symmetry

MDPI

Article

The Malware Detection Approach in the Design of Mobile Applications

Doaa Aboshady [1,*], Naglaa Ghannam [2], Eman Elsayed [2,3] and Lamiaa Diab [2]

[1] Department of Mathematics, Faculty of Science, Tanta University, Tanta 31511, Egypt
[2] Department of Mathematics, Faculty of Science, Al-Azhar University (Girls Branch), Cairo 11884, Egypt; naglaasaeed@azhar.edu.eg (N.G.); emankaran10@azhar.edu.eg (E.E.); lamiadeiab@azhar.edu.eg (L.D.)
[3] School of Computer Science, Canadian International College (CIC), Cairo 11835, Egypt
* Correspondence: doaa_aboshady@science.tanta.edu.eg; Tel.: +20-1119799933

Abstract: Background: security has become a major concern for smartphone users in line with the increasing use of mobile applications, which can be downloaded from unofficial sources. These applications make users vulnerable to penetration and viruses. Malicious software (malware) is unwanted software that is frequently used by cybercriminals to launch cyber-attacks. Therefore, the motive of the research was to detect malware early before infection by discovering it at the application-design level and not at the code level, where the virus will have already damaged the system. Methods: in this article, we proposed a malware detection method at the design level based on reverse engineering, the unified modeling language (UML) environment, and the web ontology language (OWL). The proposed method detected "Data_Send_Trojan" malware by designing a UML model that simulated the structure of the malware. Then, by generating the ontology of the model, and using RDF query language (SPARQL) to create certain queries, the malware was correctly detected. In addition, we proposed a new classification of malware that was suitable for design detection. Results: the proposed method detected Trojan malware that appeared 552 times in a sample of 600 infected android application packages (APK). The experimental results showed a good performance in detecting malware at the design level with precision and recall of 92% and 91%, respectively. As the dataset increased, the accuracy of detection increased significantly, which made this methodology promising.

Keywords: malware detection; mobile applications; ontology; software quality; UML; revers engineering

Citation: Aboshady, D.; Ghannam, N.; Elsayed, E.; Diab, L. The Malware Detection Approach in the Design of Mobile Applications. *Symmetry* **2022**, *14*, 839. https://doi.org/10.3390/sym14050839

Academic Editor: Mihai Postolache

Received: 7 March 2022
Accepted: 14 April 2022
Published: 19 April 2022

1. Introduction

Producing secure and high-quality software remains an ongoing research challenge. Software systems must fulfill quality characteristics such as reliability, usability, and maintainability. Over the last few years, it was proven that design patterns play a vital role in software engineering, IoT, security, mobile applications, and many other fields of computer science [1–4]. A good design pattern produces a perfect software design [1,5]. Most of the time, software engineers reuse existing design patterns for developing software systems and for solving similar issues such as errors, high costs, and high time consumption [2,6]. There are many design patterns on the internet for reusing purposes. The use of these patterns containing undesirable elements will result in poor quality and poor safety. Both anti-patterns and malware have similar bad factors that cause negative impacts on software quality [5,7]. Many studies have assessed the negative impact of anti-patterns on change-proneness, comprehension, reliability, fault proneness, security, performance, and energy efficiency [8–10]. At the same time, many empirical studies have assessed the negative impact of malware on performance [11,12], reliability [13], energy consumption [14], and other quality elements [7]. We noted that most of the research detected malware at the source

code level, which is considered late [15–17]. The proposed approach detected malware at the design level, which is a new detection method for malware. In addition, the design level is a phase before the coding phase, which shows the model of the application using any modeling tool; hence, this will avoid time-consuming use of patterns of applications from the Internet as new versions containing malware. The proposed approach was a detection method for malware before coding and installation.

The contribution of the proposed approach is a new detection method at the design level for malware. According to the joint effects of anti-patterns and malware, the proposed approach studied the probability of malware detection at the design level as anti-pattern detection.

Motivated by the research mentioned above, the major contributions of this paper were fivefold:

- A method for reversing the UML class model of malware using the Modelio modeling tool was presented.
- A suitable classification method for malware for design detection was presented.
- Existing solutions for anti-pattern detection were investigated to see if they could detect malware or not.
- A semantic method for detecting malware at the design level was presented.
- The evaluation of the proposed method in a dataset of 600 mobile applications was described.

The paper is structured as follows. To begin we present the related works. Next, we present the basic definition of both anti-patterns and malware. Then, the reverse engineering concept that was used in the proposed research is illustrated. After that, we describe the details of the proposed methodology, and the results and discussion are presented. Finally, the conclusions and future work are presented.

2. Related Work

Many previous studies have proven the negative impact of both malware and anti-patterns on software and its quality. Malware and anti-pattern detection have always been an active area of research in recent years. Several techniques and methods have been suggested to counter and reduce the growing amount and sophistication of malware and anti-patterns. Machine and deep learning techniques have contributed significantly to the literature for malware detection, as well as anti-pattern detection. We will refer to the latest of these approaches and learning techniques related to this research.

Malware detection techniques range from early day signature-based detection to machine and deep learning techniques [11,18,19]. There are distinct basic malware analysis techniques. In [20–24], the authors proposed a classification of analysis of malware to static and dynamic and a hybrid of both static and dynamic analyses. This analysis was an essential step in the malware detection process as it was a way of knowing how malware performs its function, how to identify it, and how to defeat it. According to [25], research has shown that the analysis was based on static PE file features of malware samples and observed that linear SVM models could be useful in detecting the evolution of malware. After that, the authors of [26] expanded on and improved upon the work in [25] in several ways. Recently, machine learning and deep learning methods (e.g., support vector machines (SVM), decision trees (DT)) have been used to detect and classify unknown samples for malware families due to their scalability, rapidity, and flexibility. In addition, machine learning and data mining techniques are combined with present detection techniques in order to facilitate the process of detection [27]. The research in [28] proposed a malware detection method called MalNet that learnt features automatically from raw data. It learned from grayscale images and opcode sequences extracted from malware files by using two deep neural networks: CNN and LSTM. The results showed that MalNet achieved 99.88% validation accuracy for malware detection. In [29], the authors proposed a hybrid model based on a deep autoencoder (DAE) and convolutional neural network (CNN) to improve the accuracy of malware detection compared with traditional machine learning methods.

In addition [30] provided a malware detection model using a deep convolutional neural network (CNN) in a metamorphic malware environment.

On the other hand, several studies proposed the detection of designed anti-patterns. The previous work, [31], introduced 40 types of design anti-patterns that formed the basis of design anti-pattern detection approaches [32–35]. The approach in [36] presented the ONTOPYTHO technique to detect smells and anti-patterns on the design of OWL Ontologies based on a metric method via the semantic web query language, SPARQL, and Python programming language. In [37], the authors focused on detecting mobile applications' anti-patterns and they proposed a method using reverse engineering and a UML modeling environment. This research presented a comparative study on nine UML tools and concluded that the Modelio UML modeler was a suitable tool for the detection process.

