A Survey on Secure Android Apps Development Life-Cycle: Vulnerabilities and Tools

Mohammed El Amin TEBIB
Mariem GRAA
Oum-El-Kheir AKTOUF
Univ. Grenoble Alpes, Grenoble INP, LCIS lab., 26000 Valence,France*
**Institute of Engineering Univ. Grenoble Alpes*
mohammed-el-amin.tebib@univ-grenoble-alpes.fr
mariem.graa@univ-grenoble-alpes.fr
oum-el-kheir.aktouf@univ-grenoble-alpes.fr

Pascal ANDRE
LS2N Lab
UMR 6004 CNRS, Nantes University,
F-44000 Nantes, France
pascal.andre@ls2n.fr

***Abstract*—Mobile devices are increasingly used in our daily lives. To fulfill the needs of smartphone users, the development of mobile applications has been growing at a high rate. As developers are not necessarily aware of security concerns, most of these applications do not address security aspects appropriately and usually contain vulnerabilities. Therefore, it is essential to incorporate security into the app development life-cycle. To help development teams to address security issues, several security integrated development environment (IDE) plugins have been proposed. In this paper, we aim to review the effectiveness of existing IDE plugins in detecting known Android vulnerabilities. We developed a classification framework that highlights the salient features related to 16 selected IDE plugins including: (1) the analysis-based approach, (2) the vulnerabilities checks coverage, and (3) the development stage, on which these tools could be employed. We proceeded to a deep analysis process where each tool effectiveness is investigated against 19 vulnerabilities. Each vulnerability has been demonstrated by executing a corresponding attack scenario on the recent version 12 of Android. The study results provide an overview of the current state of secure Android application development and highlight limitations and weaknesses. Limits such as: tools unavailability, benchmarks incompleteness, and the need of dynamic analysis approaches adoption are among the main findings of this study. The paper synthesizes valuable information for future research on IDE plugins for detecting Android-related vulnerabilities.**

Keywords-* *Android; Software development; DevSec; Secure Coding; Classification Framework; Security IDE Plugins.

I. INTRODUCTION

With the increase of mobile devices usage, a growing number of applications have been developed to satisfy Android users' requirements. In October 2022, approximately 97000 mobile apps were released through the Google Play Store[1]. However, most of these applications are developed without integrating proper security needs. Due to the lack of security awareness among developers, many of these applications contain vulnerabilities. According to the official MITRE data-source for Android vulnerabilities, *Common Vulnerabilities and Exposures (CVE) details*,[2] recent years witnessed the most significant increase of Android security threats, *"1034 vulnerabilities the last couple years"* , and it continues to increase with *"34 vulnerabilities only the two first months of 2022"*. These vulnerabilities could be exploited to create harmful actions such as developing malwares and stealing users private information. Thus, many updates and patches are provided to the published applications. Therefore, there is an urgent need to fix source code vulnerabilities at the design and development stages and to integrate security-by-design concepts and practices. More precisely, security verification must belong to Integrated Development Environments (**IDE**), as *plugins*, rather than being just external tools, otherwise developers would often consider security as a secondary concern. Security verification feedback should appear as syntax or type errors in the IDE to be part of the developer's activity. Industry and academia tools started recently to integrate security to the software development life-cycle, shifting *from* just ensuring the development speed and leaving the security checks to external stakeholders, *to* employing new software development paradigms such as *development, security, and operations (DevSecOps)* [2]. In these paradigms, developers adhere to a secure development process by means of training sessions and analysis tools.

To assist non-expert Android developers in addressing security concerns, it is essential to provide them with an up-to-date overview of IDE tools that can help to secure their applications. This is the main contribution of our article. We selected a sample of IDE plugins (tools). This sample includes the four most well-known industrial plugins, as extracted from the OWASP list (Open Source Application Security Project) [3]. Additionally, we included academic tools for more comprehensive analysis.

To study these tools, we propose a classification framework based on three dimensions: (1) the analysis-based approach (static or dynamic); (2) the covered security vulnerabilities by each tool; and (3) the development stage, on which these tools could be employed. To limit the scope of our study, we consider the following constraints: (a) Only tools usable during the development life-cycle are considered; i.e., *the tools integrated in the IDE environment*. (b) For academic tools, if the tool code is not provided, our analysis is based only on reading the corresponding published scientific publications and related documentations. (c) For industrial tools, only free available ones are considered.

