VB2021 localhost

7 - 8 October, 2021 / vblocalhost.com

UNCOVERING AUTOMATIC OBFUSCATION-AS-A-SERVICE FOR MALICIOUS ANDROID APPLICATIONS

Vit Sembera
Trend Micro, Czech Republic
vit_sembera@trendmicro.com

Masarah Paquet-Clouston
GoSecure, Canada
mcpc@gosecure.net

Sebastian Garcia
Czech Technical University, Czech Republic
sebastian.garcia@agents.fel.cvut.cz

Maria Jose Erquiaga
Cisco Systems, Czech Republic
merquiag@cisco.com

ABSTRACT

With the security community regularly developing mechanisms for malware detection, malware samples are constantly being obfuscated through various techniques. Although these changes are suspected to be automatic, there has been no research investigating how such automation works, how it is offered in the underground community, what obfuscation techniques are favoured, and whether offering automation-as-a-service is profitable.

This research presents a deep dive investigation into an obfuscation-as-a-service platform for *Android* applications advertised on underground forums. The various obfuscation techniques used by the service are uncovered and the service's efficiency is evaluated. The potential revenue made by those behind the service is also estimated based on open-source information found on various underground forums.

This research provides the first overview of such automatic service, which takes advantage of the whole malware-as-a-service industry, providing medium quality obfuscation for the *Android* malware market. Although the technical obfuscations are not state-of-the-art, the service succeeds in reducing detection for malicious *Android* applications. We conclude that the active use of the service highlights the need for the malware market to develop better obfuscation techniques, hence the good job that the security community is doing at quickly detecting changing malware. We also conclude that this service seemed to generate enough revenue for the group, given its automatic nature and purpose. Given that automatic services like this may be a greater problem in the future of malware obfuscation, this research provides a first technical analysis of the details of such obfuscation service and the possible impact in detection results.

1. INTRODUCTION

Malware is obfuscated and re-deployed continually due to the good work of the security community. This is also true for Android Application Packages (APKs), especially considering that, in 2017, a higher number of distinct *Android* malware files were observed in comparison to *Windows* malware [1]. Subsequently, there has been increasing work on *Android* malware [2], [3], as well as its obfuscation techniques [4], [5], and the efficiency of anti-malware products in detecting them [5–8]. In parallel, the cybercrime industry has emerged [9] and *Android* malware authors can turn to this industry to find new techniques to obfuscate their malicious samples.

Although obfuscation-as-a-service is a known specialization in the cybercrime industry, the mechanics behind the scene of such specialization have yet to be studied – including, for example, the challenges faced by highly specialized individuals or what kind of attackers benefit most from these services. This study provides a first overview of a specialized service offering automated obfuscation-as-a-service for malware *Android* authors. The service was reported in [10], [11] as actively used by *Android* botnet operators.

The analysis of the service was both technical and contextual. First, the obfuscation techniques used by the service were uncovered through static analysis and reverse engineering. Second, the business reality of those behind the service was explored through a series of short analyses of the service's usage, efficiency, potential profitability and market competition. The study's key takeaways are:

- The service offered average quality obfuscation, with most obfuscation techniques already known by the security industry.
- Among one novel technique, the use of Goto jumps between code blocks was ingenious as almost all Java decompilers were unable to keep track of the code flow.
- The service's clients are most likely large-scale attackers with highly malicious applications.
- Those offering the service have made a minimum revenue of USD 5,100 (conservative) to USD 61,160 (optimistic) for a six-month operation.

This study yields unique insights into an obfuscation service that can be used to start understanding the specialization processes currently trending in the cybercrime industry. Automation is key for fast adaptation in the malware world, opening a door to a new economy of services.

This paper is organized as follows: sections 2 and 3 survey the literature. Section 4 presents the methods and data. Section 5 shows the obfuscation techniques. Section 6 discusses the service's business context.

Section 7 goes over the discussion and section 8 concludes.

2. DETECTING CAT, HIDING MOUSE

Obfuscation aims at changing a program while preserving its functionality [12] and has successfully been used by malware authors to avoid detection for a long time. It makes an application more difficult to analyse, hiding its intentions [7], [12].

There are many techniques to obfuscate malware, including encryption, oligomorphic malware that changes embedded decryptors, polymorphic malware that forces a large number of decryptors, and code injection [12]. Obfuscation can be as simple as a new packing technique [12], to more complex metamorphic malware that changes itself as it propagates [2].

As the ecosystem for *Android* malware is thriving [3], [13], *Android* malware authors use these obfuscation methods to avoid anti-virus detection – a necessary means given the easy-to-reverse nature of Java [14]. Several studies have looked at the efficiency of anti-malware products to detect obfuscation methods in *Android* malware and they all concluded that trivial transformations, such as repacking, disassembling, identifier renaming and data encoding could evade most anti-virus products [5], [6], [8]. On the other hand, recently two studies focused on detecting obfuscation techniques, and in both cases, several obfuscation techniques were found to be easy to detect [4], [7]. This may explain why malware authors keep upping their game, developing new obfuscation techniques, such as run-time obfuscation, a novel technique that was used in 80% of the 2,000 malware samples studied in Wong and Li (2018) [4].

To buy toolkits to automatically obfuscate malware [15] or to discuss new strategies to avoid detection, one can now turn to the cybercrime industry.

3. SPECIALIZATION IN OBFUSCATION-AS-A-SERVICE

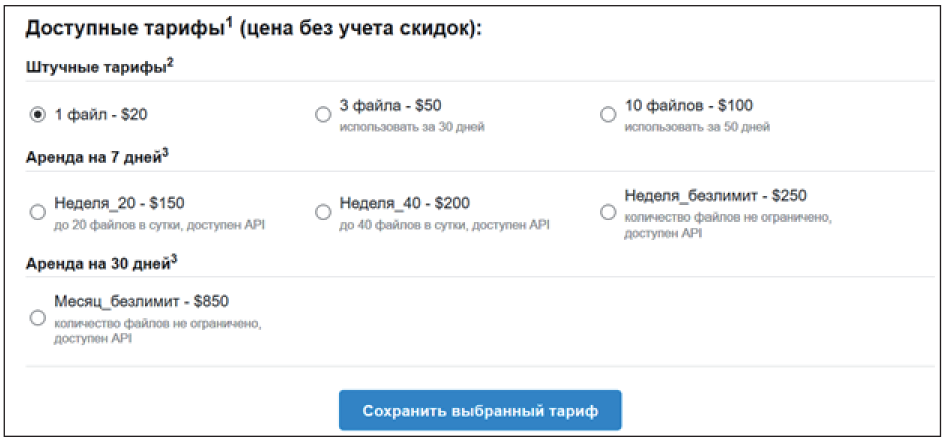
Over the past 20 years, malware authors have developed various strategies to organize and find business partners [16–19]. From online anonymous chat rooms to informal markets, and nowadays formal marketplaces, malware authors can trade their malicious skills and/or software through various means. These platforms, along with the interesting potential profitability surrounding cybercrime activities, have led to the rise of a cybercrime industry [9], [16], [20], [21], which currently accounts for half of all property crime [9]. This industry is characterized by a strong trend towards specialization and professionalization [22], [23]. Such specialization reduces the costs surrounding cybercrime through increased productivity and profitability [24]. Thus, individuals can now specialize in one skill and outsource the remaining tasks [21].