In addition, [1] introduced a general method that supported semantic and structural anti-pattern detection at the design level to automatically detect anti-patterns by using Modelio, OLED, and Protégé in a specific order to obtain positive results.

According to the symmetry between malware and anti-patterns, and the ability to detect design anti-patterns, the proposed research aimed to detect malware at the design level.

3. Background

3.1. Design Patterns and Malware

According to [38], design patterns are defined as solutions that developers reused for solving repeated problems in software systems for improving reusability and quality. Every pattern has its design and the features of the anti-pattern that is resolved. Anti-patterns exist in various levels in software development, such as design, coding, architecture, community, organization, environment, collaboration, etc. Anti-pattern examples can be a bad practice, a wrong reaction to a combination of events, a failure to predict, understand, or control a project factor, etc. Malware is defined as the class of software that may be called viruses, worms, and Trojans [39]. Malware is specifically designed to destroy, steal data, hosts or networks, or generally perform other "bad" or illegal operations on data, hosts, or networks. We contemplated the question of dealing with malware as an anti-pattern and creating a base for detecting it at the design level. While, at the same time, treating every anti-virus as a design pattern for detecting a certain malware and describing them using any modeling tool. This was a starting point for collecting all malware, its design features, and the used anti-virus. As a result, we could create a semantic catalog of malware that allowed developers to detect the existence of malware at the design level rather than later at the code level. First, we needed to answer the research questions.

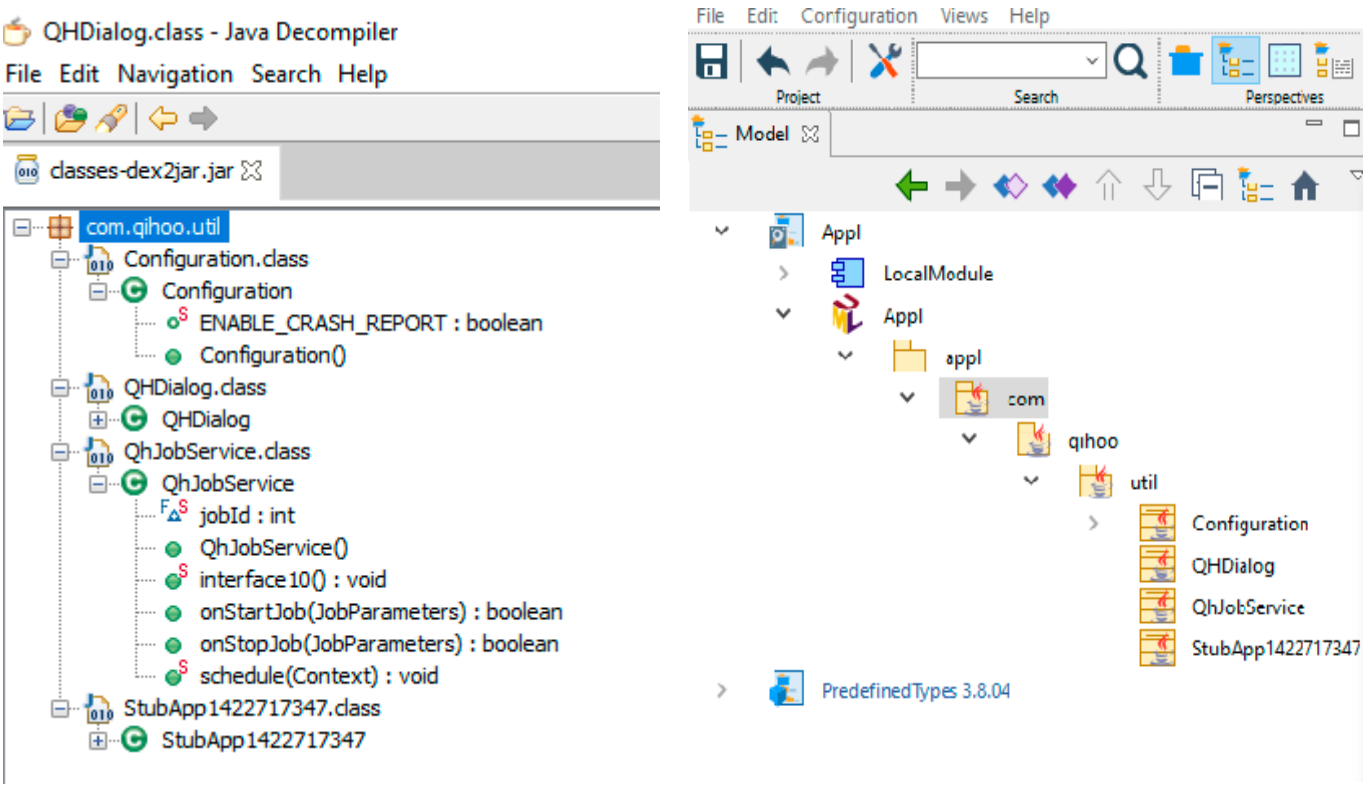
3.2. Reverse Engineering

The idea of this research was based on reverse engineering. According to the structure of android applications, we needed to reverse the source code in order to generate the design of it. First, we extracted the zip files of the APK that were in the JAR format and directly dealt with the Dex file, which was decompiled to generate java files. For the de-compilation, we used the (Android Decompiler-master) to obtain the Dex2jar files. Then, we used (JD-GUI-1.6.6) to obtain java sources. Using Modelio 3.6, we generated the class diagram model of the application. According to the results in [1], Modelio was the suitable UML modeler for reversing the class diagram and for detecting the anti-patterns. This was performed for un-infected mobile applications, but the question here was, could we do this for infected applications?

3.3. Reverse the Infected Applications

According to [1,37] the class diagram model of mobile applications was generated after reversing the java code. It was proven that the applications had many anti-patterns. According to this, the research reversed a sample of 42 APK in the dataset. Using the

mentioned tools, we reversed the class diagram model of the applications, as shown in Figure 1.



(**a**) (**b**)

Figure 1. The reversed java classes of APK: (**a**) using Java Decompiler; (**b**) using Modileio software.

This research also studied the reversal of thirteen infected programs, as in Table 1. The research study included the identification of malware across different types, different sizes, and different languages (C++, Java, Golang). According to this, we could reverse the infected software.

Table 1. The identification of the malware.