This classification framework allows to highlight salient features of 16 selected tools by a shallow analysis. We proceeded to a deep analysis process by running the tools on a relevant security benchmark. Each tool effectiveness is investigated against 19 vulnerabilities. Each vulnerability has been demonstrated by executing a corresponding attack scenario on the recent version 12 of Android. The study results allowed to establish a picture of the current secure Android application development.

This article is an extended version of a paper published in Secureware 2022 [1]. The tool information has been revisited to add more details. New material includes a detailed presentation of the

[1] https://fr.statista.com/

[2] https://www.cvedetails.com/product/19997/Google-Android.html

framework in Section V, including the list of potential vulnerabilities; a review of the security analysis approaches and a position of the topic in the Secure Software Development Life-Cycle. The main contribution of our work is an empirical evaluation of the IDE plugins (16 IDE plugins) in detecting vulnerabilities (19 vulnerabilities) for Android 12. We add new research questions in Section IV and new experiments with methodological feedback.

The article is organised as follows. Section II introduces material to understand the context and the comparison methodology. Section III summarizes the existing related works in the literature, reviewing the IDE plugins used for securing Android applications development. We describe our work methodology in Section IV. The classification framework is exposed in Section V. In Section VI, we overview the 16 tools selected according to the selection criteria provided in Section IV. Based on the proposed classification framework, we present in Section VII the main results of the search and analysis phases of our study methodology. We discuss the main findings of our study in Section VIII as well as methodological limitations. Finally, Section IX concludes the paper and provides tracks for future work.

II. Background

This section illustrates the main concepts related to Android applications. Android provides a layered software stack composed of native libraries and a framework as an environment for running Android applications. The framework exposes a set of system services in the form of Applications Programming Interfaces (APIs). An application uses a specific service by instantiating the interface of the corresponding API; the framework launches a remote procedure call to invoke the real implementation that resides on the kernel level.

Based on this software stack, developers implement different types of Android applications: (1) **native** applications that restrict their access to the APIs provided by the framework and (2) **hybrid** applications that could also be web applications. Since considering the security of hybrid applications should cover a wide range of potential security issues related to the web, our study only covers native Android applications (also called **apps** in this article).

A native Android application is built using four types of components, categorised into two families: (1) `Foreground components` such as *activities*, and (2) `Background components` such as *services*, *broadcast receivers*, and *content providers*. Activities provide graphical interfaces, which allow the user to interact with the app to perform different tasks (following the Model-View-Controller (MVC) pattern). The other components are merely used to handle background processing and communications. The core functionalities of the application are implemented through *services* that are used for long-running operations. `Content providers` manage the data layer (storage, read/write accesses), they are used to share data between apps; and finally *broadcast receivers* that receive and respond to `Broadcasts` (i.e., messages that the Android system and Android apps send when events that might affect the functionality of other apps occur). There are two types of broadcasts: system broadcasts, that are delivered by the system, and custom broadcasts, that are delivered by apps. `Custom Intent` actions are defined to create a custom broadcast. There are four ways to deliver a custom broadcast: normal, ordered, local, sticky. Normal broadcasts are sent to all the registered receivers at the same time, in an undefined order. Ordered broadcasts are sent in defined order, the `android: priority` attribute determines the order of the broadcast sending. If more than one receiver with same priority are present, the sending order is random. The intent is propagated from one receiver to the next one. A receiver has the ability to update the intent or cancel the broadcast. Local broadcasts are specifically sent to receivers within the same app. On the other hand, sticky broadcasts are a unique type of broadcast where the intent object that is sent remains in the cache even after the broadcast has been completed. This allows other components within the app to access and retrieve the intent at a later time. The system may broadcast once again sticky intents to later registrations of receivers. Unfortunately, the sticky broadcasts API suffers from numerous security-related shortcomings. Sticky broadcasts can lead to sensitive data exposure, data tampering, unauthorized access to execute behavior in another app, and denial of service. The *intents* manage the communication aspects between the application components. Communications could be implicit in case the message target is not specified. Application can pass a `PendingIntent` to another application in order to allow it to execute predefined actions. The unfilled fields of a pending intent can be modified by a malicious app and allow access to otherwise non-exported components of a vulnerable application.

