CHAPTER 3

ASParseV3: Auto-Static Parser and Customizable Visualizer

Iman Almomani, Rahaf Alkhadra, and Mohanned Ahmed

3.1 Introduction

Our modern world is rapidly moving toward digitalization and automation, where everything is converging into an automated version. As technology takes over our lives, we are at the start of the 4th industrial revolution, which mainly focuses on a world that relies heavily on technology and innovation. The use of technology not only provides us with convenience but comfort as well. However, the rapid development of technology comes at the price of ensuring cybersecurity. Attackers are finding many ways to achieve their malicious goals, which requires us to take precautions to

I. Almomani (✉)
Security Engineering Lab, Prince Sultan University, Riyadh, Saudi Arabia

Computer Science Department, The University of Jordan, Amman, Jordan
e-mail: imomani@psu.edu.sa; i.momani@ju.edu.jo

R. Alkhadra • M. Ahmed
Security Engineering Lab, Computer Science Department, Prince Sultan University, Riyadh, Saudi Arabia
e-mail: rkhadra@psu.edu.sa; mqasem@psu.edu.sa

I. Almomani et al. (eds.), *Cyber Malware*, Security Informatics and Law Enforcement, https://doi.org/10.1007/978-3-031-34969-0_3

face such security issues. One of the most popular and common forms of security invasion in our digital world is using malicious code, often referred to as malware [27]. Malware is a code written by security attackers to intrude into a specific computer system or software to perform malicious acts such as stealing data or causing damage. For example, malware could be in different forms, such as worms, viruses, trojans, spyware, adware, or ransomware. Therefore, it is essential to protect any system from malware. This can be done by detecting the malware and then classifying which type it is. A tremendous amount of research has been conducted in the past years regarding the topic of malware detection and classification [11].

According to recent reports, malware generation and creation have been increasing rapidly on a daily basis. It is estimated that around one million malware files are created daily [31]. This increase could seriously threaten the economy, both financially and technically. The increase in cyber threats and crimes costs the economy around 1 trillion dollars in 2022 for cyber insurance, which results in an increase of 50% in comparison to the past 2 years [12]. The term malware refers to any malicious entity that changes the original behavior by utilizing software flaws and vulnerabilities. In this chapter, the term malware will be used to refer to any malicious software that may include any of the following malware families, ransomware, adware, viruses, or keyloggers [11].

Depending on the purpose and behavior of the malware, it is categorized into different families. Every family has common features. For instance, stealing information, creating vulnerability, and denial of service are all examples of malware behavior. Such behaviors are essential in detecting malware since this information will be used to analyze the software and categorize it into benign or malware [35]. To differentiate between malicious and benign apps, we need to scan the program code first, extract its features, and analyze them [6]. Features extraction can be achieved through two main ways: static analysis [3] and dynamic analysis [13]. Another possible way is to use hybrid analysis [2], a combination of the previous two [25]. Static analysis is concerned with contextual data from the source code without running the program. However, dynamic analysis involves executing the program and extracting the runtime features. The hybrid analysis uses both contextual and runtime features to detect malware [11].

Over the years, researchers have been developing new techniques for malware detection. The latest trend in this field is using machine learning for malware detection. However, this technique cannot be used without analyzing the program code and extracting important features that help in discriminating the malware families [22]. It is possible to evade the risk of malware if the related features are available. Therefore, a collection of advanced detection methods using machine learning depends on feature engineering as well as reverse engineering [33]. Feature engineering is a technique used to manipulate unstructured data into features that can be understandable by the computer or machine [32]. However, other techniques, such as binary obfuscation, can be used by attackers to design a reverse engineering resistant file [30]. Moreover, deep learning can be used in an advanced model of neural networks to capture features, learn, and adapt during training. Even though a few studies report the use of deep learning, some do not discuss the scalability and different architectures enough for malware detection [5, 33].

One of the main benefits of using static analysis over any other technique is that this analysis does not require executing the program, making it a safer choice to apply [25]. Moreover, another vital benefit is examining the code without regard to the diversity of IoT architecture or the physical capabilities of an IoT device. Hence, the analysis considers all possible inspection methods with no reference to the physical performance [24]. Furthermore, due to the nature of the static analysis, the malware may not be able to avoid, hide, and/or obfuscate during the analysis process because it runs passively [34]. Finally, its automation characteristic is what makes static analysis prominent and outstanding [16].

Therefore, this chapter introduces a new comprehensive static parsing software called ASParseV3. It is an extension to ASParseV1 [1]. It is a GUI-based tool with various features such as **(a)** selecting many files or directories to be scanned in one experiment, **(b)** adding or removing keywords/features, **(c)** filtering the keywords/features and specific file types, **(d)** efficient scanning process as many files are scanned simultaneously, **(e)** providing customizable visualization dashboards with the ability to export the chart(s), and **(f)** exporting the results in different formats such as JSON and CSV.

The rest of the chapter sections present and discuss the related works regarding malware analysis techniques, malware detection, and the use of static analysis for malware detection. Moreover, they present the proposed developed software (ASParseV3), which performs static features extraction and parsing. Also, the chapter demonstrates a use case of Android OS malware static features extraction using the ASParseV3 software. Finally, conclusions with a summary of possible future works are presented.

3.2 Related Works

Parsing the features of source code is potentially utilized in estimating the software performance, reverse engineering, and static analysis [20]. However, the extracted features can be represented in different formats such as gray-scale images, structural entropy, or JSON file [15]. Moreover, the extracted features can be further deployed in various fields. For instance, the authors of [21] have developed a tool named DeepTLS to analyze encrypted traffic by extracting the features from the network packets. In [28], the python-Evtx-parser (pexp) has been developed to parse the required features to detect Lateral Movement Attacks. In a nutshell, Table 3.1 demonstrates a comparison among related works.

