Android APK on-the-fly tampering

Zdeněk Říha, Dušan Klinec and Vashek Matyáš

***Abstract*—The Android operating system is widely deployed and relied upon by both providers and users of various applications. These applications get frequently downloaded from other sources than just Google Play. This makes Android and its application treatment a popular target for attackers. We first present an automated offline attack injecting a previously prepared code to a previously unseen Android application installation file (APK) in an automatic manner. Moreover, we present a novel transparent on-the-fly extension of our attack when a proxy server performs code injection during a new APK download.**

***Index Terms*—Android security, application security, application download, code injection, malware contamination**

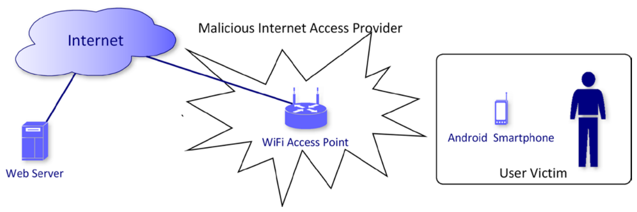

Fig. 1. The high-level architecture of our attacker's setup. The infrastructure for the Internet access (typically a WiFi access point) is under the control of an attacker who can manipulate unprotected communication.

I. INTRODUCTION

The Android mobile operating system has penetrated 88% of the smartphone operating system market by 2016 [5]. The bare operating system as delivered by phone manufacturers typically provides just a basic functionality. Therefore, it is more-or-less expected that most users install additional applications either to enhance smartphone features or just for fun. The official way to obtain Android applications is to use the Google Play service. Android phones use the pre-installed application (client) that directly connects to Google Play servers. Alternative sources of applications are also supported, but this feature is disabled by default for security reasons.

Still, many users activate the feature allowing installation of applications from other (often unknown) sources to be able to install applications with alternative distribution models, not present on the Google Play Store (e.g., tourist guide applications are often distributed locally on-site in places without Internet access).

Various and numerous applications are banned from the distribution in Google Play (e.g., advanced security scanners software requiring root privileges, copyright infringement materials, advertising blocking applications or even privacy tools aimed at stopping other applications from collecting data on users).

We demonstrate two methods of automatically injecting attacker's code (e.g., backdoor) into the APK files transparently to the user. See the high-level architecture of the typical attacker's setup in Figure 1. The user would see the APK download phase progress as usual, but receive a modified file. Since "free" installation of applications from ad hoc sources is a crucial feature for Android smartphones, we raise this issue to both the user and developer communities.

Motivations of the attackers to inject a malware into Android applications can vary. The malware can have the form of a spyware, leaking the location of the user, SMS messages and other sensitive user data, etc.; botnet client, transforming the mobile phone into a zombie, or anything else that can lead to economic or other profit (e.g., Bitcoin mining).

APK file integrity is protected with a digital signature. The list of hashes for practically all the files in a package is digitally signed and stored as a PKCS#7/CMS signature file (including the X.509 certificate of the signer). The digital signature is verified during the installation process and if the verification fails then the installation is aborted. This mechanism is able to detect integrity issues within the APK file (e.g., missing files, extra files or modified files) and addresses download errors (e.g., a truncated APK file).

Experience from other application domains (most notably, but not exclusively, the SSL/TLS) shows that users have serious issues when having to make decisions about Public Key Infrastructure (PKI) tasks and questions [6], [15]. Android is based on a very simplified PKI.

For the APK files, the signer certificate is self-signed and is generated by a developer without any aid of a Certification Authority (CA). When a new application (a new package name) is installed, any certificate is accepted. The signer's certificate is only important in some particular situations – the signer must be the same when upgrading an application, to allow applications to run in the same process and to share code or data between applications through permissions.

In our scenario, we assume that the user is downloading and installing a new application. Therefore, it is easily possible to modify the content of an APK file and afterwards to sign it with a different private key of attacker's choice.

As no CA is involved, the values of the name fields in the (public key) certificate can be arbitrarily chosen by the attacker. To sign the application, we use the same *jarsigner* utility (part of the Java Development Kit) that developers use. We generated a new private key and certificate for our experiment described below.

Affiliation: Masaryk University, Faculty of Informatics, Botanicka 68a, CZ-602 00 Brno, Czechia. (zriha | xklinec | matyas @fi.muni.cz)

A. Aims and limitations of our work

The *core contribution* of this paper is a novel approach for the on-the-fly automated modification of APK files. In the case of the offline APK modification we assume to have the complete APK file at our disposal (i.e., fully downloaded file) and run a script to inject the payload. Yet our primary contribution comes with the demonstration of a code injection (at a proxy) – the online version of the modification modifies the APK on-the-fly while being downloaded from a remote server to the Android device of the victim (i.e., during the man-in-the-middle attack). The online modification must create an impression of a continuous download of the APK. To our best knowledge, the on-the-fly attack is a novel approach not published before.

