An Insight to Cracking Solutions and Circumvention of Major Protection Methods for Android

Nils Kannengiesser
Technical University of Munich
Division for Operating Systems, Faculty of Informatics
Munich, Germany
nils.kannengiesser@tum.de

Johannes Neutze
Technical University of Munich
Division for Operating Systems, Faculty of Informatics
Munich, Germany
neutze@mytum.de

Uwe Baumgarten
Technical University of Munich
Division for Operating Systems, Faculty of Informatics
Munich, Germany
baumgaru@in.tum.de

Sejun Song
University of Missouri
Kansas City, Kansas, USA
sjsong@umkc.edu

ABSTRACT

Android is a target of software piracy. Application stores are trying to counteract piracy attempts by providing license verification libraries but even those do not stop theft of intellectual property. The scope of this paper is to present different attack methods. Insights to the attacks can be used to create countermeasures that are addressed shortly, while presenting their details in a different paper (available at this conference) instead.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized access

General Terms

SECURITY

Keywords

Android, Google, Amazon, LVL, DRM, Copy Protection, Attacks

1. INTRODUCTION

In recent years, a new operating system hit the market and depending on the country viewed, has reached market shares of up to 90% [8]. Its name? Android. While initially intended for smart-phones only, Android is now available on all kinds of devices like tablets, watches or even smart-TVs. With such a large market share comes a great responsibility to protect intellectual property, but unfortunately protecting apps and content was never a high priority on Android [23]. There is no obvious reason why the protection has not been improved since 2010, when Android's License Verification Library (LVL) was released and soon after, cracked [7]. Google likes to transfer the burden of improving security to its developers by stating "The security of your application's licensing controls ultimately relies on the design of your implementation itself." [3]. Instead, Amazon uses a different approach by injecting their protection into an app at the developer's request during the upload to the market [9]. While this approach is much more developer friendly, their security solution is not much more secure either. Here the reasons are the fundamental issues of Android in terms of the easy reengineering of apps that lies in the responsibility of Google [23]. While the main purpose of this paper is to present the known attack options on protections used by common markets, it also covers the details of one of the most widely used cracking tools, "Lucky Patcher" [1], analyzed by us in detail. In addition and to underline our intention as security researchers, a brief summary is given to possible protection topics, while further details are addressed in a different paper [1] available at this conference as well.

2. BACKGROUND

2.1 Lucky Patcher

Lucky Patcher is an Android cracking application with various functionalities. This includes basic features, such as creating backups or restoring applications, and more advanced features that include modifying the code of applications. Lucky Patcher allows the changing of application permissions, removal of advertisements and the circumventing of license verification mechanisms. Some of these features require the device to be rooted. Applying features on an application does not change the functionality of the target application but voids the license verification. Success is not guaranteed, i.e. some applications are resistant to cracking. The current version of the application, 6.2.4, can be downloaded for free on a dedicated website [1].

[1]Title:"Securing License Verification by using Native Code, Fusing Options and Indirect Method Triggering on Android"

2.2 Reengineering Tools

Besides the tools available to non-skilled users to perform the cracking of apps, more advanced users, researchers and security experts can rely on tools like the APKtool [25] used to disassemble any Android application available in the form of an APK file. The resulting assembly dialect is called smali code and may be described as a high-level assembly code. Modifications of the smali code are easily made and APKs reconstructed, signed and used as usual.

Alternatives to the APKtool are the use of usual Java decompilers that require the embedded DEX file after it has been converted to a JAR file. The benefit of having Java code, in comparison to smali-code, is that it is even easier to understand the program logic, although the reengineered code is often quite poor [23].

2.3 Java Code Interception (Xposed Framework)

The "Xposed Framework" [10] was developed a few years ago and allows modification of Android application on-the-fly by intercepting methods before and after these functions are executed. The interception is done by injecting the required library into the so-called Zygote process that is forked for any newly started Android app and therefore always present in new applications to allow the interception of any (Java) methods [10].