Several tools have been proposed to perform static parsing in Android platform [1, 8, 23]. Khalid et. al. proposed a memory parsing tool for Android applications [19]. The authors of [17] have developed Sena TLS-Parser, a tool that automates the software testing process by parsing the Android source code. Initially, the Android source code is imported into the Eclipse environment. Subsequently, Sena TLS-Parser scans the code and generates the required test cases. Another approach that utilizes static parsing in enhancing the development of Android applications is by recommending a suitable API for the Android developer based on the parsing results. In [36], the authors have developed APIMatchmaker, a tool that recommends the best API usage by parsing similar Android apps.

Parsing Android source code can further be deployed in detecting malicious applications. In [26], the authors have parsed the suspect methods of two Android apps in order to extract their similarities using their proposed tool, StrAndroid. Consequently, they identified the potential malicious behaviors that are shared between the two apps. Additionally, Android permissions can be parsed in order to rank the risk of the malicious application. Dharmalingam et al. proposed a permission grading scheme that extracts and defines the required permissions in an Android app and rates the risk of the app accordingly [14]. In their proposed scheme, the Manifest file is parsed to extract the defined permission in the app. Subsequently, the extracted permissions are fed into the feature encoder to be further utilized in the deep neural algorithm for detecting malware applications. However, static analysis can be combined with dynamic analysis to increase the efficiency of malware detection. In [2], the authors have applied static analysis as a prior stage to implementing the dynamic analysis.

Table 3.1 A comparison among existing parsing tools

Work	*Year*	*Tool name*	*Aim*	*Scanned File Type*	*Has GUI?*	*Customization*	*Number of Extracted Features*	*Exported File Format*	*Scanning Environment*	*Is a System?*	*Developing Language*
[21]	2022	DeepTLS	To analyze encrypted traffic by extracting the features from the network packets	Pcap	Yes	No	70+	JSON	NM	Yes	C++
[15]	2022	PE Parser	To parse executable binary files in order to extract required features for malware detection	Exe files	No	No	14	Gray-scale images, structural entropy.	NM	No	Python
[28]	2022	python-Evtx-parser (pexp)	To parse the required features from the network packet to detect Lateral Movement Attacks	.evtx files	No	No	NM	xml objects	Windows 10, macOS, Ubuntu	No	Python
[17]	2022	TLS-Parser	To automate the software testing process by parsing the Android source code	.java	No	Yes	NM	.java	Windows	No	Java
[26]	2020	StrAndroid,	To parse the suspect methods of two Android apps in order to extract their similarities	APK	No	No	NM	Text file	NM	Yes	Python
[36]	2022	API Matchmaker	To recommend the best API usage by parsing similar Android apps	APK	No	No	NM	Text file	NM	No	Java
[18]	2021	PetaDroid	To cluster the malware families based on static analysis for Android OS	APK	No	No	300+	Text file	NM	Yes	Python, Bash
[14]	2020	Permission Grader	To grade the risk level of Android malware app based on its extracted permissions	Manifest file	No	Yes	1000	Text file	NM	Yes	NM
[29]	2020	DroidPortrait	To utilize the extracted Android permissions and API calls in developing a malware portrait	Manifest file, class.dex	No	Yes	50,000	PNG	NM	Yes	NM
This work	2022	ASParseV3	To propose a GUI-based, customizable, and comprehensive static parsing tool with the ability to export results/charts in different formats	Flexible (Any)	Yes	Yes	Unlimited	CSV Meta-Data: Json Graph: PNG	Cross-platform	Yes	Python

The efficiency of the parsing approach highly affects the overall static analysis process. The authors of [18] applied canonical representation to enhance the parsing process for Android code by developing the static analyzing tool, PetaDroid. The core of this proposed solution is to define the application's behavior by tracking the used APIs and the app's actions. Consequently, fingerprinting the malware applications. Besides the API calls, the permissions can be utilized to determine the malicious application's behavior. In [29], the APK file has been decomposed using APKtool to retrieve the Manifest file and class.dex file. The aforementioned files were parsed to extract the permissions and the API calls, respectively. Then, multidimensional behavior analysis was conducted on the extracted features to develop a malware portrait. Even though there are many static parsing tools, they are not flexible in accepting many file systems and can extract only a limited number of features. Moreover, they do not have a customizable graphical user interface (GUI). Therefore, there is a need for a customizable GUI-based system with the ability to scan an unlimited number of features on various file systems.

3.3 Proposed System

There is a need for user-friendly, extensible, and flexible software. This chapter introduces the third version of the Android Static Parse (ASParse). The tool ASParse-V3 is an improvement to the previous versions. It is a cross-platform, portable, and general tool that performs static analysis and features parsing for any file type while supporting different operating systems. This version of ASParse is efficient and fast due to its concurrent scanning characteristic. Furthermore, ASParseV3 can be used as a preprocessing method for static feature extraction to construct datasets for subsequent processing through ML/DL models due to its feature of exporting the results to JSON and CSV files. For instance, the previous versions of the ASParse tool were used to extract static features and develop different types of datasets [1]. For example, [4, 7] utilized the ASParse tool to extract the API and permissions of thousands of Android applications. The extracted features created a dataset that helped detect Ransomware apps with high accuracy.

