operations

File operations offered by the proposed interface should aid in the detection of potentially malicious files. Malware recognition is usually achieved by signature matching, heuristic analysis, or also hash-based. We will detail in the following how our proposed interface can provide for these malware recognition techniques.

4.2.1 Signatures

Common signatures contain one or multiple byte sequences, which can be linked in various relations, such as logical (AND, OR) or regarding location in a target file. Such relations can be expressed via regular expressions. Thus, our proposed interface shall provide a method to conduct pattern matching using regular expressions. That method will only return true or false for any given pattern described by a regular expression passed to the method. This ensures user and system data privacy, while still allowing signature-based detection.
Different pattern matching algorithms can also be used, which can improve pattern matching performance or compatibility with existing vendor signature databases.

4.2.2 Hashes

Malware can also be detected using hash functions. On desktop platforms this is not as effective, as many variants with minor alterations of one malware may exist, which circumvents full-file hash matching. On mobile platforms, though, hashes are of bigger benefit. Desktop malware is usually spread highly decentralized via exploitation of software vulnerabilities, often via the Internet or web, and often the same malware may be spread by many people for whom it has been "personalized". This leads to hashes being a less useful approach. On mobile platforms, however, with difficult app vulnerability exploitation and with centralized software distribution, hash-based malware detection gains value. Malware is usually spread as part of repackaged or seemingly harmless apps, which have to be distributed via app markets to reach the highest possible number of users. Thus, malware package files will be largely identical and hardly varying. Furthermore, many malware types on the Android platform make use of dynamic fetching of malicious payload files. Such components will often be the same across many devices.
The aforementioned aspects contribute to a high level of file homogeneity, making hashes of full files a valuable characteristic for malware detection. The antivirus interface should thus provide methods to calculate typical hash functions such as MD5 or functions from the SHA family. Such a hash would be calculated given an alias or a file path. Again, the antivirus interface has to store a list mapping the aliases to file paths of the directory listed last by the antivirus app, unless a trapdoor function is used.

4.2.3 Heuristics

Apart from pattern or hash matching, antivirus software also uses heuristic techniques to detect malware. The major advantage is that potentially malicious behavior can also expose currently unknown malware, and is often unaffected by minor alterations which would circumvent pattern or hash matching. Such heuristics are mostly based on sequential execution of specific library, API, or system calls, and may contain also certain passed arguments.
Heuristic techniques can be divided into static and dynamic heuristics [17]. While the former is easier to conduct, the latter yields better results. However, it is not trivial to implement on the Android platform and introduces potential security risks.

Static Static heuristic techniques do not execute the file to be examined and do not observe dynamic behavior. Instead, they observe the existence and order of certain call patterns [9]. For example, to find a specific system call sequence in a binary file, the resulting opcodes[1] can be searched for [5]. To include certain parameters, the opcodes preceding the system call itself and parameters can also be examined. The latter works especially if constant values (such as system call numbers or other parameters) are involved and are included inline with the opcode itself (as "immediate value"). This way, register values can also be identified, such as the system call number or other parameters. Pointers to any dynamic data, however, cannot be easily evaluated due to their sole validity during runtime.
Java apps can also be examined this way, although the examination method has to be adjusted to the specific properties of Dalvik bytecode [24]. Parameters belonging to certain calls may not be easy to identify. Still, call sequences can be examined.
To support these static heuristic techniques, the mechanism devised for signature scanning can be used. For this matter, it needs to support patterns passed as byte-sequences and described via regular expressions, although granularity at byte level can also be helpful for common signature-based approaches.
The two aforementioned heuristic techniques result in a signature-representable pattern form. Further static heuristic techniques are presented by Zhou et al. [24], including analysis of an app's manifest file and the structural layout of an app's package file. These techniques can also be carried out by antivirus apps by extending the interface presented in this paper.

