Securing License Verification by using Native Code, Fusing Options and Indirect Method Triggering on Android

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

Yixiang Chen
Technical University of Munich, Faculty of Informatics, Division for Operating Systems
Munich, Germany
yx.chen@hotmail.com

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

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

ABSTRACT

To prevent massive piracy in the Android app markets, Google provides developers the License Verification Library (LVL) to implement a network-based license checking service. The application queries the Google Play app, which builds a license checking request to a trusted Google licensing server, to obtain the license status for the current user. Since this library is written in Java, it cannot protect the app from reverse engineering. In this paper, a native LVL implementation in C is proposed. This native LVL is much harder to be disassembled into comprehensible code than Java. We introduce so-called fusing options to fuse the Android app and native code together while allowing different program parts to communicate by indirect method triggering. The result is that the app cannot be executed without it anymore.

Categories and Subject Descriptors

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

General Terms

SECURITY

Keywords

Android, Google, LVL, nLVL, Copy Protection, Fusing Options, Indirect Method Triggering

1. INTRODUCTION

As introduced in [1], copy protection is an important topic to the software industry, known and researched for decades on desktop computers with lots of available solutions. Previously, simple mechanisms like serial keys for the activation or riddles were used (e.g., PC Game/Monkey Island 2), while it was sufficiently secure by that time and copy machines or the internet were not widely available to circumvent it that easily. Nevertheless, some software programs were already protected by real copy protection mechanisms preventing the copying of a physical floppy disk. Typically these old copy protection methods were based on artificially applied physical damages or by using specialties like long sectors [2] that were detected by the copy protection mechanism used before executing relevant application parts. With the introduction of CDs and Windows95 in the early 1990s, software developers also started to focus on applying similar protection mechanisms to CDs when shipping their increasingly larger-sized apps on CD. An example of a protection mechanism that was applied to CDs (and even later DVDs and BluRays) by that time is LaserLock [3]. These more advanced protections were already using encryption or signatures while still depending on artificial disk errors or other special properties that require a special - and by that time extremely expensive - burner to create successful copies. An alternative to physical copies was the use of dumped CD images while emulating physical CDROM drives and any specialties expected by the target copy protection. Highly-priced applications were protected by so-called dongles that are still in use today, when a special protection is required, for example, the "USB-eLicenser (Steinberg Key)" [4]. Today most companies have switched to online DRM solutions and software is sold in online stores, like Valve's Steam[1]. If the software was bought on a CD, an online activation is required. Of course, software is still sold on these physical mediums, but customers now prefer the convenience of software-on-demand. The most recent approach to copy protection on desktop computers is the actual protection of the license mechanism itself. For instance, the protection "Denuvo" was able to keep various computer game titles "piracy free for months" [5].

The available copyright protections on mobile devices are

15th AmiEs September 22-24th, 2016, Heraklion, Crete/Greece

[1] store.steampowered.com

limited. In fact, those are no longer real copy protections but only license validations. For instance, Google provides developers the LVL [6] that needs to be integrated by each developer. By contrast, Amazon applies its Amazon DRM to apps in its Appstore automatically at the developer's request. Both solutions are not really safe (see background section) and developers still face the risk of software piracy when using Android. In recent years, several developers, such as ustwogames[2] notified the public that only 5% of their famous "Monument Valley" game on Android are actually bought while the majority is pirated [7]. Also, Microsoft announced (as reported by BR[3]) the end of project Astoria due to reservations from their developers regarding the copy protection available on Android [8].

Therefore, this paper addresses a new approach for protecting applications on Android by using native code and Java code together while porting security related code into native code. For this purpose, the LVL by Google was ported to a native version in a proof-of-concept. Most details such as the non-published interfaces and data formats were obtained by reengineering the existing libraries and frameworks by Google. That this was easily possible, shows already the insecurity of fundamental Android frameworks. In addition, we also present so-called "fusing options"[1] which are methods meant to fuse Java and native code together to make each code dependent on the other. To achieve this goal different functions need to communicate with each other. Therefore, we present the "indirect method triggering"[1] that allows functions to communicate unsuspiciously and without immediate trace option [1].

2. BACKGROUND

2.1 LVL Overview

The LVL [6] is a framework released by Google in 2010 [9] to handle network-based license checking for Google Play[4] apps. In the early Android versions the application file was stored in a folder designed to be accessible only at system level. Without system permission, the application could not be copied or distributed. However, once an attacker gains the root right, he can easily access any resources in a device (and even intercept method calls using, for example, the Xposed framework [10]). Compared to this static copy protection mechanism, Google Play Licensing Service provides a more flexible and secure means of license verification since it is based on network requests at runtime. Applications are enforced to query the Google Play app to obtain the licensing status for the current user. Figure 1 illustrates this licensing procedure.