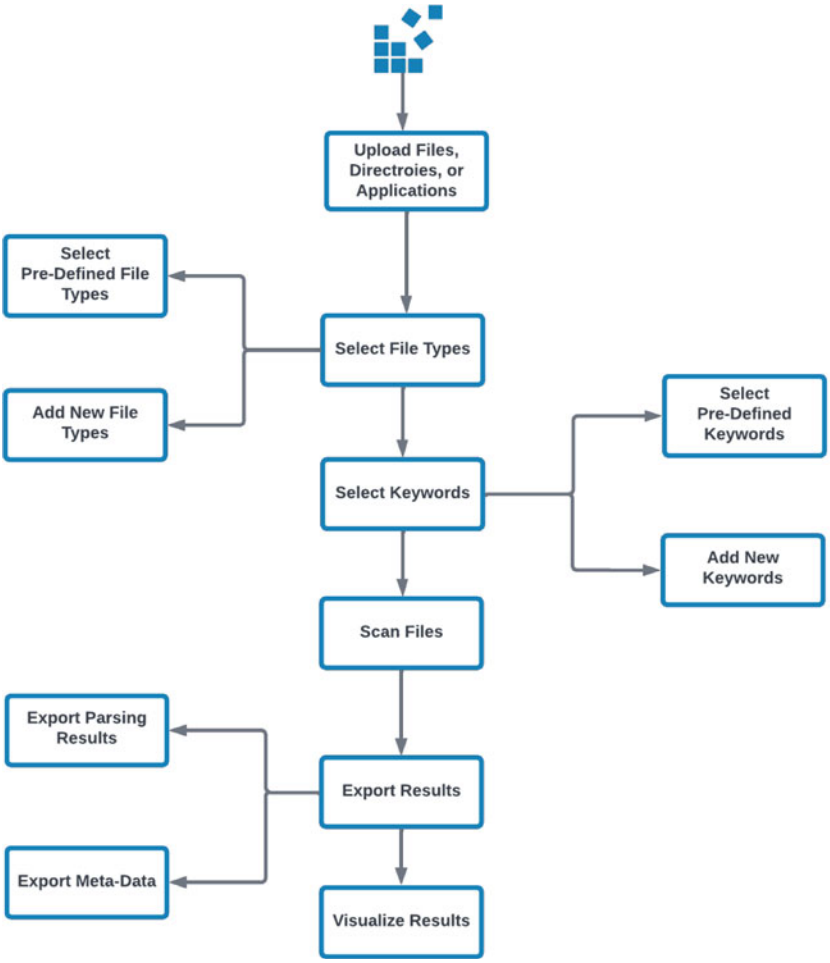


Fig. 3.1 Flow structure of the proposed system

3.3.1 System Overview

To illustrate the system flow, Fig. 3.1 shows how the ASParseV3 application generally works. The first step is uploading the files, directories, or multiple directories. The second step is choosing a set of predefined features or adding specific features. Then, moving to the third step, the system scans the files to export the results. Finally, after the results are exported, they can be visualized via a customizable dashboard.

3.3.2 Features and User Interfaces

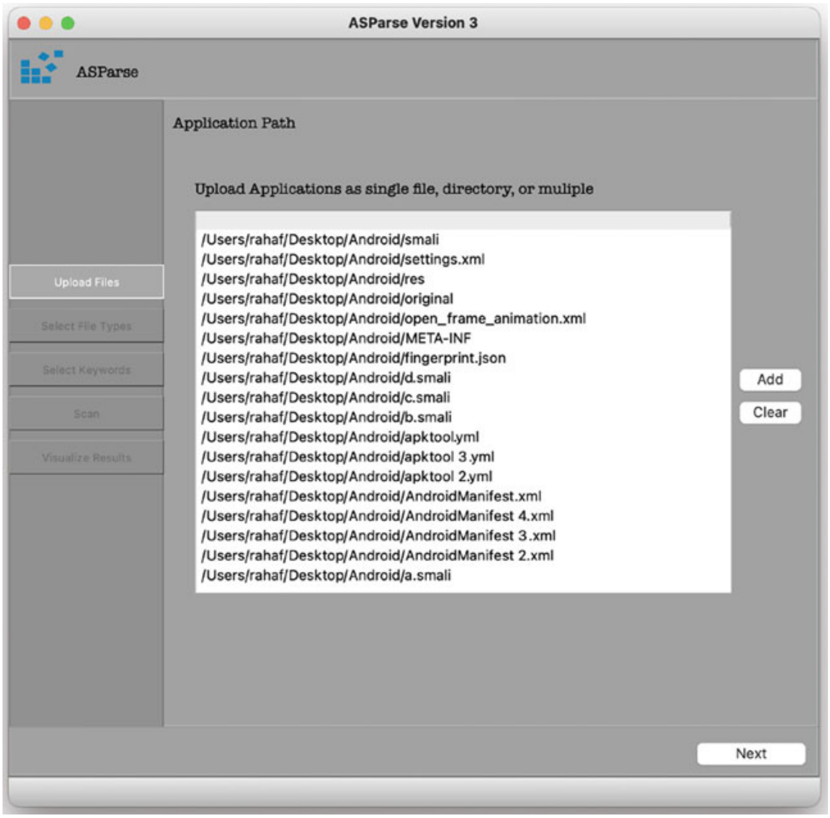
The scalability and portability of ASParseV3 are achieved by integrating it with a portable development environment that also makes the software cross-platform to be installed on various operating systems (OSs). In addition, the software's scope can be used as general and specific. For example, it can scan and parse different input formats, such as Android and Windows applications. Furthermore, ASParseV3 is user-friendly due to the modern graphical user interface (GUI) that is easy to use and its customizability based on the user's needs. For instance, the user can customize features and file types to be scanned and customize the scanning results based on the filtering feature available on the results dashboard. The system process is divided mainly into five steps: uploading files, selecting file types, choosing keywords, scanning, and results visualization. Each phase has a separate user-friendly window.

3.3.2.1 Uploading Files Window

The first window of the application is used to upload files or applications to be scanned. The user can upload multiple files, directories, or a single directory. As Fig. 3.2 illustrates, the button "Add" is clicked to upload the applications, which opens a file selector dialog window to upload files/directories. All uploaded files will be shown on a panel field. The user may also clear the uploaded files in the panel field by clicking on the "Clear" button and adding new applications when needed.