Among one specialization is obfuscation-as-a-service. Indeed, the cat-and-mouse game between attackers [2–5] and defenders [5–8] discussed above resulted in the appearance of obfuscation-as-a-service platforms. Since malware creators may not always have the skills to do good obfuscation, and they need to speed up the process, they must resort to using such a service.

Offering specialized services for malware obfuscation involves mastering a set of special technical skills, such as automating the process, generalizing the obfuscating techniques to all types of malware, adding new techniques, verifying that the techniques still work, and managing the business side.

To provide novel insights on this specialization, this study investigates an obfuscation-as-a-service, uncovering the various obfuscation techniques developed by the service, as well as providing a first understanding of the service's business context, including the service's usage, efficiency, potential profitability and competitors.

The service is named fttkit.com. It was found in a published investigation in early 2020 about a large-scale botnet operation that distributed *Android* trojans targeting Russian victims and banks [10]. The service required a coupon code to log into the platform, which we were able to find in hacking forums such as HackForums, xss.is, procrd.top, alligator-cash, or exploit.in. The service was always advertised as 'fully automated service for protection of *Android* applications'. Inside the service, the main language used was Russian, suggesting that the platform's customers were mainly Russian speakers. The prices for the service varied, ranging from USD 20 for obfuscating one APK to unlimited obfuscation of APKs for USD 850 per month, as shown in Figure 1. In order to use the service, payments needed to be sent to a one-time generated bitcoin address.



Доступные тарифы[1] (цена без учета скидок):

Штучные тарифы[2]

● 1 файл - $20	○ 3 файла - $50 использовать за 30 дней	○ 10 файлов - $100 использовать за 50 дней

Аренда на 7 дней[3]

○ Неделя_20 - $150 до 20 файлов в сутки, доступен API	○ Неделя_40 - $200 до 40 файлов в сутки, доступен API	○ Неделя_безлимит - $250 количество файлов не ограничено, доступен API

Аренда на 30 дней[3]

○ Месяц_безлимит - $850
количество файлов не ограничено, доступен API

Figure 1: Prices in the fttkit.com obfuscation service as shown inside the website.

4. METHODOLOGY

The first part of the research focuses on the technical analysis of the APKs. It consists of the reverse engineering and analysis of the selected APKs. The second part of the research focuses on quantifying the revenues related to the service. It includes an estimation of the service's efficiency, usage and revenue.

4.1. Uncovering the obfuscation techniques

To analyse how the service worked, three APKs were obfuscated and thoroughly reversed engineered. A fourth one, a benign one, was obfuscated later in the research for comparison purposes.

4.1.1. Obfuscated applications

The four applications obfuscated for the analysis are briefly presented below.

- **Android locker**. The *Android* locker, called install.apk, was analysed by pwncode.io in May 2020 [25]. This malicious APK adds a device administrator to reset the password, locks out the user from the device, and asks for a ransom. It was selected because it had an evident malicious behaviour and did not use any obfuscation techniques.
- **SMS stealer**. The second APK was an SMS stealer called Android-SMS-Stealer.apk [26]. The APK is part of a botnet and steals SMSs. Some of the strings in the code of this application were encrypted, but the layout of the application was simple. It was selected because it was easy to access the code and we had a good understanding of the APK's call functions. Both the install.apk and the Android-SMS-Stealer.apk were detected as malicious by dozens of anti-virus engines in *VirusTotal* [27], which also made them good candidates for obfuscation.
- **Potential Adware**. The third APK was called swimmingpool.apk and did not seem to have any malicious behaviour. However, when running the application, unwanted ads appeared. Also, on *VirusTotal*, two anti-virus engines flagged it as malicious. It was selected because its behaviour was 'shady', potentially spreading adware, yet not necessarily 100% malicious.
- **Benign**. Further in the research, a fourth APK was obfuscated. This APK was a benign one which showed 'Hello World' on a window[1].

4.1.2. Reverse engineering process

First, the obfuscated applications were compared with the original ones using the static analyser tool *MobSF* (*Mobile Security Framework*) [28]. *MobSF* provided a high-level comparison of the changes. Second, the obfuscated APKs were manually reversed engineered using the *JEB Android* decompiler tool [29]. The *JEB* commercial software was necessary because most open-source Java decompilers could not correctly follow the obfuscated code. Only the open-source *Ghidra* tool [30] was able to decompile the obfuscated APKs, but was unfortunately not able to properly reconstruct the initialization values.

The obfuscation techniques used by the service were the same for all applications. Thus, the results section summarizes the techniques uncovered and all examples are pulled from the obfuscated *Android* locker application .

4.2. Assessment of the business of obfuscation

To understand the business context surrounding such activity, the service's usage, efficiency, potential profitability and potential competitors were assessed through a series of analyses presented below.

4.2.1. Assessing service usage

The service's usage was assessed using *VirusTotal*, a 'free service that analyzes files and URLs for viruses, worms, trojans and other kinds of malicious content' [27]. It aggregates results from more than 65 anti-virus tools [31], [32]. Given *VirusTotal*'s wide use in the security community, most malware samples will end up being uploaded to the service by security operators within a few days [13], [31]. Thus, using *VirusTotal*, applications obfuscated by the service were fingerprinted.

Detecting other obfuscated applications in the wild

Thanks to indicators uncovered during the reverse engineering process (as explained below), specific modifications made to the obfuscated APKs could be used to identify them in the wild using YARA rules [33] on *VirusTotal*. More precisely, any ZIP or JAR file that contained a radio.ogg file in a /tracks folder could potentially be a file obfuscated by the service investigated. Using this fingerprint, we developed a YARA rule [33] and conducted two Retrohunt jobs (which allows searching for any files submitted on *VirusTotal* in the past 90 days). The first one, launched on 22 June 2020, found 2,173 files, of which 1,051 were not corrupted. The files were submitted to *VirusTotal* between 11 April and 22 June 2020. The second job, launched on 14 October 2020, extracted 2,051 files, of which 2,006 were not corrupted. The files were submitted to *VirusTotal* between 17 July and 14 October 2020.

In total, 3,058 APKs were found on *VirusTotal* over a six-month period. All decompiled files had features similar to the ones investigated: the strings in the manifests were all random and they all had a /tracks folder containing a file named radio.ogg. The files found are thus, with a high degree of certainty, obfuscated with the service investigated.

[1] Downloaded from https://apkpure.com/helloworld/my.v1rtyoz.helloworld.

Grouping obfuscated applications

These 3,058 applications needed to be grouped to find the probable clients to which they belonged. However, given the obfuscation applied on them, they were hard to distinguish. Fortunately, a mistake was made by the operators, leaking information about the original APK within the obfuscated one. In short, the /res/values folder of the obfuscated applications included strings.xml and ids.xml files with clear text strings, such as *Adobe Flash Player is a lightweight [...]*. This finding was confirmed by looking at the applications we obfuscated and comparing the clear text strings with the ones in the original files. This mistake was thus leveraged to group the obfuscated applications based on similar strings in the strings.xml file and the ids.xml files.