Initially, information about the app, such as package name, software version and nonce, which prevents replay attack, is parsed to the Google Play app via an Android-specific interprocess communication (IPC) mechanism called binder [11]. Additionally, the information required to identify the user, such as a Google Play account, user authentication token and other device-related information, is serialized and encoded into a string which is sent as an HTTPS parameter to the Google licensing server. After decoding and deserializing the request content, the Google licensing server verifies the user identity against purchase records for the application. A license status response is returned to the Google Play app and parsed to the application via an IPC mechanism. Here, the developer is in charge to act accordingly to the reply.

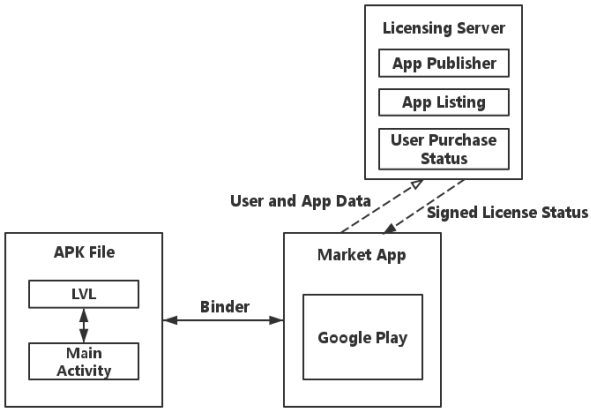

Figure 1: Licensing Overview (based on [11])

2.1.1 IPC Mechanism

In Android, IPC refers to the mechanism that enables separated processes to exchange signals and share data. In the low-level design, this communication can be simplified as a client-server model. The client process initiates a service request and transacts data to the binder driver, which then delivers data to the target process [12]. Acting like a server, the target process handles the incoming request and returns feedback. The license checking service of LVL also implements this mechanism. In LVL there is an .aidl file agreed on by both client and target process. Once the client process invokes the license checking request, application information is marshalled into a parcel which can be transferred down to binder kernel driver with the help of an AIDL [5] tool [13]. This parcel is then forwarded and unmarshalled by the Google Play app.

2.1.2 Data Serialization and Encryption

To send a large amount of application information to the Google server, one of the best practices is to serialize the information into a byte array which is encoded into a string as a single parameter of the HTTPS request. For the serialization, a language- and platform-neutral and extensible serialization mechanism called Protocol Buffers was introduced by Google [14]. Compared to the Java built-in serialization and XML serialization, the Protocol Buffers have great advantages due to its lightweightness, simplicity and fast speed [15]. Figure 2 describes the core concept of Protocol Buffers.

Most importantly, the serialization protocol must be defined to specify how the information should be structured. After compiling into the application, classes are generated providing setter and getter methods to access attribute fields. In the Google Play app, the class CheckLicenseRequestProto defines the application information such as package name,

[2]Developer of video games
[3]German TV broadcaster
[4]Appstore by Google for Android
[5]Android Interface Definition Language

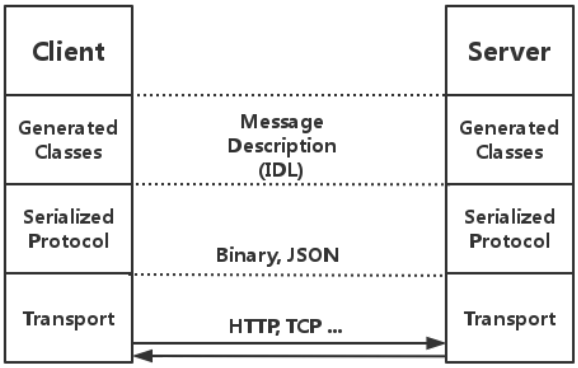

Figure 2: Data Serialization with Protocol Buffer (based on [16])

software version and nonce, and another class RequestPropertiesProto represents the data structure of user account and device information including the user authentication token, account name and operator information. Figures 3 and 4 show the structure of data serialization whereby tag is simply a numerical representation of message protocol or fields. For each protocol or field, the size block indicates the length of the bytes. Finally, based on the Base64 encoding algorithm [17], the serialized data is encrypted to an ASCII string format which is sent as a HTTPS Post request parameter.