Our paper investigates the security consequences of the feature allowing for installation of applications from unknown sources and presents a way how to automatically inject malware into Android applications. Security of the Google Play service is out of the scope of this paper.

Our ultimate goal is to demonstrate the ease of injecting an additional code so that it remains hidden from the phone user. The process of the code injection (even into previously unseen applications) is fully automated and the download of the package is not disrupted from the user point of view. The process of the modification works in a streaming mode, changing the APK file on-the-fly.

B. Paper roadmap

In Section II, we map related work on the Android attacks topic. Section III describes the basic offline variant of the attack, Section IV outlines the on-the-fly variant of the attack. Technical details are described in Section V and Section VI demonstrates the practical usability of the proposed approach in our experiments. Section VII briefly discusses possible countermeasures and the following section concludes our paper.

II. Related work

The APK file modification is not novel. It has been demonstrated several times that the APK file can be decompressed, disassembled, modified and then reassembled and repackaged [25], [1]. There are standard Java tools that can decompress the APK file, disassemble the compiled Java classes, recompile the source code and repackage the APK file including the APK signature generation. The analysis of the source code and its proper modification usually remains a manual and case-by-case work.

Code injection was also previously demonstrated with other executable file formats (e.g., Windows EXE [4]).

In [7] E. Aydogen and S. Sen generate repackaged (obfuscated) APK files using *apktool*. The resulting APK files are used to evaluate the performance of antivirus systems.

A DroidChameleon framework presented in [23] is a tool for generating malwared versions of the Android applications with use of transformation techniques, repackaging and reassembling APKs. They tested common antivirus products and commercial antimalware applications with modified APKs. All APK modifications are offline, the paper does not discuss on-the-fly APK manipulations.

ADAM [31] is another malware generator framework that uses repackaging and obfuscation to generate new malware samples to stress antivirus products. The paper focuses only on an offline APK repackaging.

In [18] J. Jeon et al. intruduces Dr. Android and Mr. Hide. Dr. Android is a tool that removes application permissions and replaces them with calls of fine-grade variants accesible through Mr. Hide (a set of Android services). Dr. Android is based on the *apktool* (to repackage the application) and *redexer* (to transform the Dalvik bytecode).

AppSpear [28] is rebuilding packed and protected applications into normal form so that they can be analyzed by standard tools.

APK files are often decompiled to analyze the behaviour of applications. In [13] W. Enck et al. presented the *ded* tool for decompiling Android DEX code to the Java source codes and carried a static code analysis on 1100 Android applications with the Fortify tool. In [8] L. Batyuk et al. statically analyze the bytecode and produce precise security reports. P. Bertholome et al. [9] extend the work and not only reports the situations where the user's privacy can be disclose, but also inject a new code to allow use to decide whether he want to prevent the operation.

Many Android vulnerabilities have been published in the past few years, but APK related vulnerabilities are not that numerous. The most serious bug on this topic is the so-called Android Master Key vulnerability [16] affecting APK installation in such a way that the tampered package is accepted as a valid one. The vulnerability is based on the fact that the APK file, having the ZIP structure, can contain multiple entries of the same name, this is quite unusual, but generally allowed in ZIP files. The existence of such duplicates is not explicitly checked by the installer. The APK installer and the signature verifier are separate components, each using a different third-party ZIP parsing library.

The core problem is that the signature verifier takes into consideration the first ZIP file entry while the installer takes the last one. Exploiting this vulnerability is straightforward. Taking an original APK file, it is sufficient to insert infected files as second ZIP entries with duplicate names. As a result, the original ZIP entries are verified while the infected files are being installed. The Android Master Key vulnerability is a serious bug that has been fixed in the Android version 4.3 (Jelly Bean).

The Android Master Key vulnerability is an effective way to infect an APK file. The core advantage of this approach is based on the fact that no modification of the original signature is required (even if the installed content has actually changed). Our approach is also based on the fact that APK file has a ZIP structure but we are not using the Android Master Key vulnerability, instead we are generating a new signature as we are changing some files inside the APK.