To ensure secure execution of Android applications, two main security artefacts are considered: *sand-boxing* and *permissions*. An application runs within its own context or sandbox, which is an isolation mechanism between Android applications. So applications cannot interact with other installed applications or the Android operating system (OS) without proper permissions. Thus, the permission system restricts the access to applications, application components and system resources (contacts, locations, images, etc.) to those having the *required permissions*. Android categorizes permissions into different types, including install-time permissions and runtime permissions. The install-time permissions allow an app to perform restricted actions that minimally affect the system or other apps. Thus, they are automatically granted when the app is installed. There are several sub-types of install-time permissions, such as normal permissions and signature permissions. Normal permissions allow access to data that present very little risk to the user's privacy. The system assigns the normal protection level to normal permissions. Signature permission are granted by system to an app only when the app is signed by the same certificate as the app or the OS defining the permission. The system assigns the signature protection level to normal permissions. Runtime permissions give the app access to restricted data such as private user data or allow the app to perform restricted actions that more substantially affect the system and other apps. Thus, runtime permissions are not automatically granted, since their access could be given or denied by the user. The system assigns the dangerous protection level to runtime permissions.

Permissions are declared by developers in the manifest file. Many studies showed that the manifest file can be the source of many security issues [4]: privilege escalation resulting from the over declaration of permissions [5], communication issues resulting from the use of undocumented message types of intents [6], etc.

III. Related works

In this section, we resume the related works aiming to review or evaluate the tools assisting developers in securing Android applications and we show how the current survey extends these works.

Significant effort has been made by the research community to assess security analysis tools for software development in general, and mobile applications more specifically. Recent works [7] and [8] present general reviews of existing IDE plugins for detecting security vulnerabilities in software applications. In [8], the authors selected plugins for the 5 main IDEs (Eclipse, Visual Studio, IntelliJ, NetBeans, Android Studio) and specifically focused on 17 plugins that provide support for input-validation-related vulnerabilities (as described in the Common Weakness Enumeration (CWE) repository). By reading documentations, they listed salient related features such as their supported IDEs, applicable languages and their capabilities in detecting security vulnerabilities. No experimentation is done to assert the documentation sources. In [7], only five open-source plugins were selected. These plugins were also studied in [8], except FindSecBugs/SpotBugs (selected as it is more recent than Findbugs). The authors evaluated coverage and performance by experimenting 14 CWE entries for the 10 top Open Worldwide Application Security

Project (OWASP) categories, using the Juliet Test Suite[3] that is a collection of intentionally vulnerable artificial code written in C, C++, C#, Java or PHP. They also evaluated usability by analysing the result quality and the plugin suggestions. Both works are complementary and present interest but none of them focuses on the Android system, its usual programming languages (Java & Kotline) and its vulnerabilities. In our work, we only focus on Android apps, and we cover even a larger scope by handling a more complete and up-to-date set of existing IDE plugins, which still miss evaluation. Consequently, we will provide a more complete and consistent analysis by covering a wider set of vulnerabilities and finer applicability assets.

In [9], Mejía et al. conducted a systematic review to establish the state of the art of secure mobile development. They found 7 assisting solutions for secure development. These solutions are classified based on: 1) the type of the use (methodologies, models, standards or strategies); and 2) the related security concern (authentication, authorisation, data storage, data access and data transfer). After analysing the results of this research, we consider that the number of handled solutions is limited regarding the real existing ones in the literature. In addition, we found that none of the presented solutions is proposed as a tool or a plugin for secure development. In our work, the search and the analysis processes are deeper. Indeed, we present a higher number of assistant solutions, which are intended to be used as IDE plugins. In addition, our classification is related to software development life-cycle of Android applications.

Janaka S. et al. [10] presented a Systematic Literature Review (SLR) on the Android vulnerability detection tools based on machine learning techniques. 118 technical studies were carefully studied. The results showed that machine learning techniques are used to increase the security of an Android application based on three steps: 1) considering analysing, 2) detecting, and 3) preventing vulnerabilities. In the same line, another SLR was proposed in [11] that classifies 300 security analysis tools based on 1) the analysis dimension (malware detection, vulnerability detection) and 2) the type of security threat (spoofing, Denial of Service, Privilege Security, etc). Regarding these works, the studies are too generic and there is no evaluation process included in the study.