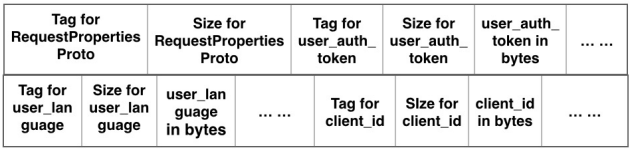

Figure 3: RequestPropertiesProto Serialization

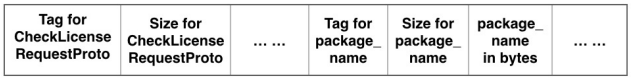

Figure 4: CheckLicenseRequestProto Serialization

2.1.3 HTTPS Request

For the network request, the Apache HttpClient is used by Google Play app to execute the http request and handle its response, while volley is utilized as a network scheduler dealing with request dispatching and caching [18]. The architecture overview of volley in Figure 5 demonstrates that volley manages a pool with three different kinds of threads: main thread, cache thread and one or more network threads. The programmer simply adds a request to the queue in the main thread without noting the threading or synchronization. In case of a cache hit, the request can be retrieved from cache and the cached response is delivered to the main thread. If a cache misses or is expired, the request would be pipelined into the request queue in network threads.

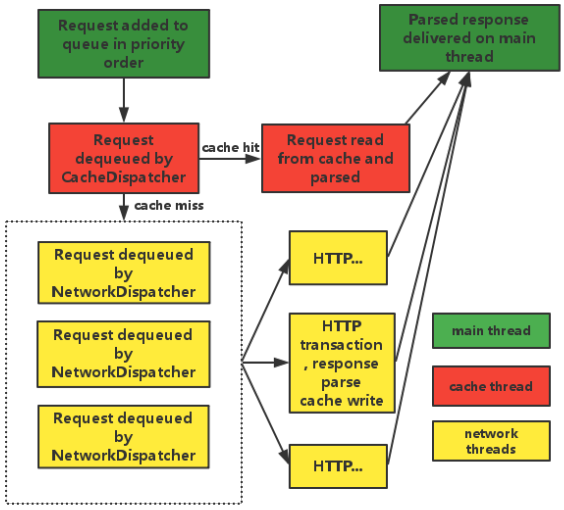

Figure 5: Architecture Volley Framework (based on [19])

2.1.4 Response Deserialization and Decoding

Firstly, the gzip compressed response data from Google server is decompressed by calling the Android built-in inflate method. Part of the decompressed bytes are listed in Table 1 [20] that gets explained below. Similar to serialization, the response data structure is also predefined by protocol message. The class ResponsePropertiesProto represents general network response data such as the result code, while CheckLicenseResponseProto contains license status specific data such as signature and signed data, which are used to verify the response. ResponseProto is a general term for both protocols mentioned above. Each protocol message or field is identified by a tag. Table 2 [20] shows the tag bytes for message protocols which are also marked red in Table 1. Table 3 [20] lists the tag bytes for each field, which are marked blue in Table 1. The bytes marked green in Table 1 are used to calculate the size of the protocol messages or their fields.

The decompressed bytes are read continuously by tag. Once a valid tag is read and the size is computed, different kinds of information such as the response code, signed data or the signature can be resolved. For example, once the byte -118 at (0, 8)[6] is read, it indicates that the check license response protocol is read. The bytes representing the type of message protocols are hard coded in protocol classes. The next 3 bytes {1, -45, 3} marked green are used to compute the number of bytes which will be read as check license response protocol. In this message protocol, there are three fields which are distinguished by tags: 8 for response code, 18 for signed data and 26 for for signature, as shown in Table 3 and marked blue in Table 1. For the signed data, for example, 116 at (1, 6) is the number of the bytes. That means the next 116 bytes will be read as signed data. The encoded signature can also be retrieved in the same way. To verify the response, the retrieved signature has to be decoded

[6]In (a,b), a represents the row number and b is the column number

based on Base64 algorithm.