3.3.2.2 Selecting File Types Window

The second window allows users to select files of specific types (file extensions) to be scanned. Figure 3.3a shows a sample of Android OS file types. The user may choose one or multiple types by checking the checkbox. Moreover, the user can customize the file types by adding or deleting types by clicking on the settings icon on the top right of Fig. 3.3a. The settings button opens a new window for editing, as Fig. 3.3b illustrates. The user can write the file types in the text field and then click on the button "Add" to add them to the current panel. The user can also delete any newly defined types by clicking on the button "Remove." By default,



Application Path

Upload Applications as single file, directory, or muliple

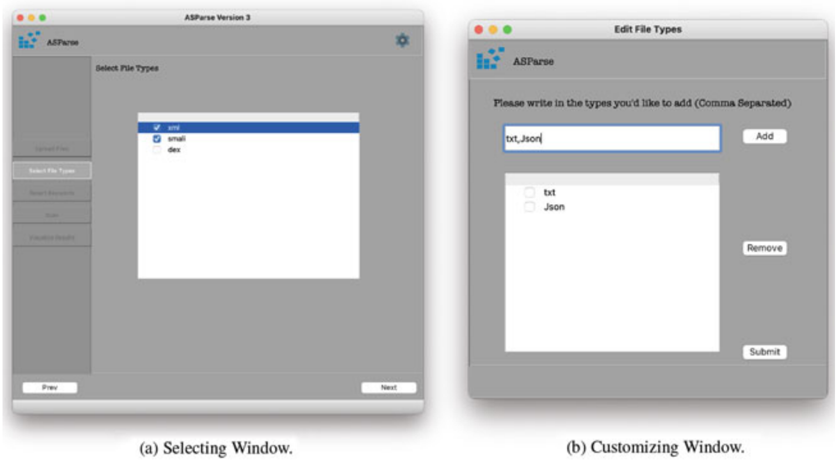
```
/Users/rahaf/Desktop/Android/smali
/Users/rahaf/Desktop/Android/settings.xml
/Users/rahaf/Desktop/Android/res
/Users/rahaf/Desktop/Android/original
/Users/rahaf/Desktop/Android/open_frame_animation.xml
/Users/rahaf/Desktop/Android/META-INF
/Users/rahaf/Desktop/Android/fingerprint.json
/Users/rahaf/Desktop/Android/d.smali
/Users/rahaf/Desktop/Android/c.smali
/Users/rahaf/Desktop/Android/b.smali
/Users/rahaf/Desktop/Android/apktool.yml
/Users/rahaf/Desktop/Android/apktool 3.yml
/Users/rahaf/Desktop/Android/apktool 2.yml
/Users/rahaf/Desktop/Android/AndroidManifest.xml
/Users/rahaf/Desktop/Android/AndroidManifest 4.xml
/Users/rahaf/Desktop/Android/AndroidManifest 3.xml
/Users/rahaf/Desktop/Android/AndroidManifest 2.xml
/Users/rahaf/Desktop/Android/a.smali
```

Fig. 3.2 Uploading applications window

if no checkboxes were chosen, all predefined file types will be included in the scanning process.

3.3.2.3 Selecting Keywords Window

The third window allows users to select the keywords to look for while scanning. Figure 3.3a shows a sample of Android OS file types. However, the user can customize the features through the settings window by adding or deleting keywords by clicking on the settings icon on the top right of the window (as shown in Fig. 3.4a). Similar to the file types editing feature,



Select File Types

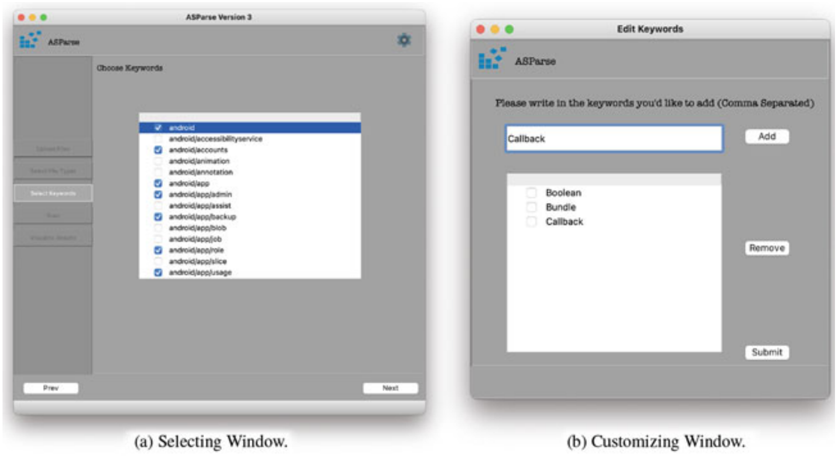
- [x] xml
- [x] smali
- [ ] dex

Please write in the types you'd like to add (Comma Separated)

txt,Json

- [ ] txt
- [ ] Json

Fig. 3.3 Selecting and customizing file types windows. (**a**) Selecting Window. (**b**) Customizing Window