III. Offline APK modification

Android applications are distributed in the form of APK [2] files. APK has internally a ZIP structure that includes primarily the following elements:

- *Manifest:* Metadata including versions, permissions and bindings (file `AndroidManifest.xml`);
- *Compiled portable code:* Java classes in DEX format (file `classes.dex`);
- *Compiled native code:* Platform dependent compiled code – separate folders for particular platforms (folder `lib` and its subfolders);
- *Precompiled resources:* For example compiled XML files (file `resources.arsc`);
- *Other resources:* Images, icons, sounds, etc. (folders `res` and `assests`);
- *Package integrity data:* The digital signature (folder `META-INF`).

In the offline APK modification we automate the use of commonly available Java tools to decompose the APK file, modify the package content and build the APK again.

In the very first step, we call the *apktool* to decompile the package. This leads to unzipping of the file structure, conversion of binary XML files to textual formats (including the `AndrodManifest.xml` file) and disassembly of the `classes.dex` file (containing all compiled Java classes) into the so-called *smali* files (textual versions of the Dalvik bytecode).

The *apktool* provides a good compatibility for the APK manipulation, but there are some packages that fail to decompile with *apktool*. This basically sets the success rate of the attacks. Particular numbers depend significantly on the source of the database of APK files and also on the version of the *apktool*. In a database of 500 APKs downloaded from the zippyshare.com about 15% of the APK files fail to decompile with the *apktool*.

The next step of the offline attack is the smali files modification. In this phase we use the pre-prepared smali code we would like to inject. Those are added to other disassembled smali files. Usually we also need to modify the existing smali files in order to start the malicious code automatically after the application startup, to provide binding to the original code or to register to system events. Smali file modification is straightforward and also automated. In our scenario we tested starting a new service and registering for interesting intents (e.g., *ACTION_BOOT_COMPLETED, SMS_RECEIVED*). The new functionality (the new service) is separate from the original code and there is no aim to actually modify the original functionality, so chances of unintended interactions/malfunctions are very low. We particularly need to avoid naming collisions.

The `AndroidManifest.xml` has to be modified in the following cases: a) our code needs permissions missing in the original application, then we add the required permissions; b) new service/activity/IntentReceiver is added, it has to be registered in `AndroidManifest.xml`. Once the modification is finished, the assembly process takes place to create a tampered APK. This includes calling the *apktool* to compile the `AndroidManifest.xml` and smali files. Then the whole package is signed with a newly generated asymmetric key using the *jarsigner* utility. An optional step is to use the *zipalign* utility that aligns zip entries at 4B boundaries. The result of this process is a tampered APK file with the injected code, still correctly signed with a new certificate and private key.

IV. ONLINE APK MODIFICATION

The previous approach works well in cases where the attacker has got a big repository of APK files available so they can be infected and then provided to users. A more universal approach is to build a proxy server that modifies on-the-fly the APKs being downloaded in order not to raise any suspicion of users about the potential malicious activities being performed on the APK file during the download process.

The online attack works in the streaming mode. Thus the APK file being downloaded is read by our proxy and on the other end the proxy produces an infected APK file. The main idea of this process is the re-ordering of files inside the APK ZIP file structure.

Usually there is no need to modify resource files in the APK and typically the resource files occupy a non-negligible amount of space in the APK. We use this fact to create an impression of continuous download. Files that have to be modified (e.g., `AndroidManifest.xml`, `classes.dex`, digital signature files) are stored sideways, postponed from being sent to the user. The rest of the files (e.g., resources) are sent to the user directly. We use a stream ZIP parser to perform this task.

A. Attack launch

Once the whole APK file is available at the proxy side, the offline attack is launched on to the downloaded APK file. Note that the user has downloaded only files that are not modified during the attack and does not have the complete APK file yet. From the user's perspective, the download process is still in progress. When the offline attack finishes, the tampered APK file is analyzed and files differing from the original APK file are transmitted to the user. This would normally lead to a download pattern where a significant part of the APK file is downloaded with a normal speed, then the connection hangs for a moment (ranging from seconds to minutes), and then the download continues with the normal speed again.

To avoid a visible delay in the middle of the download process, it is possible to artificially reduce the download speed from the beginning so that the delay in the middle is not present or minimized.

The main benefit of this approach is that download on the user side starts quite quickly (i.e., the download progress bar shows the download has really started). The naïve strategy would be to use the offline attack on the proxy side and once the attack finishes the whole infected APK file would be dumped to the user's download stream. Yet that could raise suspicion of users since the modification takes some time — from a few seconds to a few minutes — the user would see 0% at the progress bar for a significant time. This could indicate connectivity problems or indications of malicious modifications. A cautious user might tend to cancel the APK download.