The closest works to our research are [12] and [13]. In [12], Bradley et al. proposed an assessment and evaluation study of Android applications analysis tools. The tools were categorized based on the security issue they solve. To evaluate these tools, the authors recruited eight computer science students with confirmed Java programming skills for a period of 10 weeks. The students evaluated each tool against open source applications extracted from known benchmarks, such as DroidBench and GooglePlay. The results of each evaluation showed the time needed to configure the requisite environment, the problem to address, the vulnerabilities to detect and the type of analysis to perform. Many limitations related to usability, time of execution and analysis precision were outlined as results of the study. The assessment study proposed by Mitra et al. in [13] evaluated the effectiveness of vulnerability detection tools for Android applications. The authors considered 64 security analysis tools and empirically evaluated 19 of them against the *Ghera* benchmarks [14]. It captures 42 known vulnerabilities, each implemented inside a single Android application. As a result, they found that the evaluated tools for Android applications are very limited in their ability to detect known vulnerabilities. The sample of tools in these studies are oriented for use by pen-testers after the application release. All of them are not IDE plugins (except FixDroid) In addition, the evaluation process is limited to the academic tools. In our work, we are interested in academic and industrial free tools, which are specifically intended as security assisting tools. We did not find existing research work that studies precisely Android IDE plugins from a security perspective. After analysing the existing benchmarks, we consider that *Ghera* repository [14] is the most useful means for evaluating the analysis tools. Indeed, *Ghera* summarises a non-exhaustive list of well know vulnerabilities related to the development of Android applications. It provides an open source Android application implementing each vulnerability. Moreover, having numerous test cases with single vulnerabilities is better, in terms of evolution, than big test cases covering several vulnerabilities. Indeed new Android releases remove old vulnerabilities. Therefore, to conduct our study, we used the same benchmark as Mitra et al. [13] to evaluate the list of our selected plugins.

In summary, existing comparisons are interesting at first sight but have limited scope or panel of tools, some are deprecated because security evolves with the OS releases. In the next section, we propose a new methodology to cover a large range of development and security fields and a consistent panel of available tools.

IV. Research Methodology

This section illustrates the proposed methodology for conducting a precise analysis and comparison study of existing tools dealing with security concerns throughout the Android app development process.

A. Research questions

The study aims to answer the following research questions:

- `RQ1` *Which tools are being employed in the development of secure Android applications?* The goal is to conduct a review of significant related works and identify tools to aid in secure code development.
- `RQ2` *Is security considered in all the design activities during the development process of Android apps?* This research question explores the coverage of security analysis solutions throughout the entire software development life-cycle (Section V-A).
- `RQ3` *Which analysis techniques are being adopted by the existing security development solutions?* This research question aims to map tools to the existing analysis techniques described in Section V-C.
- `RQ4` *Are the studied IDE plugins effective in detecting known vulnerabilities?* Through this research question, we aim to investigate the capabilities of the IDE plugins in detecting the list of Android vulnerabilities presented in Section V-B.

B. Classification framework

A classification framework enables to structure our comparison study by grouping the search space on axes. Four main dimensions are explored:

- **what** to find as security issues;
- **where** to assist developers in detecting security vulnerabilities;
- **when** to handle the security issues during the software development;
- **how** to proceed to detect security vulnerabilities.

This framework will be detailed in Section V. It serves as a structuring basis for our analysis approach. Next, we present the followed search methodology to identify relevant security assistance tools. These tools will be examined to refine all the dimensions of the presented framework.

C. Research methodology process

The proposed research methodology is depicted in Fig. 1. Three main phases are considered in the process:

1) **Tools search & selection**: This phase highlights the tools used by designers and/or developers to prevent security issues in Android applications. To define this list, our primary source of information were mainly:
   - For academic tools, we focused on published academic reviews [7] [8], systematic mapping studies [9] [15], and public GitHub repositories [16] [17];
   - For industrial plugins, we considered only free and available ones such as SonarLint [18], Findbugs [19],

