How does Visualisation Help App Practitioners Analyse Android Apps?

Lihong Tang, Tingmin Wu, Xiao Chen, Sheng Wen, Li Li, Xin Xia, Marthie Grobler, and Yang Xiang

Abstract—Behaviour analysis is essential for the security verification of suspicious Android applications, but analysts are usually faced with a huge obstacle when conducting the app behaviour analysis. They are expected to have comprehensive knowledge of different IT fields and a strong awareness of cyber threats. However, training a new security analyst typically requires a significant amount of time and can be extremely costly. Although there are tools available to assist analysts in studying Android behaviour and security, the completion of this task still heavily relies on the experience of the analysts. To address this problem, we recognise visualisation as a promising method and conduct a series of controlled experiments to demonstrate its effectiveness in the context of Android app behaviour and security analysis. We accordingly develop a visualisation tool based on apps' call graphs (CG) (named VISUALDROID) and conduct an experiment and a follow-up interview. Compared to existing solutions, the results suggest that the CG-based visualisation solution (VISUALDROID) can lower the barriers to Android behaviour and security analysis. The user study reveals that the platform includes CG-based visualisation components leads to a statistically significant improvement in Android behaviour analysis and security awareness. More specifically, it improves APK ANALYZER, JD-GUI, JD-GUI+FLOWDROID by 71.4%, 35.7%, and 39.2% in terms of the effectiveness of behaviour analysis. Participants who use VISUALDROID also show improvements in the aspect of security awareness with an increase of 155% against APK ANALYZER, 96% against JD-GUI, and 59.3% JD-GUI+FLOWDROID.

Index Terms—Android, visualisation, application comprehension, human-computer interaction.

1 INTRODUCTION

BEHAVIOUR analysis has long been an essential activity in Android app security. In the endless war of cybersecurity, behaviour analysis helps defenders timely capture malicious behaviours of attackers, which may cause various security incidents such as the compromise of Android devices. However, behaviour analysis has never been an easy task for security analysts. They normally need to have comprehensive knowledge of different areas in IT as well as a strong skill set. For example, apart from Android programming, an analyst is expected to have knowledge of computer networks and systems, cryptography, *etc*. Analysts also need to read source code smoothly and be sensitive to potential security flaws. Almost all these capabilities can only be developed based on the experience of long-standing IT security analysis. Therefore, there exist massive barriers for IT graduates to be capable of committing security analysis on the behaviour of Android APKs.

Tools are deemed to be helpful in behaviour analysis. So far, there are several tools available in the domain, such as JD-CORE [1], APK ANALYZER [2], and FLOWDROID [3]. However, the use of these tools still requires expertise from experienced operators. This shortcoming considerably shrinks the group of capable analysts who are benefited from using the tools. Moreover, existing tools are usually not compatible with each other. When it is necessary to derive results from multiple tools, analysts may face difficulties in identifying or correlating complex security issues. Companies also claim that it is tough and costly to train a sophisticated analyst for Android security. As suggested by world-known cybersecurity companies [4], it usually takes from one to five years to train a new security analyst depending on the responsibility they have taken and their previous experience.

Software visualisation is one of the promising techniques that can help address the above problem. This technique has been widely used in the areas of software maintenance, reverse engineering, and re-engineering. In the market of Android program analysis, we can find several visualisation tools available for use, for example, ANDROGUARD [5]. The quality of these tools heavily relies on designers' understanding of how visualisation can assist analysts. Unfortunately, according to our investigation, there is no prior work that has explored the effectiveness of visualisation in analysing behaviours of Android apps. In particular, we can find some related works that focus on visualisation in analysing PC programs [6], [7], [8]. However, because the features of different platforms (*e.g.* PC, iOS, and Android) and the focus of program analysis can significantly affect what and how a program can be visualised, their experience cannot be directly borrowed and used for the Android platform.

Therefore, in this paper, we are motivated to conduct a series of controlled experiments that empirically evaluate the visualisation in Android app analysis based on the app's function call graph (*CG*).

To investigate the merits of graphical representations for source code in assisting the junior IT people with APK

- *L. Tang, S. Wen, and Y. Xiang are with the Department of Computing Technologies, Swinburne University of Technology, Australia, VIC, 3122. E-mail: {lihongtang,swen,yxiang}@swin.edu.au.*
- *L. Tang, T. Wu and M. Grobler are with CSIRO's Data61. Email: {Lihong.Tang,Tina.Wu,Marthie.Grobler}@data61.csiro.au.*
- *X. Chen and L. Li are with the Faculty of Information Technology, Monash University, Clayton, VIC 3800, Australia. Email: {Xiao.Chen,Li.Li}@monash.edu.*
- *X. Xia is with Software Engineering Application Technology Lab, Huawei. Email: xin.xia@acm.org.*
  Corresponding Authors: S. Wen and X. Chen.

Manuscript received April 19, 2022; revised April 20, 2022.

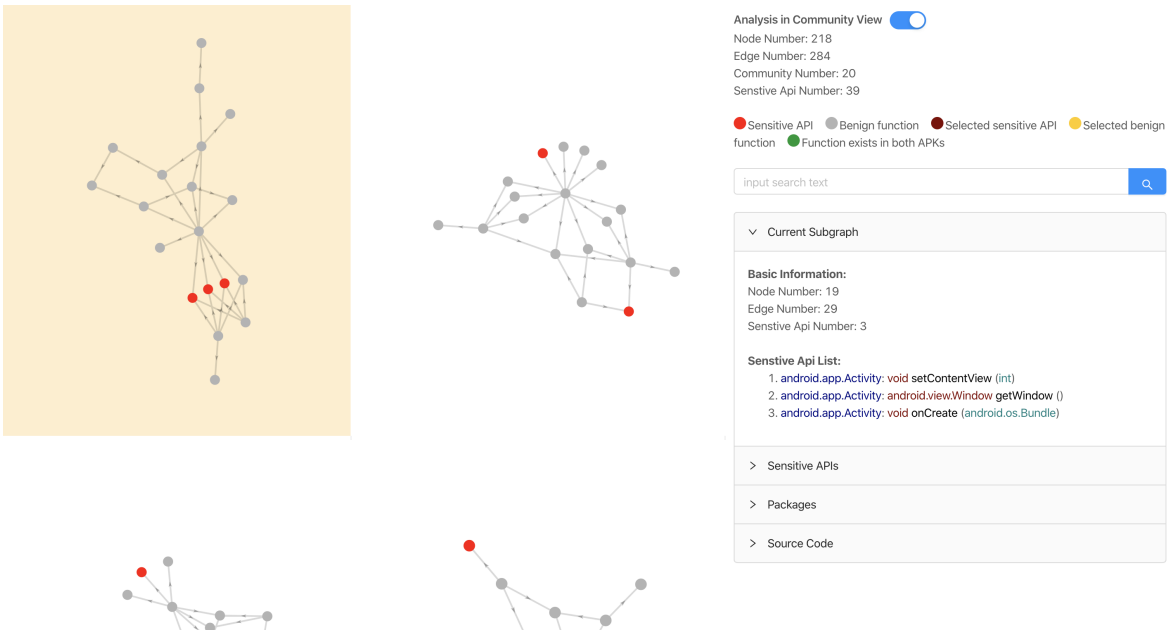

Fig. 1: A screenshot of VISUALDROID.

behaviour analysis, we develop a web-based tool named VISUALDROID and make comparisons with existing tools that employ different types of code representations (*e.g.* tree structure). VISUALDROID integrates three main functionalities: 1) visualisation for the CG of APKs, 2) interaction between CG and source code, and 3) APK comparison view. This tool provides an innovative visualisation solution for manual analysis of Android APKs. As suggested by the experiment results (cf. Section 4), users with an IT programming background (but not specialists) can easily identify the behaviours (see the example of behaviours in Table 6) of the target APKs. Specifically, this tool shows improvements in two different vectors: 1) the effectiveness of Android APK behaviour analysis and 2) users' security awareness. It improves APK ANALYZER, JD-GUI, and JD-GUI+FlowDroid by 71.4%, 35.7%, and 39.2% in terms of effectiveness, 155%, 96%, and 59.3% in terms of security awareness, respectively. From the interviews with the participants, we also find that our tool enhances the susceptibility of non-experts to suspicious APK behaviours. We summarise the major contributions as follows:

- We conduct a user study with 40 participants to evaluate the performance of our proposed visualisation framework. The results show that our approach improves the effectiveness and users' security awareness in Android APK behaviour analysis compared to existing types of visualisation.
- To facilitate the user study, we design and implement a visualisation framework called VISUALDROID. It contains the visualisation components (*e.g.* the graphical visualisation of the function call graph (*CG*) of an app), the interaction between CG and the source code, and app comparison view based on visualised CG, VISUALDROID is to be applied as a visualisation platform for Android APK behaviour analysis.

The remainder of this paper is organised as follows. We first present our preliminary study on existing app analysis tools, which provide different types of code presentation in Section 2. We also introduce our visualisation approach (VISUALDROID) and research questions in this section. We then present the detailed design of the user study in Section 3. In Section 4, we elaborate and analyse the results. In Section 5, we discuss how visualisation components help APK behaviour analysis, the implication of our study, and threats to validity, followed by the related work in Section 6 and the conclusion in Section 7.

2 EXPERIMENTAL DESIGN

In this section, we first present the preliminary study and introduce the existing analysis tools that use different types of code representation. Based on the existing available tools, we present our designed CG-based visualisation analysis tool to facilitate our user study.

2.1 Preliminary Study

In industry, apart from the information obtained from the automatic process, there are still many processes inevitably handled by manual inspection. Considering the situation that analysts can easily obtain results by running automatic tools by taking training, the difficulty of analysing APK usually relies on how analysts perform manual analysis. This is also why analysts with different professional backgrounds can draw different conclusions.

We first conduct a preliminary study on existing APK manual analysis tools. Most security companies have their own inspection tools to perform APK analysis, but they are usually not available to the public. Therefore, we search exhaustively on Stack Overflow and Reddit for the most common Android manual analysis tools. We select the tools to analyse if they provide the functions of inspecting the source code. Finally, we select APK ANALYZER, JD-GUI and the combination of FLOWDROID and JD-GUI as representatives from industry and academia. These three app

analysis tools provide different types of code representation, *e.g.* tree structure, general source code, and linked function call in text. Android Studio is a well-known tool to develop android apps, and it also provides an extended function for APK inspection called APK ANALYZER [2]. APK ANALYZER provides immediate insight into the APK when the Android project has already been built. It allows analysts to import APK files directly. Analysts can then inspect the $classes.dex$ file and some other resources within the APK. By clicking the $classes.dex$ file, APK ANALYZER first lists the package folders, after expanding the package folder, it shows all the classes in the package and furthermore lists all methods that the selected class contains. It displays the APK source code information in a tree structure. JD-GUI [1] is another common tool for analysts to conduct APK inspection on APK source code. Unlike the APK ANALYZER, the target APK needs to be decompiled and converted to a $.jar$ file to be imported into JD-GUI. The Java source code that is zipped in the $.jar$ file will show in JD-GUI's user interface. Analysts can then browse the reconstructed source code with instant access to all methods and fields. It is regarded as another code representation that programmers are most family with, in which source code is presented in a text manner within a GUI. The third tool we selected is FLOWDROID. It is a popular tool in academia when analysing APKs. It presents APK source code as data flows in Android apps and Java programs. Function calls are linked if they have a caller-callee relationship. This visualises the app source code into a number of chains, which is known as the data flow. Most researchers [9], [10], [11] utilise the data flow generated from FLOWDROID and the source code displayed in the JD-GUI to perform a further in-depth analysis for APKs. The three tools provide different code representations to assist analysts in analysing APK and the functionalities of different tools can be found in Table 1.