2.4 Native Code Interception (LD_PRELOAD etc.)

Intercepting native code is also possible, although it is more difficult in comparison to the interception of Java code. While there have been frameworks available in the past, like "Cydia Substrate" [11] that cannot be used with Android versions after 4.3, a remaining option might be "Frida" [12] that uses JavaScript for control. An even easier approach could be to replace a function, when a method name is known using LD_PRELOAD directive as shown in [14]. Replacing a function using that directive seems to work even on recent Android versions (6.0.1) [23].

2.5 Typical App Protections

This section introduces the common protection methods. The License Verification Library (LVL) by Google, the injected Amazon DRM in their related Appstore and examples for third party protections.

2.5.1 *License Verification Library*

Google introduced the LVL back in 2010 as a tool to fight piracy of applications [2]. It is a simple solution for developers used to verify users against applications purchased on Google Play Store. The LVL library provides the implementation of the verification while leaving full control over handling the result to the developer. The library is based on a network service which queries the trusted Google Play license servers to validate whether the user has purchased the application. Since the source code is provided as part of the Android SDK, the developer just has to implement the actions to be taken when the response is returned. Google suggests to modify the library to create a unique implementation that is less vulnerable to code cracking attacks [3][4]. The classes are the LicenseValidator class, which is responsible for decrypting and verifying the server's response. The policy classes for storing the response, namely APKExpansionPolicy class and ServerManagedPolicy class, which are provided by Google. The LicenseChecker class is repsonsible for initiating the license check and the ILicenseResultListener class, which provides the interface for verifying the license [18].

2.5.2 *Amazon DRM*

Amazon uses a different approach to protect developer's apps in their Appstore by using a technique to inject code into these apps. Here, it does not matter if the app actually uses the DRM protection; Amazon's libraries are injected anyway even if the protection will not be activated. Amazon points out that this approach is required to interact with their App-Store app [9].

2.5.3 *Other protections*

Besides the widely known protection mechanisms, other markets like SlideMe provide their own license verifications that have similarities to Google's LVL [13]. Furthermore, there are various protections by other researchers available, but none of them seem to be used widely or are recommend by Google or Amazon. Examples to be mentioned are the solution using smart-cards and encryption of the application as presented by Shoaib et al. [20]. Another option is the solution approach called "DIVILAR" by Wu Zhou et al. [22] introducing a virtual machine protection for older Android versions. In addition, protections using dongles that can execute code in a secure manner could be developed using [21]. Nevertheless, these solutions are quite often meant for Java code, which keeps them vulnerable to attacks. In our research [23], we focused on protections using native code that is briefly addressed in the section Defense strategies and outlined in a another paper by us.

3. CRACKING APPLICATIONS

The following section covers the available options for cracking an Android app. One option is to use the tool Lucky Patcher that will be introduced in detail based on a black-box analysis as performed by [5]. Moreover, the typical way for attacking an app using reengineering tools is shown, before outlining an universal cracking option using the Xposed Framework as researched by [16].

3.1 Cracking applications by using tools (example Lucky Patcher)

3.1.1 *Modes*

Lucky Patcher, as analyzed in [5], offers different options for modifying applications. This paper focuses on the option called Universal Patching. The option contains three different modes of operation for modifying Google's LVL and another for Amazon's license solution. A further mode of operation for Samsung's library is out of scope in this paper [5].