B. File size issue

There are two major ways of transfer encoding using HTTP as the download protocol. The server can use either the chunked encoding or the normal mode. In the chunked encoding, the overall content length of the payload is not sent to the user in the HTTP headers. This mode of transport enables to generate the content dynamically or support scenarios where the content length is not known to the server at the beginning. The downside of this approach is the missing progress illustration of the download. The other option is to use the normal transfer encoding with the content length header present among the HTTP headers.

Since we want to mimic the normal file download with the progress displayed to the user, we need to use the latter option. The problem is that the APK modification inflates the APK file by some amount of data, from bytes to kilobytes. Since estimation of the difference of the file sizes is not reliable, we cannot tell the resulting APK file size precisely before the APK tampering takes place.

Our approach is to estimate the resulting APK size by adding some extra space as a reserve. It should hold that real APK size is smaller than the APK size sent in headers, otherwise the user receives an incomplete and thus invalid APK file. Then we have additional bytes of data that need to be added to the APK somehow. Each ZIP file entry has an *Extra field* according to the ZIP standard [21]. This Extra field is of a variable size, taking at most 65535 B. It serves for storing a special application/platform information in a ZIP file and provides extensibility to the ZIP format. In most APK files it is usually unused. This is the place where we put additional bytes to obtain a “padded” version of the APK file where file size matches the one sent in the beginning of the transfer.

Padding is done once the APK tampering has been finished on the server side and the size of padding bytes is calculated. In this phase of the attack we have the list of files that need to be sent to the victim. The padding bytes are spread across the extra fields of this file entries and sent to the user. As the ZIP structure itself is not a subject of the signature the padding or the order of files in the ZIP does not affect the sigature of the package.

V. Technical details

The offline attack as described above was implemented in Java. The attacker has to prepare the code to be injected. The code being injected needs to be expressed in the smali language, but an attacker can prepare the malware code in Java, compile it (using standard development tools) and decompile it (using the *apktool*) to obtain the needed smali code. This needs to be done only once and the same malware smali code can be injected into many APK files.

The offline attack wraps the whole process of decompression, decompilation, injection, compilation, signature and compression. The process is a simple sequence of a few steps heavily using the standard *apktool* utility, thus providing good compatibility with APK files of various types. The disadvantages of the utility include the complexity (of what it is doing in a single command) and therefore also relatively slow speed.

The online attack was also implemented in order to demonstrate its applicability. Please note that it works only for HTTP, not for HTTPS. In our setup we have used a dedicated proxy server (a Fedora Linux box[1]) with two network interfaces. `Interface 1` is an Ethernet type connected to the Internet. `Interface 2` is a WiFi card for a hotspot emulation. We have chosen the *hostap*[2] software that emulates a wireless access point on this interface.

The hostap is connected to the *TinyProxy*[3], which is an open-source lightweight transparent proxy. We modified the source code of the TinyProxy to hook all `HTTP GET` requests for files with the APK extension. If an APK is being downloaded, it invokes the Java implementation of the online attack and passes the download stream to its standard input while sending its standard output to the user. TinyProxy also estimates the final file size of the APK after modification by adding a fixed amount of bytes to the total size.

The online attack application is written in Java, using the ZIP stream parser from the *Apache Commons*[4] Library. It implements the attack described earlier, together with a simple download speed limit algorithm.

Our testbed implementation is placed into public domain under the Apache License v2 hosted on GitHub: `https://github.com/ph4r05/ZIPStream`.

VI. Experimental Validation

In order to validate our approach, we tested our setup with a dozen of real APK files on real smartphones (Samsung Galaxy S3, HTC One X, Samsung Galaxy S2 mini, Motorola Moto G and Sony XPeria Z2) with Android of versions 2.3.7, 4.2.2, 4.3, 4.4.4 and 5.0.2. The principles of the APK file modification are independent on the Android version and do not depend on a particular vulnerability of the OS. As long as the signature of the APK file can be made by any signer, the attack will basically work. All tested APK files worked on all tested phones.

As the repackaging itself is automated, it is easy to perform in a larger scale. We repackaged over 100 of APK files. On other hand, installation, running and verification that the original functionality of the APK was not affected is a manual work. We tested that on few dozens of applications.

Our tests aim to show the times needed to download sample APK files. We performed these tests with our transparent proxy server connected to a fast local network with the web server. The download of the original file that was not intervened by the proxy server was very fast. When the file was modified on-the-fly on the server then the modification needed a non-trivial amount of time and the download took significantly more time (and the file being downloaded was larger). The on-the-fly modification suffers from a well visible signature of the download process when the download practically stops for a long moment. Our solution to this problem is based on

[1]The basic hardware configuration was intentionally chosen as a low performance (router-like) computer: Intel Pentium 4 CPU 3GHz Dual Core, 2GB RAM, 7200 RPM SATA hard disk.

