electronics

MDPI

Review

Exploring Android Obfuscators and Deobfuscators: An Empirical Investigation

Shouki A. Ebad * and Abdulbasit A. Darem

Department of Computer Science, Faculty of Science, Northern Border University, Arar 91431, Saudi Arabia; basit.darem@nbu.edu.sa

* Correspondence: shouki.abbad@nbu.edu.sa; Tel.: +966-14-661-5607

Abstract: Researchers have proposed different obfuscation transformations supported by numerous smartphone protection tools (obfuscators and deobfuscators). However, there is a need for a comprehensive study to empirically characterize these tools that belong to different categories of transformations. We propose a property-based framework to systematically classify twenty cutting-edge tools according to their features, analysis type, programming language support, licensing, applied obfuscation transformations, and general technical drawbacks. Our analysis predominantly reveals that very few tools work at the dynamic level, and most tools (which are static-based) work for Java or Java-based ecosystems (e.g., Android). The findings also show that the widespread adoption of renaming transformations is followed by formatting and code injection. In addition, this paper pinpoints the technical shortcomings of each tool; some of these drawbacks are common in static-based analyzers (e.g., resource consumption), and other drawbacks have negative effects on the experiment conducted by students (e.g., a third-party library involved). According to these critical limitations, we provide some timely recommendations for further research. This study can assist not only Android developers and researchers to improve the overall health of their apps but also the managers of computer science and cybersecurity academic programs to embed suitable obfuscation tools in their curricula.

Keywords: software obfuscation; tools; malware detection; Android; drawbacks

check for updates

Citation: Ebad, S.A.; Darem, A.A. Exploring Android Obfuscators and Deobfuscators: An Empirical Investigation. *Electronics* **2024**, *13*, 2272. https://doi.org/10.3390/electronics13122272

Academic Editors: Hsi-Min Chen and Shang-Pin Ma

Received: 4 May 2024
Revised: 29 May 2024
Accepted: 6 June 2024
Published: 10 June 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Obfuscation refers to obscuring the intended meaning of the source or machine code. It can then be seen as a technique to maximize software protection. Obfuscation often deters reverse engineering and prevents code tampering by making the code difficult to understand and analyze. On the other hand, deobfuscation means making the code clear enough to be understandable and readable; it is a reverse engineering obfuscated code and is useful for understanding the obfuscated code. In this paper, the programs that do obfuscation and deobfuscation are called obfuscator and deobfuscator, respectively [1]. However, there is some overlap between these concepts with others, such as optimizer, analyzer, beautifier, minifier, and compressor. Although the concept of obfuscation goes back to the 1990s, researchers have paid much attention to this area and become competent to discuss obfuscation transformations over the last few years [2]. This heightened interest stems from the escalating proliferation of malicious software in recent years when the adoption of Android smartphones has increased tremendously. Up to January 2024, Android has maintained its dominant position within the smartphone market, leading to a corresponding increase in its users and developer base [3].

The significance of these tools stems from their pervasive implementation in mobile apps. Obfuscators and deobfuscators are extensively utilized in mobile app development, especially Android apps. When creating an app, the plugins are integrated into the default toolset for the app's construction process. In practical terms, legitimate software firms

must assess which obfuscator is optimal for safeguarding their app's intellectual property. Simultaneously, malware analysts seek to identify the most effective deobfuscator for efficiently analyzing malicious software.

Several obfuscation transformations have been proposed, such as variable renaming and inserting a dead code. To assist in the quality of such transformations, different quality attributes have also been used, including potency, resilience, cost, stealth, and similarity, which are the most widely used metrics to evaluate the quality of obfuscation techniques [2]. Among these attributes, the commonly recognized one is similarity, which refers to the degree two code samples exhibit identical functionality. It is measured in terms of a variety of numeric values such as Euclidean distance, longest common sequence, and string matching. This attribute has received more attention from researchers because a large proportion of Android apps represent cloned or repackaged versions built by third-party developers to efficiently construct and spread malware [4]. To acquaint the reader with the terminology employed in this paper, the following two subsections furnish an outline of obfuscation transformations and the structure of an Android app package.

1.1. Obfuscation Transformations

Numerous obfuscation transformations have been proposed, predominantly drawing inspiration from Collberge's work [5]. Subsequently, additional transformations have also been considered. All of these transformations make the source code less informative to a reverse engineer. Almost all transformations are summarized and categorized in Table 1. It is important to notice that there is no unified categorization for these types of transformations; different authors give different names to the same things; see, for example, [5–7].

Table 1. Summary of obfuscation transformations.

Category	Sub-Category
Formatting	Removing comments Removing debugging information (may be conducted during compiling) Removing whitespaces Removing indentations Removing unreachable code Printing empty strings
Renaming/modifying	Renaming identifiers (class name, method name, attribute/field name) Renaming/modifying variable declarations Replacing the variable with an expression Replacing the numeric values of array elements with arithmetic expressions Placing string literals into arrays Changing the array indices Modifying composite data types Modifying loop conditions
Code injection	Inserting redundant code Inserting dead/junk code Inserting redundant operands (using algebraic laws in arithmetic expressions) Inserting dummy statements Opaque predicate (a conditional expression whose outcome is discernible to the obfuscator but challenging for an adversary to determine statically. It is commonly employed to generate a bogus code within programs and is particularly prevalent in control flow obfuscations).
Other transformations (merging, splitting, flattening, dynamic dispatcher, etc.)	Overloading Encryption Merging two or more scalar variables into one variable Breaking up one variable into two or more variables Splitting the array into sub-arrays Merging two or more arrays into one Folding the array: increase the number of dimensions Flattening the array: decrease the number of dimensions Flattening the code makes the basic blocks look like they have the same set of predecessors and successors, including control flow flattening Parallelizing code Using aliased pointers to determine the exact indirect addresses of the dispatcher

1.2. Android App Packaging

There are several obfuscators and deobfuscators that aim at discovering similarities among Android apps; here, a brief description of the structure of any Android app (packaged as an APK file) is provided [8].