In total, seven distinct groups were found, an eighth included all outliers and a ninth included the applications obfuscated for this research. To have a better idea of what the groups represented, a dozen APKs were sampled randomly from each group and investigated using the *apklab.io* service. This mobile-threat intelligence platform is created and maintained by *Avast* and allows quick static and dynamic analyses of APKs.

4.2.2. Service efficiency

The service's efficiency was assessed using the *VirusTotal* platform. The platform aggregates the reports of whether a file is malicious or not from more than 65 anti-virus tools, as mentioned above. If the obfuscated applications are detected as malicious by several anti-virus engines on *VirusTotal*, then the service's efficiency is considered low, as the malicious intent of the application could not be successfully concealed.

4.2.3. Potential revenue

Operators' potential revenue was estimated considering the 3,058 applications found on *VirusTotal* and the different price bundles offered by the platform (see Figure 1). Two possible revenue scenarios were developed. The first scenario, a conservative one, considered that all APKs come from the same client, who bought an API price bundle at USD 850/month. The second scenario, an optimistic one, considered that all APKs found on *VirusTotal* were paid at the highest price of USD 20/APK.

Assessing competitors

Potential competitors were found by searching for the keywords 'crypt' (which is a 'slang' word used to talk about the service), 'obfuscation' and 'protection' in two darknet monitoring services: *Flare.systems* [34] and *Sixgill* [35]. Since January 2020, six potential competitors were found, advertising 'APK crypt service' on different underground forums. The platforms on which they advertised their services and their prices are presented in the results.

4.3. Ethical considerations

One ethical consideration that has to be outlined is that USD 80.76 was sent to a service known to conduct obfuscation-as-a-service for malware authors. However, since the aim of the research was to assess the service, such action was inevitable. No human interaction was involved and we received ethical approval from the Czech Technical University (N.CS11232) prior to starting the research. The hope is that the outcome of this research will surpass the harms created by contributing once to a service financing malware authors.

5. DEOBFUSCATING OBFUSCATION

This section analyses the obfuscation techniques used by the service. It starts with a static analysis, followed by manual reverse engineering.

Note that part of the obfuscation process was uncovered in a blog post [36] focused on understanding an *Android* banking trojan named Geost [10], [11]. The Geost application was obfuscated with the service. In the results section, the main differences between the results of the blog [36] and our results are highlighted, illustrating how the service has evolved.

5.1. Static analysis results

The first analysis of the obfuscated APKs was a static analysis using the *MobSF* tool [37]. This tool provided an overview of the main differences between the original APK and obfuscated ones. These differences are shown in Table 1 for the *Android* locker APK. Note that *MobSF* uses the Common Vulnerability Scoring System (CVSS) [38] to score vulnerabilities found inside an APK. CVSS scores are low (0.1–3.9), medium (4.0–6.9) or high (7.0–10). The security score, on the other hand, represents a calculation of risk, values can be critical (0–15), high (16–40), medium (41–70), and low (71–100).

Table 1 shows that the CVSS score of the APK changed from 5.9 to zero after obfuscation because *MobSF* was not able to find any vulnerability. Also, the overall *Security Score* increased for the obfuscated application (100/100) compared to the original one (75/100). Finally, the package name of the obfuscated file changed to pseudo-random letters and the file included additional services. This provides evidence that the service is good at evading detection by static tools.

	Original	**Obfuscated**
CVSS	5.9	0
Security score	75/100	100/100
Package name	com.chrome.beta	com. lorxvclhz. Veskdmzqmv
SDK (target and min)	8	8
Exported activities	0	0
Services	1	3
Exported services	0	0
Receivers	4	4
Exported receivers	4	4
Providers	0	0
Trackers	0	0
Signer certificate	good	good

Table 1: Main differences between the original and obfuscated file for the Android Locker APK.

5.2. Reverse engineering results

After the static analysis, we performed a manual reverse engineering of the obfuscated APKs to find the specific techniques used and how they changed.

The original *Android* locker APK consisted of a Dalvik executable with the following structure:

- Manifest file
- Signing certificate
- classes.dex
- Resources directory

After reverse engineering, the structure of the obfuscated APK was as follows:

1. A Dalvik executable dropper and launcher in clases.dex file.
2. An encrypted second-stage Dalvik executable payload (original classes.dex code with partially obfuscated symbol names) stored in tracks/radio.ogg.
3. Manifest file with obfuscated object names.
4. Resources directory.
5. Tracks directory containing several files with randomly generated names and radio.ogg file.
6. Signing certificate (different from original).

Consecutive stages of the APK lifetime are discussed below.

5.2.1. Dalvik executable dropper and launcher

The main purpose of the dropper is to open, decrypt and store a second-stage Dalvik executable payload. The dropper also initializes the second stage, links missing references in the Manifest file to the initialized code methods, and redirects code execution to the Application.onCreate() method in the payload. For this Dalvik file, the following obfuscation methods were used:

- Injected code
- Symbol obfuscation
- RC4 strings encryption

- Java reflection code
- Statically linked libraries

Comparing the obfuscated *Android* locker APK with the original APK from the Geost botnet (that triggered this research), portions of the dropper code had evolved slightly and the amount of injected code had increased. This indicates that those behind the protection service are actively updating their service.

Injected code

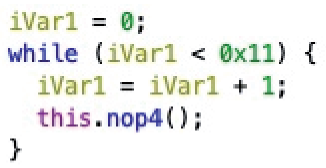
The service injected randomly generated junk code structures, including:

1. Boolean, integer, integer arrays and string variables.
2. Pseudo-randomly generated Boolean, integer and string literals (constants).
3. Assignment and calculation statements using variables and literals mentioned in points 1 and 2.
4. if/then/else statements using variables, literals and statements mentioned in points 1, 2 and 3, for hardly predictable code branching.
5. switch/case/default statements with the same purpose as in point 4.
6. try/catch blocks.
7. Loops, decompiled as while or for blocks.
8. Goto jumps between blocks mentioned in points 4, 5 and 6.
9. Class methods with randomly generated arguments using variables and literals as in points 1 and 2. These methods are frequently nested, calling each other.
10. Classes with variables and literals from points 1 and 2 and their constructors.

The whole obfuscated dropper consists of four classes, the attached libraries contain 416 classes, and the remaining 715 classes are junk code. Thus, 63% of the classes are junk. The overall size of the dropper Dex file is 941KB, 188 times larger than the original 5KB Dex file of the second stage. Also, compared with the original *Android* locker APK of 39.6KB, the obfuscated APK is 1.2MB, being 30 times larger.

The Goto jumps between code blocks represent a novel ingenious obfuscation method as almost all Java decompilers failed to keep track of the code flow. For example, the *Jadx* tool gave warnings such as '*Sanity check requires truncation of jumptable*, *Bad instruction - Truncating control flow* and *Second-state recovery error*'. Only the *Ghidra* tool seemed to be able to decompile the application, but it was not able to properly reconstruct the initialization values.