[2]http://wireless.kernel.org/en/users/Documentation/hostapd

[3]https://banu.com/tinyproxy/

[4]https://commons.apache.org/

APK file	Original size	New size	Time (orig. file)	Time (modified file)	Time (modif'd file with speed lim.)
Navigation utility ('C')	293 kB	793 kB	1 s	13 s	51 s
Game ('F')	4126 kB	4626 kB	2 s	19 s	53 s
Antivirus solution ('E')	3523 kB	4023 kB	2 s	31 s	62 s

TABLE I
THE TIME STATISTICS OF THE DOWNLOAD PROCESS OF THREE TYPES OF APPLICATIONS.

slowing down the user download speed from the beginning so that the on-the-fly modification can be masked by a slow link behavior.

Table I illustrates the times needed to download our sample APK files (injecting a sample privacy-related malicious code):

- File 'C' is a very basic navigation utility.
- File 'F' is a basic game with rich graphics and a simple logic.
- File 'E' is a leading provider's antivirus solution with a complex code.

These three files represent various characteristics of applications, in particular the size of the application and the proportion of the code and resources. Note that the sizes of files 'E' and 'F' are similar, but the processing times of the files differ significantly. We need to emphasize that the most time consuming part of this effort is the DEX file processing. The more Java code (the larger the `classes.dex` file) the longer the package processing takes as the decompilation and compilation of the Java/smali code is more complex than processing of other files (e.g., images and other resources). This explains the longer processing time of the file 'E', which contains more Java code than file 'F'.

Our tests were performed using a very low performance computer to imitate single purpose devices acting as routers or access points (e.g., Linux based APs). A more powerful computer can perform better. E.g., we ran some of the tests on a notebook based on Intel Core i7-3540 (3GHz) and an SSD disc – such a setting can reduce the modification times by about one half.

The speed limitation needs to estimate the ideal speed that can be achieved constantly. The time needed to process the package depends on the proportion of the files that need not be modified (and can be therefore sent directly) and files that will be (potentially) modified (and will be sent after the package modification and resigning). The speed also depends on the size of the Java code that significantly influences the decompilation and compilation times. Setting the compression level may introduce another dimension to choose between the compression level and slowdown in the modification/transfer speed.

Figure 2 demonstrates one of the performed tests. It represents the download process of the file 'F' in three tested situations. The green (solid) graph line shows the download without the proxy, the red (dotted) line shows the download process when the proxy modifies the file, and the yellow (dashed) line shows the download process with a modified file and with the speed limitation enabled. Note that the file size of the modified file is about 500 kB bigger (the injected code is accessing local resources/data and transmitting them to a remote server).

Our speed estimate is conservative and is suitable also for packages with a larger amount of Java code (like the file 'E'). Looking at Figure 2, it is possible to conclude that in this particular case the download speed could be faster and the download would still not suffer from the download stall in the middle.

VII. POSSIBLE COUNTERMEASURES

The simplest countermeasure is to use the SSL/TLS to protect the communication (e.g., the HTTPS protocol). The security issues of SSL/TLS (including possible attacks) are out of the scope of this paper.

Requesting permissions at run time in newer versions of Android may reveal undesired hidden functionality of an application, but only if that feature is active and the user pays attention to the popping up messages.

The application can check the signing certificate at runtime (so called certificate pinning) [3]. An attacker could modify the reference certificate during the repackaging attack, but doing so automatically would assume a particular process of the certificate verification in the application.

Y. Zhauniarovich et al. [29] are suggesting the use of a secondary signature of the APK file that would be added by the application store. This method requires a trusted certificate of the store to be available in the Android before the application installation.

M. Conti et al. introduce OASIS [11], a trusted component processing sensitive data on behalf of applications. Applications cannot access data directly, but only via a handle. Therefore the OASIS system can enforce complex policies like allowing access to contacts to display them but prohibiting sending contacts over the Internet (even if Internet access is generally allowed).

Increasingly popular are anti-decompilation, anti-cracking and anti-reverse-engineering mechanisms. These can be as simple as calling one particular method in the Java code (e.g., [27]) or more complex (e.g., [22], [28]).