Response Protocol

Response Properties Protocol

Nr.	0	1	2	3	4	5	6	7	8
0	11	18	5	8	0	32	-24	7	-118
1	1	-45	3	8	1	18	116	49	124
2	45	49	48	53	55	50	55	48	56
3	56	56	124	99	111	109	46	112	111
4	108	105	99	101	108	105	103	104	116
5	95	98	101	116	97	46	97	112	112
				Check License Response Protocol					
6	115	111	108	117	116	105	111	110	46
7	112	111	108	105	99	101	108	105	103
8	104	116	95	98	101	116	97	124	50
9	124	65	78	108	79	72	81	80	50
10	68	118	118	108	97	114	111	102	70
11	75	113	47	43	48	107	82	114	105
12	116	55	67	110	72	118	75	65	61
13	61	124	49	52	53	54	56	55	54
14	55	54	48	49	52	49	26	-40	2
15	76	111	73	112	48	122	57	105	90
16	86	85	70	82	114	52	56	113	116
17	101	77	83	84	56	122	109	89	104
...					... ...				

Table 1: Response Byte Stream

2.2 Attacks on existing solutions

As mentioned in the Introduction, the currently available protection options are not sufficiently secure and can be circumvented in different ways; ranging from very simple approaches by using cracking tools like Lucky Patcher [21], to more complex approaches using reengineering techniques (e.g. apktool) or even universal attacks by intercepting and replacing various calls dynamically. The following subsections are meant to provide the reader a quick overview on various attack techniques against major copyright protections used by the two major Appstores - Google Play Store and Amazon Appstore. Hereby, the actual reasons for the requirement of a better copy protection on Android are shown. The introduced attacks are presented in a summarized way only and addressed in detail in a different paper of this conference [7].

Byte	Type of Message Protocol
11	Response Protocol
18	Response Properties Protocol
-118	Check License Response Protocol

Table 2: Bytes for Types of Message Protocols

Byte	Tag	Message Protocol
8	result	Response Properties Protocol
8	response code	Check License Response Protocol
18	signed data	Check License Response Protocol
26	signature	Check License Response Protocol

Table 3: Bytes for Tags in Message Protocols

2.2.1 *Google's LVL and similar libraries*

While Google's LVL was actually cracked years ago already [22], it can be circumvented by non-savvy customers using the necessary tools like Lucky Patcher. More advanced users with IT-skills and a favor for hacking are even able to find tools, like apktool, on the internet themselves, which allows the modification of apps. The actual cracking of default LVL implementations is achieved in minutes. Essentially, it requires to redirect some switch-cases (representing failed license response) to a licensed state only [22], before recompiling and signing the APK file to allow its installation on a device again.

Furthermore, an advanced attack as implemented by [10] allows the interception and replacement of LVL messages using the Xposed Framework[8]. Our developed Xposed module exchanges the return value of Google's license server while replacing the used signature and public key of that reply. The result is a faked, but valid response (from the view of an app).

Even it was not verified by us, it is more than reasonable that these attacks work with third party copyright protections libraries like SlideMe's SlideLock and Samsung's Zirconia [24] [25]. Both of them have to be also implemented by the developer himself while the license server supplies the app with a response. It has to be assumed that the circumvention can be performed in a similar way as above [1].

2.2.2 *Amazon's DRM*

In comparison to aforementioned solutions, Amazon uses a developer-friendly approach in terms of time-requirements for the integration of a basic protection. Their DRM is integrated into any app on request, besides the injection of other Amazon services (used for the communication with the Amazon market app) upon upload to the market [26]. As analyzed by us in 2014 by [10], we were able to circumvent the protection, before Amazon applied modifications somehow in early 2016 as noticed in a review by [1]. In another analysis performed in June 2016 [27], we were able to identify an additional DRM switch that allows attackers to disable the protection instantly.

3. PROPOSED SOLUTION

Since the reengineering of Android apps is fairly easy [1] and the assembly dialect smali reads, with some training almost like source code, we evaluated into options on how

[7]Title:"An Insight to Cracking Solutions and Circumvention of Major Protection Methods for Android"

[8]The Xposed Framework allows the interception of Java function calls on rooted Android devices. Developers can create modules to modify apps on-the-fly [23]

to remove these references and the issue of an almost fully understandable source code (even when developers applied obfuscation methods). The most likely approach was to try native code, as it is commonly known from the desktop world that assembly code is difficult to understand due to all the lost references. Moreover, there are numerous options for obfuscation including using the static directives to hide internal functions [20], using pragmas or visibility attributes to hide functions [28] or hiding critical information (e.g. keys) in binary using the naked attribute resulting in confusion of all disassemblers [29]. In addition, obfuscators like obfuscator-LLVM may be used. Even by default, the Android NDK compiles any native code using the -O3[9] parameter, which acts like a soft-obfuscation and makes it difficult for decompilers to recover the code as evaluated in [1].

In general, customization is the key to protecting apps from getting cracked [1].