An Android application, typically packaged as an APK file, is essentially an archive, akin to a ZIP file, containing a variety of files (as illustrated in Figure 1), primarily comprising the following:

1. AndroidManifest.xml: this configuration file holds crucial details like declared permissions and a list of components.
2. classes.dex: this bytecode file represents the primary application code in the Dalvik Executable (DEX) format.
3. The res directory: this directory houses diverse resource files, including images and layouts that define the visual appearance of the app.
4. Other components may include the META-INF directory, storing authors' certificates, and the assets directory, which serves as storage for raw data.

```
|Example.apk
|-- AndroidManifest.xml //main config.
|-- classes.dex //main app code
|-- res //resource files
|    |-- drawable
|    |-- layout
|    |-- ... ... // menu, raw, etc.
|-- META-INF //signature
|-- ... ... //lib, assets, etc.
```

Figure 1. The components of an Android APK.

There have been a number of obfuscators and deobfuscators (hereafter, tools) in the market as a solution against many types of attacks, especially in the context of Android. Such tools differ in their quality and the number and type of obfuscation transformations. This paper provides an in-depth analysis of the existing tools, including classifications and drawbacks. This categorization facilitates the identification of technical domains that have not been thoroughly explored yet, highlighting the need for further development in the future.

The contribution of this paper can be summarized as follows:

1. To systematically characterize the smartphone obfuscation tools (obfuscators and deobfuscators).
2. Secondly, an attribute-based framework is used to classify twenty tools according to their features, including analysis type, the programming languages supported, licensing, applied obfuscation transformations, and general technical drawbacks. This framework could be used by other researchers to conduct a similar study.
3. The drawbacks and limitations of each obfuscation tool are discussed in detail. Timely recommendations are provided accordingly.

This study can assist three types of practitioners: (a) mobile app developers to understand the characteristics and the capability of the studied obfuscators and deobfuscators, (b) researchers and engineers to address the identified drawbacks and enhance the overall health of smartphone ecosystems, and (c) managers of cybersecurity academic programs to embed suitable obfuscation tools in their curricula [9].

This paper is organized as follows. First, related works and their knowledge gaps are discussed in Section 2. Next, the required background is presented in Section 3. The research design and methodology that was followed to perform this research are provided in Section 4. Then, the main findings and observations are discussed in Section 5. After

that, the limitations of this study are provided in Section 6. Finally, this study's conclusion is drawn in Section 7.

2. Related Work

This section delineates the research conducted on obfuscation tools, encompassing those that compare two or more artifacts to ascertain their similarity rate or to identify specific features or patterns. Examples include tools for malware detection, plagiarism detection, similarity detection, clone detection, and reverse engineering. Each of these tool types, to varying degrees, incorporates the concepts of obfuscation and deobfuscation.

Rauti and Leppänen [1] conducted a comparative analysis of online JavaScript obfuscators, assessing their resilience, effectiveness, and associated costs. They discovered that although many obfuscators theoretically offer robust protection, their outputs could be easily reversed by automated deobfuscators. Duchêne et al. [10] conducted a survey focusing on network protocol reverse engineering tools developed from 2004 to 2015. Their study centered on tools dedicated to inferring the format of individual messages or the grammar of message sequences. The authors categorized these tools based on various criteria to offer valuable insights into the techniques employed by each. Wermke et al. [11] introduced Obfuscan, a tool designed to analyze obfuscation usage in free Android apps on Google Play. Their findings revealed that out of 439,232 apps, a significant portion were obfuscated by developers, leaving over 75% vulnerable to malicious repackaging. Through an online survey targeting Android developers, they also discovered that while 78% were familiar with obfuscation, only 48% actually utilized software obfuscation techniques.

Martins et al. [12] illustrated the state of the art in software plagiarism detection tools by comparing their features and testing them against simple source codes. Roy et al. [6] focused on clone detection techniques and tools, providing a concise but comprehensive survey and a hypothetical evaluation based on editing scenarios. Recently, this work was extended by Svajlenko and Roy [13] to include more clone detection tools. Amro [14] summarized most of the malware detection techniques used by Android and iOS, and he presented a brief description of each technique. Tchakounté et al. [15] proposed an Android malware detection tool, LimonDroid, and they tested the tool with tens of malicious and benign apps. This was aimed at establishing a robust signature database able to identify malicious trends in Android apps. You et al. [16] investigated the characteristics and performance of four optimization and obfuscation tools used in Android apps. The tools included R8, ReDex, Obfuscapk, and DeGuard. Foltýnek et al. [17] tested 15 web-based text-matching systems that could be used when plagiarism is suspected. The test was conducted using material in eight different languages. Although some systems could identify some plagiarized content, they did not find all materials to be plagiarized, and at times, also identified non-plagiarized material as problematic.

In conclusion, obfuscators have not attracted the attention of sufficient researchers. Most attempts have focused on a small number of obfuscators [1,11,12,14–16]. Others narrowed their research to specific types of obfuscators, such as plagiarism/cloning [6,13,17], network protocols [10], or JavaScript [1]. The rest summarized some of the obfuscation transformations with no mention of tools [14]. In addition, some works are quite old, so the tools discussed are out of the market today [6]. Accordingly, there is a lack of comprehensive studies to classify the existing obfuscators and their practical limitations.

3. Research Design

3.1. Goals and Research Questions

The main goal of this study is defined as follows: "The purpose of this study is to identify the characteristics and drawbacks of a set of tools that belong to different obfuscation transformations from the point of view of software practitioners in the context of industry".

In order to achieve the above goal, the following three research questions (RQs) are answered through this paper:

RQ1: How do existing obfuscators differ based on their analysis approach, supported programming languages, file formats, and licensing?
RQ2: What is the most widely used obfuscation transformation in existing tools?
RQ3: What are the drawbacks of these existing tools?

The rationale of these RQs is to provide an in-depth analysis of the state-of-the-art obfuscators and their weaknesses, as well as help software practitioners determine the tool that suits their requirements according to the results of the predefined framework. Additionally, the answer to the above RQs provides the researchers as well as tool vendors with new insights that could be exploited to enhance the quality of obfuscators and deobfuscators.