Dynamic Dynamic heuristics test executable files by running or emulating them [9]. This is done, for example, by sandboxes or emulation environments, but other techniques may also be used. They can follow the execution flow of a program without harming the host system.
On the Android system, when devising a possible antivirus interface to support

[1] E.g., the traditional software interrupt `int 0x80` or current methods like `syscall` on the AMD64 architecture and `sysenter` on the x86 architecture each have specific opcodes

these type of operations, multiple problems arise. Foremost, letting one app execute any code by another app is a breach with the Android security architecture. Depending on the implementation of the dynamic heuristics component, this may put the host system at risk. Furthermore, very sophisticated malware may also attack and compromise its environment. Such attacks have already been carried out successfully against Google's Bouncer service [21], which is also a dynamic heuristic malware recognition software. With the capabilities granted to antivirus software by the antivirus interface described in this paper, malware would gain widespread access to the system. Furthermore, implementation problems arise. If a sandbox or emulation mechanism is used, all files the executable to be examined is dependent on need also to be loaded into the execution environment. This may be a large number of files, and could, e.g., contain most of the working directory of an app, but also system libraries.
If such dynamic heuristics were to be implemented, much of its functionality would have to be placed in the antivirus interface itself. We regard a direct interface for antivirus software to load executables (both binaries and apps) a very high risk which should be avoided. Also, typical Linux kernel techniques such as `chroot` and interception of file system writes or redirection to a RAM-disk for sandbox execution should only be carried out by trusted, well-controlled software. Hence, possible approaches, e.g. `chroot`-based or with interception of any potentially harmful writing activity to the file system, to the network interface, or to other interfaces, ought to be implemented by the antivirus interface itself and not by antivirus software. A feasible implementation for binaries would be a safe and controlled execution of the `strings`, `strace`, and `ltrace` utilities in a `chroot` environment or without any write access to network interfaces or the file system for the file to be examined. Java app execution can also be conducted in a Dalvik VM instance with blocked write access and an attached debugger.

4.2.4 Limiting Operations to Executable File Types

To limit operations to suitable filetypes, i.e. heuristics operations to executable files, the antivirus interface should also offer a method returning the filetype of a file, given its alias or path, e.g. as a MIME type. Such functionality is already included in the `file` utility. Alternatively, it may also provide a boolean method stating whether a file is executable or not. This preserves resources and ensures that only executable files are loaded into heuristic environments. Only returning a boolean value instead of a MIME type increases data privacy.

4.3 File System Monitoring

File system monitoring, often referred to as "on-access scanning" or "guard", checks for any changes made to the file system and tests new or changed files automatically. We identified multiple approaches to facilitate this feature which, however, cannot monitor the whole file system.

Monitoring of the Package Database The Android package database, located in `/data/system/packages.xml`, may be monitored for changes by the Android antivirus software or by the antivirus interface. Such monitoring can be implemented by means of the `inotify` Linux kernel mechanism [19], which allows monitoring software to be notified automatically if changes to a file system object, i.e. a file or a directory, occur. This way, the antivirus interface can inform the antivirus software via Intents about changes to the package database. In such Intents, paths to the new working directory and to the application package file of newly installed apps can be passed to the antivirus software for scanning. If the monitoring is performed by the antivirus software itself, it needs to extract this information from the package database on its own. While the working directory itself is not included in the package database, it may be derived from the package name.

Intents upon App Installation When a new package is installed, the Android system broadcasts a `PACKAGE_ADDED` Intent which contains the newly installed app's package name. This way antivirus software can be informed of changes to the package database without having to manually monitor it. Upon receiciving of this intent the package database can be read to identify the new app's package file's path and the working directory. The path of those can then be passed to the file operation methods offered by the antivirus interface to perform malware detection techniques on the newly added files and directories. Furthermore, the event of a new package being added can also be used by the antivirus interface itself to place new `inotify` handles to monitor the file system.

Monitoring of the File System Monitoring of the whole file system cannot be accomplished without major efforts and complexity. However, single directories can be monitored by means of the `inotify` interface. As described above, it automatically notifies programs if changes to file system objects occur. `inotify` can be applied at different levels. For example, a `inotify` handle can be placed on the `/data/app` and `/data/data` directories to monitor for new app installations. To monitor single app activity more closely, `inotify` handles can also be placed to all working directories of apps (in `/data/data/[package name]`) and their subdirectories. If such an `inotify` handle notifies the antivirus interface of any changes, the antivirus interface sends an Intent containing one or multiple paths or aliases to the antivirus app. The antivirus app can then perform its operations. This approach, if used on subdirectories of working directories too, allows for close observation of file system activity by any user-installed app.
To ensure full monitoring of the working directories of all apps, if an `inotify` handle notifies the antivirus interface (and subsequently, the antivirus app) of a newly added directory, the antivirus interface can again place an `inotify` handle on the newly added directory.