[3] https://samate.nist.gov/SRD/testsuite.php

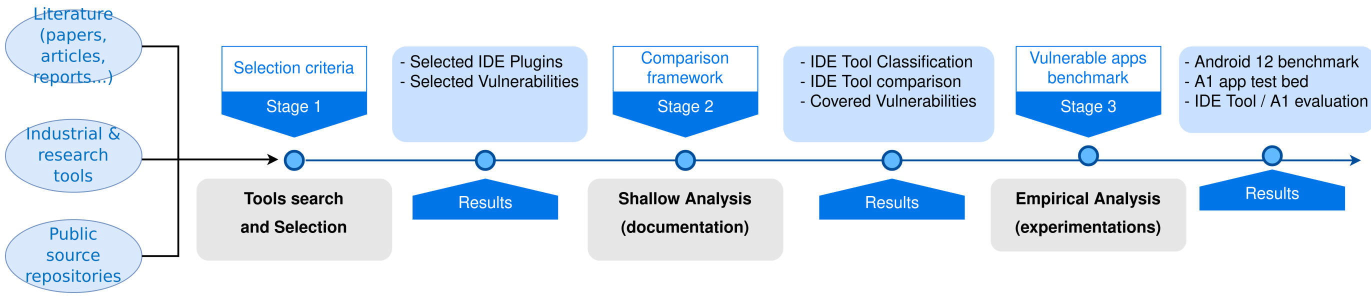

Literature (papers, articles, reports...)
Industrial & research tools
Public source repositories
Selection criteria
Stage 1
- Selected IDE Plugins
- Selected Vulnerabilities
Comparison framework
Stage 2
- IDE Tool Classification
- IDE Tool comparison
- Covered Vulnerabilities
Vulnerable apps benchmark
Stage 3
- Android 12 benchmark
- A1 app test bed
- IDE Tool / A1 evaluation
Tools search and Selection
Results
Shallow Analysis (documentation)
Results
Empirical Analysis (experimentations)
Results

Fig. 1. Proposed Research Methodology

AndroidLint [20], and Snyck [21]. We had the opportunity to experiment them during the development of many IT industrial projects in France. They are also listed among the well known tools extracted from the OWASP list [3]. We also included solutions built on top of AndroidLint [20] such as: Lintent [22], and icc−lint [23].

- As an additional selection criteria, we included the tools that we had the opportunity to investigate while we are building the PermDroid tool [24] in a complementary research work. This tool is based on formal analysis to prevent permissions related security issues in Android applications.

To continue the selection process, we also considered the following excluding criteria:

- Tools that are not oriented for detecting vulnerabilities during the development process like Anadroid [25], which is used for malware detection, and MassVet [26], which is used for analysing packaged applications in Google-Play store;
- Tools that cannot be used within the IDE, e.g., ComDroid [27], which warns pen-testers of exploitable inter applications communication errors related to the released applications. The investigation of this kind of tools has already been done by Ul Haq et al. in [15], and J. Mitra in [13];
- Tools that are integrated in the IDE but are not concerned by security vulnerabilities, like PMD [28], and CheckStyle [29]. These tools are used for checking coding standards, class design problems, but cannot be used for identifying code smells related to security issues;
- Industrial tools such as: Fortify [30] and Checkmarx [31], which are well-known tools, but we were unable to install them due to their pay access policy.

2) **Shallow analysis**: In this phase, we conduct an evaluation of the IDE plugins security coverage against the list of vulnerabilities presented in Section V-B. This analysis is *shallow* because it is only performed through investigating the available documentation and/or the published papers. Three teams are constituted: Team1 is the first author. Team 2 is a group of three final year students (Rémi AGUILAR, Tristan Boura and Nicolas MEGE) of a cyber-security curriculum in Grenoble Alpes Univ., France. Team 3 is composed of all the authors. The dig of the documentations is performed by different teams in three iterations. Team 2 conducted the initial investigation of all vulnerabilities. Then, Team 1 performed a second iteration. Finally, the results were reviewed by Team 3.

3) **Deep analysis**: In this phase, we perform an experimental analysis that completes the shallow analysis. It consists of performing an empirical evaluation of the selected tools against a subset of vulnerabilities. The evaluation process is realized by Team 2 led by Team 1 and reviewed by Team 3. It was organized according to agile practices. The backlog describes the tasks focusing on the list of vulnerabilities to check on the plugins:

- To DO branch: contains the list of vulnerabilities to be investigated during the week;
- Doing branch: contains the list of vulnerabilities under investigation, this helped to perform a quick feedback between the team members;
- Review: contains the list of investigated vulnerabilities during the week;
- Done: contains the work verified and validated.

To accomplish each task among those presented in the backlog, we follow an ordered list of steps described in Fig. 2 and below:

a) We install the various tools and plugins on the Intellij IDE, the goal is to check whether a tool detects a given