Name	Type	Size	Effects (Threat Type)	Platform	File Type (s)	Source Language
Sasser worm	Internet worm	15,872 bytes	Operating system and system performance	MS Windows	.exe	C++
THANATOS virus	Multi-vector worm	50 KB	Memory and files locker	MS Windows	.exe, .pif, .scr	C++
ExfilDocs virus	Computer virus	2.17 KB	Memory	MS Windows	.exe	Golang
Outlook Exfil virus	Computer virus	6 KB	Memory	MS Windows	.exe	Golang
Screen Shotter virus	Computer virus	1.72 KB	Memory	MS Windows	.exe	Golang

Table 1. *Cont.*

Name	Type	Size	Effects (Threat Type)	Platform	File Type (s)	Source Language
Dropper virus	Computer virus	1.77 KB	System performance and memory	MS Windows	.exe	Golang
Lion worm	Internet worm	860,160 bytes	System performance	Linux	.tgz, .sh, ELF	Shell Script
CIH virus	Computer virus	1 KB	System BIOS and drives	Windows x9	.exe	Assembly
Elk Cloner virus	Boot sector	-	Floppy disk drive and memory	Apple DOS 3.3 OS	Apple II	Assembly
Beanhive virus	File virus	7890 (.cab), 10,204 (.jar)	Java apps and programs	Java Runtime Environment	.cab, .jar, .class	Java
Strangebrew	File virus	3894 bytes	Java apps and programs	Java Runtime Environment	.class	Java
NetSky worm	Mail worm	29,568 bytes	Memory and emails	W32	.tmp	C++
Mimail worm	Mass mailer worm	-	Emails	MS Windows	.htm, .exe, .zip	C++

3.4. Malware Identification

From the sample in Table 1, we noted that there were many different types of malware. Therefore, as a start, we reversed all the types and were ready for detection. However, we should note that only infected applications could be checked at the design level. This was because we controlled the software and we could check it before installing it. We knew that malware, such as worms, infected the system through spam emails, which were not controlled by the user. Therefore, we could not detect all malware types at the design level.

4. Proposed Methodology

The main purpose of this research was to study the detection of malware at the design level. To do so, we needed to classify malware according to the way it infected systems. This helped us to determine the type of malware that the proposed methodology could detect at the design level.

4.1. Malware Classification

The initial step in the proposed methodology was malware classification. In general, malware can spread like a virus, a worm, or a Trojan. Every type has its infection method. Malware may infect your device by a visit to a hacked website, opening spam emails, downloading infected files or applications, using infected discs, or pressing on any untrusted advertisements; these are the malware sources. We tried to detect malware in the downloaded applications before installing them. The proposed research examined the code reversed from the APK by generating a class diagram of it.

4.2. Proposed Detection Method

In this section, we propose the detection method for detecting malware at the design level. The pseudo code of the proposed method is presented in Algorithm 1.

Algorithm 1. Malware detection algorithm.

Input: APK of mobile applications.
Output: a list of detected malwares

```
Convert classes.dex to classes.de2jar;
Reverse the java code to class diagram model;
run Modelio checker;
if the result is ≥1;
   print "malware detected"
else print "not detected";
convert the model to OWL Ontology;
run the reasoner;
If the result ≥1;
    Print "malware detected in ontology";
else run the detection method algorithm;
      print "Detected Malwares";
else print (" ");
   End.
```

In addition to reversing the UML class diagram model, we generated the OWL Ontology of the model using the Ontology editor. The model could then be checked for anti-patterns and malware for improving quality and security, as proposed in Figure 2.

4.3. Case Study in "Data_Send_Trojan"

To explain the proposed method, we presented a snapshot of it in the case study of "Data_Send_Trojan". Data_Send_Trojan is a kind of Trojan virus that always retrieves the important information of the users. The information most of the time includes credit card information, email addresses, passwords, instant messaging contact lists, log files, and so on. It is a part of the Trojan banker. For implementing the proposed method, we used the UML modeler and Modelio for presenting the class diagram model of the application. In addition, we examined the ability of Modelio to detect malware at the design level. Next, by converting the model to the OWL Ontology model and running SPARQL queries, the detection was performed. Figure 3 presents the UML class diagram model for posing as Data_Send_Trojan. The model contained three classes: the first class presented the user information, which was the attributes of the class; the second class presented the class of the virus containing the operations that would obtain the user information; finally, the third class presented the hacker that would receive the user information from the virus class.

The main components of an OWL Ontology are classes, datatype properties that present the attributes in UML models, and object properties, which present the operations. Every object property has a rang and a domain. The rang presents the class holding the operation, while the domain presents the class of the values. By converting the UML model of the virus to OWL Ontology, we obtained the ontology model in Figure 4.

Bank:Bank is the root class of the ontology. It had three subclasses, which were Bank:Data_send_Trojan, Bank:Hacker, and Bank:Users. There were two object properties, the first was (Bank:getUserInformation) and the second was (Bank:receiveUserInformation).

Bank:Data_send_Trojan—Bank:getUserInformation (Domain > Range)—–> Bank:Users

Bank:Hacker—Bank:receiveUserInformation (Domain > Range)—–> Bank:Data_send_Trojan