4.4 Securing Access to the Interface

While much attention has been paid to ensuring data privacy and security of user and of system files, the devised antivirus interface still has to be adequately secured from unauthorized access. Like many interfaces of the Android API, this can be achieved through a permission specifically securing access to this interface. This permission has to be guarded by a suitable *protection level*. There are four different *protection levels* [1]. They determine which apps may be granted the respective permissions.

- *normal*: Lower-risk permissions with minimal risks to other applications or user data. Granted automatically.
- *dangerous*: Higher-risk permissions granting a requesting app access to private user data or potentially security relevant system resources. Has to be confirmed at installation time by the user.
- *signature*: Very high risk permissions which are only granted to apps signed with the same key as the app providing the requested resource.
- *signatureOrSystem*: Same as signature, but also granted to apps stored on the read-only system partition of an Android device.

The permission of the antivirus interface designed in this paper ought to be secured with a protection level of *signature*. The signature has to be provided by Google. As the number of antivirus products is very limited[2], it is a reasonable effort for Google to perform thorough checks of antivirus software before it is signed by Google. We consider this feasible because a capable antivirus interface, enabling antivirus software to use desktop-equivalent detection techniques, would significantly improve the current situation of malware recognition on Android. Largely trusted and well-established companies might also be less thoroughly checked, decreasing the effort required by Google even more. The *signature* protection level ensures that only apps checked and certified by Google can use the antivirus interface, protecting it from access by all other apps.

5 Discussion

We have described above the basic mechanisms which have to be offered by an antivirus interface to carry out malware recognition techniques using signatures and heuristics, to traverse the file system, and to monitor important parts of the file system. Regarding implementation, several issues, options and additions arise which can optimize the performance of the interface and thus of antivirus apps using this interface. Other matters of interest include data and especially file system path privacy, the limited possibilities for action after malware detection, and related work. All of these will be discussed in the following.

[2] 41 products were tested by AVTest [4]

5.1 Aliases for Addressing

We proposed aliases for input and output of the directory listing and file system traversal mechanism. For this mechanism to work, the antivirus system needs to be able to translate aliases back into file system paths. Different options exist:
(1) The antivirus interface maintains a database of all file system objects and their related hashes. We discourage this option, as it requires file system monitoring and a lot of resources.
(2) Dynamic calculation of hashes and file system neighbours (including parents): For the latest directory listing requested by an antivirus app, the antivirus interface calculates and stores the hashes of all file system objects which have been returned by the listing function. Also, it stores the hash of the direct parent to allow traversal up in the file system hierarchy. This eliminates the need for file system monitoring, as all hashes are calculated dynamically, and requires only few resources. Arbitrary aliases, however cannot be passed to the antivirus interface, for example for specific single files and directories. Still, commonly this feature would be requested by the user, who would enter a file path instead of an alias.
(3) Trapdoor functions: Trapdoor functions ($f(path, t) \longrightarrow alias$, $f'(alias, t) \longrightarrow path$) are two-way functions which map an input value to an output value and vice versa, given a *trapdoor* parameter. Only a party in possession of the trapdoor parameter may compute the functions correctly. The antivirus interface may use such a function to translate between paths and aliases. Keeping the trapdoor parameter a secret from antivirus software, data privacy is ensured completely. Antivirus software is only presented with the aliases. Also, this allows for the passing of arbitrary aliases to the antivirus interface, which is not possible with option (2). A possible trapdoor function, for example, can be RSA.

5.2 File System Access Backend

The antivirus interface itself needs to be granted access to the whole file system, or to specified parts of it which should be scanned, for example the package file directories (`/data/app`) and the working directories (`/data/data`). This access is required to translate real file system paths to aliases and to allow for operations on file system objects which are to be provided to the antivirus software.
To enable this access, the interface can be assigned a permanent, hard-coded user ID with corresponding capabilities, i.e. read-only access to the whole or parts of the file system. In this case, the interface would need its own process. Alternatively, legitimate antivirus apps can receive the predefined user ID and then call the interface's methods, which would be the preferred method. Read-only access to the while file system (or only to specific portions like the working directories and all subdirectories) can be granted using access control lists, as defined in POSIX-1003.1e, which are supported by the Linux kernel [13, 18].