3.1.2 *Patch Patterns*

Lucky Patcher uses Patch Patterns to find and apply changes to the license verification implementation. A Patch Pattern consists of a pair of two sequences, the search pattern and the replace pattern, which are applied on DEX bytecode level. Each of the patterns is a bytecode mask of fixed length, containing the target instruction and the context in the form of fixpoints and wildcards as displayed in Figure 1. While the search pattern is used to locate the instruction of

```
@@ Search pattern @@
2C ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? 0A ?? 0F ?? 1A ?? ?? ?? ?? ?? ?? ?? ??
    ?? 28 ?? 1A ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? 0A ?? ?? ??
   ?? ?? ?? ?? 0C ?? ?? ?? ?? ?? ?? ?? 0C ?? ?? ?? ?? ?? ?? ?? 28

@@ Replace pattern @@
2C ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? 0A ?? 0F ?? 1A ?? ?? ?? ?? ?? ?? ?? ??
    ?? 28 ?? 1A ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? 12 ?? ?? ??
   ?? ?? ?? ?? 0C ?? ?? ?? ?? ?? ?? ?? 0C ?? ?? ?? ?? ?? ?? ?? 28
```

Figure 1: Search and Replace Pattern example [5]

interest, the replace pattern is used to apply the changes. There is no bytecode added or removed in the process, but the control flow is altered to enforce a specific outcome of evaluations. This is achieved by manipulating single instructions of the bytecode by either changing its opcode or arguments. Since implementations never look the same, Patch Patterns have different bytecode patterns for changing the same functionality. All of the Patch Patterns are tried when patching and when a Search Pattern fits, the Replace Pattern is applied to change the instruction. The patterns applied by the chosen mode are shown when the manipulation has finished. There are seven Patch Patterns for the LVL, called N1 to N7. The following gives a detailed description on when the patterns are used, the targeted function and class, and resulting changes [5].
Patch Pattern N1 and Patch Pattern N2 are part of the auto and auto inverse mode. These patterns target the verify() method inside the LicenseValidator class. Patch Pattern

```
@@ Patch Result N1 @@
# case LICENSED:
- case LICENSED_OLD_KEY: handleResponse(); break;
- case NOT_LICENSED: handleError(); break;
+ case NOT_LICENSED: handleResponse(); break;
+ case LICENSED_OLD_KEY: handleError(); break;
```

Figure 2: Result of Patch Pattern N1 [5]

N1 attacks the method's switch case by switching the cases for LICENSE_OLD_KEY and NOT_LICENSED as seen in Figure 2. This results in NOT_LICENSED responses are being handled the same way as LICENSED responses and thus the result of the method always allows access. After

```
@@ Patch Result N2 @@
- if (sig.verify(Base64.decode(signature))) {...;}
+ sig.verify(Base64.decode(signature); ...;
```

Figure 3: Result of Patch Pattern N2 [5]

Patch Pattern N2 (Figure 3) is applied, the signature is still verified but the evaluation is skipped and the former condition is always executed so the signature's validity is no longer required [5]. Patch Pattern N3 has two different variants, called N3 in the auto mode and N3i in the auto inverse mode. Target of both patterns is the allowAccess() of APKExpansionPolicy and ServerManagedPolicy class. Both Patch Patterns attack the same instruction but use opposing values. While N3 bets on true as solution, N3i initiates the value as false (Figure 4) . The result of the Patch Pattern is

```
@@ Patch Result N3 @@
- result = false;
+ result = true;
# ...
# return result;

@@ Patch Result N3i @@
- result = true;
+ result = false;
# ...
# return result;
```

Figure 4: Result of Patch Pattern N3 [5]

that the allowAccess() method is always successful since the result variable is initialized with the desired outcome. Since Lucky Patcher is not able to analyze or predict the code and the implementation is easily modified at this location, two opposing patterns are provided to cover both scenarios [5]. Patch Pattern N4 is applied by the auto and auto inverse mode and attacks the checkAccess() method inside the LicenseChecker class. The pattern replaces the check for the

```
@@ Patch Result N4 @@
- if( ! mPolicy.allow()) {...}
+ if(mPolicy.allow() != mPolicy.allow()) {...}
```

Figure 5: Result of Patch Pattern N4 [5]

validity of the policy of the response with an evaluation with itself as in Figure 5. This way the result of the evaluation is always false and the condition code block is never executed [5].