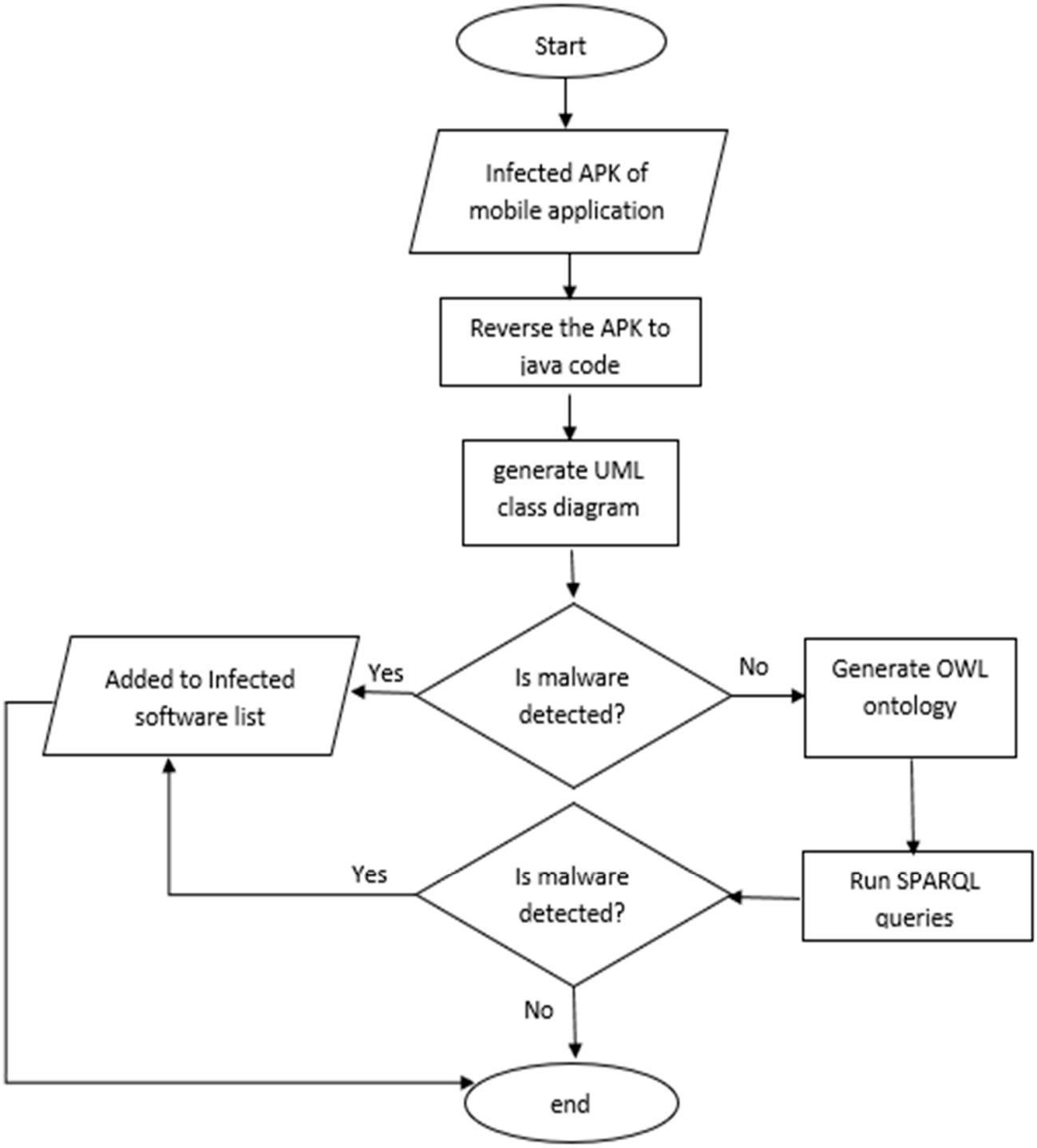


Figure 2. The Proposed methodology.

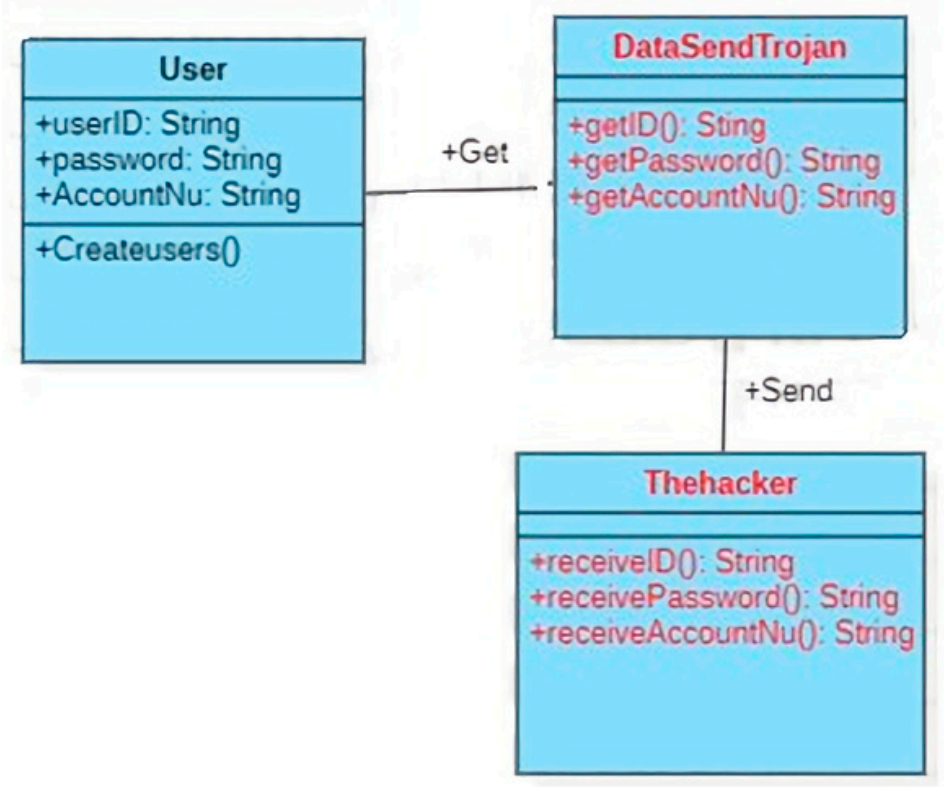


Figure 3. UML class diagram model for posing as Data_Send_Trojan.

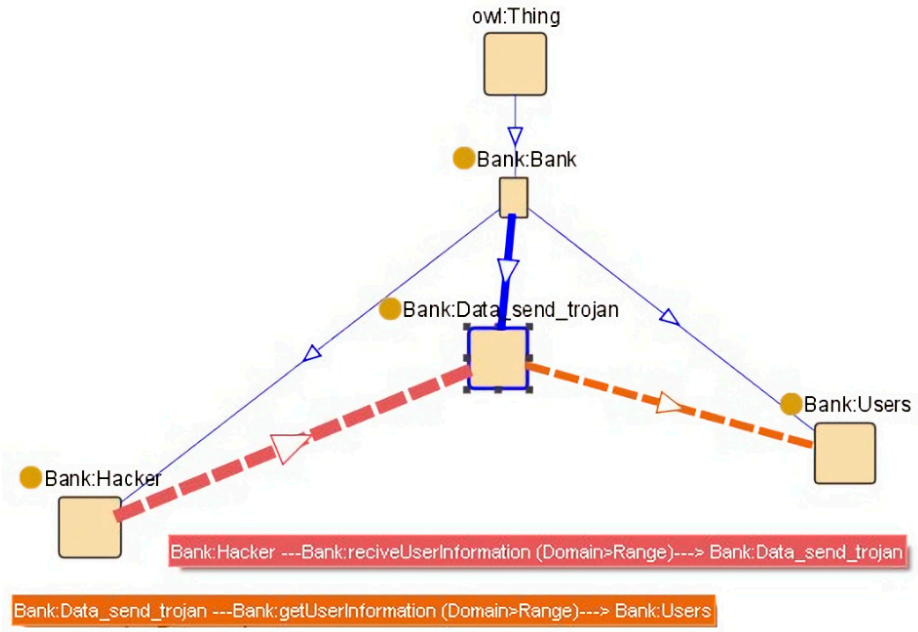


Figure 4. OWL Ontology for posing as Data_Send_Trojan.

The range of the (getUserInformation) object property was the (Data_Send _Trojan) class, which held the object property. While the domain of it was the (Users) class which had user information as datatype properties. The range of the (receiveUserInformation) object property was the (Hacker) class, which held the object property. The domain was the class (Data_send _Trojan), which, in this case, presented the class containing user information.

The challenge was getting all user information and sending it to the hacker class. In other words, both the object properties had to contain the same values and the same values must be the user information.

The next was the SPARQL queries for retrieving the user information and the values of both object properties.

Q1. SPARQL query for retrieving user information from the (Users) class.

```
SELECT ?Users ?AccountNumber ?Password ?UserID
WHERE {?Users rdf:type Bank:Users.
?Users Bank:AccountNu ?AccountNumber.
?Users Bank:Password ?Password.
?Users Bank:UserID ?UserID.
}
```

The result of Q1 is shown in Figure 5.