TABLE 1: Functionalities provided in each platform

Functionalities	CT_1	CT_2	CT_3	TM
Fuzzy Search	✓	✗	✓	✓
Graphic Visualisation	✗	✗	✗	✓
Data Flow Presentation	✗	✗	✓	✓
All Neighbouring Data Flow Presentation	✗	✗	✗	✓
Subsection Analysis	✗	✗	✗	✓
Source Code Inspection	✗	✓	✓	✓
Redirect to Class/Method Declaration	✗	✓	✓	✗
Sensitive API Highlight	✗	✗	✓	✓
Connect Data Flow with Source Code	✗	✗	✗	✓

CT_1: APK ANALYZER, CT_2: JD-GUI,
CT_3: JD-GUI+FLOWDROID, TM: VisualDroid

2.2 VisualDroid

In this section, we introduce and justify the functionalities designed to facilitate our user study.

2.2.1 Justification of VisualDroid's functionalities

After we select the baselines for our study, we analyse the functions provided by each tool and then design a graphical representation to facilitate our user study. We have several functionalities identified in the visualisation platform, including CG-visual components, source code inspection, and APK comparison.

Visual components: This research aims to show the effectiveness and efficiency of graphical solutions based on comparing existing tools that utilise different code presentations. Therefore, VISUALDROID integrates three sub-visual components to facilitate the Android APK analysis in our controlled experiment, including CG visualisation, subgraphs visualisation, and node highlighting. To discuss the benefits gained from the visualisation, we decide to visualise the CG as our main visualisation resource. Firstly, based on the investigation of the code presentations provided by the available tools, the naming scheme for the functions in the CG is very similar to the scheme applied in other tools. In this way, we can control the information presented to our participants in different groups has the same types of data information. Besides, analysts who understand source code can easily read from CG. Considering not many participants are able to read from other types of source code information (*e.g.* smali code), CG is the best visualisation resource under our experiment settings. Apart from the visualisation of the CG, we also implement subgraph visualisation. The original CG of an APK contains up to thousands of nodes and edges, making it difficult to perform APK behaviour analysis. Subgraph analysis can help improve the efficiency of APK behaviour analysis. Additionally, to emphasise the security aspect, we use the highlight effect for sensitive API nodes. The highlight has been proved that it can effectively attract users' attention in previous studies [12], [13]. It can also help users quickly distinguish the highlight objects from other objects. Therefore, it is an effective solution to deliver visual information of the sensitive APIs in our design.

Interaction between CG and source code: While CG can effectively provide the relationship of the function calls in a single view, analysts can not entirely rely on the CG. Source code can offer some other detailed information, such as program statements and parameters. Program statements and parameters can be leveraged to commit malicious behaviours. Therefore, it is also important to have a further look at the source code when analysing the CG. To avoid inconveniences, we add the interaction between the CG and the source code to make it more user friendly. In this way, analysts can quickly locate the relevant source code while performing CG analysis.

APK comparison view: A study shows that 80% malware samples are built via repackaging other apps [14], for example, by simply unpacking a benign app and injecting malicious code before repackaging it. Therefore, the capability of comparing two similar apps is important to a static analysis tool. Existing tools used for manual analysis rarely provide a comparison function. Users usually need to open multiple windows or tabs to perform comparative analysis. Because we aim to discuss the effectiveness and efficiency of the visual solution, we integrate the comparative view of two APKs' CG with the comparison algorithm running at the back-end.

2.2.2 VisualDroid Design

We design and implement VISUALDROID, a visual solution for Android APK analysis to facilitate our user study. VISUALDROID is an online platform developed under the React

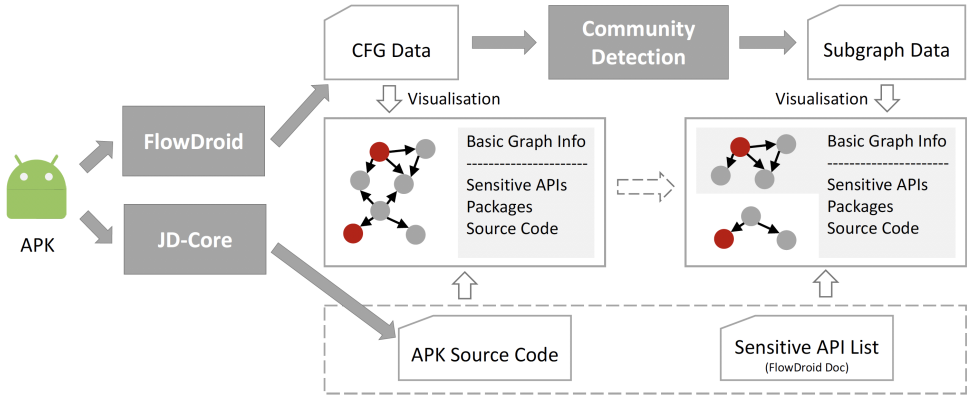

Fig. 2: The overall architecture of VISUALDROID.

framework[1].

Fig. 1 demonstrates an example user interface (UI) of VISUALDROID. The UI contains a visualisation panel (on the left of Fig. 1) and an information panel (on the right of Fig. 1). The visualisation panel displays the CG of the analysed APK. A CG illustrates the relationships between the caller and the callee of a function call with a directed graph. We highlight the sensitive APIs in red. The information panel provides the general information of the APK (*e.g.* the number of nodes and edges in the CG, a list of sensitive APIs, etc.) and the information of the selected subgraph. VISUALDROID supports a broad range of facilities to interact with, such as 1) CG visualisation, 2) source code inspection, and 3) APK comparison. In the following subsections, we introduce the workflow and the core functionalities of VISUALDROID in detail.

Visual components: CG is a directed graph that presents the function flow in the APK. VISUALDROID incorporates FLOWDROID [3] to decompile the APK first and generates its CG for the visualisation purpose at the first stage. Analysts can further choose to partition the CG by using a supported community algorithm (*e.g.* Infomap [15]). Then they can inspect the visualised subgraphs with the relevant information (*e.g.* sensitive APIs, packages, source code, *etc.*) to assist the APK analysis, as shown in Fig. 2. In VISUALDROID, after the test APK is decompiled into CG, it is visualised as a few 2-dimensional graphs. Each node denotes a function call, and each edge declares a caller-callee relationship between two connected function calls. The node at the arrowhead represents the function caller, and the node at the tail is the callee. To ease the APK inspection, VISUALDROID also provides subgraph analysis, in which the original CG is partitioned into several subgraphs by utilising a community detection algorithm. After the community detection, each generated subgraph has fewer edges and nodes. Users can then focus on the subgraph they are particularly interested in. The community network refers to the structure that nodes in the network can be categorised into different sets, and nodes in each set are densely connected internally. Previous studies [16], [17], [18] have demonstrated CG is the network that has the community features. Because Java is a significant object-oriented language, methods are grouped together in the same class or package to perform highly relevant operations among objects [19].

To facilitate the security analysis, we integrate the sensitive API list provided by FLOWDROID [3]. Users can find the sensitive APIs in the collapsible panel on the right side (see Fig. 1) or hover over the highlighted function node in the CG. Sensitive APIs (*e.g.* $requestLocationUpdates()$) are typically protected through different $Permissions$ in Android APKs [20]. If the user grants the permissions, the APK can utilise the sensitive APIs particular managed by those permissions. This security mechanism provided by the Android platform is to mitigate the threats caused by the misuse of sensitive APIs. The sensitive APIs can be utilised by malware writers to steal users' information, and the inappropriate use of sensitive APIs will also expose vulnerability to attackers. Because utilising sensitive APIs is one of the critical features of malicious apps, the function calls that are regarded as sensitive APIs are highlighted in red in the CG, and non-sensitive function calls are displayed in grey. Analysts can also hover over the node to view more detailed information, such as the package, the class name, the function name, and the type(s) of parameters it has.

The examples of subgraphs with highlighted function nodes show in Fig. 3. The subgraph in Fig. 3(a) shows the behaviour of creating an alarm, obtaining the current time, and then setting the alarm as a pending event. In Fig. 3(b), the app creates a web view to display the HTML content. The behaviour in Fig. 3(c) is to send a text message to a specific number without giving any notice to the users. Analysts can hover over different nodes and edges to inspect the app behaviours from the graphic view.

Interaction between CG and source code: VISUALDROID also supports source code inspection of the nodes in the CG, providing additional information besides the function call flow. When a node is selected, its source code will be displayed in the source code collapse panel. VISUALDROID utilises JD-CORE [1] to convert $.dex$ bytecode into readable $.java$ source code (see the details in Fig. 2), and then extracts the relevant code segments from the source code directory. As the name of function call specifies the package and the class it belongs to (*e.g.* com.software.application.Actor:write()), we can quickly locate the desired $.java$ file. After the target $.java$ file is found, our tool will search for the function that has the same name

1. https://reactjs.org

(a) Set an alarm (b) Create a web view content (c) Obtain $SmsManager$ with a subscription ID

Fig. 3: Subgraphs with highlighted function nodes

inside the target $.java$ file. Because Java has the polymorphism features, there might be several functions from the same $.java$ file having the same function name but different parameters. If there are results returned, VISUALDROID will further check the number, sequence, and parameter types of the method. Once the method is found, our tool will highlight the function in the source code, so that the analysts can quickly locate the relevant code segment and analyse the source code in the context.

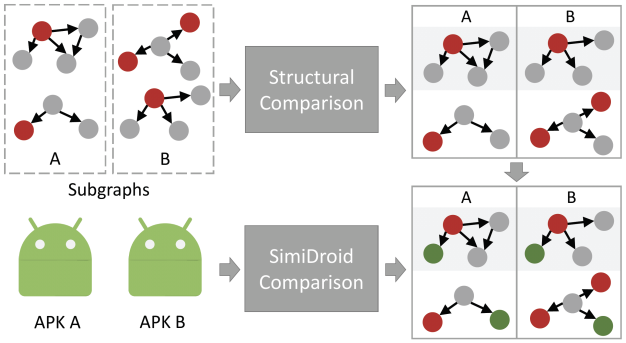

Fig. 4: VISUALDROID comparison flow graph.

APK comparison view: VISUALDROID provides two comparison strategies, namely code comparison, and structural comparison, to tackle different needs. The workflow is shown in Fig. 4. VISUALDROID integrates SimiDroid [21] for code comparison, which converts the APK's bytecode into an intermediate representation (i.e., Jimple [22]), and identifies the identical or similar methods in two APKs. However, due to the intrinsic limitation of static code analysis tools, the code comparison strategy is vulnerable to code obfuscation. Therefore, we also implement the structural comparison strategy, which can effectively tackle the code obfuscation problem. The structural comparison strategy takes the subgraphs from two APKs as input and then compares the sensitive APIs and the structure of the input graphs. We employ the graph similarity algorithm proposed in [18]. We first calculate the shortest path distance between the sensitive API nodes in the same subgraph. Then, we measure the similarity of the same sensitive API nodes in both subgraphs through a standard cosine metric. If two graphs have the same sensitive API(s) and a similar graph structure, they are most likely to perform the same behaviour. VISUALDROID displays similar subgraphs in pairs for further analysis.