3.1 nLVL

Since the existing LVL is easy to crack as outlined previously, we analyzed and reengineered its communication to the Google license server using the Xposed framework in [20] and developed a completely native port of that library called "nLVL" that bypasses any insecure local frameworks (cf. binder / Play Store App). The nLVL communicates directly with the Google servers via an HTTPS connection. Therefore, the nLVL not only arranges the license verification in a more secure way, it is also immune against existing cracking methods as performed by the Lucky Patcher and the provided modded Play Store by [21]. An overview on the implementation is provided in Figure 6 below. As shown in that figure, the Android app itself can communicate with the library using JNI[10] calls (other possibilities are available and used by the so-called fusing options presented in the next section). The nLVL itself uses libCurl[11] to establish the direct and encrypted network connections to the Google license server.

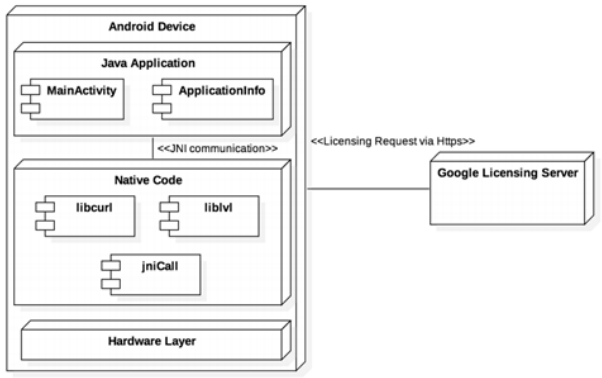

Figure 6: High-level overview of the nLVL [20]

Basically it does exactly the same licensing verification as the LVL with some minor differences (e.g., currently missing signature validation/proof-of-concept implementation) and major benefits (e.g., much more obfuscated and difficult to reengineer for average developers). In addition, while the Google frameworks have privileged rights and can access any required information, the nLVL is not able to obtain this information without the help of the existing Java frameworks and (the Android app) requires special permissions for that purpose [20]. By implementing apps using the nLVL for evaluation purposes, the following four permissions were required as stated in [1]:

- android.permission.GET_ACCOUNTS
- com.google.android.providers.gsf.permission.READ_GSERVICES
- android.permission.AUTHENTICATE_ACCOUNTS
- android.permission.USE_CREDENTIALS

In addition, lots of information on the user or device is requested and transmitted to the Google server upon each license request as shown in [20] and below. Futhermore, in our preparations for the evaluation apps we noticed that this type of information is not important for the actual license request, but gathered by Google for perhaps statistical purposes. By contrast, some devices do not provide all required attributes like the so-called loggingID on a Nexus 5 [1].

As explained in [20] the following attributes need to be fetched (or generated) to satisfy the requirements by the license server for the license request:

- Package name
- App version code
- Nonce
- User authentication token
- Android-ID
- Softwareversion
- Operatorname
- Operatorname numeric
- SIM operatorname
- SIM operatorname numeric
- Market-ID
- Logging-ID
- Device name
- User country
- User language

In reality as shown in [1] most of the values may be replaced by fixed values and the only important attributes for the license requests are the user-authentication-token, the Android-ID, the package name and the app version code. For security purposes the nonce should be modified each time, too. It is included in the license response later on and prevents replay attacks, since requests and its license-response with the signature would look exactly the same

[9] maximum optimization
[10] Java Native Interface
[11] File transfer library supporting various protocols

each time otherwise. All other attributes above may be replaced by fixed values instead and had (in our tests) no impact on the license response.

After acquiring above information [20] through JNI from the Java helper methods, the actual license request can be created, base64 encoded and finally appended to an URL call targeting the Google license server. In return the license server returns a zipped file "ApiRequest" [1] with the license response.

By default, it would be required to verify the response using the embedded public key in the Android app and check the signed license data against its signature. Since our solution is a proof-of-concept only, this step was skipped and falls under the category of known issues that should be addressed for productive usage to secure the nLVL against MITM attacks [1].

Obviously the remaining question is how to exchange the license reply with the actual Android app in a secure way, while preventing attackers from just removing the native code. This issue is addressed in the next section in which we present our so-called "fusing options" and "indirect method triggering" [1].

3.2 Fusing Options and Indirect Method Triggering

An intense problem remaining when using any native code is how to exchange the license information of a native LVL in a secure way while preventing attackers from just replacing the JNI calls in Java code with fixed return values. Here, the solution is so-called "fusing options" [1] that make the native code mandatory for the the Java code and vice-versa.