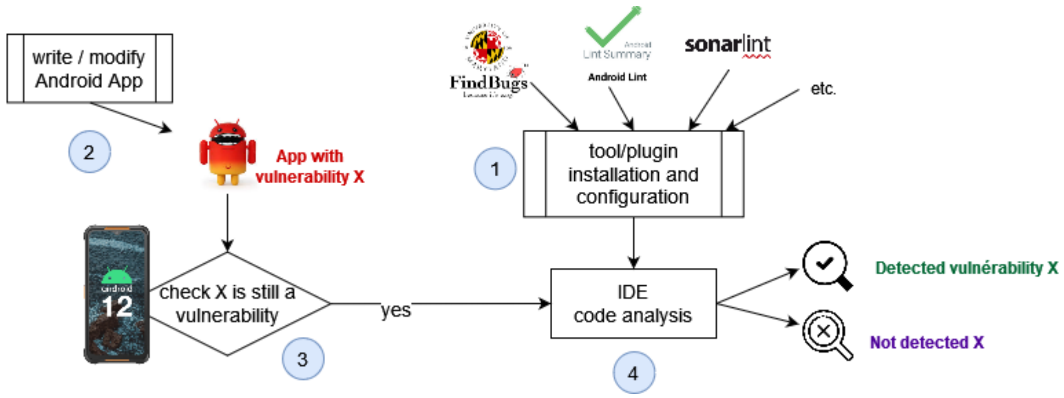


Fig. 2. Deep Analysis Process (experimentation part)

vulnerability in the entry application.

b) We write or modify Android applications in order to have vulnerable applications. The vulnerabilities are those we selected from the Ghera repository [14].

c) We check that Android 12 does not prevent the vulnerability (the issue has not been closed yet). It is useless to require from a tool to detect ”old” vulnerabilities.

d) Once we ensure that the vulnerability could be exploited on the recent Android version, we analyse the vulnerable application by the selected IDE plugins and we report the results.

This evaluation has been conducted only with available and free tools such as SonarLint [18], FindBugs [19], Android−Lint [20], lint −icc [23], and FixDroid [32]. We attempted to experiment more tools but it was not possible due to the unavailability of the tools. We contacted the authors of PerHelper [33], 9Fix [34] and Vandroid [35] but we did not receive an answer yet. Consequently, we decided to perform a second iteration on the documentation analysis for the unavailable tools instead of experimenting them (which was not possible). Finally, as our study focuses on vulnerabilities that could be found at the code level (*cf.* Section V-B), our deep analysis could not apply to tools such as Sema [36], PoliDroid−As [37], Page [38] because the inputs of these tools are respectively: GUI Storyboards for Sema [36], Textual specification of the application for PoliDroid−As [37] and Page [38], and not the application source code.

In the following, we present our framework in Section V, the selected tools in Section VI and the analysis and evaluation results in Section VII.

V. Classification Framework

The classification framework is a comprehensive analysis of the relevant criteria of security concerns in the development cycle of Android applications. Our classification framework contains four main dimensions as illustrated in Fig. 3.

- **IDE plugins**: As a first dimension the goal is to define the IDE plugins Where the presented security vulnerabilities will be checked. We conducted a review of significant related works and identified tools to aid in secure code development. We thoroughly discussed each tool, its capabilities, and the mitigation process during development. Finally, we determined the vulnerabilities covered by each tool.
- **Design level**: This dimension explores When the selected tools could be employed regarding the entire software development life cycle. The study in this dimension is conducted based on the engineering phases presented in Section V-A.
- **Analysis approaches**: In this dimension, we manually inspected How the selected tools behave to analyse security vulnerabilities. This enabled us to figure out the adopted analysis approaches of IDE security plugins regarding the analysis methods presented in Section V-C such as Fuzzing, Instrumentation, Symbolic Execution, and Formal verification.
- **Security vulnerabilities**: In the final dimension, we examine the selected tools effectiveness against a non exhaustive list of vulnerabilities that we selected from the Ghera repository [14] (*cf.* Section V-B). The goal is to investigate What are the well known vulnerabilities covered by the selected tools. In addition, we tested each vulnerability based on corresponding attack scenarios on recent versions of the Android OS to confirm the associated risks.

Next, we present each dimension in detail.

A. Secure Development Life-cycle (when)

Software developers are pointed out in many exploratory studies [40] [41] as the main reason of security vulnerabilities. This is because they often consider security as an afterthought concern. Software developers are mostly not security experts. Considering security requirements would need many interactions with software security experts, hence adding delays to the core software development. Consequently, these interactions are not considered in the development life cycle like many non-functional requirements, while functional testing starts in the requirements analysis phase by providing user scenarios that will be the source of acceptance testing. To overcome these limitations, software organizations have recently initiated new software development paradigms allowing to secure the SDLC [42], [43] (see Fig. 4 borrowed from [39]). The goal is to ensure security requirements throughout the entire software development pipeline: requirements analysis, design, coding, testing deployment/runtime and decommissioning. With this regards, software development and operations (DevOps) approaches are of high interest as one of their objectives is to improve communication and interaction between involved actors in the software development process.