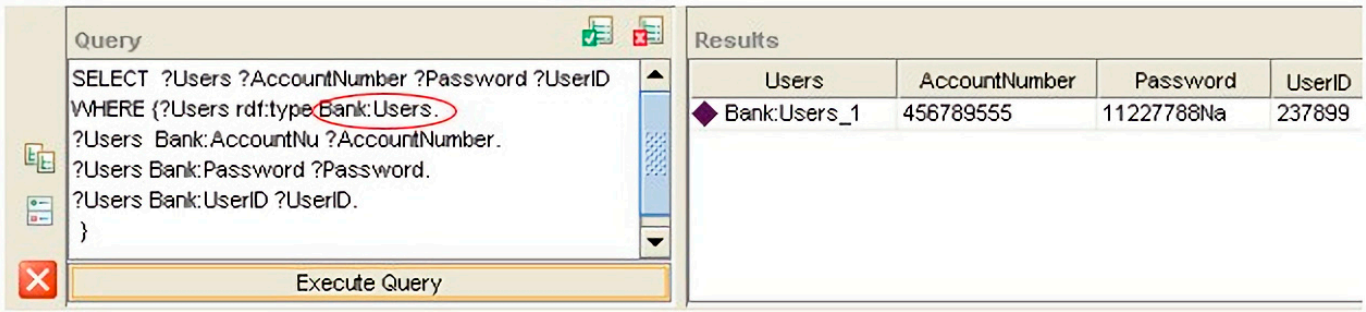


Figure 5. Result of Q1.

Q2. SPARQL query for retrieving (getUserInformation) object property of (Data_send_Trojan) class.

```
SELECT ?Users ?Information ?Values
WHERE {?x a owl:ObjectProperty.
?x ?n Bank:Data_send_Trojan.
?y ?x ?z.
?a a owl:ObjectProperty.
?z ?a ?Users.
?Information a owl:DatatypeProperty.
?Users ?Information ?Values.
}
```

The result of Q2 is shown in Figure 6.

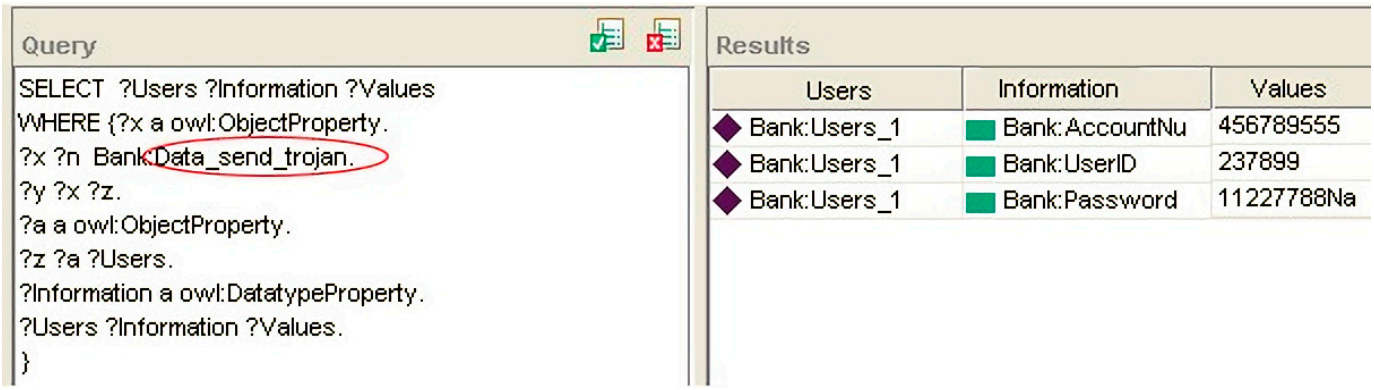


Figure 6. Result of Q2.

Q3. SPARQL query for retrieving (receiveUserInformation) object property of (Hacker) class.

```
SELECT ?RecivedInformation
WHERE {?x a owl:ObjectProperty.
?x ?n Bank:Hacker.
?y ?x ?z.
?a a owl:ObjectProperty.
?z ?a ?RecivedInformation.
}
```

The result of Q3 is shown in Figure 7.

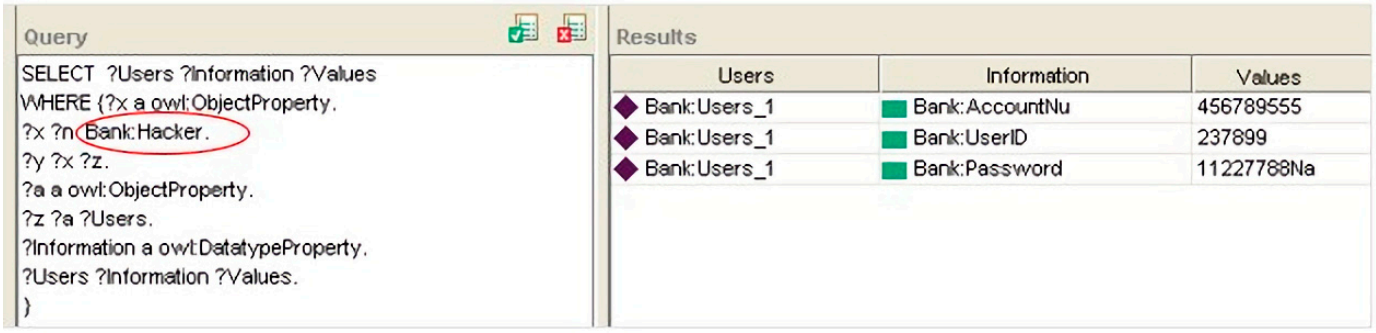


Figure 7. Result of Q3.

We noted from the results in Figures 5–7 that the values of the object properties of Data_Send_Trojan and the hacker classes were the same user information.

5. Experiments and Results

5.1. Evaluation Measures

To evaluate the detection performance successfully, it was necessary to identify appropriate performance metrics. The following five measures were employed to evaluate the proposed model performance.

$$\text{True Positive Rate(TPR)} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{1}$$

$$\text{False Positive Rate(FPR)} = \frac{\text{FP}}{\text{TN} + \text{FP}} \tag{2}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3}$$

$$\text{Recall} = \text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{4}$$

$$\text{F} - \text{measure} = \frac{2 * \text{Recall} * \text{Precision}}{\text{Recall} + \text{Precision}} \tag{5}$$

where,

TP (true positive): the number of correctly identified benign mobile applications.
FP (false positive): the number of incorrectly identified mobile malware applications.
FN (false negative): the number of incorrectly identified benign mobile applications.
TN (true negative): the number of correctly identified mobile malware applications.

Precision returned the rate of relevant results rather than irrelevant results. Recall was the sensitivity for the most relevant result. The F-Measure was the value that estimated the entire system performance by combining precision and recall into a single number. The maximum value of 1.000 indicated the best result.