3.2. Predefined Framework

In our extensive review of existing tools, we identified a comprehensive set of features crucial for classifying obfuscation and deobfuscation tools, directly addressing our research questions (RQs). This section expands upon the predefined framework established through our survey, detailing the criteria and rationale behind the classification schema, as encapsulated in Table 2. To augment the utility and applicability of our framework, we incorporated an in-depth analysis of each feature, underscored by examples from the tools evaluated. This approach not only facilitates a nuanced understanding of the tool landscape but also enriches our framework with insights into emerging trends and capabilities within the domain. Furthermore, we explore the implications of these features for tool effectiveness, user accessibility, and their overall impact on software security practices, offering a forward-looking perspective on the evolution of obfuscation and deobfuscation methodologies.

Table 2. Description of our proposed predefined framework.

#	Feature	Description
1	Objective	Understanding the objectives is critical for evaluating any tool.
2	Open source	It is important to determine if the source code was released under an open-source license.
3	Analysis approach	According to the literature, there are two main approaches used by detection tools: static analysis, which checks the app's maliciousness without executing the app., and dynamic analysis, which examines the application during execution.
4	PLs	The PLs supported by the tool are important to consider.
5	Obfuscation transformation	There are four main categories as follows: formatting, renaming/modifying, code injection, and others including merging, splitting, flattening, dynamic dispatcher, etc.; each category has multiple subcategories shown in Table 1
6	Drawback	Limitations, drawbacks, and disadvantages from technical perspectives are considered.

3.3. Experiment Procedure

A total of 40 students enrolled in the final year of the Computer Science BSc program at a public university participated in our study, comprising 12 males and 28 females. They were organized into pairs, with each group tasked with selecting a tool to assess features and technical limitations. The procedure unfolded as follows: Initially, each group installed the chosen tool on their respective machines, encountering technical challenges during installation due to certain tool configurations, which are elaborated upon later. Consequently, this stage spanned a minimum of several days. Subsequently, the students utilized sample programs as the input for each tool, sourced either from programming languages supported by the tool or downloaded from the Internet, ensuring the validity of the source codes. Following this, each group provided their collective observations on the tool's performance, focusing on characteristics outlined within a predefined framework, including any identified drawbacks or limitations. Recognizing instances where students expressed personal opinions rather than strictly assessing the tool's limitations, the instructor implemented an additional stage to refine the observations (the final stage). This refinement process involved excluding the work of certain groups that failed to properly install the tool. Moreover, the instructor cross-checked data from select participants who

claimed insufficient time to complete all the tasks successfully. Consequently, a total of 20 tools underwent investigation, as depicted in Figure 2.

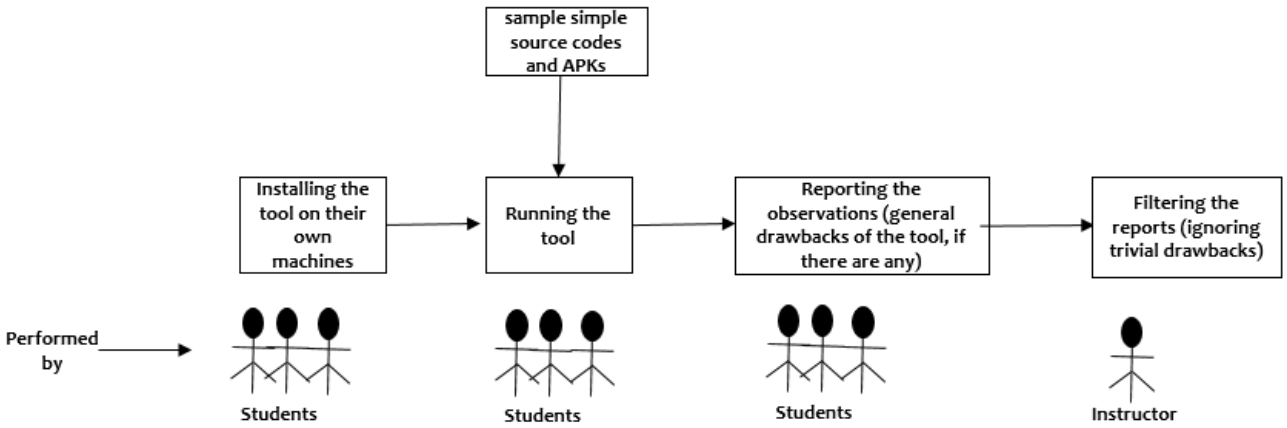


Figure 2. Methodology of conducting the experiment.

3.4. Material

We came across several obfuscation tools throughout our research journey. Some of those obfuscators were downloaded, installed, and run according to the procedure shown in Figure 2. Other tools, like Limon sandbox (https://cysinfo.com/limon-sandbox-for-analyzing-linux-malwares-2/ (accessed on 8 June 2024)), Stunnix (http://stunnix.com/prod/cxxo/ (accessed on 8 June 2024)) and Jasob (https://jasob.com/ (accessed on 8 June 2024)) were not taken into account since the students faced a problem in their configuration. The selected tools are listed in Table 3 with their URLs. In the next sections, we reference the tools by their corresponding Tool # as indicated in this table, just for the sake of simplification.

Table 3. Compilation of obfuscators and deobfuscators analyzed in this research.