- <u>Specification</u>: considering security at software specification level is recommended at the head of the 10 proactive controls

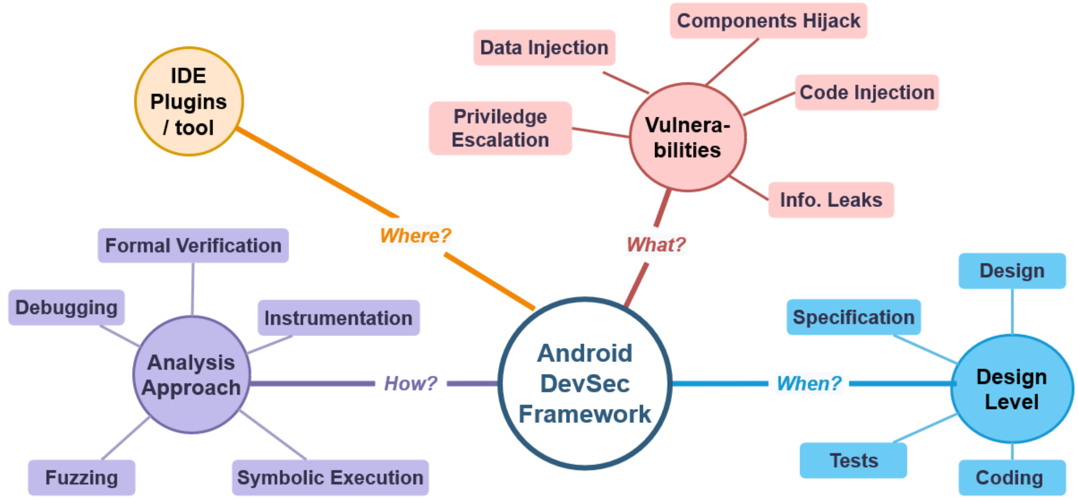


Fig. 3. Android DevSec Classification Framework

list of OWASP[4]. The goal is to ensure that the software design meets security requirements and minimizes the risk of security vulnerabilities. Potential vulnerabilities analysis, security requirement and risk assessment become part of this phase. Security requirements define or redefine features to solve specific security problems or eliminate potential vulnerabilities.

- Design: At this level, security may be ensured based on the following steps: (1) to conduct threat modeling by designing potential security threats and vulnerabilities; and (2) to select well known design patterns for fundamental aspects such as: access control, encryption, etc.
- Implementation: Considering security during the coding phase can help to identify and fix security vulnerabilities early in the development process, before they become serious issues that could be exploited by attackers. At this stage, security could be ensured in many ways such as: implementing applications by following secure coding best practices, using static and dynamic code analysis, using secure libraries, and automate security checks.
- Testing: considering security at the testing stage by performing different types of structural and functional tests such as: unit, instrumentation, fuzzing tests. *Pentests* or security tests should also be applied at this stage.
- Deployment & Post-Deployment: at this stage, operational security processes and tools are needed. In particular, they include testing build routines and user acceptance testing.

One of the first approaches that have been proposed to consider security needs in the software development lifecycle is the Security Development Lifecycle (SDL) by Microsoft. SDL defines twelve stages, in which objective is to consider security throughout the whole development process. Among these stages, education and awareness, security best practices, security documentation, etc. have been highlighted. Today, these security specific stages are implemented throughout security practices.

OWASP has strengthened the Software Development Lifecycle with a Software Assurance Maturity Model (SAMM) that encompasses security aspects throughout the development process. In the OWASP SAMM model, security governance is considered as a specific stage and highlights, as in the Microsoft Secure Development Lifecycle, the importance of training and education. Traditionally, IT people were in charge of security testing, once the application has been released. Such delayed security tests are no longer sufficient in the nowadays context, with highly interconnected applications and numerous security breaches.

Thus, *DevSecOps* approaches have been introduced. At the core of DevSecOps is the principle of keeping security as a priority and adding security controls and practices into the DevOps cycle [44]. All these initiatives that aim to handle security aspects put the light on the methodological concerns including models, processes and people, tools and automation.

In the context of Secure Android Apps Development Life-Cycle, models are those of the vulnerabilities, actors are mobile app developers, process, tools and automation are implemented and integrated in IDEs. We will detail these aspects in the next sections.

B. Security vulnerabilities related to the development process (what)

This section introduces the identified vulnerabilities in the recent Android versions and illustrates the attack scenarios we will implement.