Patch Pattern N5 and Patch Pattern N6 are part of the extreme mode. Both patterns target the LicenseValidator class's verify() method. After applying Pattern N5, the

```
@@ Patch Result N5 @@
- if (data.responseCode != responseCode) {
+ data.responseCode;
+ if (LICENSED != responseCode) {
```

Figure 6: Result of Patch Pattern N5 [5]

parsed response code is no longer evaluated and instead the constant for LICENSED is used (Figure 6) [5]. Patch Pattern N6 forces the evaluation to always be true. In addition, the switch case returns always the same result since no longer the response code is used but a constant (Figure 7) [5]. Patch Pattern N7 is applied when patching with the extreme mode. It targets the onTransact() method of the ILicenseResultListener class. Figure 8 shows the result of Patch Pattern N7. The pattern ensures that the verifyLicense()'s parameter is set to LICENSED instead of the response code from the server [5].

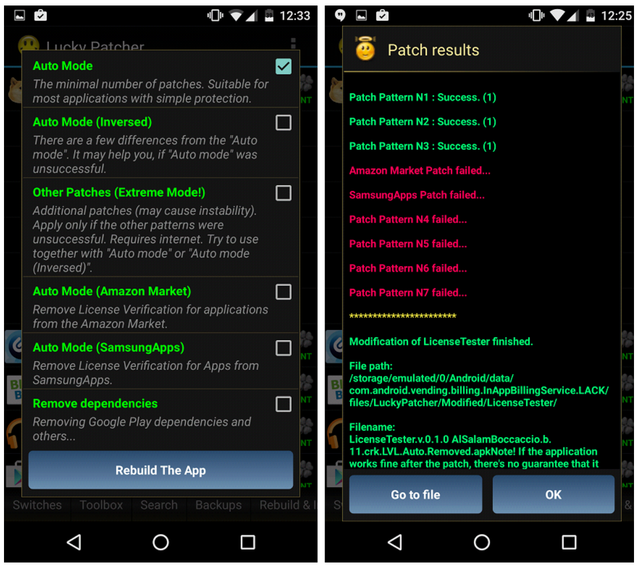
Figure 9 shows a screenshot of Lucky Patcher with all the modes and the performed cracking approach on one of our test apps. The figure shows that Lucky Patcher just tries out the pattern and hopes for a match. A general cracking approaches may be used to circumvent a specific protection by using custom recipes [5].

```
@@ Patch Result N6 @@
- if (responseCode != LICENSED || responseCode != NOT_LICENSED ||
    responseCode != LICENSED_OLD_KEY) {
- switch (responseCode) {
+ responseCode = LICENSED;
+ if ((LICENSED != LICENSED) && (LICENSED != LICENSED_OLD_KEY)) {
+ switch (LICENSED) {
```

Figure 7: Result of Patch Pattern N6 [5]

```
@@ Patch Result N7 @@
- this.verifyLicense(data.responseCode, data.signedData, data.signature);
+ data.responseCode;
+ this.verifyLicense(LICENSED, data.signedData, data.signature);
```

Figure 8: Result of Patch Pattern N7 [5]



Auto Mode [✓]
The minimal number of patches. Suitable for most applications with simple protection.

Auto Mode (Inversed)
There are a few differences from the "Auto mode". It may help you, if "Auto mode" was unsuccessful.

Other Patches (Extreme Mode!)
Additional patches (may cause instability). Apply only if the other patterns were unsuccessful. Requires internet. Try to use together with "Auto mode" or "Auto mode (Inversed)".

Auto Mode (Amazon Market)
Remove License Verification for applications from the Amazon Market.

Auto Mode (SamsungApps)
Remove License Verification for Apps from SamsungApps.