Choose Keywords

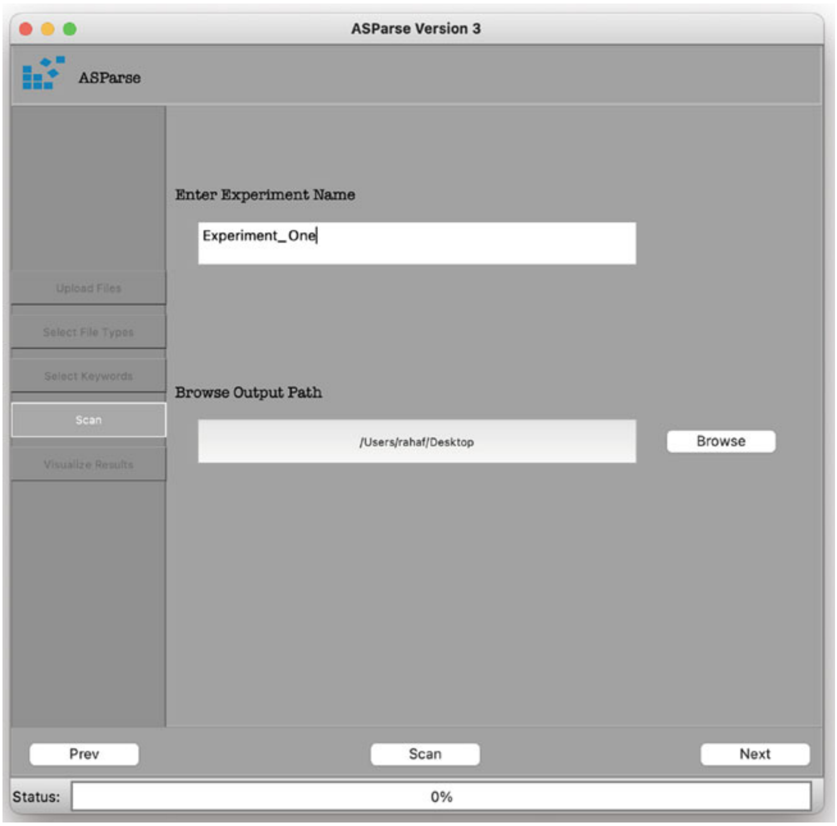
- [x] android
- [ ] android/accessibilityservice
- [x] android/accounts
- [ ] android/animation
- [ ] android/annotation
- [x] android/app
- [x] android/app/admin
- [ ] android/app/assist
- [x] android/app/backup
- [ ] android/app/blob
- [ ] android/app/job
- [x] android/app/role
- [ ] android/app/slice
- [x] android/app/usage

Please write in the keywords you'd like to add (Comma Separated)

Callback

- [ ] Boolean
- [ ] Bundle
- [ ] Callback

Fig. 3.4 Selecting and customizing keywords windows. (**a**) Selecting Window. (**b**) Customizing Window



Enter Experiment Name

Experiment_One

Browse Output Path

/Users/rahaf/Desktop

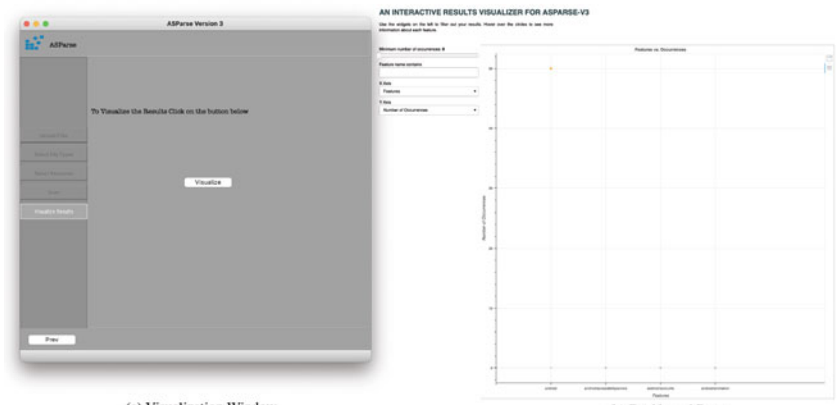
Status: 0%

Fig. 3.5 Scanning window

the settings button can be used to edit the list of keywords, as illustrated in Fig. 3.4b.

3.3.2.4 Scanning Window

The fourth window allows users to add the configuration values of an experiment, such as the experiment name and the path used to save the results, as shown in Fig. 3.5. Then, the scanning process begins by clicking


ASParse Version 3
ASParse
To Visualize the Results Click on the button below
Visualize
Prev
AN INTERACTIVE RESULTS VISUALIZER FOR ASPARSE-V3

(a) Visualization Window. (b) Dashboard Page.

Fig. 3.6 Visualization window and page. (**a**) Visualization Window. (**b**) Dashboard Page

on the "Scan" button. Finally, the progress bar provides the user with real-time updates on the scanning progress.

3.3.2.5 Visualizing Results and Dashboard Window

The fifth and final window links the tool to the visualization dashboard. After completing the scanning progress, the user can move to the visualization window and click on the "Visualize" button as shown in Fig. 3.6a to display the results in terms of a plot. The actions performed in this window do not affect the scanning results. It is a complimentary step for results visualization and filtering. However, this step cannot be completed without performing the scanning. When visualization is activated, a dashboard page opens in the browser. The dashboard is where the user can visualize the parsing results. The plot's X-axis represents the features (keywords), and the Y-axis represents the number of occurrences. As Fig. 3.6b illustrates, the dashboard is customizable based on the user's preference. For instance, the user may filter out and visualize the results according to the minimum number of feature occurrences and features containing a specific string or substring. Also, the resulting graph (plot) can be exported as an image using the saving button on the right of the plot. This can help the researchers/experts to share their results conveniently.

3.3.3 Use Case

To demonstrate the tool, Android benign samples and malware samples were used. The samples come in the form of an Android Package Kit (APK). The APKs contain all software details, including source code, permissions, and APIs used. However, APKs are compressed files that need reverse engineering to recover the application code [9]. APKTool[1]s was used to decompile the apps and extract the source files. Afterward, the decompiled APKs were fed to the ASParse tool.

3.3.3.1 Data Collection

For data collection, two sources were used, Drebin Dataset [10] and APKCombo.[2] The Drebin Dataset contained 5560 malware samples belonging to 179 malware families. On the other hand, the benign data samples were downloaded through APKCombo. Ten samples were randomly chosen from the Drebin dataset, along with ten samples from APKCombo. To ensure that the apps downloaded from APKCombo are benign, they were scanned by a well-known website called VirusTotal.[3] This website offers tens of Antivirus engines that are specialized in detecting different types of malware.

3.3.3.2 Tests and Results

The experiment was performed on a sample of 10 benign APKs and 10 malicious samples from the Derbin dataset. First, all files were added to the application upload field. Then, all predefined file types were chosen. Afterward, six keywords from the predefined ones were chosen, including android, android/animation, and android/app. In addition to the keywords Bundle and Button and Callback. After clicking on the visualization button in the final window, the application will shift to the dashboard, where the plot will be displayed with the ability to save the plot after customizing it. Figure 3.7 illustrates the saved plot sample. Moreover, Fig. 3.8 illustrates a sample of the saved plot where it illustrates the details of each data point on the plot. Furthermore, Table 3.2 demonstrates a sample of the resulting CSV. Finally, Fig. 3.9 represents the JSON metadata file resulting from the scan.