5.2. Result and Discussion

In this section, we assessed how well the proposed approach could predict the security of android applications. For evaluating the proposed approach, we applied it to a sample of 600 malware of the CICMalDroid 2020 dataset presented by the Canadian Institute for Cybersecurity. The CICMalDroid 2020 dataset has 11,598 android samples with five distinct categories. This dataset can be found in [40].

The (Data_Send_Trojan) malware was detected 552 times by using the general SPARQL query Q4.

Q4. SPARQL query for detecting the "Data_Send_Trojan" generally.

```
SELECT ?h
WHERE {?d a owl:ObjectProperty.
?d rdfs:domain ?gg.
?r rdfs:range ?gg.
?y ?d ?v.
?a a owl:Class.
?v rdf:type ?a}
```

We applied the proposed approach in a sample of 600 Trojan malware using certain SPARQL queries. We compared the proposed approach to other approaches in the section of related works [18,19,23,27,29,41] as in Table 2. The related references detected malware using different methods. References [19,27,29] used a deep convolutional neural network and two deep learning models, DexCNN and DexCRNN, while reference [18] used ensemble learning and big data. Reference [23] used social network properties and community detection while reference [41] used meta-learning. The proposed approach used a semantic environment for detecting malware. The comparison criteria included the method of detection, the used dataset, the accuracy, and the detection level. It is worth noting that there were some limitations that were encountered when applying the proposed method at the preprocessing and data converting stages; this was because it was performed semi-manually, one by one, separately and took a long time. This led to the experiment of the proposed method being on a relatively small number of samples compared to other research that detected malware at the code level.

Table 2. Comparison among malware detection methods.

Reference	Method	Data Set	Accuracy	Detection Level
[27]	Deep neural networks CNN and LSTM	42,386 samples	99.88%	Image and Opcode sequence
[18]	Ensemble learning and big data	198,350 Windows files	99.5%	Code level
[29]	Deep autoencoder (DAE) and convolutional neural network (CNN).	10,000 benign apps and 13,000 malicious apps	(99.80–99.82%)	Code level
[19]	Two deep learning models, DexCNN and DexCRNN.	16 k	93.4% and 95.8%, respectively	Code level
[23]	Social network properties and community detection	60,000 files	97%	Code level
[41]	meta-learning	414,291 samples	(94.5–99.9%)	Code level
Proposed	Semantic	600 samples	92% and 91%, respectively	Design level

From Table 3, the proposed model yielded 82.8% TPR, 17.2% FPR, 92% precision, 91% recall, and 91.4 f-measure, respectively. We considered this result as a good starting point for detecting malware at the design level. Additionally, Figure 8 illustrates our experimental receiver operating characteristic (ROC) curve plot of the proposed model. The ROC curve is a plot of performance measure that was determined by the true-positive rate versus the false. The *X*-axis represents the true-positive rate (TPR) and *Y*-axis represents the false-positive rate (FPR). The area below the ROC curve, known as AUC, was widely utilized to evaluate the performance of the malware detection models. A data point in the upper left corner and the higher AUC value corresponded to optimal and high performance. As can be seen from Figure 8, the proposed method yielded 91% of the AUC score.

Table 3. Performance of the proposed method.

	TPR (%)	FPR (%)	Precision	Recall	F-Measure
Proposed model	82.2	17.2	92%	91%	91.4

Figure 8. The AUC curve for the proposed detection model.

According to this result, we could detect malware at the design level. However, the ability of the used tools in anti-pattern detection to detect malware needs to be verified. This research used a Modelio checker to check the reversed model against the malware. However, the result of the check was zero detection, as in Figure 9. We could see the model and the tool check against any quality problems and the result was zero for errors and warnings.

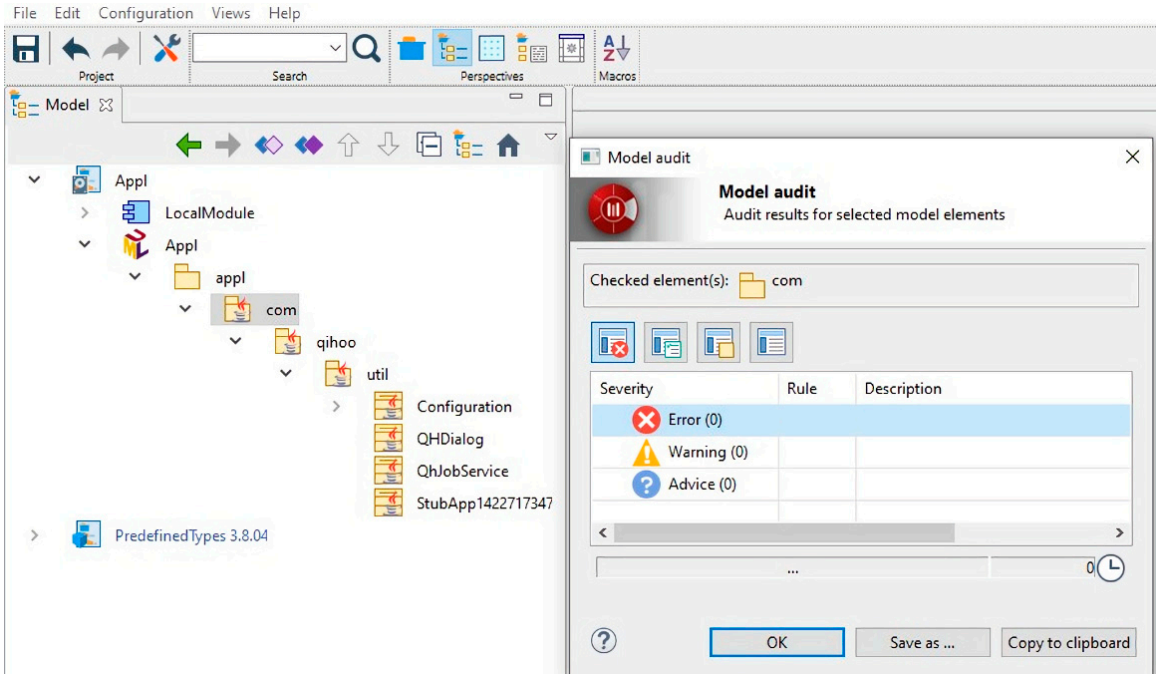


Figure 9. Result of using Modelio.

In addition, the researchers used the reasoner of ontology and the ONTOPYTHO approach that was presented in [37]. ONTOPYTHO was used to detect anti-patterns at the design of OWL Ontologies and the same result was zero detected, as in Figure 10. According to these results, the tools that were used to detect anti-patterns at the design level could not detect malware at the same level.