Remove dependencies
Removing Google Play dependencies and others...

```
Patch Pattern N1 : Success. (1)
Patch Pattern N2 : Success. (1)
Patch Pattern N3 : Success. (1)
Amazon Market Patch failed...
SamsungApps Patch failed...
Patch Pattern N4 failed...
Patch Pattern N5 failed...
Patch Pattern N6 failed...
Patch Pattern N7 failed...
************************
Modification of LicenseTester finished.

File path:
/storage/emulated/0/Android/data/
com.android.vending.billing.InAppBillingService.LACK/
files/LuckyPatcher/Modified/LicenseTester/

Filename:
LicenseTester.v.0.1.0 AlSalamBoccaccio.b.
11.crk.LVL.Auto.Removed.apkNote! If the application
works fine after the patch, there's no guarantee that it
```

Figure 9: Lucky Patcher in action [5]

3.2 Cracking Applications using Typical Re-Engineering (example LVL and Amazon's DRM)

Google's LVL was cracked years ago [7]. While the easiest option to crack LVL is to use "Lucky Patcher" [15] and even non-savvy customers can crack and circumvent the protection with this tool, a more advanced attack requires some Android and reengineering knowledge by editing a single application itself by using the apktool. As outlined in [7] the attacker is required to find the LicenseValidator.smali file that may be obfuscated, but typical structures and calls are still imaginable. Here and within the verify() method the attacker needs to look for a switch block. Values like 0x00 or 0x03 represent a valid license state, which links to the desired switch case. Ultimately the trick is to redirect the other cases (representing a failed license response) to the valid ones. By saving, recompiling and signing it, the app is cracked.

Amazon uses a different approach to integrate their DRM protection by injecting their code into existing apps, regardless if DRM is requested or not, since the injected code is also used for communication purposes with the Appstore [9]. In 2014, Amazon's DRM was analyzed by [16] for the first time and our research revealed that there are essentially three function calls within the so-called kiwi-class, which need to be removed to disable the protection. In a recent review, it was discovered that the protection has been modified, but to our surprise, a new Boolean variable responsible for enabling or disabling DRM allows the deactivation of the protection within seconds for any protected app [17]. Since the protection is integrated by Amazon, developers can do almost nothing about it and depend on Amazon to improve their license protection. In theory, developers could integrate their own additional methods, but that is beyond the scope of this paper. Nevertheless, some options are presented in section Defense strategies.

3.3 Cracking Applications by Interception/Manipulation (example LVL)

A more sophisticated attack solution is a universal crack that may be implemented using the Xposed Framework. It is usable with all default LVL implementations [16] if developers ignore Google's request to modify it. The Xposed framework works on rooted devices only and hooks itself into Android's Zygote process responsible for launching all apps by forking child processes that inherit most of its loaded libraries [10] .

As analyzed and implemented by [16], the attack works in a summarized way by intercepting the onTransact() method as well as the verify() method that is used internally by the LVL. The license response by the Google license server consists of six values including the actual (LICENSED) response code and the package name (by example). Moreover, a signature to allow the application to verify the response for any modifications is also added by the license server. On Android the whole response is supplied in a parcel that needs to be modified to circumvent the LVL. The parcel needs to be intercepted and altered, before executing the actual onTransact() method by changing the included response code to 0x00 (LICENSED) using our developed Xposed module. In addition, the signature has to be altered. Since the private key for creating the signature is known by Google only, a new public/private key pair needs to be created. The newly created private key can then be used to generate the valid signature for the changed response and both values, the new signature and the new public key, will replace the previous values before being forwarded to the verify() method. The app itself will receive the valid license response and continue as if everything is fine.

4. DEFENSE STRATEGIES

Since this paper is focused on the attack options, defense methods are addressed only briefly and presented in a different paper by us; we present a short summary for the sake of completeness only.

In general, we propose the use of native code for increased protection [23], since reengineering is much more difficult and, according to our surveys of computer science students, many of these students are not used to ARM assembly and non-computer science students are not familiar with basic reengineering tools either. Moreover, available decompilers, like retdec.com, are often also not able to recover the entire code. In addition, native code can be protected and its obfuscation has been researched for decades.