[1] https://ibotpeaches.github.io/Apktool/.
[2] https://apkcombo.com/.
[3] https://www.virustotal.com/gui/home/upload.

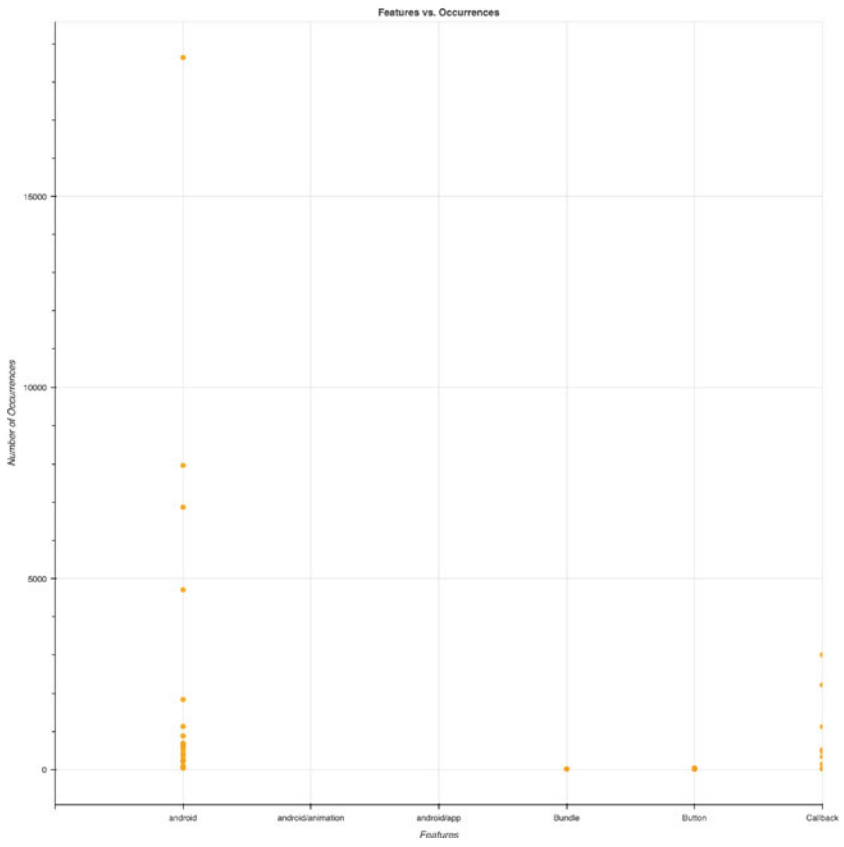


Fig. 3.7 Features vs. Occurrences Plot

3.3.3.3 Validation

The validation process for ASParseV3 was carried out thoroughly to ensure that its performance, user interface (UI), and user experience (UX) met the required needs. The Security Engineering Lab (SEL) conducted the validation and compared the scanning results of ASParseV3 with previous releases of ASParse. In addition, VirusTotal was used to retrieve information such as permissions used in the applications/APKs to compare with ASParseV3 and verify further its scanning results' accuracy. To validate the use case, VirusTotal was used to collect the permissions used by the APK.

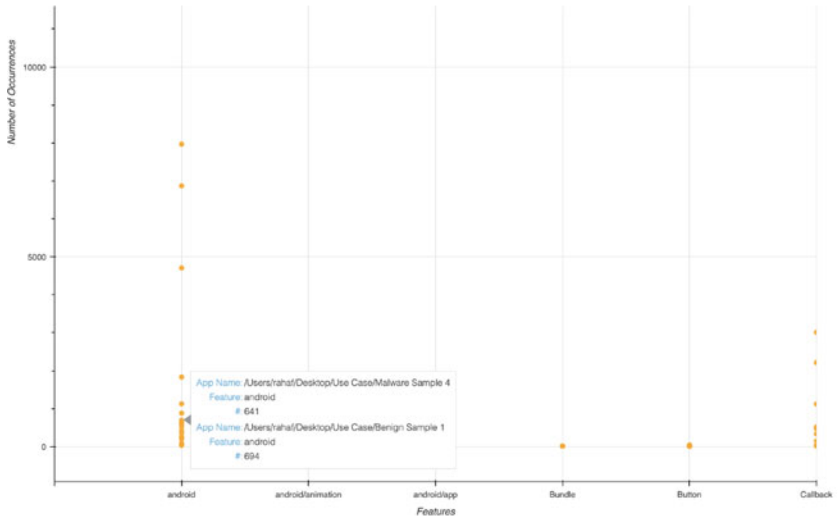


Fig. 3.8 Data point details

Figure 3.10 shows a sample of the permissions used by the APK validation test sample. The resulting permissions were then used to scan the same APK using ASParseV3. The results showed that ASParseV3could scan the uploaded APK and accurately report the number of occurrences for each permission. Overall, the validation process demonstrates that ASParseV3 is a reliable and efficient tool for scanning applications and APKs features such as permissions. The comparison with previous releases and the use of VirusTotal helped ensure the scanning results' accuracy. For example, Table 3.3 illustrates the number of occurrences of each permission found by ASParseV3 during the validation process. Moreover, using ASParseV3 to scan the same application without specifying any keywords has resulted in showing additional permissions/API calls other than the ones retrieved from VirusTotal as Table 3.4 illustrates. Hence, this validates the accuracy of the ASParseV3 and its additional capabilities compared with similar tools.