The modification of the code flow by inserting if/then/else, switch/case and try/catch constructions was ingenious. All of these constructions used a fake variable that was assigned a specific value to select the correct code path. An example of the if/then/else technique is shown in Figure 2.



```
iVar1 = 0;
while (iVar1 < 0x11) {
  iVar1 = iVar1 + 1;
  this.nop4();
}
```

Figure 2: Obfuscation method that inserts junk control flow structures: a fake while loop, a fixed variable value to force the output, and unique returns. The nop4() method was renamed during the deobfuscation analysis.

Symbol obfuscation

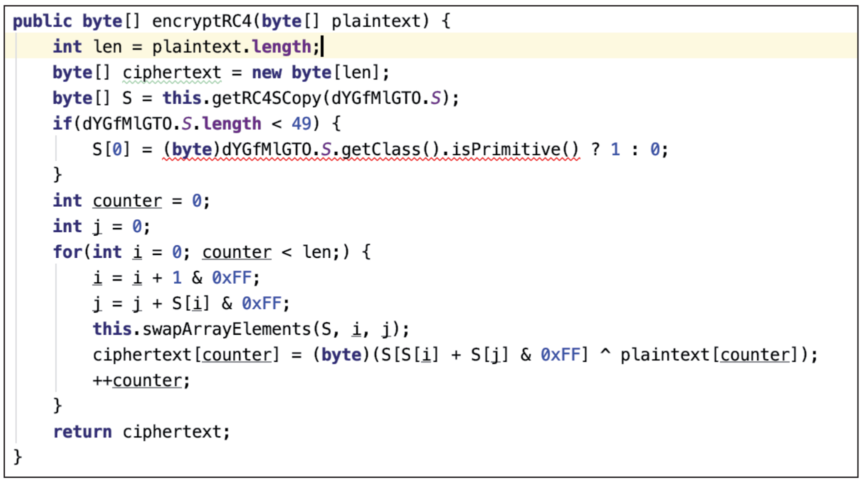
All symbols used in the obfuscated application, both for the dropper code and injected variables, have obfuscated names. These names are six to 12 characters long and consist of pseudo-randomly generated alphanumeric strings with both lower case and upper case letters. The same types of strings were used for symbols in the second-stage executable (which are also referenced in the Manifest file for the application to run properly).

RC4 strings encryption

Most of the strings in the dropper were replaced with in-lined RC4 decryption function calls, taking an integer array of encrypted text characters as an argument. Indeed, the RC4 algorithm for string encryption is divided into six separated methods to avoid easy recognition. It took a significant effort to understand this obfuscation strategy.

From the six separated methods in the RC4 function, the first easily recognizable one swaps array elements for the RC4 decryption. Once deobfuscated and its return value had been changed to void (not int), it was easier to find that the RC4 initialization was actually split into two methods, both heavily obfuscated with junk code.

The function inside the RC4 method is an array copy method. After an in-depth analysis, the complete deobfuscation of the RC4 function was possible, as shown in Figure 3.



```
public byte[] encryptRC4(byte[] plaintext) {
    int len = plaintext.length;
    byte[] ciphertext = new byte[len];
    byte[] S = this.getRC4SCopy(dYGfMlGTO.S);
    if(dYGfMlGTO.S.length < 49) {
        S[0] = (byte)dYGfMlGTO.S.getClass().isPrimitive() ? 1 : 0;
    }
    int counter = 0;
    int j = 0;
    for(int i = 0; counter < len;) {
        i = i + 1 & 0xFF;
        j = j + S[i] & 0xFF;
        this.swapArrayElements(S, i, j);
        ciphertext[counter] = (byte)(S[S[i] + S[j] & 0xFF] ^ plaintext[counter]);
        ++counter;
    }
    return ciphertext;
}
```

Figure 3: Deobfuscated code of the RC4 encryption function.

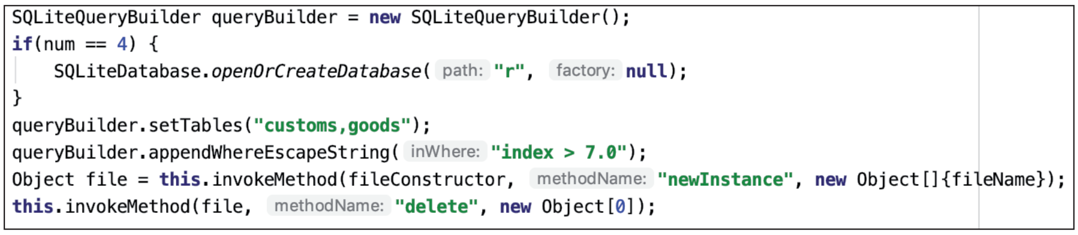
After deobfuscation, the RC4 private key could be uncovered. However, the service assigns a new RC4 private key to each APK, so there is no reason for publishing it. Overall, these techniques made the reverse engineering process more difficult while also ensuring that publishing the RC4 keys of one APK would not compromise other obfuscated ones.

Fortunately, the data is not encrypted as a stream, but as a block (without keeping RC4 internal state between decryption calls) so each string array could be decrypted separately and easily. Based on this finding, the discovered private key could be used to decrypt all the strings.

Java reflection code

The dropper code manipulates the encoded second stage using library methods with names like open(), read() and write(), which indeed attracts the attention of the reverse engineer. It also needs to initialize second-stage classes and launch the code in them, as well as call noticeable methods like getConstructor(). To hide these sensitive names, the dropper uses a statically linked Java ASM reflection library with obfuscated symbols. The authors behind the obfuscating process seem to be cautious, since some encrypted strings are even split into two parts, and at run-time, they are concatenated.

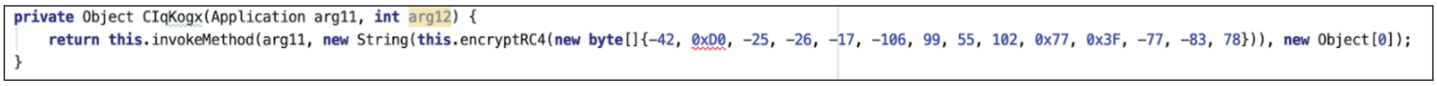
Besides all the encrypted strings used in stage 1, a few unencrypted decoy strings were left in the obfuscated APK. These strings are probably intended to divert the attention of the analyst. For example, the deleteFile() method contains SQL strings but they don't have any usage in the APK, as shown in Figure 4.



```
SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();
if(num == 4) {
    SQLiteDatabase.openOrCreateDatabase( path: "r",  factory: null);
}
queryBuilder.setTables("customs,goods");
queryBuilder.appendWhereEscapeString( inWhere: "index > 7.0");
Object file = this.invokeMethod(fileConstructor,  methodName: "newInstance", new Object[]{fileName});
this.invokeMethod(file,  methodName: "delete", new Object[0]);
```

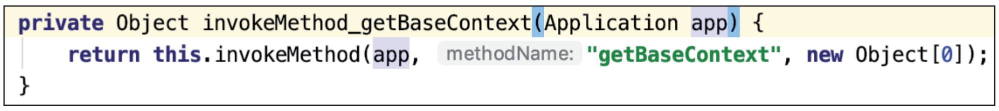
Figure 4: Decoy strings inside the deleteFile() function with fake SQL manipulations that are not used.