To achieve fusing options, we implemented methods in native code that replace values upon runtime (and upon valid license response) live in memory based on the ideas of self-modifying code as outlined in [30] as well as findings by [31] for replacing byte arrays in memory. For instance, the actual Java code may contain if-statements as follows:

```
// Byte arrays have to be declared
// before calling native code
// so that they can be found in memory
byte [] str1 = "NILS2K".getBytes();
byte [] str2 = "NILS2K".getBytes();
// native function call (nLVL)
nativeLVLcall();
// fusing method
if (Arrays.equals(str1, str2)) {
   exit();
}
```

Listing 1: Example for fusing option (Java) [1]

By default as shown in [1], the app would now instantly quit. Nevertheless, the previously called native code can check the license response by the Google license server and, upon positive license response, modify the if-statement in memory so that the quitting is no longer executed. The following code listing shows an example.

```
char adress[13];
FILE* fp;
char line[2048];

fp = fopen("/proc/self/maps", "r");
if (fp == NULL){
  // ignore in example
}
long long int mp;
void* vp;
char* lowerLimit;
char* upperLimit;
char *egg_end = 0;
int asize, incre;
while (fgets(line, 2048, fp) != NULL) {
    if((strstr(line, "rw-p") != NULL)) {
        if (line[8] == '-') {
            // address 0x00001111
            asize = 8;
            incre = 9;
        } else {
            // address 0x000011112222
            asize = 12;
            incre = 13;
        }

        strncpy(adress,line,asize);
        adress[asize+1] = '\0';
        mp = (long long int)strtoll(adress, NULL,
        ↪  16);
        vp = (void*)mp;
        lowerLimit = (char*) vp;

        strncpy(adress,line+incre,asize);
        adress[asize+1] = '\0';
        mp = (long long int)strtoll(adress, NULL,
        ↪  16);
        vp = (void*)mp;
        upperLimit = (char*) vp;
        int egg_count = 65;
        char* string_a = 0;
        for (char* i = lowerLimit; i < upperLimit
        ↪  - 6; i++){
            if (i[0] == 'N' && i[1] == 'I' &&
            ↪  i[2] == 'L'
                && i[3] == 'S' && i[4] == '2' &&
                ↪  i[5] == 'K'){
                // modify to disable
                // if-statement
                i[0] = (char) egg_count;
                i[1] = (char) egg_count;
                i[2] = (char) egg_count;
                i[3] = (char) egg_count;
                egg_count++;
            }
        }
    }
}
fp->_close;
```

Listing 2: Example for fusing option (C) [1] [31]

This fusing option approach may be performed in various variations. Its implementations need to be hidden in the Java code by hiding its intention using so-called "indirect method triggering" [1]. For instance, instead of calling the System.exit() method directly (attackers look for these calls as recognized in our evaluation), the app can create a file in the private app space. While that file looks non-suspicious to attackers, another method in the app is just awaiting its creation for modifying, e.g., a critical graphic source path resulting in a crash of the application at some later point. Moreover, there are no imminent traces to that file creation that might have been implemented differently (e.g., via shell command echo or Java-IO methods or even native code methods). An attacker is likely to be confused by the occurring errors and the error messages have no direct connection to the previous path change or file creation [1]. In a similar manner, the app can just be killed in the native code, which generally results in strange null-pointer exceptions when hitting any buttons (surprisingly Android does not kill the UI). This scenario certainly confuses attackers as evaluated during our practical evaluation tests with students trying to circumvent the protection [1]. There are various communication channels for this indirect method triggering. Besides those mentioned, the environment variables can be used for these purposes as well for allowing the secret setting of variables that may trigger actions in other parts of the program. Furthermore, the hardware may be misused here by switching devices on and off, causing the sending of broadcast messages throughout the system that can act like Morse-codes to activate certain functions. The options are certainly endless. Moreover, the resulting actions range from crashing the app to modifying it to annoy the software pirates (cf. strange app behavior) [1].

4. EVALUATION

Since it is difficult to rate the security improvement and the VdS [12] in addition to Thomas Goebl from the Denuvo company [13] confirmed us that there are no standardized certifications, we performed a hallway test with computer science students with different skill levels and monitored their approaches for a given timeframe of 20h to simulate attackers trying to circumvent the protection [1].

4.1 Preparation

In preparation of the evaluation, an unpublished demo application was selected from our own Android courses and the aforementioned protection methods applied (nLVL, fusing options and indirect method triggering). We also used ProGuard for shrinking and basic obfuscation [1].