For improved protection (cf. Lucky Patcher), it is sufficient to slightly modify the default LVL implementation by for example, renaming the packages and functions (also recommended by Google [3]). More advanced attacks using reengineering tools or interception cannot be prevented this way but aforementioned methods do buy some time. This issue applies to any protection because of the general insecurity of Android and related root access [23], but the time factor can be optimized to the advantage of the developer (copyright holder) using various methods. For instance, we developed a native version of the LVL called ”nLVL” [6] that is immune to Lucky Patcher, Xposed Framework and to typical proxy attacks due to its direct connection to the Google license servers while benefiting from the power of obfuscation methods for native code (e.g. static directive [6]). In addition, we investigated and developed methods to bind native and Java code together so that a separation of the app from the more secure native code is no longer easily possible and a separation will trigger certain protection methods thus render the application useless [23].

5. RELATED WORK

There are various similar research works, but only few of them focus on similar topics like attacking libraries. In [26] Collin Mulliner et al. attacked the In-App-Billing Library by method call interception using their own libraries. In comparison we used the Xposed Framework for our LVL attacks instead. Most other related works focus on copyright protection for Android. One work, for example, focuses on the usage of smart-cards and application encryption using dynamic code loading for copy protection as presented by Shoaib et al. [20]. Other researchers focus on improved copyright security by modifying opcodes to prevent the reengineering as described in the paper about ”DIVILAR” by Wu Zhou et al. [22]. Moreover, security companies provide external dongles to allow the execution of code in a more secure manner as outlined in e.g., [21].

6. CONCLUSION

Summarizing the status on Android in terms of copyright protection, the situation can be described as devastating. While many developers do not seem to be aware of the issues of reengineering in general, the available copyright protection used in major app markets is seriously deficient and can be circumvented by customers using cracking tools like Lucky Patcher. Our observation is shared by Eric Lafortune from GuardSquare, who collected some figures on the use of their obfuscation tools, stating, ”We have collected some statistics on the protection of the top European banking apps. It seems that about 65% use ProGuard, 15% use DexGuard, and 20% are unprotected [...] Considering that ProGuard only offers very basic protection (name obfuscation), most developers indeed seem to be unaware [of reengineering issues]” [24]

In general, the situation on Android will not improve as long as Google provides access to the DEX file that is still included in current OAT formats (new format used by ART VM) [19] [23].

7. FUTURE WORK

Since there are no additional, secure solutions available that are widely used, there is no imminent future work on the same topic of introducing cracking possibilties possible. Nevertheless, alternative solutions by other researchers as well as our own solutions may be re-evaluated in the future, while current evaluations of our solutions have already provided a significant improvement against existing copyright protection solutions.

APPENDIX

LEGAL

Both companies - Google and Amazon - have been notified about the problems related to their license verification in 2015 and in early 2016. In addition, Google was notified about protection solutions and invited to participate in the project which brings native protection to a broad range of developers. Furthermore, any presented attacks are not described in great detail to prevent malicious usage right away. More information or source codes are available upon justified request.

UNPUBLISHED WORKS

While the 1st author's dissertation is submitted for review and currently not yet publicly available, other unpublished and used references are not accessible due to legal reasons. Upon justified request these works are available and can be requested from the 1st author.

ACKNOWLEDGMENT

We would like to thank our former and current students Marius Muntean, Sebastian Schleemilch, Yixiang Chen, Jonas Raedle and Gabriel Michels that provided us input via their research works and theses, but were not involved in the actual writing of this paper.

ABBREVIATIONS

APK Application Package

ART	ART VM for Android by Google
DEX	Dalvik Executable
DRM	Digital Right Management
JAR	Java Archive
LP	Lucky Patcher
LVL	License Verification Library
SDK	Self Development Kit
OAT	Ahead of time

8. REFERENCES

[1] ChelpuS, Lucky Patcher, `http://lucky-patcher.netbew.com/`. Last access: 07-18-2016.

[2] E. Chu. Licensing Service For Android Applications, `http://android-developers.blogspot.de/2010/07/licensing-service-for-android.html`, Last access: 07-18-2016.