A good example of a double technique is the combination of encrypted strings and reflection, as shown in Figure 5, where the call to the encrypted RC4 method can be seen. After deobfuscating it, the method is more understandable, as shown in Figure 6.



```
private Object CIqKogx(Application arg11, int arg12) {
    return this.invokeMethod(arg11, new String(this.encryptRC4(new byte[]{-42, 0xD0, -25, -26, -17, -106, 99, 55, 102, 0x77, 0x3F, -77, -83, 78})), new Object[0]);
}
```

Figure 5: An example of a double technique, an encrypted function that is also a wrapper method.



```
private Object invokeMethod_getBaseContext(Application app) {
    return this.invokeMethod(app,  methodName: "getBaseContext", new Object[0]);
}
```

Figure 6: Decrypted version of the wrapped code of Figure 5.

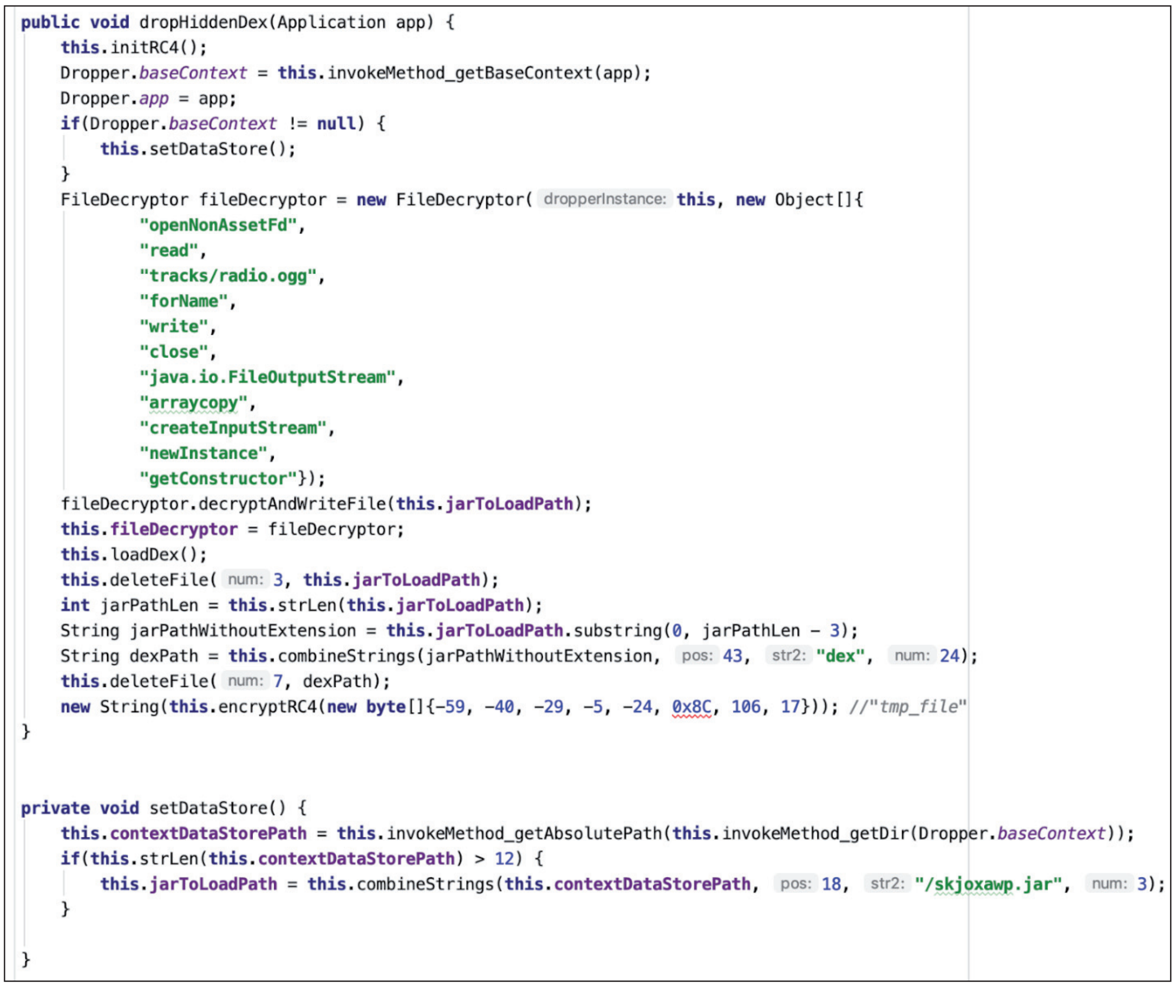
The evolution of the service can be seen in the use of the invokeMethod() method. This function is the core of all wrapper methods. It is much more complicated in this version than in the original Geost APK. Since the output of the method is the same, it is unclear why the method has been changed. In the current case of the *Android* locker, two new supporting methods were added, and the function looks completely different.

Statically linked libraries

One of the main problems of using dynamically linked classes and method references is that their names would reveal details about the code structure. Therefore, the service obfuscated and statically linked some libraries into the dropper. Some symbols and strings in the libraries were not obfuscated to avoid linking problems with additional *Android* libraries.

5.2.2. Second-stage executable

The second stage of the reversing process was to look into the code responsible for dropping and launching the Dalvik executable (Dex file from the original APK). There is a separate class definition for the hidden obfuscated application Dex file decryptor, the dropper, the launcher and the application onCreate() wrapper. The authors used adequate measures to hide this critical process by splitting already encoded strings. An example of all these techniques are shown in the code for dropping the Dex file in Figure 7.



```
public void dropHiddenDex(Application app) {
    this.initRC4();
    Dropper.baseContext = this.invokeMethod_getBaseContext(app);
    Dropper.app = app;
    if(Dropper.baseContext != null) {
        this.setDataStore();
    }
    FileDecryptor fileDecryptor = new FileDecryptor( dropperInstance: this, new Object[]{
            "openNonAssetFd",
            "read",
            "tracks/radio.ogg",
            "forName",
            "write",
            "close",
            "java.io.FileOutputStream",
            "arraycopy",
            "createInputStream",
            "newInstance",
            "getConstructor"});
    fileDecryptor.decryptAndWriteFile(this.jarToLoadPath);
    this.fileDecryptor = fileDecryptor;
    this.loadDex();
    this.deleteFile( num: 3, this.jarToLoadPath);
    int jarPathLen = this.strLen(this.jarToLoadPath);
    String jarPathWithoutExtension = this.jarToLoadPath.substring(0, jarPathLen - 3);
    String dexPath = this.combineStrings(jarPathWithoutExtension,  pos: 43,  str2: "dex",  num: 24);
    this.deleteFile( num: 7, dexPath);
    new String(this.encryptRC4(new byte[]{-59, -40, -29, -5, -24, 0x8C, 106, 17})); //"tmp_file"
}


private void setDataStore() {
    this.contextDataStorePath = this.invokeMethod_getAbsolutePath(this.invokeMethod_getDir(Dropper.baseContext));
    if(this.strLen(this.contextDataStorePath) > 12) {
        this.jarToLoadPath = this.combineStrings(this.contextDataStorePath,  pos: 18,  str2: "/skjoxawp.jar",  num: 3);
    }

}
```

Figure 7. Code to drop the second-stage Dalvik executable (original APK to be obfuscated). The technique uses multiple protection methods, such as separate class definitions, wrappers and splitting encoded strings.