Tool Number	Tool Name	URL/Source
T1	UPX	https://upx.github.io (accessed on 8 June 2024)
T2	DNADroid	https://github.com/amirhgharib/dnadroid (accessed on 8 June 2024)
T3	ProGuard	https://www.guardsquare.com/proguard (accessed on 8 June 2024)
T4	DexGuard	https://www.guardsquare.com/dexguard (accessed on 8 June 2024)
T5	Androguard	https://github.com/androguard/androguard (accessed on 8 June 2024)
T6	FSquaDRA	https://github.com/zyrikby/FSquaDRA (accessed on 8 June 2024)
T7	FlowDroid	https://github.com/secure-software-engineering/FlowDroid (accessed on 8 June 2024)
T8	DroidMOSS	https://github.com/MIchicho/Repackaged-APPs-detection/tree/master/DroidMOSS (accessed on 8 June 2024)
T9	ScanDal	[16] https://code.google.com/archive/p/andromaly (accessed on 8 June 2024)
T10	Andromaly	https://link.springer.com/article/10.1007/s10844-010-0148-x (accessed on 8 June 2024)
T11	NiCad	https://github.com/bumper-app/nicad (accessed on 8 June 2024)
T12	DroidEvolve	https://github.com/DroidEvolver/DroidEvolver (accessed on 8 June 2024)
T13	Mangle-It	https://mangle-it-c-source-code-obfuscator.updatestar.com (accessed on 8 June 2024)
T14	CodeMorph	http://www.sourceformat.com/code-obfuscator-cpp.htm (accessed on 8 June 2024)

Table 3. *Cont.*

Tool Number	Tool Name	URL/Source
T15	Sherlock-Sydney (The original of this tool came from two programs, sig and comp. The first generates the digital signatures and stores them in a file. The second can then be used to compare the signature files and report the similarities. Both programs are then combined into one program called Sherlock. It should be noted that two different tools have the same name: one is from Sydney, which is used here, and the other is from Warwick (http://warwick.ac.uk/iasgroup/software/sherlock (accessed on 8 June 2024))	https://github.com/diogocabral/sherlock (accessed on 8 June 2024)
T16	UglifyJS	https://github.com/mishoo/UglifyJS (accessed on 8 June 2024)
T17	SIM by Grune	https://dickgrune.com/Programs/similarity_tester (accessed on 8 June 2024)
T18	ReDex	https://fbredex.com (accessed on 8 June 2024)
T19	Java-deobfuscator	https://github.com/java-deobfuscator/deobfuscator (accessed on 8 June 2024)
T20	DroidAnalytics	https://github.com/zhengmin1989/DroidAnalytics (accessed on 8 June 2024)

4. Results: Analysis and Discussion

This section presents the evaluation results of the existing tools using the predefined criteria. After performing the experiment using each tool individually and recording their corresponding results, the final collation of the data was made to separate the objective characteristics from the subjective characteristics.

4.1. Tools Classification (Answering RQ1)

The tools for obfuscation and deobfuscation under examination, as detailed in Table 3, have been categorized based on Research Question 1 (RQ1), specifically in terms of their analysis approach, programming languages (PLs)/formats, and licensing. Figure 3 depicts the classification across all examined tools, where the key findings and primary observations can be summarized as follows:

4.1.1. Analysis Approach

Regarding the analysis approach, all obfuscators and deobfuscators under investigation primarily operate at the static level, with the exception of T10, which functions dynamically. Static analysis, dependent on signature matching for malware detection, often falls short in identifying malware that has undergone obfuscation or transformation. Conversely, T3, integrated within the official Android Studio IDE, provides rudimentary protection against static analysis. Its more advanced counterpart, T4, extends its capabilities to encompass both static and dynamic analyses, marking a crucial distinction from T3. This distinction is particularly significant as attackers of Android applications frequently employ a blend of both methods in their reverse engineering efforts. T10, as the sole analyzer utilizing dynamic analysis, leverages artificial intelligence (AI) algorithms to differentiate between malicious and benign applications by monitoring a range of indicators, including CPU usage, network traffic, active process counts, and battery consumption. In general, AI-based malware detection and hybrid approaches that combine static analysis and dynamic analysis may provide an effective way to classify malware, especially when applied

to malware gray-scale images. Some dynamic-based tools rely on static-based ones; for example, RepassDroid [18], a dynamic obfuscator, uses T7, a static tool, to extract features.