In addition, each student was requested to fill out a questionnaire to acquire any existing skills, while it consisted of a self-rating with several questions in addition so that the instructor was able to verify the self-rating [1].

The groups were divided into beginners, intermediates, experts and experts 2. While the beginners had only some knowledge about Android, the intermediate group was aware of basic reengineering or IT-security knowledge. The expert group showed good knowledge of Android reengineering tools and skills in IT security, while we introduced to expert 2 group additionally the protection used and its methods in detail.

[12]German company for security certifications - www.vds.de

[13]Austrian company famous for highly secure x86 copyright protection solutions - www.denuvo.com

4.2 Results

All student groups, as outlined in [32] and [1], started with gathering reenginering knowledge in the first minutes. The beginners had to use Google instead initially, but found various resources quickly and were able to decompile and recompile the APK file while circumventing one of the fusing options. The beginners and intermediate teams tried the general cracking tools like Lucky Patcher or AntiLVL (without success). They also tried the decoupling of Java and native parts and looked for special values (e.g. exit calls) as well as using a deobfuscation tool in anticipation of recovering the code (all without success). In the end, the beginner team even approached the decompilation of the native code, but did not succeed. The expert 2 group instead tried various advanced approaches by using disassemblers like gdb or IDA to reveal the native code functions and patched and replaced certain opcodes (without success). Finally, the expert 2 group attempted to sniff the traffic by using a proxy server, but was unable to collect any data and ultimately gave up.

In summary, none of the teams was able to break our latest copyright protection methods.

5. RELATED WORK

Besides the copyright protection solutions of the major Appstores (cf. LVL and Amazon DRM as outlined before), other smaller markets provide their own solutions including SlideMe with its SlideLock [24] and Samsung with its Zirkonia library [25]. These both are similar to Google's LVL and have to be implemented by the developer. Therefore, both of them face similar issues and cannot be considered safe. In addition, other researchers focused on Android in recent years, providing ideas by using smartcards, encryption and dynamic code loading as presented by Shoaib et al. [33]. Instead, Wu Zhou et al. [34] have shown an approach using different opcodes to prevent any form of reengineering in their "DIVILAR"[34] solution. Furthermore, companies like [35] work on providing solutions by outsourcing code to a dongle. That idea is even similar to one of our own ideas (not presented in this paper) of using secure elements for gaining additional protection [1].

6. CONCLUSIONS

While the results (taken from [1]) in our evaluation are of limited value due to certain limitations in group size and timing, they still allow an assumption on the protection of Android apps in comparison to the existing copyright protection solutions provided by Google or Amazon. While these default solutions can be circumvented with cracking tools like Lucky Patcher or by manual patching using the apktool, our solution is momentarily highly protected against these attacks, because it is much more difficult to reengineer the native code than any Java-based Android application. The fusing options come in several variations using several communication options, which makes it very difficult to discover

and crack them. As outlined previously, individualism is the key and developers are requested to apply their own modifications to avoid issues with general cracking tools. We believe native code is the right way to provide much more protection, in terms of copyright protection, to the mobile world of Android. We verified the solution up to Android version 6.01.

7. FUTURE WORKS

As outlined previously, the nLVL has open flaws when it comes to the validation of the signature; these flaws need to be fixed for productive use. Furthermore, tools for injecting fusing option could be developed as most of the current protections were and have to be implemented manually. A disadvantage of an automatic approach would take place at costs of customization.
In addition, the nLVL needs to be reviewed and further hardened against possible interception options available for native code including Frida[14] or LD_PRELOAD directive[15].

8. LEGAL

Google and Amazon were notified about any issues related to their license verification either in 2015 or early 2016. In addition, Google was notified about our nLVL solution and invited to participate. Moreover, the presented attacks are not presented in detail to prevent any intended or actual malicious use of the information we present. Detailed information is available upon justified request. That applies also to any source codes.

UNPUBLISHED WORKS

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

9. ACKNOWLEDGMENTS

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

ABBREVIATIONS

AIDL Android Interface Definition Language

ASCII American Standard Code for Information Interchange

APK Application Package

DRM Digital Right Management

HTTPS HyperText Transfer Protocol Secure

HTTP HyperText Transfer Protocol

IDL Interface Definition Language

IPC Interprocess Communication

IT Information Technology

JNI Java Native Interface

JSON JavaScript Object Notation

LVL License Verification Library

LLVM Brand name (in former times referring to Low-Level-Virtual-Machine)

MITM Man in the middle

NDK Native Development Kit

nLVL Native License Verification Library

SIM Subscriber identity module

UI User interface

XML Extensible Markup Language

[14]www.frida.re

[15]www.cedricvb.be/post/intercepting-android-native-library-calls

10. REFERENCES

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

[2] Peter Rittwage. "Copy Protection". `http://diskpreservation.com/protection`. Last access: 07-15-2016.