One of the important data entries is *param[2]*, where the file name of the encrypted second-stage Dalvik executable is stored. In this case, the name is tracks/radio.ogg. The file is stored within the application files. The file is named as a radio audio track.

In Figure 7 the method setDataStore() can be seen, which contains the name of the dropped Dex. In this case, the name of the file is /skjoxawp.jar and the file type is changed to .dex afterwards. The content is decrypted with the same RC4 key as the strings, resulting in a zipped Dalvik module that is loaded with the loadDex() method. The loadDex() method creates an instance of a DexLoader class, which initializes the standard Java ClassLoader, DexClassLoader and a Dropper class instance. The method then calls the openOutputStreamAndWriteAll() method from the FileDecryptor class on unzipped data, doing all the read, decrypt and write work.

For manual second-stage file decryption, the first four bytes of the file containing the length of the data must be skipped. Therefore, to analyse the obfuscated second-stage Dalvik file, we need first to find the private RC4 key in the first stage code, then decode the name of the encrypted second-stage file (or it can just be guessed by the size of the file and its contents), then remove the first four bytes, and finally decrypt and unzip the file.

The extracted and decompiled second-stage Dalvik modules are the same as in the Dalvik file within the original APK that was sent to obfuscate, just with obfuscated symbol names which are referenced in the Manifest file.

5.2.3. Analysis of Manifest file with obfuscated object names

The manifest file of the obfuscated APK is quite normal, except for the fact that the symbol references are obfuscated. A quick analysis revealed that the application requests quite a lot of user permissions, which may be enough to conclude that the application is suspicious. One can also see that some symbols in the Manifest files for services and intents are referencing non-existent objects, illustrating a suspicious behaviour. The reason for this is that objects are loaded and initialized at run-time by the dropper.

6. ASSESSMENT OF THE BUSINESS OF OBFUSCATION

Given the various techniques of the obfuscation service now uncovered, the following section focuses on understanding the business context surrounding the service, including its usage, efficiency, potential revenue and competitors.

6.1. Service usage: clientele size

From the 3,058 applications, seven distinct groups were found. An eighth group encompassed all the outliers, and a ninth included the APKs obfuscated for this study. Table 2 shows details for each group: the number of APKs, what they tried to pass as based on the title and the photo, and additional insights on those that were investigated. The additional insights illustrate that the APKs within each group behave similarly, connecting to the same network domains or to similar domains (such as ccc1ccc.ru or bbb1bbb.ru).

This analysis indicates that the obfuscation service is probably used by several actors involved in spreading malicious applications related to malware. For example, the third group is associated with domains that are linked to an *Android* banking trojan botnet [39] while the fourth group includes APKs that connect to the rakason.ru domain, which is related to the Flexnet malware [40].

In sum, these APKs most likely do not belong to thousands of clients, but rather a small number of them (probably fewer than ten).

ID	APK faked as	# APKs	Insights on samples investigated
1	Flash Player, Instagram Shared	1,697	Connected to http://static.66.170.99.88.clients.your-server.de
2	Sistem Guncelles¸tirmesi¨ (system update)	416	Connected to http://orucakacdkkaldi.com (104.217.127.209), http://ba2a.com (108.187.35.84), selammigo34.com (34.91.209.109) and http://gunaydinmorroc.com (104.217.127.131)
3	Android Guncelleme (Android update) Browser Guncellemesi (browser update)	251	Connected to https://hnoraip.world, https://kalyanshop.best, https://dontworryman.club, https://Placeoftomcat.club. All hosted on IP 46.227.68.99 kalyanshop.best is associated with an Android banking trojan
4	MOD (a lot of money)	49	Communicated to http://rakason.ru (81.177.139.80). Related to Flexnet malware
5	Flash Player, Romance Mod, Spotify++	115	Connected to https://ccc1ccc.ru, https://eee5eee.ru, https://twitter.com. Hosted on IP 194.58.112.174
6	Flash Player, GoogleGPS, Android Guncellemesi (Android Update), Google Update	462	No connections and http://217.8.117.15
7	Notification (SMS app)	4	Connected to 142.250.102.188 and myluckycorp.com (107.161.23.204, 209.141.38.71 and 192.161.187.200)
8	Other	60	Various APKs
9	Install (Android Locker) Swimming Pool (Adware), Spy Mouse	3	Our APK submitted to the obfuscation service

Table 2: Estimated groups of clients of the obfuscation service based on APKs found in the wild that used this service.

6.2. Efficiency at obfuscating

The obfuscated and original APKs were uploaded to *VirusTotal* to assess the number of anti-virus engines that detected them. The more anti-virus engines that detect the files, the less efficient the service is. Anti-virus detections are shown in Table 3.

As shown in Table 3, the original *Android* locker and SMS stealer were detected by 27 and 29 of the anti-virus engines respectively. Using the obfuscation service reduced the detection to 16 and 13 respectively. Thus, when the file is malicious the service is efficient at reducing the detection rate by nearly half. On the other hand, the detection rate for the potential adware APK increased from two detections to ten, showing that the service is useful only for very malicious APKs.

	AV detections for original	**AV detections for obfuscated**
Android locker APK	27	16
SMS-stealer APK	29	13
Adware APK	2	10

Table 3: Detection rates of the original and protected APKs when uploaded to VirusTotal.

To investigate further, an application that was not malicious – called *benign* APK in the methodology – was obfuscated using the service. When uploaded to *VirusTotal*, the original application had no detections, but once obfuscated, eight anti-virus engines out of 65 flagged it as malicious, even tagging it as a *banker*. These findings confirmed that non-malicious applications could see their AV detection rate increase when using the obfuscation service.

Moreover, the three other obfuscated applications were also tagged as bankers by anti-virus engines, illustrating that the obfuscation service is closely linked to *Android* banking trojans, to the point that many anti-virus engines detect the obfuscated applications as bankers even if they are not.

The same process was conducted for the 3,058 applications found on *VirusTotal*. With little surprise, all of them were flagged as malicious. The minimum number of anti-virus detections was eight, just like the benign application above, and the maximum number of detections was 32. On average, the APKs found on *VirusTotal* were flagged as malicious by 18 anti-virus engines (with a standard deviation of 4.79).

6.3. Revenue scenarios

Leveraging the prices found on the platform as well as the number of APKs found on *VirusTotal*, two revenue scenarios were developed to assess the potential monetary success of those behind the platform.