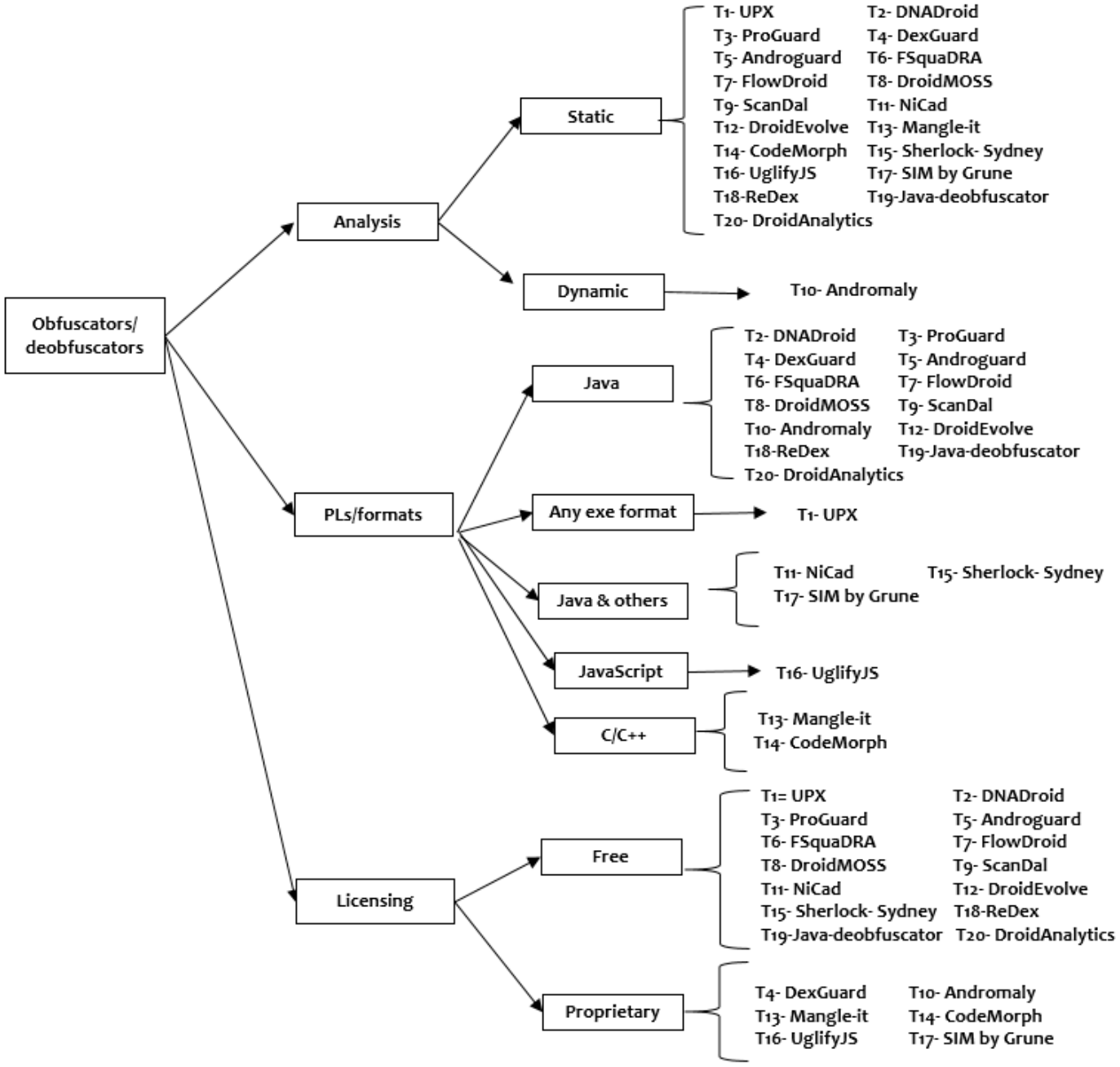


Figure 3. Classification of obfuscators and deobfuscators.

4.1.2. PLs and File Formats

Most obfuscators and deobfuscators (16 out of 20) work for Java programs or Java-based apps. This means Android and Java-based ecosystems are the target for most tools. The others work for C/C++, T13, and T14, i.e., for standalone applications or JavaScript, and T16, i.e., for WebApps. It is clear that few tools (T11, T15, and T17) support different programming paradigms, procedural PLs, and object-oriented PLs, including C, C++, Pascal, C#, Lisp, Java, Python, and Web Services Description Language, Modula-2, Miranda, 8086 assembler code, and normal language (i.e., texts). T1 has advantages compared with the others; it supports different executable formats, including Windows programs and DLLs, MAC apps, and Linux executables. Besides the PLs, T15, and T17 support normal

languages, and such tools cannot only detect similarity between source codes but also between any text document, and this cannot be useful for the academic programs that are planning to embed these tools in their curricula as mentioned in Section 1.

4.1.3. Licensing

The majority of the tools examined (14 out of 20) are available without the need for licensing and are freely accessible. However, the students encountered accessibility issues with a subset of the tools—specifically, T4, T10, T13, T14, T16, and T17—due to their proprietary nature. Consequently, they resorted to using more restrictive, temporary 30-day trial versions of these tools.

4.2. The Most Widely Used Obfuscation Transformations in Obfuscators and Deobfuscators (Answering RQ2)

Figure 4 presents a Venn Diagram illustrating the distribution of the analyzed obfuscators and deobfuscators according to the principal obfuscation transformations detailed in Table 1. Notably, the renaming transformation is employed by 11 of the 20 tools examined, making it the most prevalent. This is followed by source code formatting and code injection transformations, utilized by 6 and 5 of the 20 tools, respectively.

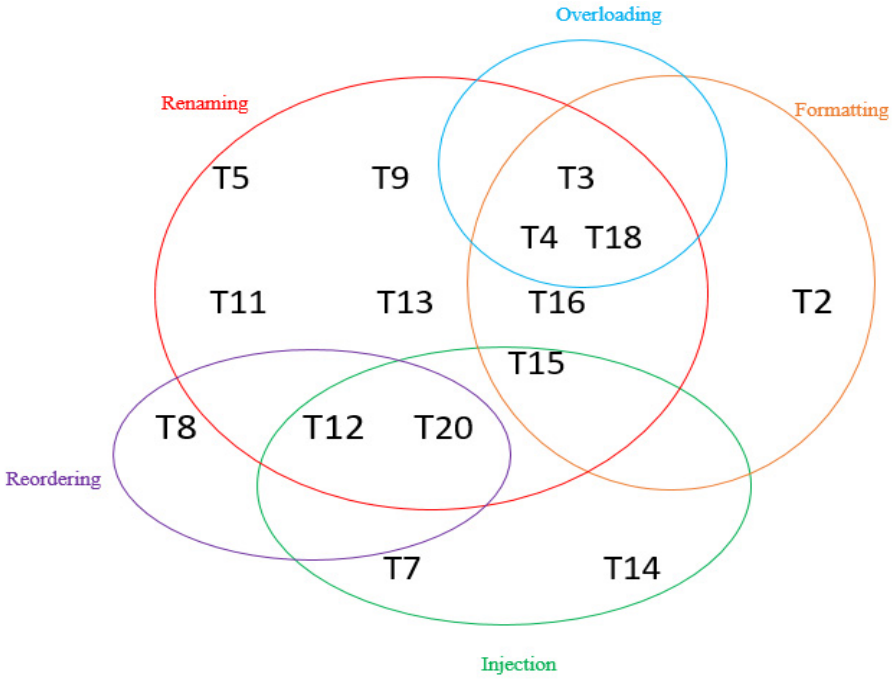


Figure 4. Allocation of analyzed tools based on employed obfuscation techniques.