5.3 Efficiency

To increase efficiency, the functionality of the antivirus interface can be partly parallelized and automated. Instead of playing an intermediary role for antivirus software, which traverses the file system through information provided by the interface and which issues pattern matching commands for one file at a time, more functionality can be shifted to the interface itself. For example, as in our approach the signatures are patterns for regular expression matching, the interface could be provided with the whole signature database at the start of a scan in a compatible format. The interface (which would become more of a system service) can then traverse the file system on its own, also eliminating the need for aliases. Also, the signature database can be cached, instead of passing signatures every time when processing a file. However, this way antivirus software would be reduced to supplying the signature database at the start for signature-based and static heuristic detection.

5.4 Dynamic Heuristics

While for signature and static heuristics recognition antivirus software can be reduced to supplying pattern databases, dynamic heuristics usually require more complex data structures and analysis techniques. As argued in the section on dynamic heuristics during the devising of our interface, a multitude of problems emerges, including implementation difficulties and system security, as demonstrated by attacks against Google's Bouncer service [21]. Different approaches for implementation exist, such as a `chroot` environment, emulation, or intercepting write access to the file system or network interfaces when running tools such as `strings`, `strace` and `ltrace`. Other researches have already developed several dynamic heuristics-based analysis systems for Android apps. Most of them, however, are not suitable for deployment on devices. Hence, approaches to dynamic on-device heuristics still have to be developed and will be researched by us in the future.

5.5 Post-Detection

This paper has described different components of an interface for malware recognition for the Android platform. Removal of malicious files has not been part of the described interface. We argue that removal practice should be discussed separately, as several different options and problems arise. Introducing the possibility to remove arbitrary files would violate Android's security architecture. Also, antivirus software equipped with such capabilities would become a very valuable target for manipulation and attack. Furthermore, system instability may occur when single files can be removed by third-party software oblivious to file dependencies. If malicious files are only found in one app's working directory, that app may be removed automatically or the user may be informed of the apps potentially malicious behavior.
As it requires thorough consideration, if and in what way antivirus software

should be able to remove files at all, we conclude that this matter ought to be discussed separately. Current practice by antivirus software is to initiate an uninstallation dialog presented to the user, who then has to confirm the uninstallation.

5.6 Related Work

In Android 4.2, Google introduced a reputation-based system for notifying users of possibly malicious apps installed on Android devices. However, analysis suggests that its warnings are primarily based on SHA1 hashes and package names and there is much room for improvement [15]. In the beginning of 2012, Google also put their Bouncer into operation, which conducts dynamic heuristic malware recognition for new app submissions to Google Play (formerly the Android Market). These two systems show that malware recognition is considered an important aspect of Android security by Google. However, they are not deployed on devices, and have known weaknesses. Our approach addresses effective on-device malware recognition.
Many other researches have already developed various Android malware detection systems, both static and dynamic. In [24] the authors establish different classification schemes for Android malware through permission-based behavioral footprinting and heuristic tests on app bytecode, also including call graphs and data flow analysis. A crowdsourcing approach which uses call trace files from Android devices is developed in [7], also giving a good overview of the state of static and dynamic malware recognition techniques. The techniques developed in that publication make use of `strace`, as also proposed in this paper. Practical analysis and detection systems include Mobile-Sandbox, to be presented at SAC'13 and already in operation at *mobilesandbox.org*, AASandbox [6], and App-Ray developed by Fraunhofer AISEC [12]. Parts of the techniques described in these publications and used in the aforementioned systems can also be implemented as additions to our antivirus interface presented in this paper.

6 Conclusion

Due to current platform limitations, antivirus software on Android is not very effective. Especially, it is oblivious to activities in apps' working directories and to dynamically downloaded malicious files. However, as malware sample and family numbers increase and device manufacturers are reluctant about supplying security updates, effective malware detection and warning techniques gain in importance. This is also shown by efforts by Google, which introduced a reputation-based warning system for potentially malicious apps in Android 4.2. This adds another layer of on-device security after Google's Bouncer service, which may be circumvented by dynamic code fetching techniques, or may even be attacked directly [21].
The main contribution by this paper is the possibility for antivirus software to scan arbitrary files on a device, without violating Android's security architec-