2.3 Research Questions

This user study is designed to answer the following questions:

- **RQ1**: How effective is VISUALDROID compared to existing tools with other code representations in analysing Android APK behaviour, regardless of the Android APK complexity?
  Motivation: VISUALDROID enhances the behaviour analysis by providing CG visualisation and searching based on CG. We would like to investigate if VISUALDROID can assist in better behaviour understanding of the APK.
- **RQ2**: How efficient is VISUALDROID compared to existing tools with other code representations in analysing Android APK behaviour, regardless of the Android APK complexity?
  Motivation: VISUALDROID elaborates visual component, querying and source code inspection to facilitate the APK behaviour analysis. We would like to investigate if VISUALDROID can help analysts understand the APK behaviours faster.
- **RQ3**: Does the use of VISUALDROID increase the security awareness of users when performing APK behaviour analysis?
  Motivation: VISUALDROID provides security information (*e.g.* the usage of sensitive APIs) and highlights the relevant sensitive nodes in the CG. We would like to investigate if analysts will have increased security awareness when the security information is provided in a particular visualisation context.

3 USER STUDY SETUP

In this section, we present the details of the participant recruitment, the design and data collection of our user study, and the procedures conducted during the experiment. This will enable a better evaluation of the user study design and results, and allow for repeatability of the experiment as is, or reuse the experimental design for a variance experiment. Before we conducted the experiment, we obtained the ethics approval[2] from the CSIRO Social and Interdisciplinary Human Research Ethics Committee.

2. Ethics Clearance 192/19

TABLE 2: Experimental groups.

Tool	Description
VISUALDROID (TM)	The source code of the APKs is presented in a directed graph view.
APK ANALYZER (CT_1)	The source code of the APKs is converted into tree management view with packages, classes and method information.
JD-GUI (CT_2)	The source code of the APKs is reconstructed into a readable version and user can access to methods and fields.
FLOWDROID +JD-GUI (CT_3)	CG is generated by FLOWDROID in a text manner. The source code of the APKs is reconstructed into a readable version and user can access to methods and fields.

3.1 Recruitment

Our study is mainly designed for junior analysts who have basic IT backgrounds. To investigate how visualisation can help junior IT people perform APK behaviour analysis. We recruit 40 university students with a software engineering major and who have experience with Java programming. All the participants are required to be over 18 years old, and they do not receive any IT security training or an IT security degree before the experiment. We advertise the project and distribute an informational handout at the participating universities. Potential participants contact us via email after checking the requirements for participation specified on the handout. Participants who provided written consent were given an information sheet that specifies the tasks in the user study. It is voluntary to participate in this experiment, and participants' consent is implied by signing the consent form. A gift card valued at 40 dollars is presented to each participant after successfully completing the entire user study.

3.2 Experiment Design

In this subsection, we detail the design of the experiment, including the APK behaviour analysis tests involved in the experiment, how we select the test APKs, and the APK behaviours listed in the answer sheet.

3.2.1 Test $Single$ and $Comparison$

We regard the behaviour of an APK as the way the APK works or functions. In the APK behaviour analysis tests, there are two different tests, a single APK analysis test (referred to as *Test Single*) and a comparison test (referred to as *Test Comparison*). The *Test Single* is to simulate a general APK behaviour analysis in real-life practice, and *Test Comparison* is mainly designed for the analysts if they focus more on evolution and piggyback perspectives.

An answer sheet is prepared, consisting of a table for each test. Examples of the tables in *Test Single* are shown in Table 3 and Table 4 respectively. Table 6 shows a list of APK behaviours that is provided for both *Test Single* and *Test Comparison*. Participants are required to identify if the given behaviours exist in the test APKs, and mark the suspicious behaviour based on their instinct. After completing the first table, participants are required to complete the second table (cf. Table 4) and provide evidence if they reckon the behaviours exist. The answer sheet for *Test Comparison* has similar tables to *Test Single*, but participants need to clearly specify which APK has the behaviour(s) and indicate in which APK the existing behaviour is suspicious. We further explain the experiments' focus on each test as follows:

TABLE 3: *Test Single* Table 1 example.

Id	APK Behaviours	E	S
1	Set the colour of the text in an alertbox	✓	
2	Listen to a location change	✗	
3	check network status and then send text message	✓	✓
...	...	...	...

E: ✓if this behaviour exists, otherwise ✗.
S: ✓if the existing behaviour is suspicious, otherwise ✗.

TABLE 4: *Test Single* Table 2 example.

Id	APK Behaviours	E	R
1	Set the colour of the text in an alertbox	e1	
2	Listen to a location change		
3	check network status and then send text message	e2, e3	r
...	...	...	...

E: if this behaviour exists, please list all the relevant evidence.
R: why do you think this existing behaviour is suspicious?

Test Single: In *Test Single*, participants are asked to analyse a malicious APK on its own, and gain insights from the following APK analysis tasks:

- Identify APK behaviours and provide evidence.
- Identify if the behaviours are suspicious.

We will examine the data from the following aspects:

- Observe the time used in completing the first table (i.e., Table 3).
- Count how many behaviours are correctly identified by participants with adequate evidence.
- Count how many behaviours are identified as suspicious behaviours.

Test Comparison: In *Test Comparison*, two APKs are provided, with one being an evolved malicious version of the other. Participants are supposed to gain insights from the following APK analysis tasks:

- Identify APK behaviours from both APKs and provide adequate evidence of these behaviours.
- Describe the unique behaviours of each APK and the common behaviours.
- Identify the malicious APK.
- Infer which APK is the later version with evidence.

We will examine the data from the following aspects:

- Observe the time spent in completing the first table.
- For each APK, count how many behaviours are correctly identified with sufficient evidence by participants.
- Observe whether participants can infer the correct evolutionary relationship with proper explanations.
- Observe whether participants can correctly describe how the later malicious APK evolved in terms of the security aspect.

TABLE 5: Selected test APKs.

Complex Level	APK ID	No. Class	No. Method	Obfuscation
1	B_{Single}	59	291	NO
	A_{Cmp1}	39	168	
	A_{Cmp2}	37	142	
2	A_{Single}	148	843	YES
	B_{Cmp1}	130	657	
	B_{Cmp2}	157	744	.

3.2.2 Test APK Selection

We select our candidate APKs from the AndroZoo dataset [23], which collects APKs from various sources, including official Google Play and alternative app markets. Candidate APKs for *Test Single* are selected from the AndroZoo dataset. Candidate APK pairs used in *Test Comparison* are randomly chosen from a list of piggybacked APK pairs as suggested in [24]. We obtain the MD5 hashes of the APK pairs and retrieve the APK files from the AndroZoo dataset. Selected APKs are decompiled and carefully inspected before analysis to ensure no program exceptions due to decompilation failure. We also excluded APKs with a large size (*i.e.,* LOC > 50k) to limit the time required for our user study. More importantly, to discuss the impact caused by the complexity of the test APKs, we examine the candidate APKs in terms of the number of classes and methods and obfuscation technology they have.

In the end, we select six APKs and cluster them into two sets according to their levels of complexity, as shown in Table 5. Set 1 has less complicated APKs with 45 (Max: 59, Min: 37) classes and 200 (Max: 291, Min: 142) methods on average. APKs in Set 2 are more complicated with more classes, methods, and obfuscated code. They have 145 classes (Max: 157, Min: 130) and 748 methods (Max: 843, Min: 744) on average. Furthermore, APKs in Set 2 have a large portion of code obfuscated with identifier renaming. All test APKs used in our experiment are under licensed use and are not executed during the experiment due to ethical conditions.

3.2.3 Behaviour Selection

During the experiments, participants need to identify if the behaviours listed in the answer sheet exist in the test APKs. To mitigate the bias and ensure that the behaviours listed on the answer sheet can be gathered by leveraging all the comparing tools (i.e., APK ANALYZER, JD-GUI, FLOWDROID, and VISUALDROID), we analyse the APKs with all the tools mentioned above and select the behaviours that can be identified by all of them.

To prepare the behaviour list in *Test Single*, 18 candidate behaviours are selected and only half of them exist in the given APK. All of the behaviours contain at least one sensitive API to examine participants' security awareness. In *Test Comparison*, we have 20 candidate behaviours. Five behaviours belong to both test APKs, two groups of five behaviours belong to each test APK only, and the other five behaviours do not exist. Three authors of this paper whose speciality lies in Android malware analysis are asked to vote and form the final behaviour list.

For both *Test Single* and *Test Comparison*, we obtain a list of six behaviours at the end. To ensure participants can gain a better understanding of the test APKs, we further check if the number of candidate behaviours that exist is equal or more than three in each test APK, otherwise, we replaced the behaviours that did not exist. The final behaviours we selected are shown in Table 6, we divide them into three categories and also explained the rationals.

3.3 Experimental Group

We identify dependent and independent variables in our user study to evaluate if VISUALDROID is a better solution than other visual tools in assisting analysts to comprehend Android APK behaviours (including security issues). The independent variables identified are 1) the tools that are utilised in Android APK behaviour analysis and 2) the complexity of the APKs. The dependent variables in our user study setting are 1) the grades achieved in the APK behaviour analysis tests, 2) the time spent on APK behaviour analysis when the participants use different tools and analyse APKs with different complexity, and 3) the index indicating their security awareness when performing APK behaviour analysis.

As shown in Table 2, we obtain four large groups based on the tool they used for the experiments. In our experiments, participants are randomly divided into these four groups, with 10 participants in each group. Control group 1 (CT_1) utilise APK ANALYZER as the analysis platform. Control group 2 (CT_2) is provided with pre-decompiled APK $.jar$ files and utilised JD-GUI. Control group 3 (CT_3) is required to elaborate the results from both JD-GUI and FLOWDROID. The treatment group (TM) employ VISUALDROID to perform APK analysis. Participants are unaware of which tool was developed by us.

Each large group is further divided into two subgroups with five participants each to participate in either session A or B. To ensure that every subgroup spends time approximately evenly in the experiments, we mix APKs with different complexity levels for two tests (i.e., *Test Single* and *Test Comparison*) in the experiments (cf. Section 3.2.1). As shown in Table 2, participants who take part in experiment session A received the test APK (A_{Single}) with level 2 complexity in *Test Single*, and two APKs (A_{Cmp_1} and A_{Cmp_2}) with level 1 complexity in *Test Comparison*. Vise versa, participants who take part in experiment session B were assigned to analyse B_{Single} (complexity = 1) in *Test Single*, and two APKs B_{Cmp_1} and B_{Cmp_2} with level 2 complexity in *Test Comparison*.

3.4 Experiment Data Collection

Here, we explain how we collect data during the experiment.

3.4.1 Answer Sheet Data

We design a scheme to convert the collected answers into quantitative information. There are three types of quantitative information collected in our experiment.

The first one is the grades achieved from the APK behaviour identification by each participant. For example, in *Test Single*, if the participants correctly identify whether a

TABLE 6: Selected behaviours.