Some tools use the 'other transformations' category (number 4 in Table 1), which includes splitting, merging, and encryption. Such transformations are used by T4, T12, T17, T18, T19, and T20. It is observed that obfuscated applications are more difficult to reverse with DexGuard (T4), which is the extended commercial version of ProGuard (T3). This comes from the various obfuscation transformations used by this tool; for example, identifier renaming, class repackaging, excessive overloading, the encryption of literal strings, encryption of classes, native libraries, and assets, Java reflection, removing debugging information, and splitting .dex file (see Figure 1). In fact, the deobfuscators that appear in Figure 4 (e.g., T12, T18, T20) do not use those transformations, but they work against them like in the case of DroidEvolve (T12), ReDex (T18), DroidAnalytics (T20). Accordingly, they can deobfuscate obfuscators for Java, including commercially available types, such as Allatori, DashO, and DexGuard (T4).

Some tools apply a single obfuscation transformation, such as T2, that uses formatting only. Although such transformation contains subcategories (as shown in Table 1), it still

focuses on a single mythology, i.e., removing lines or words from the source code. This can be insufficient for securing new applications working under the IoT paradigm because the source code of these applications—at least in the current simple IoT architecture—is vulnerable to tampering due to various factors, including the heterogeneous nature of the IoT devices and their limited resources, which are rapidly increasing [19].

We did not find a tool that offers all of the obfuscation transformations. However, T4 might achieve good quality in terms of its number of transformations; it applied renaming, overloading, formatting, and splitting transformations. This was followed by T3 and T18, which used renaming, overloading, and formatting transformations, and T12 and T20, which implemented renaming, reordering, and injection. Honestly, it is not necessary that all of the above obfuscation transformations are offered by the existing tools; if they are offered, it is not also necessary the tools are effective against vulnerability.

4.3. Drawbacks (Answering RQ3)

Table 4 summarizes the main technical limitations of the existing obfuscators and deobfuscators. From Table 4, limitations are discussed relating to (a) resource consumption, including computing power, CPU usage, and memory requirement, and (b) false positives that are unable to detect unknown malware types or which classified them as malware even when they were not, i.e., most of the malware remained undetected. However, these drawbacks are common in signature and Dalvik-based static analysis obfuscators and deobfuscators such as in T1, T2, T6, T7, T8, and T9. An additional observation pertains to the configuration of the tools, as outlined in Section 4. The experiment, primarily conducted by students, encountered certain limitations that hindered or even prevented the successful installation and operation of the tools on the students' own computers. These challenges were twofold as follows: firstly, there was the necessity for third-party libraries, as encountered with tools T2 and T6, and secondly, the absence of a graphical user interface (GUI) in tools T1, T15, and T17, which lacked any form of graphical interaction.

Table 4. Overview of key technical limitations identified in the analyzed tools.

Tool Number	Tool Name	Tool Drawbacks
T1	UPX	- Relies on signature-based detection, which struggles with unknown or custom-made packers, reducing its effectiveness in malware detection. - Lacks a graphical user interface (GUI), operating solely through command-line interactions. - Issues have been reported where programs, after compression by UPX, do not function correctly despite a successful compression ratio. This problem persists across both the free and commercial variants of similar tools, as noted by industry experts.
T2	DNADroid	- It utilizes a third-party library, such as dex2jar (https://github.com/pxb1988/dex2jar (accessed on 8 June 2024)), to transform the Dalvik bytecode into the Java bytecode, enabling the construction of a Program Dependence Graph, PDG. (PDG is a representation of the dependencies between different elements in a program, such as variables, statements, and expressions) for each method. - PDG is employed for comparisons to identify similarities between two applications. However, the reliance on PDG, which incorporates semantic program information, raises concerns about accuracy, particularly in capturing dynamic dependencies that manifest only during the execution of a program.
T3	ProGuard	- It only provides basic obfuscation features.

Table 4. *Cont.*

Tool Number	Tool Name	Tool Drawbacks
T4	DexGuard	- Although it is an extended version of T3, this tool only protects Android apps, while T3 is a generic optimizer for the Java bytecode (i.e., it can obfuscate desktop apps, embedded apps, and Android apps). - This tool does not offer any explanation of the differences among similar apps, i.e., failing to provide opportunities for further analysis.
T5	Androguard	- In terms of code-based similarity, T5 was compared less for quality with other tools, specifically with T4. - This tool does not offer any explanation of the differences among similar apps, i.e., failing to provide opportunities for further analysis.
T6	FSquaDRA	- It requires a third-party library to parse command line arguments, in particular jCommander (https://jcommander.org (accessed on 8 June 2024)).
T7	FlowDroid	- It suffers from computing power and memory requirements. Moreover, false positives may arise due to over-approximation.
T8	DroidMOSS	- It can only pinpoint known malware, so it may miss some repackaged applications. In other words, the limited signature database makes this tool unreliable for detecting zero-day malware apps. This limitation is common in signature-based tools that use hash algorithms. - The tool only considers the DEX code (see Figure 1) and can be bypassed by native code execution.
T9	ScanDal	- It only handles simple cases of reflection by analyzing string values. In other words, it cannot detect apps that use reflection for data leakage. - In addition, similar to Dalvik-based tools, T9 consumes a lot of resources and storage space since they are performed at the op-code level.
T10	Andromaly	- If a legitimate app invokes too many system calls, it may be classified as malicious (false positive). This is because this tool is designed to target a particular type of malware to the detriment of other types. - It misses quick actions, and the analysis model needs to be reviewed
T11	NiCad	- It has a quadratic time complexity with respect to the number of extracted code fragments for comparison. - Because T11 is parser-based and language-specific, adding a new language or granularity needs to be conducted to obtain a TXL (TXL is a special-purpose programming language used to create, manipulate, and prototype language descriptions, tools, and applications [20]) with a parser or extractor.
T12	DroidEvolve	- It cannot detect malware in the following cases: • If the malware can only be detected based on more complicated features, such as API call graphs and bytecode semantics; • If the API calls of malware are not visible in the static analysis, such as dynamically loaded malicious code and runtime malicious behaviors; • If the tool is vulnerable to certain malware and poisoning attacks, the attackers may deliberately craft the initialization dataset for this tool.
T13	Mangle-It	- Portability limitation: it only runs on Windows OS and works with C++ only. - The tool just modifies the code structure (i.e., no optimization).