ture. For this, our interface acts as a proxy service. Arbitrary file system objects can be traversed, checked, and file system portions can also be monitored. This applies especially to apps' working directories. Consequently, apps can no longer download malicious files at runtime and execute them without antivirus software noticing. For carrying out recognition techniques on file system objects, our devised methods allow for a number of operations. These include pattern matching operations to aid in signature-based malware recognition approaches, but possible approaches for heuristics have also been described. Access to our interface is restricted to apps which are trusted by Google, using mechanisms already featured by the Android platform.
The antivirus interface designed in this paper enables Android antivirus apps to deploy techniques similar to those of desktop malware recognition products not only on package files and small file system parts, but on a whole device. The increase in effectiveness is significant, as up until now package names and app package files were the primary criteria for recognition of malware installed on devices. Seemingly harmless package files which contain code to download arbitrary malicious files are currently not detected. With our interface, recognition techniques can also be applied to working directories of apps, and also to other file system areas.
We consider signature and static heuristics-based recognition techniques for potentially arbitrary file system objects a valuable addition to Android device security. Monitoring of portions of the file system helps against dynamically downloaded files. Antivirus software with said capabilities would be an additional layer of security for Android devices, improving the level of protection offered by antivirus software on Android by far. Research into on-device dynamic heuristics and post-detection courses of action will be part of our future work. Parts of the techniques already developed by other researchers for off-device analysis and included in the section on related work can potentially also be used for on-device recognition.

References

[1] Android Developer's Guide. <permission>. https://developer.android.com/guide/topics/manifest/permission-element.html.

[2] Android Developer's Guide. Saving files. https://developer.android.com/training/basics/data-storage/files.html.

[3] Android Open Source Project. Android Security Overview, October 2012. http://source.android.com/tech/security/.

[4] AV-Test. Test Report: Anti-Malware solutions for Android. Technical report, March 2012. http://www.av-test.org/fileadmin/pdf/avtest_2012-02_android_anti-malware_report_english.pdf.

[5] D. Bilar. Opcodes as predictor for malware. *International Journal of Electronic Security and Digital Forensics*, 1:156–168, 2007. Available at: http://inderscience.metapress.com/content/N760240L16832162.

[6] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. Camtepe, and S. Albayrak. An android application sandbox system for suspicious software detection. In *5th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 55–62, Oct. 2010.

[7] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and Privacy in Smartphones and Mobile devices*, SPSM '11, pages 15–26, New York, NY, USA, 2011. ACM.

[8] J. Burns. Exploratory Android surgery (talk slides). In *Black Hat Technical Security Conference USA*, May 2009. Available at: https://www.blackhat.com/html/bh-usa-09/bh-usa-09-archives.html.

[9] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium*, SSYM'03, Berkeley, CA, USA, 2003. USENIX Association.

[10] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *IEEE Security & Privacy*, 7(1):50–57, Jan.–Feb. 2009.

[11] R. Fedler, C. Banse, C. Krauß, and V. Fusenig. Android OS security: Risks and limitations. Technical report, Fraunhofer AISEC, May 2012.

[12] Fraunhofer AISEC. App-Ray. http://app-ray.de/.

[13] A. Grünbacher. POSIX access control lists on Linux. In *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference*. USENIX Association, June 2003.

[14] S. Höbarth and R. Mayrhofer. A framework for on-device privilege escalation exploit execution on Android. In *3rd International Workshop on Security and Privacy in Spontaneous Interaction and Mobile Device Use (IWSSI/SPMU 2011)*, June 2011.

[15] X. Jiang. An Evaluation of the Application ("App") Verification Service in Android 4.2, December 2012. http://www.cs.ncsu.edu/faculty/jiang/appverify/.

[16] X. Jiang. New RootSmart Android Malware Utilizes the GingerBreak Root Exploit, February 3, 2012. http://www.csc.ncsu.edu/faculty/jiang/RootSmart/.

[17] E. Konstantinou and S. Wolthusen. Metamorphic virus: Analysis and detection. Technical report, Information Security Group at Royal Holloway, University of London, 2009. Available at:

http://media.techtarget.com/searchSecurityUK/downloads/RH5_Evgenios.pdf.

[18] Linux manual page. acl(5). Available at: http://linux.die.net/man/5/acl.

[19] Linux manual page. inotify(7). Available at: https://www.kernel.org/doc/man-pages/online/pages/man7/inotify.7.html.

[20] H. Lockheimer. Android and security, February 2012. http://googlemobile.blogspot.com/2012/02/android-and-security.html.

[21] J. Oberheide and C. Miller. Dissecting the Android Bouncer. In *SummerCon '12*, June 2012. http://jon.oberheide.org/files/summercon12-bouncer.pdf.

[22] 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.

[23] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy (SP)*, pages 95–109, May 2012.

[24] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th Network and Distributed System Security Symposium*, February 2012.