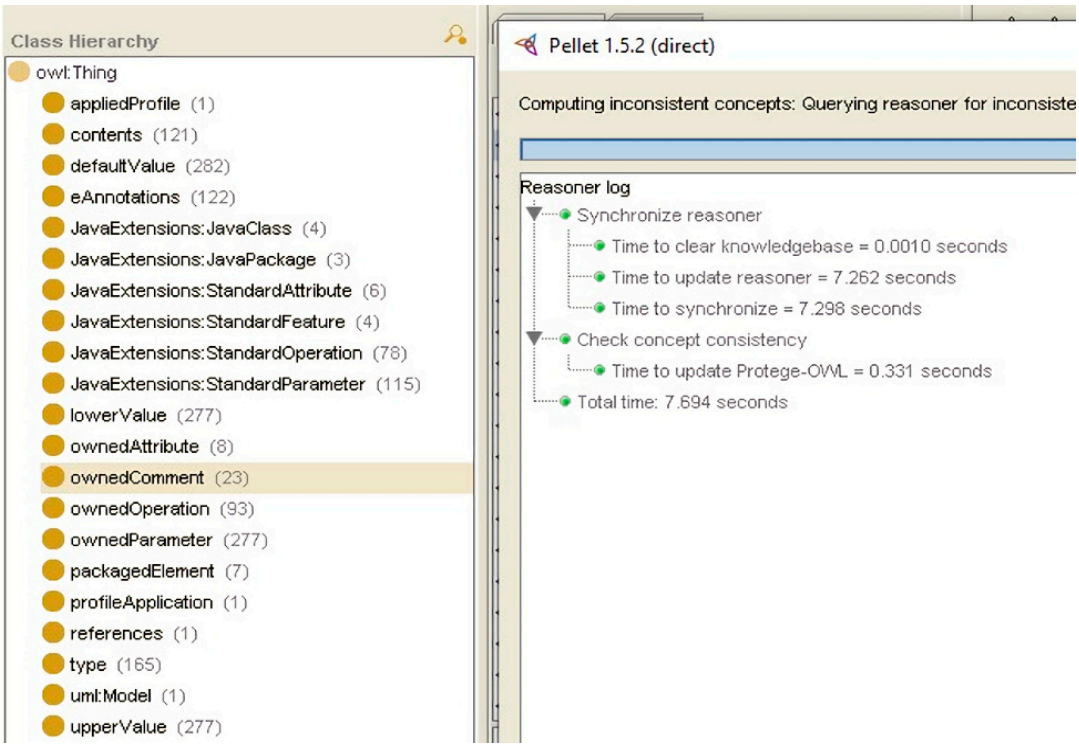


Figure 10. Result of using the Reasoner.

6. Conclusions

This study presented a malware detection method based on OWL Ontology, reverse engineering, and the semantic web query language SPARQL. This method detected malware in the design of mobile applications. By reversing the source code, the UML class diagram model was generated. This UML model was converted to OWL Ontology to detect the malware. To evaluate the method's performance, a sample of 600 APK mobile applications were selected from the CICMalDroid 2020 dataset. This sample was infected by Trojan malware, which appeared 552 times through running special SPARQL queries on the design of ontologies. The experimental results showed that the proposed method detected Data_Send_Trojan malware at the design level with 92% precision, 91% recall, 91.4% f-measure, and 91% of the AUC score, respectively. Furthermore, the proposed method showed that anti-pattern detection tools were not suitable to detect malware. The proposed method was considered the first method for detecting malware at the design level, compared to state-of-the-art methods; however, we performed most of the works manually, which resulted in a small number of implemented samples.

In future research, we will continue to explore the use of other methods to detect other malware at the design level. In addition, we will try to make a cloud system to do the conversion steps at the preprocessing automatically instead of manually; this would solve problems related to the size of the dataset, which will improve the results and be able to discover a greater number of malware.

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

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author. The data are not publicly available due to security issues.

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

References

1. Elsayed, E.K.; ElDahshan, K.A.; El-Sharawy, E.E.; Ghannam, N.E. Reverse engineering approach for improving the quality of mobile applications. *PeerJ Comput. Sci.* **2019**, *5*, e212. [CrossRef] [PubMed]
2. Krupitzer, C.; Temizer, T.; Prantl, T.; Raibulet, C. An Overview of Design Patterns for Self-Adaptive Systems in the Context of the Internet of Things. *IEEE Access* **2020**, *8*, 187384–187399. [CrossRef]
3. Volk, M.J.; Lourentzou, I.; Mishra, S.; Vo, L.T.; Zhai, C.; Zhao, H. Biosystems Design by Machine Learning. *ACS Synth. Biol.* **2020**, *9*, 1514–1533. [CrossRef] [PubMed]
4. Li, Q.; Yan, L. Older adults' use of mobile device: Usability challenges while navigating various interfaces. *Behav. Inf. Technol.* **2019**, *39*, 837–861. [CrossRef]
5. Kermansaravi, Z.A.; Rahman, S.; Khomh, F.; Jaafar, F.; Guéhéneuc, Y.-G. Investigating design anti-pattern and design pattern mutations and their change- and fault-proneness. *Empir. Softw. Eng.* **2021**, *26*, 1–47. [CrossRef]
6. Naqvi, B.; Clarke, N.; Porras, J. Incorporating the human facet of security in developing systems and services. *Inf. Comput. Secur.* **2021**, *29*, 49–72. [CrossRef]
7. Mercaldo, F.; Di Sorbo, A.; Visaggio, C.A.; Cimitile, A.; Martinelli, F. An exploratory study on the evolution of Android malware quality. *J. Softw. Evol. Process* **2018**, *30*, e1978. [CrossRef]
8. Rasool, G.; Ali, A. Recovering Android Bad Smells from Android Applications. *Arab. J. Sci. Eng.* **2020**, *45*, 3289–3315. [CrossRef]
9. Ramadan, Q.; Strüber, D.; Salnitri, M.; Jürjens, J.; Riediger, V.; Staab, S. A semi-automated BPMN-based framework for detecting conflicts between security, data-minimization, and fairness requirements. *Softw. Syst. Model.* **2020**, *19*, 1191–1227. [CrossRef]
10. Politowski, C.; Khomh, F.; Romano, S.; Scanniello, G.; Petrillo, F.; Guéhéneuc, Y.-G.; Maiga, A. A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension. *Inf. Softw. Technol.* **2020**, *122*, 106278. [CrossRef]