Category	Behaviours	Rationales
User Interface Related	Set up a web view to display the HTML content	This behaviour involves some sensitive APIs related to the built-in web view. Some APKs utilise this to display advertisement aggressively or to cover malicious trace at the background
	Create a dialog box with "Yes" and "Cancel" buttons, when any button is clicked by the user, the predefined UI will always be displayed.	If a button is wired with a listener, it trigger the code written inside the $onClick()$ after the user presses the button. It can be possibly utilised by malware.
	Set up a listener to a "Yes" button, and then set the visibility of text views	
	Create a button and set a listener to the button, and change the span style of the text view	
	Monitor if user has pressed the keyboard	This behaviour can give the information if the key down event is occured, it can be possibly utilised by malware.
	Integrate a calendar for user to check the date	Information obtained from the calendar can be possibly utilised by malware.
Data Sources and Sinks	Request the features of the app window	This behaviour is to ask the system to include or exclude some of windows UI features (*e.g.* toolbar, actionbar).
	Obtain device location based on a set of criteria	To provide a better service, some APKs need the location information, but it is considered as dangerous to share location to untrusted APKs.
	Obtain user device's location and also request the location update	
	Register the device on a remote server, or obtain the information from SharedPreferences and then unregister the device from the remote server	Store or obtain information in the device database or sharedPreference. Malicious APKs can utilise these to store and obtain sensitive information.
	Create database and record the status of interface	
	Check country code and operator code, and then send text message	This will result in sending message automatically. Malware can send premium text message or possibly leak personal information through text.
	Utilise Dex class loader to load a user defined class from external library	It enables APK developers to load external libraries and invoke methods directly from external. Malware can utilise DexClassLoader to cover its malicious intent inside the external library.
	Download APKs from the given links	An APK downloads other APKs from a list of links. Malware can download unwanted APKs secretly without user's permission.
Condition control	Check if the airplane mode is on	When APK detects that user is back on service, APK can then perform some tasks, which typically require the internet. Malware sometimes need to check this condition before they perform some malicious behaviour related to internet.
	Create an alarm, get the current time, and set the alarm as a pending event	APK can trigger some events at a specific time. Malware can therefore set predefined task at specific time to evade the dynamic detection.

certain behaviour exists in Table 3 and provide adequate evidence in Table 4, they will be awarded one score. Moreover, participants need to carefully identify all the sub-behaviours involved in every behaviour listed in the table and provide relevant evidence. For example, in Table 3, the third row includes two sub-behaviours: 1) check network status, and then 2) send text message. If the participants think this behaviour exists, they need to provide evidence for both sub-behaviours. Any incomplete or incorrect evidence results in losing a score. Besides, participants are required to provide the suggestion for each behaviour. They need to tick the behaviour if it exists and cross out the behaviour if it does not exist. If no suggestion is provided, it also will result in losing a score. In both *Test Single* and *Test Comparison*, there are six behaviours listed for participants to judge if they exist. Only one APK is provided in *Test Single*. Therefore, the total score for *Test Single* is six. In *Test Comparison*, participants need to analyse two test APKs, and judge if the listed six behaviours exist in both APKs. Therefore, the total score for *Test Comparison* is twelve.

Apart from the grades obtained from behaviour analysis, *Test Comparison* has additional two open questions. We want to further investigate the APK evolution or repackage analysis based on the visualisation solution. In the first question, if the correct evolutionary sequence is identified with correct reasons, one score will be rewarded. In the second question, if malicious changes are correctly identified, then they can have one score. The total grade for the open questions is two.

The third type of quantitative information is the number (N_{SB}) of behaviours that are regarded as suspicious. All the listed behaviours contain at least one sensitive API, which might lead to a security compromise. Our participants are supposed to link sensitive APIs with potential security issues. It is subjective that the user might judge differently by their intuition when they perform the APK behaviour analysis, and a larger number can reflect that the participant has a higher security awareness.

The participants in different groups need to provide the evidence based on the tools to complete the second table in the experiment (i.e., Table 4). Control group 1 (CT_1) needs to specify the supporting functions including the entire package path, the class it belongs to, and the parameters it holds. Valid evidence for CT_1 shows in Example 3.4.1.

EXAMPLE 3.4.1: CT_1 VALID EVIDENCE

Package.Subpackage...Class: void function(param,...)

For control group 2 (CT_2), participants need to provide the relevant code segment where the core function is called. Participants are also required to highlight the core function related to the existing behaviour. An example of valid evidence is shown in Listing 1.

```
23 private void a(){
24     Uri localUri = Uri.fromFile(new File(this.b));
25     Intent localIntent = new Intent("android.intent.
        action.VIEW");
26     localIntent.setFlags(268435456);
27     localIntent.setDataAndType(localUri,
28     "application/vnd.android.package-archive");
29     startActivity(localIntent);
30     finish();
31 }
32
33 public void onCancel(DialogInterface
        paramDialogInterface){
34     paramDialogInterface.dismiss();
35     a();
36 }
37
38 public void onClick(DialogInterface paramDialogInterface
       , int paramInt){
39     paramDialogInterface.dismiss();
40     a();
41 }
```

Listing 1: An Example of Valid Evidence for Control Group 2 (CT_2).

For control group 3 (CT_3), since participants are required to elaborate the result obtained from FLOWDROID and JD-GUI, they can either provide the relevant code segment as shown in Listing 1, or the screenshot specifies the relevant caller-callee relationship as shown in Example 3.4.2.

EXAMPLE 3.4.2: CT_3 VALID EVIDENCE

com.geinimi.ads: boolean b() -> java.util.Date: void ()

Participants who belong to the treatment group (TM) are required to take a screenshot of the subgraph as evidence. The screenshot must show the ID of the subgraph, and the name of the core function that is highly related to the behaviour. An example of valid evidence for TM shows in Fig. 5.

3.4.2 Completion Time

During the experiments, we only record the time that participants spend on completing the first table, as the time spent on the second table may vary, and recording of this may introduce bias. For example, some participants may prefer to complete the evidence as sentences and tend to write more detailed answers, whilst others may prefer listed points to summarise their answers. Therefore, instead of timing the whole test, we stop the timer when the participants finish the first table. Participants are not allowed to fill out the second table until the first table is completed. We set the maximum time to complete the first table at 20 minutes in *Test Single* and 30 minutes in *Test Comparison*. Once the time limit is reached, participants have to proceed to complete the second table.

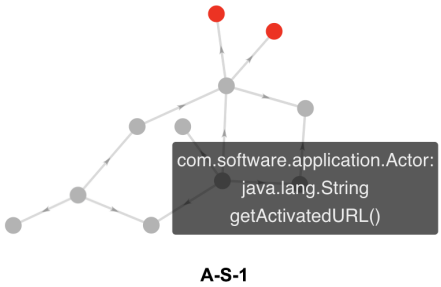

Fig. 5: An Example of the Valid Evidence for the Treatment Group (TM).

3.5 Procedure

The user study includes an experiment and a follow-up interview. The experiment and the interview are conducted consecutively with only one participant at a time. Both are conducted in person.

Before commencing the experiment of the user study, we introduce the project to the participants and provide an information sheet with details about the study. After participants read the information sheet and confirmed that there is no confusion, we provide them with a consent form. The experiment starts after the participants agree to participate and signed the consent form.

At the start of the experiment, we hand out the experiment instruction sheet and provided a quick tutorial. We explain the technical terms in the document. During the tutorial, we first show the participant how to utilise the analysis tool to perform the app behaviour analysis. Then participants are required to follow the experiment instructions and analyse the sample APK by the given tool. The sample answer sheet in the tutorial is pre-filled with correct answers for reference. Participants are welcome to ask any questions during the tutorial. After the experiment starts, participants are required to utilise the given tool to analyse the test APKs and complete the answer sheet. To mitigate the bias, participants can only utilise the resource we provide them during the experiment, In such a case, we can validate the benefits from visualisation without errors caused by unexpected operations of participants, which we might not be aware of during the empirical studies. Every participant is assigned a random ID. All the data produced by the participant are logged under that ID. During the experiment, participants are free to ask clarification questions, not related to the answers.

After the participants successfully complete *Test Single* and *Test Comparison* in the experiment, we invite them to the follow-up interview. The purpose of the interview is to understand the results we gather from the experiment and improve the current version of VISUALDROID. The interview consists of two parts. In the first part, the researchers ask the participants if they have ticked or crossed the behaviours they intended to. Then researchers validate each answer

(evidence and reason) they provide in the second table (cf. Table 4) to mitigate the bias caused by the misunderstanding of the answers. In the second part, participants are required to leave feedback, including sharing their user experience and what they have learned during the experiment. We ask the participants to share the obstacles that they encounter during the experiment. Participants from the VISUALDROID group are asked to rank the functionalities according to how helpful they were in APK behaviour analysis. Participants are also required to give advice on the functionalities, such as how they can be improved. We then ask the participants what interesting insights they have gained about Android app security from our experiment.

4 RESULTS

We have two independent variables (i.e., the tool used in the experiment and the complexity of the test APKs) and two dependent variables (i.e., grades and the completion time). In this section, we present the results of these two independent variables.

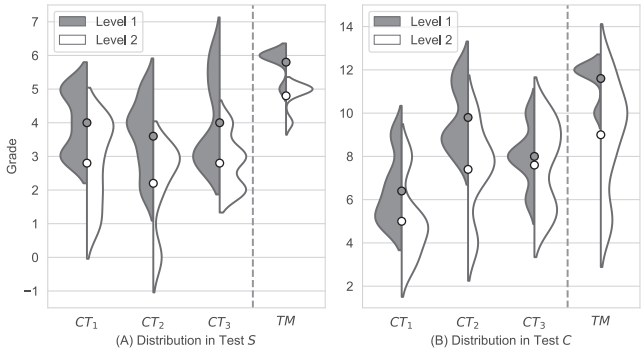

Fig. 6: The distribution of the grades achieved by using APK ANALYZER, JD-GUI, JD-GUI+FLOWDROID, and VISUALDROID in (A) *Test Single* and (B) *Test Comparison*.

4.1 RQ1: Effectiveness of VisualDroid

In RQ1, we present the grades obtained under different experiment settings and analyse the results to reveal the insights regarding the effectiveness of VISUALDROID.

4.1.1 The effect of tool and APK complexity on grade

Fig. 6 shows the distribution of the grades by using different tools in *Test Single* and *Test Comparison*. Each test has two groups that use APKs with either level 1 complexity or level 2 complexity (illustrated in grey and white, respectively). The mean value of each group is marked with a circle filled with the corresponding colour.

In all four different experimental groups, the mean grades (G_{S_1} and G_{C_1}) of the tests that provide less complicated APKs always score higher than the tests that provide complicated APKs (G_{S_2} and G_{S_2}). The VISUALDROID group obtains the highest grades in both *Test Single* and *Test Comparison*, regardless of the complexity levels of the test APKs in the behaviour analysis. Specifically, in *Test Single*, the average grade of G_{S_1} is 5.8 out of 6, while the average grade of G_{S_2} is 4.8 out of 6. In *Test Comparison*, VISUALDROID scores 11.6 (G_{C_1}) and 9 (G_{C_2}) out of 12 in test APKs with level 1 and 2 complexity, respectively. In *Test Single*, APK ANALYZER and JD-GUI+FLOWDROID score the second highest (i.e., G_{S_1} = 4, G_{S_2} = 2.8) with a slight advantage over JD-GUI (G_{S_1} = 3.6, G_{S_2} = 2.2). However, in *Test Comparison*, JD-GUI outperforms APK ANALYZER and JD-GUI+FLOWDROID, and obtains mean scores of 9.8 (G_{C_1}) and 7.2 (G_{C_2}) in test APKs that have the complexity level 1 and 2, respectively. Overall, there is an increase of 35.7% in the grade achieved by VISUALDROID over JD-GUI, 39.2% in the grade over JD-GUI+FLOWDROID, and 71.4% in the grade over APK ANALYZER. Participants who use VISUALDROID outperform other participants in both *Test Single* and *Test Comparison*. While VISUALDROID group achieves best grades in both tests, a distinct advantage over other tools is observed in *Test Single* than in *Test Comparison*. We further apply the Wilcoxon rank-sum test [25] to measure whether the difference between the grade achieved by the four tools is significant. The result suggests that the group using VISUALDROID achieves significantly better grades compared to APK ANALYZER ($p < 0.001$), JD-GUI ($p = 0.016$) and JD-GUI+FLOWDROID ($p = 0.015$). Meanwhile, APK ANALYZER, JD-GUI and JD-GUI+FLOWDROID perform similarly in assisting users with app behaviour analysis ($p > 0.13$ for each of the three pairs). Additionally, we compare the grades of test APKs with different complexity. We find participants analyse APKs of complexity level 1 more accurately than those with complexity level 2 ($p = 0.013$).