Table 3.2 The resulting CSV from the use case

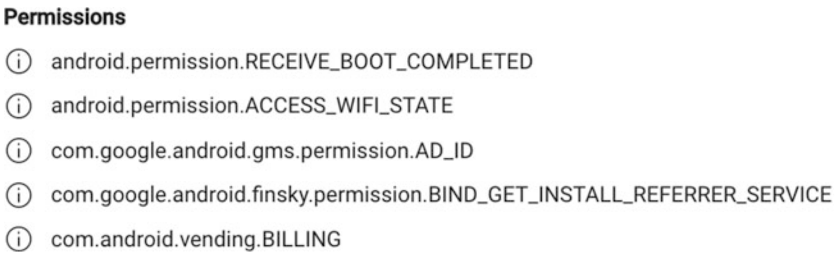
fileName	*Android*	*Android/ animation*	*Android/app*	*Button*	*Bundle*	*Callback*
/Users/rahaf/Desktop/Use Case/Malware Sample 5	212	0	0	10	0	0
/Users/rahaf/Desktop/Use Case/Malware Sample 7	221	0	0	4	0	0
/Users/rahaf/Desktop/Use Case/Malware Sample 8	53	0	0	0	0	0
/Users/rahaf/Desktop/Use Case/Malware Sample 4	641	0	0	15	1	9
/Users/rahaf/Desktop/Use Case/Malware Sample 1	1834	0	0	0	1	0
/Users/rahaf/Desktop/Use Case/Malware Sample 10	355	0	0	0	0	0
/Users/rahaf/Desktop/Use Case/Malware Sample 3	535	0	0	9	0	0
/Users/rahaf/Desktop/Use Case/Malware Sample 6	254	0	0	2	1	5
/Users/rahaf/Desktop/Use Case/Benign Sample 5	37	0	0	0	0	0
/Users/rahaf/Desktop/Use Case/Malware Sample 2	96	0	0	0	1	0
/Users/rahaf/Desktop/Use Case/Benign Sample 3	0	0	0	0	0	0
/Users/rahaf/Desktop/Use Case/Malware Sample 9	430	0	0	9	0	18
/Users/rahaf/Desktop/Use Case/Benign Sample 4	596	0	0	4	1	36
/Users/rahaf/Desktop/Use Case/Benign Sample 1	694	0	0	23	4	148
/Users/rahaf/Desktop/Use Case/Benign Sample 7	880	0	0	5	5	471
/Users/rahaf/Desktop/Use Case/Benign Sample 2	1128	0	0	13	7	1121
/Users/rahaf/Desktop/Use Case/Benign Sample 10	4709	0	0	21	7	333
/Users/rahaf/Desktop/Use Case/Benign Sample 9	6871	0	0	10	3	517
/Users/rahaf/Desktop/Use Case/Benign Sample 6	7968	0	0	38	15	2215
/Users/rahaf/Desktop/Use Case/Benign Sample 8	18638	0	0	40	16	3010

```
{
  {
    "ApplicationPath": [
        "/Users/rahaf/Desktop/Use Case/Malware Sample 5",
        "/Users/rahaf/Desktop/Use Case/Malware Sample 4",
        "/Users/rahaf/Desktop/Use Case/Malware Sample 3",
        "/Users/rahaf/Desktop/Use Case/Malware Sample 2",
        "/Users/rahaf/Desktop/Use Case/Malware Sample 1",
        ...
        "/Users/rahaf/Desktop/Use Case/Benign Sample 5",
        "/Users/rahaf/Desktop/Use Case/Benign Sample 4",
        "/Users/rahaf/Desktop/Use Case/Benign Sample 3",
        "/Users/rahaf/Desktop/Use Case/Benign Sample 2",
        "/Users/rahaf/Desktop/Use Case/Benign Sample 1"
    ],
    "OutputPath": "/Users/rahaf/Desktop",
    "filetypes": [
        "xml",
        "smali",
        "dex"
    ],
    "selectedFileTypes": [
        "smali",
        "xml",
        "dex"
    ],
    "keywords": [
        "android",
      "android/accessibilityservice",
        "android/accounts",
        "android/animation",
        "android/annotation",
        "android/app",
        "android/app/admin",
        "android/app/assist",
        "android/app/backup",
        "android/app/blob",
        "android/app/job",
        "android/app/role",
        "android/app/slice",
        "android/app/usage",
        "android/appwidget",
        "android/bluetooth",
        "Button",
        "Bundle",
        "Callback"
        ...

    ],
    "selectedKeywords": [
        "android",
        "android/animation",
        "android/app",
        "Button",
        "Bundle",
        "Callback"
    ],
    "ExperimentName": "Experiment_One"
  }

}
```

Fig. 3.9 Metadata JSON content for the use case



Permissions

- ⓘ android.permission.RECEIVE_BOOT_COMPLETED
- ⓘ android.permission.ACCESS_WIFI_STATE
- ⓘ com.google.android.gms.permission.AD_ID
- ⓘ com.google.android.finsky.permission.BIND_GET_INSTALL_REFERRER_SERVICE
- ⓘ com.android.vending.BILLING

Fig. 3.10 APK permissions from VirusTotal

Table 3.3 Validation results

\/Users/rahaf/Desktop/PSU/Use Case/Benign Sample 1	
Permissions	Occurrences
android.permission.RECEIVE_BOOT_COMPLETED	1
android.permission.ACCESS_WIFI_STATE	4
com.google.android.gms.permission.AD_ID	3
com.google.android.finsky.permission.BIND_GET_INSTALL_REFERRER_SERVICE	2
com.android.vending.BILLING	1

Table 3.4 ASParseV3 additional permissions and calls

\/Users/rahaf/Desktop/PSU/Use Case/Benign Sample 1	
Permissions and calls	Occurrences
Android	18474
CallbackHandler	117
CameraAccessException	14
Certificate	285
Connection	1522
CookieSyncManager	1
DownloadRequest	8
FragmentHostCallback	3
LruCache	2
INTERNET	25

3.4 Conclusion and Future Work

This chapter proposed a third version of ASParse software as a parsing and static analysis tool. The analysis results can be used to feed machine learning algorithms and deep learning models for malware analysis and detection. Moreover, a demonstration was presented on Android OS applications showing the system's capabilities. In future work, the ASParse tool will be used to carry on with malware detection using ML and DL algorithms and models. Moreover, it will be enhanced in terms of performance and user experience.