[3] LaserLock. "Product Features". `http://www.laserlock.com/product_features.html#disc_check`. Last access: 07-15-2016.

[4] Steinberg. "USB-eLicenser (Steinberg Key)". `http://www.steinberg.net/en/products/accessories/usb_elicenser.html`. Last access: 07-15-2016.

[5] Denuvo. "FAQ". `http://www.denuvo.com/#page-2`. Last access: 07-15-2016.

[6] Google. "App Licensing". `http://developer.android.com/google/play/licensing/index.html`. Last access: 07-31-2016.

[7] ustwogames. Twitter account ustwogames. `https://twitter.com/ustwogames/status/552136427904184320`. Last access: 07-15-2016.

[8] Achim Killer. "Keine Android-Apps unter Windows 10". `http://www.br.de/themen/ratgeber/inhalt/computer/astoria-eingestellt-windows-10-android-apps-100.html`. Last access: 07-15-2016.

[9] Der Standard. "Kopierschutz von Android Market geknackt". `http://derstandard.at/1282273487603/App-Piraterie-Kopierschutz-vonAndroid-Market-geknackt`. Last access: 07-25-2016.

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

[11] Google. "Licensing Overview". `https://developer.android.com/google/play/licensing/overview.html`. Last access: 07-31-2016.

[12] Aleksandar Gargenta. "Deep dive into android ipc/binder framework". In AnDevCon: The Android Developer Conference, 2012.

[13] Google. "Android Interface Definition Language (AIDL)". https://developer.android.com/guide/components/aidl.html. Last access: 07-31-2016.

[14] Google. "Protocol Buffer". https://developers.google.com/protocol-buffers. Last access: 07-31-2016.

[15] Google. "Protocol Buffer Basics: Java". https://developers.google.com/protocol-buffers/docs/javatutorial. Last access: 07-31-2016.

[16] Bormi Toch. "Serialisation : Thrift et Protocol Buers, principes et aperu". http://blog.octo.com/serialisation-thrift-et-protocol-buffers. Last access: 07-31-2016.

[17] Simon Josefsson. "The base16, base32, and base64 data encodings". https://tools.ietf.org/html/rfc4648. Last access: 07-31-2016.

[18] Google. "Transmitting Network Data Using Volley". http://developer.android.com/training/volley/index.html. Last access: 07-31-2016.

[19] Google. "Sending a Simple Request". http://developer.android.com/training/volley/simple.html. Last access: 07-31-2016.

[20] Yixiang Chen. "Analysis and Reengineering of Google Frameworks for Development of a Native, Improved Version of License Verication Library (LVL)". unpublished.

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

[22] 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.

[23] rovo89. "Developer Tutorial". https://github.com/rovo89/XposedBridge/wiki/Development-tutorial. Last access: 07-15-2016.

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

[25] Samsung. "How to protect your app from illegal copy using Samsung Application License Management (Zirconia)", Feb. 12 2015". http://developer.samsung.com/technical-doc/view.do?v=T000000062L. Last access: 07-30-2016.

[26] Amazon. "Publishing Android Apps to the Amazon Appstore". https://developer.amazon.com/public/support/submitting-your-app/tech-docs/submitting-your-app. Last access: 07-15-2016.

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

[28] Apple. "Controlling Symbol Visibility". https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/CppRuntimeEnv/Articles/SymbolVisibility.html. Last access: 07-15-2016.

[29] Simone Margartelli. "Using ARM Inline Assembly and Naked Functions to Fool Disassemblers". https://www.evilsocket.net/2015/05/02/using-inline-assembly-and-naked-functions-to-fool-disassemblers/. Last access: 07-15-2016.

[30] Daniel Hugenroth and Fatih Kilic. Seminar Reverse Code Engineering: Obfuscation of Source Code and Intermediary Artifacts with Special Regard to the Android Platform, 2014. unpublished.

[31] 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.

[32] Students of Android Practical Course SS2016. TUM Android Practical Course - Research Task reports. unpublished.

[33] 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.

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

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