The first scenario, quite conservative, considered that all obfuscated applications would have been purchased with one API access (USD 850/month) through six months of operation. In such a case, those behind the obfuscation-as-a-service platform would have made USD 5,100. The second scenario, quite optimistic, considered that all the applications would have been purchased at the highest price of USD 20 per APK. This would have yielded a revenue of USD 61,160.

These two scenarios allowed us to estimate a range. This range represents operators' potential lower bound revenue for all APKs found on *VirusTotal*.

6.4. Comparing the service with competitors

Since January 2020, six competitors have been found on different underground forums. They are presented in Table 4. The prices advertised for each competitor were higher than the prices of the service investigated. Moreover, none of them offered an automated platform with API access. Instead, they all asked potential clients to contact them via messaging applications like *Jabber* or *Telegram*.

Potentially, these competitors conduct their obfuscations manually, rather than automatically, which could explain the higher prices. This also means that the obfuscation-as-a-service investigated may have had a competitive edge by offering an 'automated service'. Otherwise, these competitors could also have an automated platform, but the access to the platform would not be available publicly, as it was possible for the service to be investigated.

7. DISCUSSION

This study is, to the best of our knowledge, the first to assess an obfuscation-as-a-service platform by reverse engineering the obfuscated APKs and comparing them with the original ones, while also looking at other high-level features of the service, such as its usage, efficiency, potential profitability and potential market competitors.

Service	Forum	Date	Prices
Competitor 1	XXS	August 2020	$30 1 APK, $80 4 APKs (1 week), $135 12 APKs (1 week) $250 25 APKs (1 week), $300 45 APKs (1 week)
Competitor 2	XSS, Club2crd Dark Market Devil Team, CenterClub	January 2020	$20 1 APK, $100 week for 10 APKs/day
Competitor 3	Club2crd	August 2020	$100 for 1 APK
Competitor 4	Hackforums	July 2020	$25 1 APK, $70 3 APKs, $99 5 APKs
Competitor 5	Ufolabs	October 2020	$30 1 APK, $150 4 APKs (1 week), $350 12 APKs (1 week) $550 25 APKs (1 week), $1,000 45 APKS (1 week)
Competitor 6	SKYNETZONE CHAT telegram group	November 2020	$150 4 APKs (1 week), $350 12 APKs (1 week), $550 25 APKs (1 week) $1,000 45 APKS (1 week)

Table 4: List of potential competitors.

An average service

The obfuscation service modifies the applications using techniques such as complex strings splitting, decoy strings, and nested junk flow control that increased the amount of code to analyse by x188. Most of the techniques used by the obfuscation service have already been mentioned in studies investigating obfuscation techniques used by malware samples [4], [5], [7], [14]. A novel technique discovered was the use of a high number of goto jumps, which obstructed almost all Java decompilers as they could not keep track of the code flow.

Overall, this platform seems to provide a service that is not complex, but sophisticated enough to reduce anti-virus detection of highly malicious files.

Limited potential clientele and average revenue

In terms of the size of the clientele, based on the APKs found, the service seems to have few clients. Although the assessment is limited to *VirusTotal*, given the tool's high visibility in the community, it is unlikely that the service is used by hundreds of clients. Moreover, these potential clients are most likely those behind highly malicious applications since our efficiency analysis illustrated that highly malicious applications would be less well detected while benign ones would see an increase in detection rate.

The revenue scenarios illustrated that those behind the obfuscation platform made a minimum revenue of between USD 5,100 and USD 61,160 for about six months of operation. It is far from the *millions* of dollars sometimes reported in the cybersecurity industry [41], but rather close to what seems to be an *average* revenue for an *average* cybercriminal. It is similar to the amounts found for the less successful ransomware families [42]. Please note that these revenue scenarios are estimations and are not comprehensive. They are based on the fingerprinted APKs found on *VirusTotal* and thus represent a lower bound.

Overall, the service is quite niche: it has a few clients who most likely need to conceal highly malicious applications and need an automated platform to obfuscate several samples. The market competition analysis illustrated that the service was the only one with a publicly available platform and offering lower prices. The success around the criminal business seems to be moderate.

Benefiting large-scale attackers

Even for an average service, the obfuscation prices are not marginal: a minimum of $850 per month for unlimited access or USD 20/APK are substantial costs. The clients seem to be large-scale attackers making enough money to pay for the service. The marginal effect of the obfuscation service in decreasing detection rates is enough to increase the chances of avoiding detection. Those benefiting the most from such obfuscation services may be those leveraging the specialization for large-scale attacks, rather than those offering it.

Study limitations and future research

An important limitation of this study is the relatively small sample size, since the estimation and analysis is limited to *VirusTotal*. Future studies will use other platforms, such as *Virus Share* [43]. However, the use of these platforms requires further fingerprinting strategies. Also, only three APKs were obfuscated because (i) it was too expensive, and (ii) there was no indication that the service may differ. The analysis could be expanded to other obfuscation services, allowing a greater understanding of the market for obfuscation-as-a-service. Nevertheless, the explanatory nature of this study yields unique insights into the business reality of those behind specialized services in the cybercrime industry.

8. CONCLUSION

This study investigated an online service specialized in *Android* applications obfuscation. The techniques used were uncovered and the business reality of those behind the service explored. On a technical level, the various techniques described and discussed may be helpful to security analysts and reverse engineers who face obscure applications every day. On a broader level, this research contributes to understanding the specialization processes behind the cybercrime industry. The investigated service was found to be average, using a few novel obfuscation techniques. It had a potentially small clientele formed of mainly large-scale attackers. Operators offering the service were estimated to have made a minimum revenue ranging from USD 5,100 (conservative) to USD 61,160 (optimistic) for a six-month operation. In the end, those who are likely to benefit the most from the specialized service are large-scale attackers who are able to pay the service's high fees in exchange for a slight decrease in the detection rate of their highly malicious applications. This study opens the door to further research questions regarding automated specialized services related to the cybercrime industry, their relative efficiency and who benefits the most from them.

ACKNOWLEDGEMENTS

We would like to thank the Stratosphere Laboratory and specifically Veronica Valeros for helping us throughout the research. This work was achieved through a collaboration with the Czech Technical University, *GoSecure Inc.* and *Trend Micro*. It was also partially funded by *Avast Software* through the Aposemat Lab.

REFERENCES

[1] Nokia Threat Intelligence. Nokia threat intelligence 2017. Technical report, Nokia, 2017. Accessed 2020-10-14.

[2] Wong, W.; Stamp, M. Hunting for metamorphic engines. Journal in Computer Virology, 2(3):211–229, 2006.

[3] Zhou, Y.; Jiang, X. Dissecting Android malware: Characterization and evolution. In 2012 IEEE symposium on security and privacy, pages 95–109. IEEE, 2012.

[4] Wong, M. Y.; Lie, D. Tackling runtime-based obfuscation in android with {TIRO}. In 27th {USENIX} Security Symposium ({USENIX} Security 18), pages 1247–1262, 2018.