4.1.2 The effect of tool on APK evolution analysis

We calculate the scores of the open questions in *Test Comparison*. We find that participants using APK ANALYZER are less likely to infer the correct evolution direction and identify the updates regarding malicious intent. They obtain 0.8 on average when less complicated APKs are provided and scored 0 on average when more complicated APKs are used in the test. Participants using JD-GUI achieve 1 out of 2 when less complicated APKs are provided and 0.4 out of 2 when complicated APKs are provided in *Test Comparison*. In JD-GUI+FLOWDROID, participants obtain 1.2 and 0.8 respectively in the test that provides less complicated APKs, and the test that provides more complicated APKs. VISUALDROID participants show the best result in identifying the evolution direction and malicious updates, they obtain a score of 1.4 out of 2 when less complicated test APKs are used in the test. The mean grade reaches 1 when increasing the complexity of test APKs.

> VISUALDROID subgraph visualisation analysis has an advantage over other forms of visualisation of effectively presenting the caller-callee relationship among functions. It improves APK ANALYZER, JD-GUI, and JD-GUI+FLOWDROID by 71.4%, 35.7%, and 39.2% in terms of the effectiveness of behaviour analysis.

4.1.3 Reflection on the grades

We further investigate the reasons why different tools result in different grade distributions in APK behaviour analysis, as well as the rationale for the complexity of the test APKs

TABLE 7: Reasons of why the score is deducted.

Category	Description
Not Found	Participant does not correctly identify an existing behaviour
False Label	Participant wrongly identifies a behaviour that does not exist
Incorrect	Participants tick an existing behaviour, but they do not provide the correct evidence
Incomplete	Participants tick an existing behaviour, but the evidence is not fully provided
No Label	The behaviour is not labelled as *Exist* or *Does not exist*, because the participant exceeds the time limitation in the experiments

having less impact on the grade obtained by VISUALDROID compared with the other two tools. We extract the incorrect answers from the collected answer sheets and categorise these into five groups based on the reasons why the grade is negatively affected (*i.e.* points are deducted from the overall grade). We conclude five categories: 1) Not Found, 2) False Label, 3) Incorrect, 4) Incomplete, and 5) No Label, as shown in Table 7. In the end, we collect 89, 65, 68 and 24 answer samples in the groups of APK ANALYZER, JD-GUI, JD-GUI+FLOWDROID, and VISUALDROID respectively. The categorisation result is shown in Fig. 7. The pie chart (A) illustrates the proportion of the reasons categorised based on the tools used by participants. The bar chart (B) shows the distribution of the deducted scores categorised by five different reasons.

As seen from the pie chart in Fig. 7 (A), APK ANALYZER and JD-GUI have a similar distribution of reasons in the deducted score, participants from both groups struggle most with providing correct answers, accounting for 38.0% and 33.8%, respectively. 31.5% of the deducted scores from the APK ANALYZER group (CT_1) are attributed to the behaviour not being found, and 21.3% is because participants identify non-existing behaviours. The second common mistake for participants using JD-GUI (CT_2) is identifying behaviours that do not exist (29.0%), followed by the "Not Found" category (26.2%). The "Incomplete" category shows a relatively low percentage in both groups, and the "No label" category only features in JD-GUI. In JD-GUI+FLOWDROID group, except the "No Label" category, the other four categories account for approximately a quarter each. Most answers are categorised as "Incorrect"(33.8%). Compared to other groups, most issues within the VISUALDROID group are caused by wrongly identified non-existing behaviours, followed by the "Not Found" (37.5%) and "Incorrect" category (20.8%).

The bar chart in Fig. 7 (B) emphasises the comparison of each reason's distribution among the four tools used in the experiments. It suggests that the group using APK ANALYSER experiences a larger deduction in overall grade as a result of "Incorrect", "Incomplete", "False Label", and "Not Found" answers. Participants from the JD-GUI and JD-GUI+FLOWDROID groups achieve a slightly better performance, with fewer incorrect, incomplete answers and more existing behaviours found by the participants. VISUALDROID participants made the smallest number of mistakes which only included three cases: 1) behaviour is provided with incorrect evidence, 2) a non-existing behaviour is marked as "Exist", and 3) existing behaviour is not found.

We further reference the participants' feedback from the three control groups (CT_1, CT_2, CT_3) to explain the reasons why different groups have different types of wrong cases in the behaviour analysis, and why VISUALDROID assists better in APK evolution analysis. **JD-GUI** provides direct access to the entire source of the Java files, which requires participants to have a strong ability to read the source code and understand the logic flow. Even though JD-GUI provides the links between functions and classes, participants need to click the function name or the class name to inspect the connections between a caller and a callee. They have to go back and forth to sort out the relationship among the function calls. After several re-directions from one method to another, participants usually lose track of the logic flow and miss the relevant information. Because JD-GUI does not visualise the relationship. The grade decreases the most compared to the other tools when the source code is obfuscated, and most of the methods do not have a reasonable name (*e.g.* a.a.c()). The above reasons mainly contribute to the cases of "Incorrect" and "Incomplete", and JD-GUI is the only group that has "No Label" cases.

In the **APK Analyzer** group, the source code file ($class.dex$) is presented in a tree structure. Classes are grouped under the corresponding package folders, with all the functions called inside the class listed with their names only. Therefore, this tree-based code presentation is difficult for participants from the APK ANALYZER group to obtain the connections among methods. Since insufficient context is provided in APK ANALYSER, participants are more likely to consider an incorrect method as the evidence of specific behaviour, because the method might have a confusing name that is similar to the target behaviour' description. Participants might mistakenly think one behaviour exists or think one method is the correct evidence of certain behaviour. This is the reason why there is a large number of wrong cases in the "Incorrect" and "False Label" categories in APK ANALYSER.

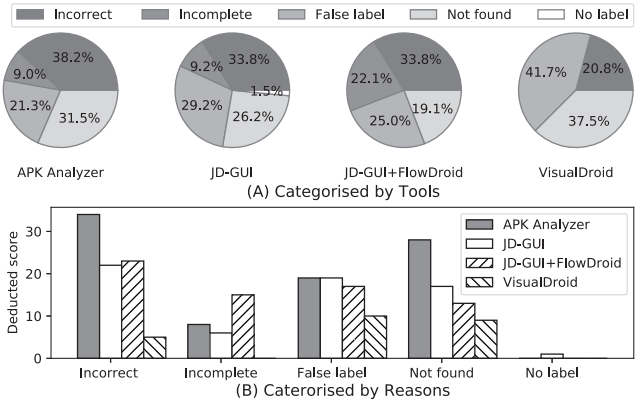

Fig. 7: Distribution of the reasons why the grade is deducted.

Similar to the **JD-GUI+FlowDroid** group, a large number of "Incorrect" and "False Label" cases occur when participants highly rely on the data flow of APK behaviour analysis. Providing only one caller and callee information

at a time makes it difficult to judge if relevant behaviour is found. As for the participants elaborating the results from both JD-GUI and FLOWDROID in control group 3, they suggest the importance to have interaction between JD-GUI and FLOWDROID, so they could be directed to the relevant source code when a certain function is clicked from FLOWDROID.

VisualDroid has ten cases of "False Label", constituting 41.7% of its total cases. We carefully examine the answer sheet and find that four participants think the $Calendar$ object in Android only refers to the digital Calendar we interact with. However, in the test APK, this $Calendar$ object is to obtain the time rather than create a digital calendar. Another example is that two participants think the function $getLastKnownLocation()$ in Android refers to a location update request. Thus, the lack of knowledge of the Android platform code caused the deduction of ten scores. However, VISUALDROID provides a visualisation solution where participants can interact with the subgraph. Callers and callees are connected with arrows, and participants can inspect a sequence of sub-behaviours the connections between the nodes all at once without going back and forth between different $.java$ files. The visualisation leads to the fewest cases of "Incomplete".

The category of "Not Found" constitutes a relatively large portion of all four groups. It is mainly because of insufficient search functions in all four different tools. JD-GUI does not support fuzzy search. Therefore, participants need to switch between uppercase and lowercase. Sometimes, the participants also need to try to match the exact string that exists in the source code. APK ANALYZER and VISUALDROID support fuzzy search, and relevant content will be highlighted when participants try to search the function by keywords, but APK ANALYZER, JD-GUI+FLOWDROID and VISUALDROID do not rule out the irrelevant content. Participants can easily miss the key content if the test APK contains a lot of information. However, as we can see in the bar chart, VISUALDROID introduce the lowest cases in the "Not Found" category compared to other tools. Even though VISUALDROID can not rule out the irrelevant content, participants can choose to analyse the subgraphs that contain sensitive API(s) only, in this case, it will reduce the large number of subgraphs to be analysed. Therefore, instead of searching the relevant information in the entire APK, VISUALDROID can remove less sensitive subgraphs for the analyst and then reduce the cases of "Not Found".

In the comparison analysis test, participants from three control groups (CT_1, CT_2, and CT_3) mention that they have to constantly switch between tabs or windows to compare the APKs because the three tools do not support separate views for different APKs. To ease the comparison analysis between APKs, VISUALDROID separates the visualisation panel into two columns, where each column demonstrates a list of subgraphs for one APK. Additionally, because a similar pair of subgraphs are organised in the same row, participants can further analyse the differences by comparing the pairs of subgraphs instead of browsing the entire APK source code information.

4.2 RQ2: Efficiency of VisualDroid

In RQ2, we reveal how the selection of the tool and the complexity of the test APKs affect the completion time, and then we reflect on the results that we obtained.

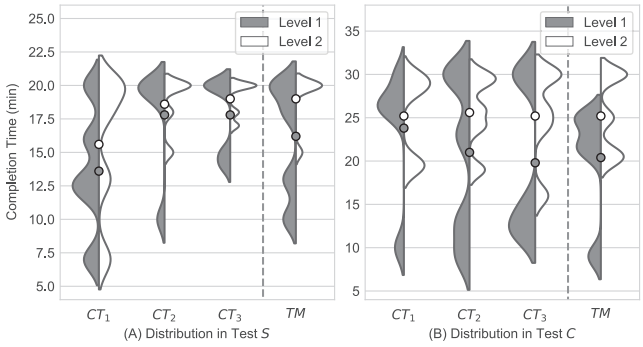

Fig. 8: The distribution of completion time in using APK ANALYZER, JD-GUI, JD-GUI+FLOWDROID, and VISUALDROID in (A) *Test Single* and (B) *Test Comparison*.

4.2.1 The effect of tool and test APKs on completion time

Fig. 8 shows the distribution of the completion time in terms of tools used and the complexity of test APKs. The grey areas denote the completion time distribution when less complicated APKs are provided, and the white areas indicate the distribution when more complex APKs are provided. The mean value of each group is marked in a circle filled with a corresponding colour.