Acknowledgments The authors would like to thank the support of Prince Sultan University. Moreover, this research was done during the author Iman Almomani's sabbatical year 2021/2022 from the University of Jordan, Amman—Jordan.

References

1. Al Khayer A, Almomani I, Elkawlak K (2020) ASAF: android static analysis framework. In: 2020 first international conference of smart systems and emerging technologies (SMARTTECH). IEEE, New York, pp 197–202
2. Almohaini R, Almomani I, AlKhayer A (2021) Hybrid-based analysis impact on ransomware detection for Android systems. Appl Sci 11(22):10976
3. Almomani I, Ahmed M, El-Shafai W (2022) Android malware analysis in a nutshell. PloS One 17(7):e0270647
4. Almomani I, AlKhayer A, Ahmed M (2021) An efficient machine learning-based approach for Android v. 11 ransomware detection. In: 2021 1st international conference on artificial intelligence and data analytics (CAIDA). IEEE, New York, pp 240–244
5. Almomani I, Alkhayer A, El-Shafai W (2022) An automated vision-based deep learning model for efficient detection of android malware attacks. IEEE Access 10:2700–2720
6. Almomani I, Khayer A (2019) Android applications scanning: the guide. In: 2019 International conference on computer and information sciences (ICCIS). IEEE, New York, pp 1–5
7. Alsoghyer S, Almomani I (2019) Ransomware detection system for Android applications. Electronics 8(8):868
8. Anupama ML, et al (2021) Detection and robustness evaluation of android malware classifiers. J Comput Virol Hacking Tech 18(3):1–24
9. Ardito L, et al (2020) Automated test selection for Android apps based on APK and activity classification. IEEE Access 8:187648–187670

10. Arp D, et al (2014) Drebin: effective and explainable detection of android malware in your pocket. In: NDSS, vol. 14, pp 23–26
11. Aslan ÖA, Samet R (2020) A comprehensive review on malware detection approaches. IEEE Access 8:6249–6271
12. Cremer F, et al (2022) Cyber risk and cybersecurity: a systematic review of data availability. In: The Geneva Papers on Risk and Insurance-Issues and Practice, pp 1–39
13. Dai Y, et al (2019) SMASH: a malware detection method based on multifeature ensemble learning. IEEE Access 7:112588–112597
14. Dharmalingam VP, Palanisamy V (2021) A novel permission ranking system for android malware detection—the permission grader. J Ambient Intell Humaniz Comput 12(5):5071–5081
15. Gibert D (2022) PE Parser: A Python package for Portable Executable files processing. Software Impacts 13:100365
16. Gosain A, Sharma G (2015) Static analysis: a survey of techniques and tools. In: Intelligent computing and applications. Springer, Berlin, pp 581–591
17. Ibrahim R, et al (2022) Sena TLS-Parser: a software testing tool for generating test cases. Int J Adv Comput Sci Appl 13(6):397–403
18. Karbab EB, Debbabi M (2021) Resilient and adaptive framework for large scale android malware fingerprinting using deep learning and NLP techniques. arXiv e-prints arXiv–2105
19. Khalid Z, et al (2022) Forensic investigation of Cisco WebEx desktop client, web, and Android smartphone applications. Ann Telecommun 78:1–26
20. Laaber C, Basmaci M, Salza P (2021) Predicting unstable software benchmarks using static source code features. Empir Softw Eng 26(6):1–53
21. Liu Z (2022) DeepTLS: comprehensive and high-performance feature extraction for encrypted traffic. arXiv preprint arXiv:2208.03862
22. Lu T, et al (2020) Android malware detection based on a hybrid deep learning model. Secur Commun Netw 2020:1–11
23. Mahr A, et al 2022 Auto-Parser: Android Auto and Apple CarPlay Forensics. In: International Conference on Digital Forensics and Cyber Crime. Springer, Berlin, pp 52–71

24. Ngo Q-D, et al (2020) A survey of IoT malware and detection methods based on static features. ICT Express 6(4):280–286
25. Omer MA, et al (2021) Efficiency of malware detection in android system: a survey. Asian J Res Comput Sci 7(4):59–69
26. Pasetto M, Marastoni N, Preda MD (2020) Revealing similarities in android malware by dissecting their methods. In: 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). IEEE, New York, pp 625–634
27. Shukla S (2022) Design of secure and robust cognitive system for malware detection. arXiv preprint arXiv:2208.02310
28. Smiliotopoulos C (2022) Use of Sysmon tool to detect lateral movement attacks
29. Su X, et al (2020) DroidPortrait: android malware portrait construction based on multidimensional behavior analysis. Appl Sci 10(11):3978
30. Talukder S, Talukder Z (2020) A survey on malware detection and analysis tools. In: International Journal of Network Security and Its Applications (IJNSA), vol 12
31. Ugarte-Pedrero X, Graziano M, Balzarotti D (2019) A close look at a daily dataset of malware samples. ACM Trans Privacy Secur (TOPS) 22(1):1–30
32. Verdonck T, Baesens B, Óskarsdóttir M, et al (2021) Special issue on feature engineering editorial. In: Machine learning, pp 1–12
33. Vinayakumar R, et al (2019) Robust intelligent malware detection using deep learning. IEEE Access 7:46717–46738
34. Wu Q, Zhu X, Liu B (2021) A survey of android malware static detection technology based on machine learning. Mob Inf Syst 2021:1–18
35. Ye Y, et al (2017) A survey on malware detection using data mining techniques. ACM Comput Surv (CSUR) 50(3):1–40
36. Zhao Y, et al (2022) APIMatchmaker: matching the right APIs for supporting the development of Android apps. IEEE Trans Softw Eng 49(1):113–130