11. Darabian, H.; Dehghantanha, A.; Hashemi, S.; Taheri, M.; Azmoodeh, A.; Homayoun, S.; Choo, K.-K.R.; Parizi, R.M. A multiview learning method for malware threat hunting: Windows, IoT and android as case studies. *World Wide Web* **2020**, *23*, 1241–1260. [CrossRef]
12. Kadiyala, S.P.; Jadhav, P.; Lam, S.-K.; Srikanthan, T. Hardware Performance Counter-Based Fine-Grained Malware Detection. *ACM Trans. Embed. Comput. Syst.* **2020**, *19*, 1–17. [CrossRef]
13. Sebastio, S.; Baranov, E.; Biondi, F.; Decourbe, O.; Given-Wilson, T.; Legay, A.; Puodzius, C.; Quilbeuf, J. Optimizing symbolic execution for malware behavior classification. *Comput. Secur.* **2020**, *93*, 101775. [CrossRef]
14. Maevsky, D.A.; Maevskaya, E.J.; Stetsuyk, E.D.; Shapa, L.N. Malicious Software Effect on the Mobile Devices Power Consumption. In *Structural Equation Modelling*; Springer Science and Business Media LLC: Berlin/Heidelberg, Germany, 2017; pp. 155–171.
15. Akram, J.; Mumtaz, M.; Jabeen, G.; Luo, P. DroidMD: An efficient and scalable Android malware detection approach at source code level. *Int. J. Inf. Comput. Secur.* **2021**, *15*, 299. [CrossRef]
16. Tang, J.; Li, R.; Jiang, Y.; Gu, X.; Li, Y. Android malware obfuscation variants detection method based on multi-granularity opcode features. *Futur. Gener. Comput. Syst.* **2021**, *129*, 141–151. [CrossRef]
17. Darem, A.; Abawajy, J.; Makkar, A.; Alhashmi, A.; Alanazi, S. Visualization and deep-learning-based malware variant detection using OpCode-level features. *Future Gener. Comput. Syst.* **2021**, *125*, 314–323. [CrossRef]
18. Gupta, D.; Rani, R. Improving malware detection using big data and ensemble learning. *Comput. Electr. Eng.* **2020**, *86*, 106729. [CrossRef]
19. Ren, Z.; Wu, H.; Ning, Q.; Hussain, I.; Chen, B. End-to-end malware detection for android IoT devices using deep learning. *Ad Hoc Netw.* **2020**, *101*, 102098. [CrossRef]
20. Wressnegger, C.; Freeman, K.; Yamaguchi, F.; Rieck, K. Automatically Inferring Malware Signatures for Anti-Virus Assisted Attacks. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, Abu Dhabi, United Arab Emirates, 2–6 April 2017; pp. 587–598.
21. Abusitta, A.; Li, M.Q.; Fung, B.C. Malware classification and composition analysis: A survey of recent developments. *J. Inf. Secur. Appl.* **2021**, *59*, 102828. [CrossRef]
22. Singh, J.; Thakur, D.; Gera, T.; Shah, B.; Abuhmed, T.; Ali, F. Classification and Analysis of Android Malware Images Using Feature Fusion Technique. *IEEE Access* **2021**, *9*, 90102–90117. [CrossRef]
23. Reddy, V.; Kolli, N.; Balakrishnan, N. Malware detection and classification using community detection and social network analysis. *J. Comput. Virol. Hacking Tech.* **2021**, *17*, 333–346. [CrossRef]
24. Willems, C.; Holz, T.; Freiling, F. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Secur. Priv.* **2007**, *5*, 32–39. [CrossRef]
25. Wadkar, M.; Di Troia, F.; Stamp, M. Detecting malware evolution using support vector machines. *Expert Syst. Appl.* **2020**, *143*, 113022. [CrossRef]
26. Paul, S.; Stamp, M. Word Embedding Techniques for Malware Evolution Detection. In *Malware Analysis Using Artificial Intelligence and Deep Learning*; Springer Science and Business Media LLC: Berlin/Heidelberg, Germany, 2021; pp. 321–343.
27. Sharma, N.; Arora, B. Data Mining and Machine Learning Techniques for Malware Detection. In *Advances in Intelligent Systems and Computing*; Springer Science and Business Media LLC: Berlin/Heidelberg, Germany, 2020; pp. 557–567.
28. Yan, J.; Qi, Y.; Rao, Q. Detecting Malware with an Ensemble Method Based on Deep Neural Network. *Secur. Commun. Netw.* **2018**, *2018*, 1–16. [CrossRef]
29. Wang, W.; Zhao, M.; Wang, J. Effective android malware detection with a hybrid model based on deep autoencoder and convolutional neural network. *J. Ambient. Intell. Humaniz. Comput.* **2019**, *10*, 3035–3043. [CrossRef]
30. Catak, F.O.; Ahmed, J.; Sahinbas, K.; Khand, Z.H. Data augmentation based malware detection using convolutional neural networks. *PeerJ Comput. Sci.* **2021**, *7*, e346. [CrossRef]
31. Brown, W.H.; Malveau, R.C.; McCormick, H.W.; Mowbray, T.J. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*; John Wiley & Sons, Inc.: Hoboken, NJ, USA, 1998.
32. Mann, C. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. *Kybernetes* **2007**, *36*. [CrossRef]
33. Moha, N.; Gueheneuc, Y.-G.; Duchien, L.; Le Meur, A.-F. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Trans. Softw. Eng.* **2010**, *36*, 20–36. [CrossRef]
34. Van Emden, E.; Moonen, L. Java quality assurance by detecting code smells. In Proceedings of the Ninth Working Conference on Reverse Engineering, Richmond, VA, USA, 29 October–1 November 2002; pp. 97–106.
35. Settas, D.; Cerone, A.; Fenz, S. Enhancing ontology-based antipattern detection using Bayesian networks. *Expert Syst. Appl.* **2012**, *39*, 9041–9053. [CrossRef]
36. Elsayed, E.K.; Ghannam, N.E. Metric Method for Long Life Semantic Applications. *Int. J. Intell. Eng. Syst.* **2019**, *12*, 25–36. [CrossRef]
37. El-Dahshan, K.A.; Elsayed, E.K.; Ghannam, N.E. Comparative Study for Detecting Mobile Application's Anti-Patterns. In Proceedings of the 2019 8th International Conference on Software and Information Engineering, Cairo, Egypt, 9–12 April 2019.
38. Svensson, R.; Tatrous, A.; Palma, F. Defining Design Patterns for IoT APIs. In Proceedings of the Communications in Computer and Information Science; Springer Science and Business Media LLC: Berlin/Heidelberg, Germany, 2020; pp. 443–458.

39. Mat, S.R.; Razak, M.F.; Kahar, M.N.; Arif, J.M.; Mohamad, S.; Firdaus, A. Towards a systematic description of the field using bibliometric analysis: Malware evolution. *Scientometrics* **2021**, *9*, 1–43. [CrossRef] [PubMed]
40. Mahdavifar, S.; Kadir, A.F.A.; Fatemi, R.; Alhadidi, D.; Ghorbani, A.A. Dynamic Android Malware Category Classification using Semi-Supervised Deep Learning. In Proceedings of the 18th IEEE International Conference on Dependable, Autonomic, and Secure Computing (DASC), Calgary, AB, Canada, 17–24 August 2020; Available online: https://www.unb.ca/cic/datasets/maldroid-2020.html (accessed on 10 March 2021).
41. Jia, Z.; Yao, Y.; Wang, Q.; Wang, X.; Liu, B.; Jiang, Z. Trojan Traffic Detection Based on Meta-learning. In *Proceedings of the Swarm, Evolutionary, and Memetic Computing*; Springer Science and Business Media LLC: Berlin/Heidelberg, Germany, 2021; pp. 167–180.