Overall, the participants spend more time on the tests that provides more complex test APKs in all four experimental groups. The VISUALDROID group (TM) does not show an advantage over other groups in terms of the time spent on *Test Single* and *Test Comparison*. We notice that APK ANALYZER participants spend relatively less time (T_{S1} = 13.6 minutes, T_{S2} = 15.6 minutes) on *Test Single*, compared to other tools, regardless the complexity of the test APKs. However, in *Test Comparison*, the mean completion time shows no advantage over others (T_{C1} = 23.8 minutes, T_{C2} = 25 minutes). Participants from JD-GUI (T_{S1} = 17.8 minutes, T_{S2} = 18.6 minutes, T_{C1} = 21 minutes, T_{C2} = 25.6 minutes), JD-GUI+FLOWDROID (T_{S1} = 17.8 minutes, T_{S2} = 19 minutes, T_{C1} = 19.8 minutes, T_{C2} = 25.2 minutes). Compared with JD-GUI+FLOWDROID, VISUALDROID (T_{S1} = 19 minutes, T_{S2} = 20.4 minutes, T_{C1} = 19.25 minutes, T_{C2} = 25.2 minutes) spend similar amount of time. We further use the Wilcoxon rank-sum test [25] to compare the completion time spent by the three tools. The result indicates there is no significant difference between APK ANALYZER, JD-GUI, JD-GUI+FLOWDROID and VISUALDROID in time spent ($p > 0.29$ for any pair). Besides, analysing APKs with higher complexity level cost more time for completion ($p = 0.036$).

> VISUALDROID participants who only employ the visual component achieve higher grades in less time compared to other participants from the other visual groups.

4.2.2 Reflection on completion time

Different from the other tools used in our control groups (CT_1, CT_2, CT_3), VISUALDROID provides various functionalities regarding to the app behaviour analysis, (*e.g.* subgraph analysis, source code inspection). To further investigate what functionality helps save time spent on app analysis, we asked our participants to share the time allocation for each functionality. The results suggest that all participants rely on the functionalities of the subgraph analysis and source code inspection, but their focus is different. Three participants from user study session A and one participant from session B rely more on the source code inspection than the subgraph analysis. The completion time distribution of VISUALDROID is shown in Fig. 8. We notice they all have two projecting portions at the head and the tail of the violin plot. Only one small projecting area is observed in the middle of the violin plot from *Test Comparison* (APK complexity = 2). The projecting area at the head denotes the completion time of the participants who rely mainly on the source code inspection, while the projecting area at the tail represents the completion time of the participants who rely more on the subgraph analysis. We further extract their grades accordingly and find that participants who employ source code inspection increased their grades by 7.45% overall.

Since VISUALDROID and JD-GUI both provide direct access to the source code for participants to inspect the methods and fields, we further compare the time spent on the tests that utilise JD-GUI and VISUALDROID. Participants from the VISUALDROID group who rely more on source code inspection spent more time (2.5%) compared to the participants from JD-GUI. Because they mentioned they check the information from both resources (source code and the CG), less completion time is observed for participants who focus more on subgraph analysis and rarely rely on source code with a decrease of 12.6% overall compared with the JD-GUI group. Moreover, participants from VISUALDROID, regardless of the functionalities they focus on, achieve higher overall grades compared to participants using the traditional tool JD-GUI. Therefore, the visualisation component of VISUALDROID can help increase the effectiveness and efficiency of APK behaviour analysis.

4.3 RQ3: Security Awareness Enhancement by Using VisualDroid

VISUALDROID is designed to provide security-related information as guidance to a security analyst, to increase the security awareness of the analyst. To address this question, we specifically target individuals who do not receive any formal security training or hold an IT security degree to participate in our study. In our study, we record the number of behaviours (N_{SB}) that are labelled as suspicious in the Table 3. We look into the numbers of selected suspicious behaviours in different experiment sessions to compare participants' security awareness among the four groups. The result shows that high security awareness is observed in the VISUALDROID group in both experiment session A and B, whilst participants from the APK ANALYZER and JD-GUI groups do not show deep concern of the behaviours that contain sensitive APIs. A slight increase in security awareness is noticed in JD-GUI+FLOWDROID. The numbers of labelled suspicious behaviours in the VISUALDROID group in both session A (N_{SB} = 30) and B (N_{SB} = 21) are higher than any number obtained from the APK ANALYZER, JD-GUI, and JD-GUI+FLOWDROID group. In experiment session A, the number of selected suspicious behaviours collected from APK ANALYZER and JD-GUI is the same (N_{SB} = 13). More behaviours are labelled as suspicious in JD-GUI+FLOWDROID group (N_{SB} = 17). In experiment session B, participants have labelled 13 and 15 suspicious behaviours using JD-GUI and JD-GUI+FLOWDROID, respectively. In comparison, only seven behaviours are marked as suspicious when leveraging APK ANALYZER. The result indicates that listing the sensitive APIs' information and highlighting the sensitive API nodes might increase the security awareness of the analyst.

We also receive feedback from the VISUALDROID participants who share what they have learned during the experiments in terms of the security aspect. We list selected feedback below.

- "I got some ideas about what suspicious behaviours are."
- "I have learned what kind of functions can be sensitive, while sensitive functions can be suspicious but it does not necessarily mean it is malicious."
- "Some existing functions provided by Android platform might be leveraged for implementing malicious behaviours."
- "Android programmers are easy to code for the purpose of hacking. Users should be careful about the access of android apps to protect their privacy."
- "Sensitive APIs are not necessarily malicious; it depends on when/where/how they are invoked."
- "Only knowing whether an API exists cannot determine whether the app is malicious; it also relies on the call sequence."

As we can see from the number of suspicious behaviour labelled by participants and the feedback obtained from the VISUALDROID group, our participants have show increased security awareness.

> Participants who use VISUALDROID show improvements in the aspect of security awareness with an increase of 155% against APK ANALYZER, 96% against JD-GUI, and 59.3% against JD-GUI+FLOWDROID.

5 DISCUSSION

In this section, we present how our visualisation solution VISUALDROID helps analysts perform the analysis of APK behaviours. We then propose some future directions and discuss the threats to validity.

5.1 How visualisation helps analyse APK behaviours

In this section, we present how participants in our treatment group think of the functionalities provided by VISUALDROID. We also conclude the lessons learned, and further make several recommendations for improving the visualisation of the APK analysis.

TABLE 8: Functionality ranking.

Functionality	Rank
Search keyword to highlight the related nodes	1
Subgraph presentation	2
Sensitive API highlight	2
Source code inspection	3
Highlight node by clicking the sensitive API info	4
Highlight node by clicking the package info	5

5.1.1 Functionality ranking in VisualDroid

We discuss how visualisation helps APK behaviour analysis by investigating the functionalities preferred by users for the APK visualisation platform. We review the feedback provided by the participants who used VISUALDROID at the end of our user study.

We prepare a list of VISUALDROID functionalities for participants. Then, our participants are required to rank the functionalities based on which assist them most in analysing APK behaviours. Table 8 presents the overall ranking we gathered from our participants from the VISUALDROID group. The keyword search function is considered as the most useful assistant for behaviour analysis, followed by the subgraph presentation (directed call graphs) and the highlighted nodes of sensitive API in the graph (to emphasise security-related information). The source code inspection ranks third, followed by clicking the sensitive API information to highlight the node and clicking the package information to highlight the node.

To ensure the time spent on the experiments is under control and make the result quantifiable, we provide a list of behaviours for participants to identify if they exist in the test APKs. This design encourages participants to search the keywords from the given behaviours to judge whether this behaviour exists. Therefore, the search function becomes a critical functionality in our experiments. However, in real-world practice, when performing the app behaviour analysis manually, security analysts usually need to start from scratch. Therefore, a more straightforward presentation and useful security information might weigh higher than the search functionality in real-world practice.

5.1.2 Recommendations for visualisation based tool

We ask our participants from four different groups (CT_1, CT_2, CT_3, and TM) to provide their advice on used tools, and then share their desired functionalities for APK behaviour analysis. We discover four categories of feedback as follows.

Data flow visualisation and interaction with source code: The control group 3 (CT_3) provides data flow information of the test APKs and shows improvements in both grade and time spent on APK behaviour analysis to other control groups. Even though the links between function calls are also provided in JD-GUI and JD-GUI+FLOWDROID, the presentations of the code in both tools do not maximise the benefits it can bring to the app analysis. Compared to JD-GUI, which allows users to inspect the relationships by clicking the name of the classes and functions, JD-GUI+FLOWDROID reveals the relationship between every two function calls. Even though the path of the entire data flow can be further generated, participants are still not able to view all the connected function calls in a single view due to the limitation of text-based information. Instead, VISUALDROID visualises the data flow and shows multiple pairs of the caller-callee relationship in a single view all at once. It also provides a subgraph view to mitigate the problem of analysing a complex network.

Another fact we observe from the control group 3 (CT_3) is that participants need to elaborate on the results from both FLOWDROID and JD-GUI. Because the data flow and the source code can both provide important information to assist APK behaviour analysis. However, it is difficult for participants to form a solid connection between the data flow and source code when there is no interaction provided between FLOWDROID and JD-GUI. However, VISUALDROID allows participants to click on the function node to further inspect the relevant code segment, which realises the interaction between the data flow visualisation and source code inspection.

Comparative view: In some cases, we need to compare similar APKs as we show in subsection 2.2.2. However, most of the APK analysis tools do not support APK comparison. Six participants from our control groups mentioned they found it challenging to perform APK comparison analysis. They needed to open two tabs or windows and constantly switch between them, which made the analysis process inconvenient and distracting. VISUALDROID provides a comparative view, automatically presents the most similar pairs of subgraphs to the least similar pairs between two APKs, and highlights the same nodes in each pair of subgraphs to help the similarity analysis.

Additional Information about sensitive APIs: From our security awareness result, participants from VISUALDROID group have security awareness increased, which can help them judge the potential malicious behaviours of an APK. Therefore, compared to text-based security information, the visualisation tools for security analysis can highlight suspicious behaviours so that analysts can perform further analysis of suspicious behaviour. Additionally, more details about why the highlighted function is a sensitive API can be provided, otherwise, developers might feel confused when commonly used APIs are referred to as security problems. Indeed, we can find many APIs that are too common to get developers aware of the potential security risks. Two potential solutions are 1) to have a toolbox to explain why this API can be sensitive (suspicious), and 2) to categorise the sensitive APIs into several groups based on how dangerous they can be. For example, some APIs related to location information might be labelled as relatively dangerous APIs.

Java function documentation: Apart from the confusion raised from sensitive APIs, we also find participants make mistakes because they do not know what exactly a specific API does. It happens when it comes to junior IT people, and it is also a challenge for experienced developers to remember the full documentation of the exhaustive Java APIs. Therefore, it will be helpful if the tool can provide explanations for the API when users hover on an API node, and a link for users to read more details in Android Java documentation.

Smart Search: In real-world practice, users will not heavily rely on the search functionality as our participants

in the experiments, but they still need to search and find relevant content in some cases. It is important to enhance the efficiency of app behaviour analysis by showing relevant results only through fuzzy search.

5.2 Implications

Based on our user study, we proved the effectiveness and efficiency of VISUALDROID on Android app analysis and the raise of security awareness compared with other visual solutions (*e.g.* APK ANALYZER, JD-GUI). We put forward several directions that we can explore further.