Table 4. *Cont.*

Tool Number	Tool Name	Tool Drawbacks
T14	CodeMorph	- It is exclusive to C/C++ and Delphi and does not work on Android apps.
T15	Sherlock- Sydney	- Particular optimizations have been made only for Java. - No graphical user interface (GUI) is available apart from a command line; - Although the tool works well to find the similarity between essays handed in by humans, it is not well suited to finding that of email messages due to their continuous arrival, preventing the availability of a complete set to analyze. Introducing intermediate (sig) files could expedite the process by precomputing and storing signatures for older items, reducing the time required for comparing new email items with existing ones. However, without utilizing (sig) files, the tool would need to repeatedly read and compute signatures for all older files each time a comparison is necessary.
T16	UglifyJS	- It only works with JavaScript code, i.e., suitable for web developers.
T17	SIM by Grune	- No graphical user interface (GUI) is available apart from a command line. - A group of files as a submission is not considered.
T18	ReDex	- This tool cannot handle the advanced control-flow obfuscation directly, i.e., it does not remove global opaque variables. Accordingly, it regards global variables as non-opaque variables.
T19	Java-deobfuscator	- The tool fails in optimization due to the loss in converting .dex files to .jar files. - Although this tool supports Java, it is not a tool for Android apps. To work with Android apps, it requires a third-party library for that conversion, in particular, enjarify (https://github.com/storyyeller/enjarify (accessed on 8 June 2024)) or dex2jar (mentioned in T2).
T20	DroidAnalytics	- It classifies the apps as malware on the basis of classes mostly used by malware families. Consequently, the similarity metric used for the detection of repackaged malware does not provide a good result in terms of accuracy and false positives. This has made the tool classify legitimate apps as malware.

5. Recommendation for Future Work

As we mentioned at the end of Section 4.2, it is not necessary that all obfuscation transformations are applied by a single tool; if they are applied, it is also not necessary that this tool is efficient against attacks. However, the tools should cover a large selection of such transformations and not only the simple ones, such as variable renaming or comments on removal, because these transformations are used by many pieces of malicious code to avoid detection. Not only this but the tools are also expected to offer a new unconventional way to obfuscate, such as embedding code within an image so it cannot be read directly from the source code of the web page in the case of JavaScript, as occurred for the tool JavaScript2img (http://javascript2img.com.bitverzo.com (accessed on 8 June 2024)). Another unconventional way to obfuscate is writing all code using a small number of characters (six characters in tool JSFuck (https://jsfuck.com (accessed on 8 June 2024))); this would make programs extremely verbose and unreadable. It is worth mentioning that Java Script has not attracted much attention from sufficient tool developers compared with Java. One more open point for further research and development is that there is a need to develop obfuscators and deobfuscators to support non-Android-based apps, and the number of these apps is increasing in the market.

6. Limitations and Threats to Validity

Compared with related works, this study covered a large number of obfuscators and deobfuscators. However, still, this number (20 tools) remains small compared with the number of tools in the entire market. The reason was technical issues in some tools that the students faced when they tried to install or run the tool. Most of these issues stem from a complicated configuration of tools, including running them with a third-party library. In addition, trial versions of some commercial tools are held so that most of the features of the tool are not allowed. This point should be taken into account when someone plans to repeat this experiment or a similar one. All such difficulties should be considered when someone plans to repeat the same experiment in the next few years. Moreover, we can furnish precise duplicates of all materials utilized in our study to facilitate other researchers who may wish to replicate it.

Threats to the study's validity should also discuss the fact of using students for the evaluation. Herein, the limited representativeness of the sample is another threat, whereby individuals (students) are grouped after the fact rather than through a planned sampling approach based on experience levels. Ideally, a follow-up investigation involving professionals utilizing their own analytical tools within a more authentic setting, such as obfuscation analysis, would be valuable. However, the scarcity of resources for obfuscation analysis in the professional sphere poses a significant obstacle to pursuing such research endeavors. The last limitation is related to the predefined framework that was the guide to conducting the experiment; this experiment lacks analysis of some key issues, such as the crucial indicators of implementation costs in obfuscation and resistance to malicious analysis.

7. Conclusions

Android apps are regularly plagiarized or repackaged, and program obfuscation is a recommended security technique against these practices. Researchers proposed numerous obfuscation transformations supported by tools (obfuscators and deobfuscators). However, the existence of several tools imposes headaches on the Android community to decide upon the appropriate one based on their requirements. Furthermore, comprehensive studies on the available obfuscation tools have not been paid much attention to. This paper aims to conduct an empirical study to systematically characterize these tools that belong to different categories of obfuscation transformations. Accordingly, we propose a framework based on the process of surveying existing works. The framework is used to systematically classify twenty tools according to their features, including analysis type, programming languages supported, licensing, applied obfuscation transformations, and general technical drawbacks. The findings show that very few tools work at a dynamic level, and most tools also work for Java or Java-based ecosystems (e.g., Android), i.e., other languages were not supported enough by the existing tools. In addition, renaming transformation was applied by more than 50% of the existing tools, followed by formatting transformations and code injection. Other transformations were not attracted by researchers, e.g., splitting, merging, flattening, and parallelization. This paper shows the technical drawbacks of each tool in detail. Some of these drawbacks are common in static-based analyzers; others affect negatively conducting experiments, e.g., those related to using a third-party library and the lack of GUI. This paper discusses the consequences of such drawbacks and provides recommendations related to improving obfuscation. For example, the tools should not only implement the simple categories of obfuscation transformations, such as identifier renaming but also cover a large selection of such transformations shown in Table 1 because the simple ones are used by many malicious codes to avoid detection. Another recommendation is to offer a new unconventional way to obfuscate, such as embedding codes within an image so it cannot be read directly from the source code of the web page, in the case of JavaScript. In general, this study would assist three types of practitioners: (a) software developers to understand the characteristics and the capability of the studied obfuscators and deobfuscators, (b) researchers to address the identified

drawbacks and enhance the overall health of Android ecosystems, and (c) managers of the computer science and cybersecurity academic programs to embed suitable obfuscation tools in their curricula. This paper also discusses potential threats to the validity of the study and proposes unresolved areas for future research.