[5] Zheng, M.; Lee, P. PC.; Lui, J. CS. Adam: an automatic and extensible platform to stress test Android anti-virus systems. In International conference on detection of intrusions and malware, and vulnerability assessment, pp.82–101. Springer, 2012.

[6] Canfora, G.; Di Sorbo, A.; Mercaldo, F.; Visaggio, C. A. Obfuscation techniques against signature based detection: a case study. In 2015 Mobile Systems Technologies Workshop (MST), pp.21–26. IEEE, 2015.

[7] Mirzaei, O.; de Fuentes, J.M.; Tapiador, J.; Gonzalez-Manzano, L. Androdet: An adaptive Android obfuscation detector. Future Generation Computer Systems, 90:240–261, 2019.

[8] Rastogi, V.; Chen, Y.; Jiang, X. Catch me if you can: Evaluating android anti-malware against transformation attacks. IEEE Transactions on Information Forensics and Security, 9(1):99– 108, 2013.

[9] Anderson, R.; Barton, C.; Bolme, R.; Clayton, R.; Ganan, C.; Grasso, T.; Levi, M.; Moore, T.; Vasek, M. Measuring the changing cost of cybercrime. 2019.

[10] Shirokova A.; Garino C. G.; Garcia S.; Erquiaga M. J. Geost botnet. operational security failures of a new Android banking threat. IEEE European Symposium on Security and Privacy Workshops (EuroSP), (1):406–409, 2019.

[11] Shirokova, S.; García, S.; Erquiaga, M. J. Geost botnet. The story of the discovery of a new Android banking trojan from an OpSec error. 2019. Accessed 2020-10-14.

[12] You, I.; Yim, K. Malware obfuscation techniques: A brief survey. In 2010 International Conference on Broadband, Wireless Computing, Communication and Applications, pp.297–300. IEEE, 2010.

[13] Suarez-Tangil, G.; Stringhini, G. Eight years of rider measurement in the Android malware ecosystem: Evolution and lessons learned. CoRR, abs/1801.08115, 2018.

[14] Dong, S.; Li, M.; Diao, W.; Liu, X.; Liu, J.; Li, Z.; Xu, F.; Chen, K.; Wang, X.; Zhang, K. Understanding Android obfuscation techniques: A large-scale investigation in the wild. In the International Conference on Security and Privacy in Communication Systems, pp.172–192. Springer, 2018.

[15] Birla, K.; Campus, G. Evolution and detection of polymorphic and metamorphic malwares: A survey. 2014.

[16] Afroz, S.; Garg, V.; McCoy, D.; Greenstadt, R. Honor among thieves: A common's analysis of cybercrime economies. In 2013 APWG eCrime Researchers Summit, pp.1–11, 2013.

[17] Dupont, B.; Côté, A.-M.; Boutin, J.-I.; Fernandez, J. Darkode: Recruitment patterns and transactional features of "the most dangerous cybercrime forum in the world". American Behavioral Scientist, 61(11):1219–1243, 2017.

[18] Holt, T.J. Examining the forces shaping cybercrime markets online. Social Science Computer Review, 31(2):165–177, 2013.

[19] Holt, T.J. Exploring the social organisation and structure of stolen data markets. Global Crime, 14(2-3):155–174, 2013.

[20] Collier, B.; Clayton, R.; Hutchings, A.; Thomas, D.R. Cybercrime is (often) boring: maintaining the infrastructure of cybercrime economies. Workshop on the Economics of Information Security, WEIS, 2020.

[21] van Wegberg, R.; Tajalizadehkhoob, S.; Soska, K.; Akyazi, U.; Hernández Ganan, C.; Klievink, B.; Christin, N.; van Eeten, M. Plug and prey? measuring the commoditization of cybercrime via online anonymous markets. In 27th USENIX Security Symposium (USENIX Security 18), pp.1009–1026, Baltimore, MD, August 2018. USENIX Association.

[22] Hutchings, A.; Holt, T. J. A crime script analysis of the online stolen data market. British Journal of Criminology, 55(3):596– 614, 2015.

[23] Thomas, K.; Huang, D.; Wang, D.; Bursztein, E.; Grier, C.; Holt, T. J.; Kruegel, C.; McCoy, D.; Savage, S.; Vigna, G. Framing dependencies introduced by underground commoditization. 2015.

[24] Moore, T.; Clayton, R.; Anderson, R. The economics of online crime. Journal of Economic Perspectives, 23(3):3–20, 2009.

[25] c0d3inj3cT. Android locker targeting russian users. Technical report, pwncode, 2020. Accessed 2020-10-14.

[26] Kersten, M. Android SMS-stealer. Technical report, Security through explanation, 2018. Accessed 2020-10-14.

[27] VirusTotal. https://www.virustotal .com/. Accessed 15 October 2020.

[28] Mobile security framework. https: //github.com/MobSF/Mobile-Security-Framework-MobSF. Accessed 15 October 2020.

[29] Jeb Android decompiler. https: //www.pnfsoftware.com/. Accessed 15 October 2020.

[30] Ghidra tool, 2020. https://ghidra-sre.org. Accessed 15 October 2020.

[31] Song, L.; Huang, H.; Zhou, W.; Wu, W.; Zhang, Y. Learning from big malwares. In Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems, pp.1–8, 2016.

[32] Ugarte-Pedrero, X.; Graziano, M.; Balzarotti, D. A close look at a daily dataset of malware samples. ACM Transactions on Privacy and Security (TOPS), 22(1):1–30, 2019.

[33] Naik, N.; Jenkins, P.; Savage, N.; Yang, L.; Boongoen, T.; Iam-On, N.; Naik, K.; Song, J. Embedded yara rules: strengthening yara rules utilising fuzzy hashing and fuzzy rules for malware analysis. Complex & Intelligent Systems, 7(2):687–702, 2021.

[34] Flare.systems https://flare.systems/.

[35] Sixgill. https://www.cybersixgill.com/.

[36] Sembera, V. Geost: Anatomy of the Android Trojan Targeting Russia. Technical report, TrendMicro, 2020. Accessed 14 October 2020.

[37] Bergadano, F.; Boetti, M.; Cogno, F.; Costamagna, V.; Mario Leone, M.; Evangelisti, M. A modular framework for mobile security analysis. Information Security Journal: A Global Perspective, 29(5):220–243, 2020.

[38] Mokris, K. CVSS scores for mobile apps: Guidelines for measuring mobile risk. Technical report, NowSecure, 2017. Accessed 14 October 2020.

[39] https://twitter.com/rebensk/status/1269233752397557760?lang=en.

[40] https://twitter.com/verovaleros/status/1268248014596059136

[41] Goodchild, J. Cybercrime's most lucrative careers. Technical report, DarkReading, 2021. Accessed 14 April 2021.

[42] Paquet-Clouston, M.; Haslhofer, B.; Dupont, B. Ransomware payments in the bitcoin ecosystem. Journal of Cybersecurity, 5(1):tyz003, 2019.

[43] Virus Share. https://virusshare.com. Accessed 15 October 2020.