5.2.1 Android APK behaviour pattern and security analysis

VISUALDROID converts source code to the directed CG, which assists analysts in understanding an app's behaviours. Because it demonstrates the context in the graph, analysts can directly view the call sequence of the function nodes. They can also learn the behaviour patterns through the graph, where some functions always appear together in a fixed sequence. For example, if an APK requires installing an external application programmatically, it usually has two behaviour patterns that depend on whether the installation permission is granted ($android.permission.INSTALL\_PACKAGES$). First, if the permission is granted, APK will create a new `Intent`, then invoke the function called $setDataAndType()$, and finally start a new activity ($startActivity(intent)$). Second, if the permission is not granted, APK needs to request root permission, and then execute the command line from the inside of the APK to install external APKs. VISUALDROID highlights sensitive APIs in the subgraph. This function helps analysts consider how the sensitive API collaborates with other non-sensitive functions. It also helps analysts investigate if malicious behaviour exists based on the context, particularly when sensitive APIs are not necessary indicators of security compromise. For example, an APK requests to read text messages but does not expose the information to third parties. Also, a navigation app requests the location update constantly to provide their service. From our results, participants from VISUALDROID group have increased security awareness, which is very important in security analysis. Only if they are aware of the risks of behaviour, they will further check if this behaviour is malicious. Thus, VISUALDROID can serve as a support tool for Android behaviour identification and security analysis.

5.2.2 Repackaging and evolution analysis

The evolution of malware has introduced new challenges for malware detection. New features are emerging along with the malware evolution. Therefore, a detection model that is trained on an older dataset often becomes obsolete and makes poor decisions with new datasets. The F_1-Score for machine learning-based malware classifier can drop down to 0.3 in the worst case [26]. Moreover, adversarial attackers can exploit the evolutionary features of malware [27]. They develop an automatic tool to generate stealthier malware samples that can successfully bypass malware detection within a short period. However, it often requires a manual inspection to understand the ever-evolving malicious behaviours. Manual inspection can be either complicated or redundant when confronted with a continuously growing stream of incoming malware samples.

VISUALDROID also supports comparison analysis of Android apps. It lists the most similar subgraphs in two Android apps with identical nodes that have been highlighted. Therefore, analysts can locate the changes on the subgraph and further inspect the differences easily. For example, in Fig. 9, subgraph C1_10 and C2_10 use the same sensitive APIs (red nodes). The two subgraphs have a similar structure, while C2_10 has a more complex structure than C2_10. The green nodes represent the identical function nodes suggested by $SimiDroid$, and the major difference can be observed between area 1 in C1_10 and area 2 in C2_10. Analysts can get the differences by comparing the red square areas, and then commit the repackage and evolution analysis.

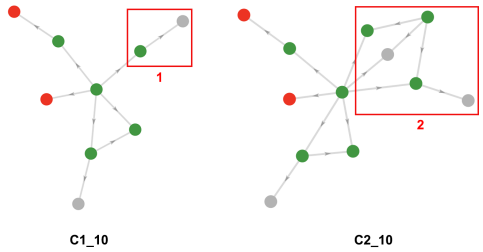

Fig. 9: Two similar subgraphs from different APKs.

5.2.3 Android APK analysis tutorial tool

Without delivering any tutorials on Android app analysis and security analysis, the group that adoptsVISUALDROID shows the best accuracy in behaviour identification and increased security awareness. In our interview session, we also asked our participants if they had obtained any new knowledge in terms of security. Our participants show an improvement in their comprehension of the APK behaviours as well as security awareness. Therefore, VISUALDROID can also be integrated into the training session as a tutorial tool for new security analysts. New security analysts can employ both the CG and the source code to comprehend the APK behaviours. Since VISUALDROID highlights sensitive APIs, analysts can reason the suspicious behaviours and learn more about the behaviour patterns related to security.

5.3 Threat to Validity

In this subsection, we will discuss the factors that could have influenced our experiment results.

Participant Selection: The participants of this study are required to have basic knowledge and experience of Java programming, but they neither receive any formal security training nor hold an IT security degree. We demonstrate that even without security training, our tool can help junior IT people raise awareness of security and identify Android app behaviour more accurately. However, the Java skill level of each participant is different, and it may affect the time spent on completing the tests in the experiments, especially for participants from the JD-GUI group (CT_2), which highly relies on source code understanding. A few participants

from the JD-GUI group spent much less time on the test than others, as we have discussed in Section 4. Though the bias is inevitable due to the diverse levels of Java skill, we mitigate this bias by selecting participants from universities who have similar degrees. This is to make sure they have a similar background in Java programming skill. The participants are then randomly separated into three control groups and one treatment group.

Baseline selection: We selected two tools for Android app inspection from industry and academia, respectively, namely JD-GUI and APK ANALYZER. JD-GUI is one of the widely adopted tools for Android/Java code analysis in academia. APK ANALYZER is an inbuilt module in Android Studio, the de facto IDE for Android app development. It supports APK code inspection and a side-by-side comparison of two APKs. There are also some other tools as far as we know. However, most of these tools used in the industry are for internal use only, and we do not have access to them. To the best of our knowledge, APK ANALYZER is one of the tools widely used in the industry and most recommended in the technical forums.

Number of participants: This study aims to demonstrate the capability of VISUALDROID in assisting Android app behaviour analysis. VISUALDROID can also increase the security awareness of people with a non-security background. We invited 40 participants, and each of them spent around two and a half hours in the user study and the interview. It is challenging to recruit a large number of participants who satisfy all of our requirements. The study with 40 participants may not sufficiently reflect the exact situation in real practice. Therefore, it may face the threat of the representatives of the subjects. Many previous studies in software engineering also invite 30 or even fewer participants to join their user study (c.f., [28], [29], [30], [31]). Our study aligns well with these previous studies. However, for each participant we selected, we have carefully checked if he/she meets our requirements and discussed any unexpected results that occurred during the experiments. In the future, we plan to invite more people to participate in our user study.

Task oriented design: The experiments in our user study are designed in a task-oriented manner. The participants are given a list of candidate behaviours and are asked to identify if the listed behaviours exist in a given Android app. However, in real practice, the analysts are not provided with a list of possible behaviours of an app. Instead, depending on the analysts' strategy, they may either start the analysis from the entry point of the app (i.e. the Main Activity) or generate a behaviour list of interest before the analysis. While VISUALDROID supports both analysis strategies, the analysis from the entry point is time-consuming and difficult to be evaluated. Asking participants to generate a behaviour list of interest heavily relies on the participants' prior knowledge of security, and will affect the evaluation results. Therefore, we provide a list of candidate behaviours to eliminate the bias caused by the differences in participants' prior knowledge.

6 RELATED WORK

In the past decade, software visualisation has been widely studied and used to assist system and software comprehension. According to our literature review, their focuses heavily vary from each other. Therefore, it is difficult to categorise them in the dimension of software visualisation. Because many research works in this area involve user studies to optimise the use of the software visualisation tools, we accordingly organise two paragraphs in the following, to present the related works that concern user studies and those which do not.

6.1 Visualisation analysis based on the user study

We first discuss the related works that involve user studies to demonstrate the effectiveness of the proposed tool. Cornelissen *et al*. [7] conducted a quantitative evaluation on EXTRAVIS, which is a tool that offers interactive visualisation views of large execution traces. In their controlled experiment, they defined eight typical comprehension tasks for participants. They measured the performance of the participants in terms of time cost and the correctness of completing the tasks. In [32], researchers presented an evaluation of four representative software visualisation (SoftVis) tools in the context of corrective maintenance. They invited four independent groups of professional software developers to participate in the evaluation. Each group used a different tool to solve the same task on a real-world code base under typical industry conditions. Fittkau *et al*. introduced hierarchical and multi-layer visualisation of large software landscapes with ExplorViz [8]. They set up a controlled experiment to compare ExplorViz with the Extravis trace visualisation approach by inviting participants to complete the predefined tasks. In [33], researchers conducted a user study to assess the city metaphor in software visualisation in terms of users' feelings, emotions, and thinking. Langelier *et al*. [34] presented a visualisation framework to perform the quality analysis and help understand large-scale software systems. They evaluated their framework by inviting 15 subjects to complete 20 tasks in their user study. Wettle *et al.* [6] presented a controlled experiment of CodeCity, which is a 3D software visualisation approach developed based on a city metaphor. They invited 41 participants from both academia and industry to validate the effectiveness and efficiency of the visualisation approach they proposed.

6.2 Visualisation analysis based on the proposed tool

There are also many works about software visualisation that were not evaluated and discussed based on user studies. For example, Somarriba *et al*. [35] monitored Android apps' suspicious behaviours at the run time and visualised the invoked malicious functions in a dendrogram, which allows analysts to inspect the malicious functions visually. In another example, Kobayashi *et al*. [36] proposed SArF Map, which used a city metaphor to visualise software architecture. They evaluated the performance of the SArF Map and the effectiveness of uncovering architectural knowledge by using open-source industrial software in their case studies. In [37], Caserta *et al*. presented real visualisation examples of their 3D-HEB technique on a software city metaphor in their evaluation. Their work facilitated the assessment of the tool effectiveness in displaying large-scale software systems. Lanza and Ducasse designed an approach (called $ClassBlueprint$) for facilitating app comprehension [38], in

which they provided a novel categorisation and visualisation of classes. They also proposed a lightweight software visualisation technique called polymetric view [39], which was evaluated on several large industrial apps. In [40], researchers designed specific tasks that were used to evaluate a particular part of their proposed software visualisation model. Panas *et al*. [41] proposed a visualisation based on Vizz3D. In their work, they also outlined several scenarios to emphasise the ways, by which their approach could facilitate collaboration and discussion among stakeholders. Griswold *et al*. [42] implemented Aspect-Browser to demonstrate the map metaphor applicability of indicating software evolution. They also carried out a case study in their evaluation to determine how the map metaphor assisted understanding of the software evolution. Balzer and Deussen [43] visualised the relations within hierarchical software structures by enhancing graphs with trees. Fittkau *et al*. [44] presented $ExplorViz$ as a live visualisation approach to monitor traces for large software. Storey *et al*. [45] developed $SHriMP$, which provides a nested graph view to present information. Hahn *et al*. [46] presented a novel visualisation technique for the interactive exploration of multi-threaded software systems, and they evaluated the performance by comparing their results with existing visualisation techniques. Wettel *et al*. implemented CodeCity [47], which has a set of visualisation techniques to support tasks related to program comprehension, design quality assessment, and evolution analysis.

In this paper, we investigated the visualisation solution (CG-based) in assisting in understanding the app behaviour on the Android platform, which was not discussed in previous studies. We also proved CG-based visualisation solution is effective in assisting the understanding of app behaviours on the Android platform.

7 Conclusion

In this paper, we present a controlled experiment aimed at evaluating the effectiveness and efficiency of a visualisation solution for Android behaviour analysis. We design VisualDroid, a CG-based analysis tool that utilises several visualisation components. In our control experiment, we conduct a user study to evaluate the effectiveness and efficiency of VisualDroid. 40 participants are recruited for our user study. The result of the experiment shows that the visualisation solution leads to a significant improvement in the effectiveness of APK behaviour analysis compared to APK Analyzer, JD-GUI, and JD-GUI+FlowDroid, with the increases of 71.4%, 35.7%, and 39.2% respectively. Besides, the highlight of the sensitive APIs in the graph also enhances participants' security awareness. In the future, we need to invite more participants, not only graduates but also experienced security analysts. We also need to observe their performance in our projects and further validate the effectiveness of our approach.

References

[1] Java decompiler. [Online]. Available: https://java-decompiler.github.io/

[2] Analyze your build with apk analyzer. [Online]. Available: https://developer.android.com/studio/build/apk-analyzer

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM SIGPLAN Notices*, vol. 49, 06 2014.

[4] It security training. [Online]. Available: http://nsclab.org/syssec/resources/Security_Training_In_Industry.pdf

[5] Welcome to androguard's documentation! [Online]. Available: https://androguard.readthedocs.io/en/latest/

[6] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: a controlled experiment," in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 551–560.

[7] B. Cornelissen, A. Zaidman, and A. Deursen, "A controlled experiment for program comprehension through trace visualization," *Software Engineering, IEEE Transactions on*, vol. 37, pp. 341 – 355, 07 2011.