A detection that the APK file was modified and repackaged can be based on multiple principles. A significant effort has been spent to detect repackaged APK files in Android markets. In these cases researchers and providers have a large number of APK files and analyze their statistical properties, e.g.,

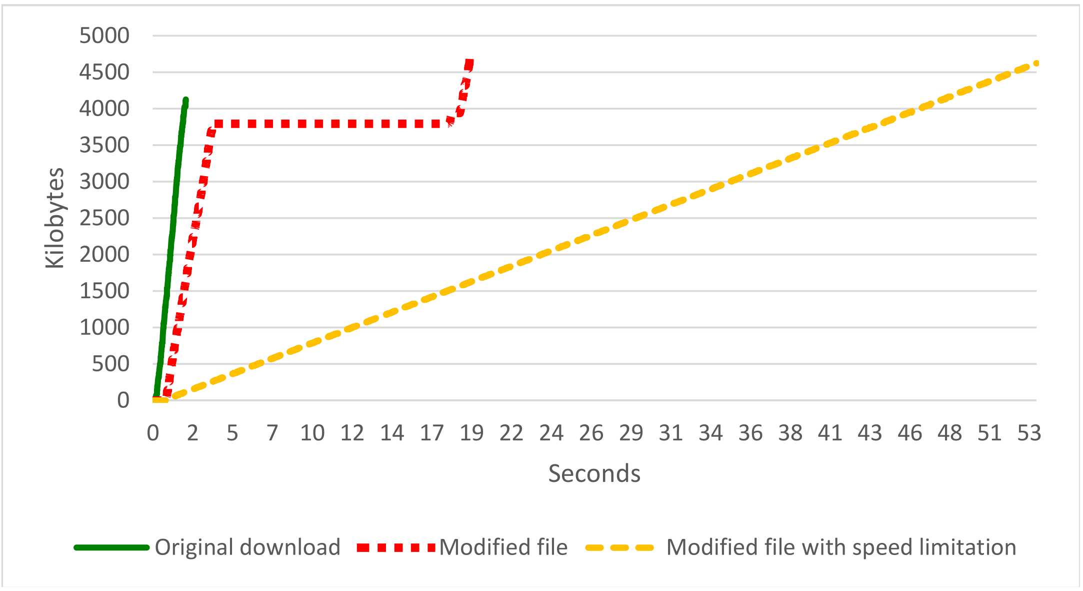

Fig. 2. The download process of a sample APK file (the game 'F'). The solid green line shows the download without the proxy, the dotted red line shows the download process when the proxy does modify the file and the yellow dashed line shows the download process with a modified file and with the speed limitation enabled.

similarity between code (including the dependency graphs between method) and resources (typically images) in packages [32], [19], [12], [30], [24], [17], [10]. The online on-the-fly attack is modifying the APK file during a particular download of a user. Focusing on the markets or file repositories does not solve the problem.

At the side of the Android device the detection can benefit from the changes the attacker has to do in the APK file. During the attack some Java code is added, the Android manifest is modified and the signature is updated with a new private key and certificate. The injected code depends strongly on the attacker and her aims. The attacker has to balance the power of the code and its detectability. The more power of the injected code the bigger is the size of the new code, the more hooks appear in the Android Manifest and the more permissions are required in the Android Manifest. Permission-based malware detection is common these days [26]. Therefore, requiring too many powerfull permissions can lead to a quick detection of an antivirus solution.

In targeted attacks, the attackers do not usually need excessive permissions and rely on commonly used permissions to get access to personal data (e.g., READ_SMS). The code is tailor-made and therefore signature based detection using known malware signature databases is not of a significant help.

In our experiment, we coded two variants of the additional functionality[5]. The first was a simple "Hello World" text appearing from time to time on the screen. The second was a realistic privacy attack collecting messages, contacts and call logs, and transmitting these to a web server located in the Internet. We tested both variants with three leading antivirus solutions and no alert was raised during our tests.

[5]These codes are not available for download.

Kirin [14] is a mobile application certification service used during application installation to check for potentially dangerous combinations of permissions.

The detection techniques can also focus on the signer. If the application is new, then the signer cannot be matched with the previous one as is the case of an application update. Basically all signers are equal and the operating system itself has difficulties deciding where a particular application should be signed by a particular signer or not. A reputation system of the signers (or of the APK files) can help [20]. The rarer is the signer or the APK the more suspicious the application is. This technique requires a community support, but is already used in some common PC-based antivirus or firewall solutions.