Author Contributions: Conceptualization, S.A.E.; methodology, S.A.E.; software, S.A.E.; validation, S.A.E. and A.A.D.; formal analysis, S.A.E. and A.A.D.; investigation, S.A.E. and A.A.D.; resources, S.A.E. and A.A.D.; data curation, S.A.E. and A.A.D.; writing—original draft preparation, S.A.E.; writing—S.A.E.; visualization, S.A.E.; supervision, S.A.E.; project administration, S.A.E.; funding acquisition, S.A.E. All authors have read and agreed to the published version of the manuscript.

Funding: The authors extend their appreciation to the Deanship of Scientific Research at Northern Border University, Arar, KSA for funding this research work through the project number NBU-FFR-2024-1564-01.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Rauti, S.; Leppänen, V. A comparison of online JavaScript obfuscators. In Proceedings of the 2018 International Conference on Software Security and Assurance (ICSSA), Seoul, Republic of Korea, 26–27 July 2018; pp. 7–12. [CrossRef]
2. Ebad, S.A.; Darem, A.A.; Abawajy, J.H. Measuring software obfuscation quality—A systematic literature review. *IEEE Access* **2021**, *9*, 99024–99038. [CrossRef]
3. Sherif, A. Mobile Operating Systems—Statistics & Facts, 10 January 2024. Available online: https://www.statista.com/topics/3778/mobile-operating-systems/#topicOverview (accessed on 8 June 2024).
4. Wang, H.; Li, H.; Li, L.; Guo, Y.; Xu, G. Why are android apps removed from google play? A large-scale empirical study. In Proceedings of the 15th International Conference on Mining Software Repositories, Gothenburg, Sweden, 27 May–3 June 2018; pp. 231–242.
5. Collberg, C.; Thomborson, C.; Low, D. *A Taxonomy of Obfuscating Transformations*; Technical Report; Department Computer Science, University of Auckland: Auckland, New Zealand, 1997; Available online: https://researchspace.auckland.ac.nz/bitstream/handle/2292/3491/TR148.pdf (accessed on 8 June 2024).
6. Roy, C.K.; Cordy, J.R.; Koschke, R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.* **2009**, *74*, 470–495. [CrossRef]
7. Madou, M.; Anckaert, B.; DeBus, B.; DeBosschere, K. On the effectiveness of source code transformations for binary obfuscation. In Proceedings of the International Conference on Software Engineering Research and Practice (SERP 2006), Las Vegas, NV, USA, 26–29 June 2006; CSREA Press: Las Vegas, NV, USA, 2006; pp. 527–533.
8. Li, L.; Bissyandé, T.F.; Wang, H.Y.; Klein, J. On identifying and explaining similarities in Android apps. *J. Comput. Sci. Technol.* **2019**, *34*, 437–455. [CrossRef]
9. Ebad, S.A. What topics should or should not be included in software security education-qualitative content analysis. *Comput. Appl. Eng. Educ.* **2022**, *30*, 1753–1773. [CrossRef]
10. Duchêne, J.; Le Guernic, C.; Alata ENicomette, V.; Kaâniche, M. State of the art of network protocol reverse engineering tools. *J. Comput. Virol. Hacking Tech.* **2018**, *14*, 53–68. [CrossRef]
11. Wermke, D.; Huaman, N.; Acar, Y.; Reaves, B.; Traynor, P.; Fahl, S. A large scale investigation of obfuscation use in google play. In Proceedings of the 34th Annual Computer Security Applications Conference (2018), San Juan, PR, USA, 3–7 December 2018; pp. 222–235.
12. Martins, V.T.; Fonte, D.; Henriques, P.R.; da Cruz, D. Plagiarism detection: A tool survey and comparison. In Proceedings of the 3rd Symposium on Languages, Applications and Technologies, Open Access Series in Informatics (OASIcs), Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Bragança, Portugal, 19–20 June 2014; Volume 38, pp. 143–158. [CrossRef]
13. Svajlenko, J.; Roy, C.K. Survey on the evaluation of clone detection performance and benchmarking. *arXiv* **2020**, arXiv:2006.15682. [CrossRef]
14. Amro, B. Malware detection techniques for mobile devices. *Int. J. Mob. Netw. Commun. Telemat.* **2017**, *7*, 1–10. [CrossRef]
15. Tchakounté, F.; Ngassi, R.C.N.; Kamla, V.C.; Udagepola, K.P. LimonDroid: A system coupling three signature-based schemes for profiling Android malware. *Iran J. Comput. Sci.* **2021**, *4*, 95–114. [CrossRef]
16. You, G.; Kim, G.; Cho, S.-j.; Han, H. A comparative study on optimization, obfuscation, and deobfuscation tools in Android. *J. Internet Serv. Inf. Secur.* **2021**, *11*, 2–15.
17. Foltýnek, T.; Dlabolová, D.; Anohina-Naumeca, A.; Raz, S.; Kravjar, J.; Kamzola, L.; Guerrero-Dib, J.; Çelik, Ö.; Weber-Wulff, D. Testing of support tools for plagiarism detection. *Int. J. Educ. Technol. High. Educ.* **2020**, *17*, 46. [CrossRef]
18. Xie, N.; Zeng, F.; Qin, X.; Zhang, Y.; Zhou, M.; Lv, C. RepassDroid: Automatic detection of Android malware based on essential permissions and semantic features of sensitive APIs. In Proceedings of the International Symposium on Theoretical Aspects of Software Engineering, TASE, Guangzhou, China, 29–31 August 2018; pp. 52–59. [CrossRef]

19. Ebad, S.A. Quantifying IoT security parameters: An assessment framework. *IEEE Access* **2023**, *11*, 101087–101097. [CrossRef]
20. Cordy, J.R. TXL-a language for programming language tools and applications. *Electron. Notes Theor. Comput. Sci.* **2004**, *110*, 3–31. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.