[8] F. Fittkau, A. Krause, and W. Hasselbring, "Software landscape and application visualization for system comprehension with explorviz," in *Information and Software Technology*, 2016.

[9] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 280–291.

[10] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1105–1116. [Online]. Available: https://doi.org/10.1145/2660267.2660359

[11] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Trans. Priv. Secur.*, vol. 21, no. 3, Apr. 2018. [Online]. Available: https://doi.org/10.1145/3183575

[12] G. Carenini, C. Conati, E. Hoque, B. Steichen, D. Toker, and J. Enns, "Highlighting interventions and user differences: Informing adaptive information visualization support," *Conference on Human Factors in Computing Systems - Proceedings*, 04 2014.

[13] B. Alper, T. Hollerer, J. Kuchera-Morin, and A. Forbes, "Stereoscopic highlighting: 2d graph visualization on stereo displays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2325–2333, 2011.

[14] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro, "Understanding android app piggybacking: A systematic study of malicious code grafting," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 6, pp. 1269–1284, 2017.

[15] M. Rosvall, D. Axelsson, and C. Bergstrom, "The map equation," in *The European Physical Journal Special Topics*, 2009, pp. 13–23.

[16] G. Concas, C. Monni, M. Orrù, and R. Tonelli, "A study of the community structure of a complex software network," in *2013 4th International Workshop on Emerging Trends in Software Metrics (WETSoM)*, 2013, pp. 14–20.

[17] Y. Qu, X. Guan, Q. Zheng, T. Liu, L. Wang, Y. Hou, and Z. Yang, "Exploring community structure of software call graph and its applications in class cohesion measurement," *Journal of Systems and Software*, vol. 108, pp. 193–210, 06 2015.

[18] M. Fan, J. Liu, X. Luo, K. Chen, Z. Tian, Q. Zheng, and T. Liu, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 8, pp. 1890–1905, 2018.

[19] Lesson: Object-oriented programming concepts. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/concepts/index.html

[20] Application security. [Online]. Available: https://source.android.com/security/overview/app-security

[21] L. Li, T. Bissyandé, and J. Klein, "Simidroid: Identifying and explaining similarities in android apps," 08 2017, pp. 136–143.

[22] R. Vallee-rai and L. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations," 01 2004.

[23] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[24] L. Li, T. F. Bissyandé, and J. Klein, "Rebooting research on detecting repackaged android apps: Literature review and benchmark," *IEEE Transactions on Software Engineering (TSE)*, 2019.

[25] F. Wilcoxon, S. Katti, and R. A. Wilcox, "Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank test," *Selected tables in mathematical statistics*, vol. 1, pp. 171–259, 1970.

[26] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "Tesseract: Eliminating experimental bias in malware classification across space and time," *CoRR*, vol. abs/1807.07838, 2018.

[27] W. Yang, D. Kong, T. Xie, and C. A. Gunter, "Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, ser. ACSAC 2017. New York, NY, USA: ACM, 2017, pp. 288–302. [Online]. Available: http://doi.acm.org/10.1145/3134600.3134642

[28] R. Latorre, "Effects of developer experience on learning and applying unit test-driven development," *IEEE Transactions on Software Engineering*, vol. 40, no. 4, pp. 381–395, 2014.

[29] G. L. Scoccia, I. Malavolta, M. Autili, A. Di Salle, and P. Inverardi, "Enhancing trustability of android applications via user-centric flexible permissions," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[30] E. M. Redmiles, A. R. Malone, and M. L. Mazurek, "I think they're trying to tell me something: Advice sources and selection for digital security," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 272–288.

[31] K. Krombholz, W. Mayer, M. Schmiedecker, and E. Weippl, ""i have no idea what i'm doing" - on the usability of deploying HTTPS," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1339–1356. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/krombholz

[32] M. Sensalire, P. Ogao, and A. Telea, "Model-based analysis of adoption factors for software visualization tools in corrective maintenance," 05 2020.

[33] S. Romano, N. Capece, U. Erra, G. Scanniello, and M. Lanza, "The city metaphor in software visualization: Feelings, emotions, and thinking," *Multimedia Tools and Applications*, 05 2019.

[34] G. Langelier, H. Sahraoui, and P. Poulin, "Visualization-based analysis of quality for large-scale software systems. ase '05," 01 2005, pp. 214–223.

[35] O. Somarriba, U. Zurutuza, R. Uribeetxeberria, L. Delosières, and S. Nadjm-Tehrani, "Detection and visualization of android malware behavior," *Journal of Electrical and Computer Engineering*, vol. 2016, pp. 1–17, 03 2016.

[36] K. Kobayashi, M. Kamimura, K. Yano, K. Kato, and A. Matsuo, "Sarf map: Visualizing software architecture from feature and layer viewpoints," 05 2013.

[37] P. Caserta, O. Zendra, and D. Bodénès, "3d hierarchical edge bundles to visualize relations in a software city metaphor," in *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, 2011, pp. 1–8.

[38] M. Lanza and S. Ducasse, "A categorization of classes based on the visualization of their internal structure: The class blueprint." vol. 36, 11 2001, pp. 300–311.

[39] M. Lanza and S. Ducasse, "Polymetric views - a lightweight visual approach to reverse engineering," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 782–795, 2003.

[40] M. J. Pacione, M. Roper, and M. Wood, "A novel software visualisation model to support software comprehension," in *11th Working Conference on Reverse Engineering*, 2004, pp. 70–79.

[41] T. Panas, T. Epperly, D. Quinlan, A. Saebjornsen, and R. Vuduc, "Communicating software architecture using a unified single-view visualization," in *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, 2007, pp. 217–228.

[42] W. G. Griswold, J. J. Yuan, and Y. Kato, "Exploiting the map metaphor in a tool for software evolution," in *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, 2001, pp. 265–274.

[43] M. Balzer and O. Deussen, "Exploring relations within software systems using treemap enhanced hierarchical graphs," in *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2005, pp. 1–6.

[44] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring, "Live trace visualization for comprehending large software landscapes: The explorviz approach," in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, 2013, pp. 1–4.

[45] M.-A. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen, "Shrimp views: an interactive environment for information visualization and navigation." 01 2002, pp. 520–521.

[46] S. Hahn, M. Trapp, N. Wuttke, and J. Döllner, "Thread city: Combined visualization of structure and activity for the exploration of multi-threaded software systems," in *2015 19th International Conference on Information Visualisation*, 2015, pp. 101–106.

[47] R. Wettel, "Visual exploration of large-scale evolving software," in *2009 31st International Conference on Software Engineering - Companion Volume*, 2009, pp. 391–394.

Lihong Tang is currently a Ph.D student at Department of Computer Science and Software Engineering, Swinburne University of Technology. She received her Bachelor degree with Honours from Deakin University, Australia. She is working on Android malware detection and evolution analysis, and her research interests are android malware analysis, adversarial attacks under the Android context, and human-centric research.

Tingmin Wu (Tina) is a research scientist at CSIRO's Data61. Prior to that, she was a research fellow at Monash University jointly with CSIRO's Data61. Her research focuses on human centric cyber security, currently with a specific focus on phishing. Her research is to study how humans interact with security tools and apply AI to optimise security to reduce the involvement of security experts.

Xiao Chen is a research fellow with the Department of Software Systems and Cybersecurity, Faculty of IT, Monash University. He received Ph.D degree from Swinburne University of Technology, Australia. His research interests include mobile software analysis, mobile security and adversarial machine learning.

Sheng Wen received the Ph.D degree in Computer Science from the School of Information Technology, Deakin University, Australia, in 2015. He is currently a Senior Lecturer at Swinburne University of Technology. His focus is on modeling of virus spread, information dissemination, and defense strategies for the Internet threats. He is also interested in the techniques of identifying information sources in networks.

Li Li is an ARC DECRA Fellow and Senior Lecturer at Monash University. He joined Monash University as a Lecturer on February 2018. Prior to that, he was a research associate in Software Engineering at the University of Luxembourg (UL) where he also obtained his Phd degree in November 2016. His research interests mainly lie in the field of Mobile Software Engineering (i.e., Mobile Security and quality assurance) and Intelligent Software Engineering (SE4AI, AI4SE).

Xin Xia is the director of the software engineering application technology lab, Huawei, China. Prior to joining Huawei, he was an ARC DECRA Fellow and a lecturer at Monash University, Australia. Xin received his Ph.D in computer science from Zhejiang University in 2014. To help developers and testers improve their productivity, his current research focuses on mining and analyzing rich data in software repositories to uncover interesting and actionable information. More information at: https://xinxia.github.io/.

Marthie Grobler received her Ph.D degree in Computer Science from University of Johannesburg. She currently holds a position as Principal Research Scientist at CSIRO, Data61 in Melbourne, Australia where she drives the research group's work on cybersecurity governance, policies and awareness. Her research interest includes human centric cybersecurity, HCI, online risk resilience, and cyber governance.

Yang Xiang received his Ph.D degree in Computer Science from Deakin University, Australia. He is currently a full professor and the Dean of Digital Research, Swinburne University of Technology, Australia. His research interests include Cybersecurity, which covers network and system security, data analytics, distributed systems, and networking. He is also leading the Blockchain initiative at Swinburne.

APPENDIX

TABLE 9: Variables used in the study

Variables	Type	Description
TM	Experimental Group	Treatment Group. Test APKs used in this group are processed by VisualDroid, and the APK information is presented in a directed graphic view
CT_1	Experimental Group	Control Group 1. Test APKs used in this group are imported into APK Analyzer, and the APK information is converted into tree management view with pack-ages, classes and method information
CT_2	Experimental Group	Control Group 2. Test APKs used in this group are reconstructed into a readable version and user can access to methods and fields
CT_3	Experimental Group	Control Group 3. CG is generated by FlowDroid in a text manner. The source code of the APKs is reconstructed into a readable version and user can access to methods and fields
Test Single	Test Type	In Test Single, participants are asked to analyse a malicious APK
Test Comparison	Test Type	In Test Comparison, two APKs are provided, with one being an evolved malicious version of the other.
A_{single}	APK ID	Test APK used in session A - Test Single. It is at the complex level 2
A_{Cmp1}	APK ID	Test APK used in session A - Test Comparison, and it is at the complex level 1
A_{Cmp2}	APK ID	Test APK used in session A - Test Comparison, and it is at the complex level 1
B_{Single}	APK ID	Test APK used in session B - Test Single, and it is at the complex level 1
B_{Cmp1}	APK ID	Test APK used in session B - Test Comparison, and it is at the complex level 2
B_{Cmp2}	APK ID	Test APK used in session B - Test Comparison, and it is at the complex level 2
G_{S_1}	Test Grade	Grade obtained in Single Test. The test APK is at the complex level 1
G_{S_2}	Test Grade	Grade obtained in Single Test. The test APK is at the complex level 2
G_{C_1}	Test Grade	Grade obtained in Comparison Test. The test APKs are at the complex level 1
G_{C_2}	Test Grade	Grade obtained in Comparison Test. The test APKs are at the complex level 2
T_{S_1}	Time Spend	Time spent on Test Single. The APK used in this test is at the complex level 1
T_{S_2}	Time Spend	Time spent on Test Single. The APK used in this test is at the complex level 2
T_{C_1}	Time Spend	Time spent on Test Comparison. The APKs used in this test are at the complex level 1
T_{C_2}	Time Spend	Time spent on Test Comparison. The APKs used in this test are at the complex level 2
N_{SB}	Number of suspicious behaviours	The number of suspicious behaviour labeled by the participants