In our implementation of the online on-the-fly attack we estimate the total new size of the file and then pad the file with zeros in the ZIP extra fields to match exactly the estimated file size. The extra fields are used also in normal APK files. The *zipalign* utility (a part of the Android Developer Tools) aligns all uncompressed data in the APK file on 4-byte boundaries by changing the size of the extra fields. After applying the *zipalign* utility the extra fields occupy single bytes, our implementation is currently adding extra fields in the order of 10 kB. This can lead to an easy detection. Once this feature leads to detections a more accurate estimation of the new file size would be needed.

VIII. Conclusion

In this paper, we practically demonstrated new fully automated offline and on-the-fly attacks on the Android APK for an arbitrary APK provided at the time of the attack. Without seeing a particular APK file before, we are able to inject a prepared code to the application and infect it this way. The practicality of this attack was demonstrated with a transparent HTTP proxy in the middle performing an APK tampering on-the-fly for all HTTP downloaded APK files.

IX. ACKNOWLEDGEMENTS

The authors thank to the anonymous reviewers for their comments. Vashek Matyáš was partly supported by the Czech Science Foundation project GBP202/12/G061.

REFERENCES

[1] Inserting keylogger code in Android SwiftKey using apktool. Online [Accessed Dec 14, 2016], Mar 2013. http://www.android-app-development.ie/blog/2013/03/06/inserting-keylogger-code-in-android-swiftkey-using-apktool/.

[2] Android Developers Reference. Online [Accessed Dec 14, 2016], 2015. http://developer.android.com/guide/developing.

[3] Retrieve the apk signature at runtime for Android. Online [Accessed Dec 14, 2016], Jun 2015. http://stackoverflow.com/questions/8682731/retrieve-the-apk-signature-at-runtime-for-android.

[4] Backdooring EXE Files. Online [Accessed Dec 14, 2016], Dec 2016. https://www.offensive-security.com/metasploit-unleashed/backdooring-exe-files/.

[5] Smartphone OS Market Share, 2016 Q2. Online [Accessed Dec 14, 2016], Sep 2016. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[6] D. Akhawe and A. P. Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, Washington, D.C., 2013. USENIX.

[7] E. Aydogan and S. Sen. Automatic generation of mobile malwares using genetic programming. In A. M. Mora and G. Squillero, editors, *Applications of Evolutionary Computation*, volume 9028 of *Lecture Notes in Computer Science*. Springer, 2015.

[8] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A.-D. Schmidt, and S. Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software*, MALWARE '11, Washington, DC, USA, 2011. IEEE Computer Society.

[9] P. Berthom, T. Fcherolle, N. Guilloteau, and J.-F. Lalande. Repackaging android applications for auditing access to private data. In *ARES*. IEEE Computer Society, 2012.

[10] J. Chen, M. Alalfi, T. Dean, and Y. Zou. Detecting android malware using clone detection. *Journal of Computer Science and Technology*, 30(5), 2015.

[11] M. Conti, E. Fernandes, J. Paupore, A. Prakash, and D. Simionato. Oasis: Operational access sandboxes for information security. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '14, New York, NY, USA, 2014. ACM.

[12] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In S. Foresti, M. Yung, and F. Martinelli, editors, *Computer Security ESORICS 2012*, volume 7459 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012.

[13] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, Berkeley, CA, USA, 2011. USENIX Association.

[14] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[15] A. P. Felt, A. Ainslie, R. W. Reeder, S. Consolvo, S. Thyagaraja, A. Bettes, H. Harris, and J. Grimes. Improving SSL Warnings: Comprehension and Adherence. In *Proceedings of the Conference on Human Factors and Computing Systems*, 2015.

[16] J. Forristal. Android Master Key Exploit – Uncovering Android Master Key That Makes 99% of Devices Vulnerable. Online [Accessed Dec 14, 2016], Mar 2013. https://uwnthesis.wordpress.com/2013/07/04/uncovering-android-master-key-that-makes-99-of-devices-vulnerable/.

[17] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A scalable system for detecting code reuse among android applications. In U. Flegel, E. Markatos, and W. Robertson, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 7591 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013.

[18] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, Raleigh, NC, USA, October 2012.

[19] S. Jiao, Y. Cheng, L. Ying, P. Su, and D. Feng. A rapid and scalable method for android application repackaging detection. In J. Lopez and Y. Wu, editors, *Information Security Practice and Experience*, volume 9065 of *Lecture Notes in Computer Science*. Springer, 2015.

[20] D. Papp, B. Kcs, T. Holczer, L. Buttyn, and B. Bencsth. Rosco: Repository of signed code. In *Proceedings of the Virus Bulletin Conference*, Prague, Czech Republic, 2015.