[3] Android Developers, Licensing Overview, `https://developer.android.com/google/play/licensing/overview.html`, Last access: 07-18-2016.

[4] Android Developers, Setting Up for Licensing, `https://developer.android.com/google/play/licensing/setting-up.html`, Last access: 07-18-2016.

[5] J. Neutze, Master's Thesis, "Analysis of Android Cracking Tools and Investigations into Countermeasures for Developers", unpublished.

[6] Master's thesis by Yixiang Chen, unpublished.

[7] Justin Case, "[EXCLUSIVE] Report: Google's Android Market License Verification Easily Circumvented, Will Not Stop Pirates", `http://www:androidpolice:com/2010/08/23/exclusive-report-googles-android-market-license-verification-easily-circumvented-will-not-stop-pirates`, Last access: 07-15-2016.

[8] ComTech. "Kantar Worldpanel", `http://www.kantarworldpanel.com/global/smartphone-os-market-share/`, Last access: 07-18-2016.

[9] Amazon, "Publishing Android Apps to the Amazon Appstore". `https://developer.amazon.com/public/support/submitting-your-app/tech-docs/submitting-your-app`, Last access: 07-18-2016.

[10] Revo89, "Development tutorial", `https://github.com/rovo89/XposedBridge/wiki/Development-tutorial`, Last access: 07-18-2016.

[11] SaurikIT LLC, "Cydia Substrate", `http://www.cydiasubstrate.com/`, Last access: 07-18-2016.

[12] NowSecure, "Frida", `http://www.frida.re/`, Last access: 07-18-2016.

[13] SlideMe, "SlideLock", `http://slideme.org/slidelock`, Last access: 07-18-2016.

[14] Cedric Van Bockhaven, "Intercepting Android native library calls", `https://cedricvb.be/post/intercepting-android-native-library-calls/`, Last access: 07-18-2016.

[15] ChelpuS, "Lucky Patcher", `http://lucky-patcher:netbew:com/`, Last access: 07-15-2016.

[16] Marius Muntean, Master's thesis - Improving License Verification in Android, unpublished.

[17] Gabriel Michels, Jonas Raedle, TUM Android Practical Course - Workshop on Amazon DRM, unpublished.

[18] Android Developers, Adding Licensing to Your App,`https://developer.android.com/google/play/licensing/adding-licensing.html#lc-lcc`, Last access: 07-27-2016.

[19] Sebastian Schleemilch, Master's thesis, "Research and Analysis of Copy Protection Mechanisms for Android Apps, as well as Implementing a Sample Application", `http://www:os:in:tum:de/fileadmin/w00bdp/www/Lehre/Abschlussarbeiten/MA_Schleemilch_Android_Copy_Protection:pdf`, Last access: 07-15-2016.

[20] Muhammad Shoaib, Noor Yasin, Abdul G. Abbassi, "Smart Card Based Protection for Dalvik Bytecode - Dynamically Loadable Component of an Android APK", `http://www.ijcte.org/vol8/1036-C040.pdf`. Last access: 07-18-2016.

[21] Aktiv Soft JSC, "Guardant Code", `http://www.guardant.com/products/all/guardant-code/`, Last access: 07-18-2016.

[22] Wu Zhou, Zhi Wang, Yajin Zhou, Xuxian Jiang, "DIVILAR: Diversitying Intermedia Language for Anti-Repackaging on Android Platform", CODASPY 2014.

[23] Nils Kannengiesser, Dissertation, "Improving Copy Protection for Mobile Apps", in review / not published.

[24] Email by Eric Lafortune by 06-24-2016, unpublished.

[25] Connor Tumbleson, Ryszard Wisniewski, "A tool for reverse engineering Android apk files", `https://ibotpeaches.github.io/Apktool/`, Last access: 07-18-2016.

[26] Collin Mulliner, William Robertson, Engin Kirda, "VirtualSwindle: An Automated Attack Against In-App", ASIA CCS, Japan 2014