[21] PKWARE. Zip file format specification. Technical Report version 6.3.2, PKWARE, 2007. http://www.pkware.com/documents/casestudies/APPNOTE.TXT.

[22] M. Protsenko and T. Muller. Protecting android apps against reverse engineering by the use of the native code. In S. Fischer-Hbner, C. Lambrinoudakis, and J. Lpez, editors, *Trust, Privacy and Security in Digital Business*, volume 9264 of *Lecture Notes in Computer Science*. Springer, 2015.

[23] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: Evaluating Android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, New York, NY, USA, 2013. ACM.

[24] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, and L. Xie. Detecting code reuse in android applications using component-based control flow graph. In N. Cuppens-Boulahia, F. Cuppens, S. Jajodia, A. Abou El Kalam, and T. Sans, editors, *ICT Systems Security and Privacy Protection*, volume 428 of *IFIP Advances in Information and Communication Technology*. Springer Berlin Heidelberg, 2014.

[25] D. Taitelbaum. Hacking for fun and for profit (mostly for fun). Online [Accessed Dec 14, 2016], Dec 2012. http://www.slideshare.net/davtbaum/hacking-for-fun-and-for-profit.

[26] F. Tchakounté. Permission-based malware detection mechanisms on Android: Analysis and perspectives. *Journal of computer science and software application*, 1(2), Dec 2014.

[27] J. Xu, S. Li, and T. Zhang. Security analysis and protection based on smali injection for android applications. In X.-h. Sun, W. Qu, I. Stojmenovic, W. Zhou, Z. Li, H. Guo, G. Min, T. Yang, Y. Wu, and L. Liu, editors, *Algorithms and Architectures for Parallel Processing*, volume 8630 of *Lecture Notes in Computer Science*. Springer, 2014.

[28] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In H. Bos, F. Monrose, and G. Blanc, editors, *Research in Attacks, Intrusions, and Defenses*, volume 9404 of *Lecture Notes in Computer Science*. Springer, 2015.

[29] Y. Zhauniarovich, O. Gadyatskaya, and B. Crispo. DEMO: Enabling Trusted Stores for Android. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, 2013.

[30] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. La Spina, and E. Moser. Fsquadra: Fast detection of repackaged applications. In V. Atluri and G. Pernul, editors, *Data and Applications Security and Privacy XXVIII*, volume 8566 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014.

[31] M. Zheng, P. P. C. Lee, and J. C. S. Lui. ADAM: an automatic and extensible platform to stress test Android anti-virus systems. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 9th International Conference, DIMVA 2012, Heraklion, Crete, Greece, July 26-27, 2012, Revised Selected Papers*, 2012.

[32] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, New York, NY, USA, 2012. ACM.

Zdeněk Říha is teaching at the Masaryk University, Faculty of Informatics, in Brno, Czech Republic. He received his PhD degree from the Faculty of Informatics, Masaryk University. In 1999 he spent 6 months on an internship at Ubilab, the research lab of the bank UBS, focusing on security and usability aspects of biometric authentication systems. Between 2005 and 2008 he was seconded as a Detached National Expert to the European Commission's Joint Research Centre in Italy. Zdenek can be contacted at zriha AT fi.muni.cz.

Dušan Klinec is security consultant, developer and research fellow at the Centre for Research on Cryptography and Security, Masaryk University, CR. Dusan focuses on secure end-to-end communication, secure cloud solutions, cryptanalysis, whitebox cryptography and software security in general. He also participated projects focused on wireless sensor networks and cryptanalysis using genetic algorithms. Dusan received his Master degree from Masaryk University, with his Master thesis on whitebox attack resistant cryptography and graduated with honours. He was an intern at CERN where he worked on projects related to GRID computing and IPv6 compliance. He can be contacted at dusan.klinec AT gmail.com.

Václav (Vashek) Matyáš is a Professor at the Masaryk University, Brno, CZ, and Vice-Dean for Industrial and Alumni Relations, Faculty of Informatics. His research interests relate to applied cryptography and security, where he published over 150 peer-reviewed papers and articles, and co-authored several books. He was a Fulbright-Masaryk Visiting Scholar with Harvard University, Center for Research on Computation and Society in 2011-12, and previously he worked also with Microsoft Research Cambridge, University College Dublin, Ubilab at UBS AG, and was a Royal Society Postdoctoral Fellow with the Cambridge University Computer Lab. Vashek edited the Computer and Communications Security Reviews, and worked on the development of Common Criteria and with ISO/IEC JTC1 SC27. He received his PhD degree from Masaryk University, Brno and can be contacted at matyas AT fi.muni.cz.