1) Identified Vulnerabilities: In this work, we are mainly interested in security vulnerabilities **(Vi)** that could be mistakenly introduced by developers and exploited to craft attacks **(Ai)**. Based on available benchmarks such as *Ghera* [14] that contains open source applications implementing vulnerabilities, we started by considering vulnerabilities (V), which belong to the following class of attacks (A): (i) privilege escalation, (ii) data injection, (iii) code injection, (iv) information leaks and (v) components hijacking. These vulnerabilities are summarised in Fig. 5. Below, we briefly describe vulnerabilities for each class.

1) **A1**. Privilege escalation **(PE)**: this attack occurs when an application with less permissions gains access to the components of a higher privileged application. Situations where such an attack can occur are mainly related to:
   - **A1.V1**. The use of *PendingIntent* with empty *base action*: a *PendingIntent* object is a token that is given to a foreign application to allow it to execute a predefined action. When a *PendingIntent* is sent, the receiver application will execute the corresponding action using the sender permission. If a malicious app receives a *PendingIntent* whose base action is empty, then the malicious app can escalate its own privilege,

[4] https://owasp.org/www-pdf-archive/OWASP_Top_10_Proactive_Controls_V3.pdf

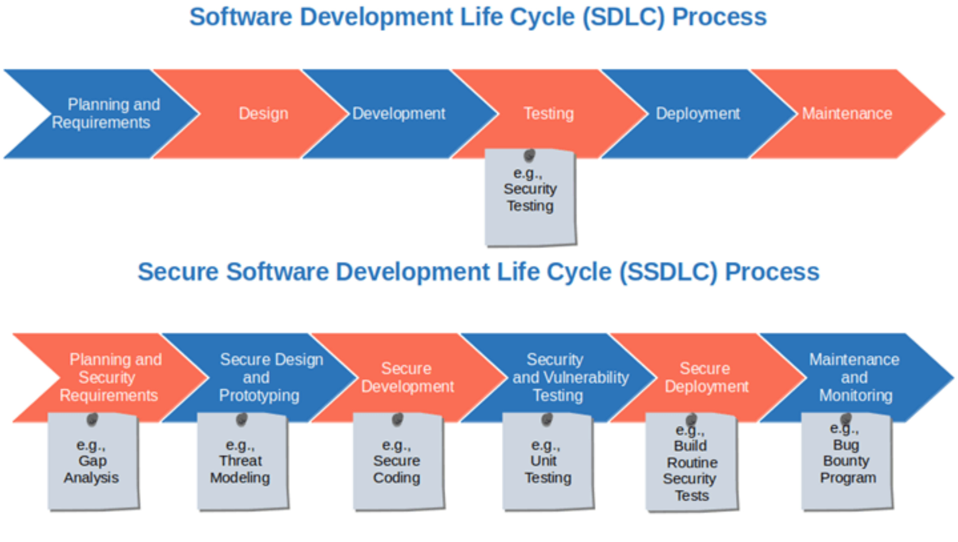

Software Development Life Cycle (SDLC) Process
Planning and Requirements
Design
Development
Testing
Deployment
Maintenance
e.g., Security Testing
Secure Software Development Life Cycle (SSDLC) Process
Planning and Security Requirements
Secure Design and Prototyping
Secure Development
Security and Vulnerability Testing
Secure Deployment
Maintenance and Monitoring
e.g., Gap Analysis
e.g., Threat Modeling
e.g., Secure Coding
e.g., Unit Testing
e.g., Build Routine Security Tests
e.g., Bug Bounty Program

Fig. 4. Secure Software Development Life-cycle (SSDLC) [39]

set an action and execute it in the context of the benign app that sent the *PendingIntent*.

- **A1.V2**. Fragments Dynamic Load: Java reflection is used to dynamically load fragments into an activity, where a fragment is a part of a user interface. But, the fragment name could be provided by a malicious application to make fragment injection attacks. For example, an activity that accepts fragment names as input from other components, and loads the fragment dynamically into the activity is vulnerable to executing a fragment provided as input by a malicious app on the device.
- **A1.V3**. High Privileged component export: If an app has the permission to perform a particular privileged operation and if that operation is performed via an app component that is exported for public consumption, then a malicious app can invoke the component method that performs the privileged operation and make the app perform the operation on behalf of the malicious app. This is a privilege escalation attack or a confused deputy attack.
- **A1.V4**. Permission over-privilege: Android applications privileges are managed through the concept of Permissions. A Permission is declared on the XML manifest configuration file as an XML attribute. Declaring a Permission on the Manifest file means that the application needs this permission to access a system resource(s) such as: CAMERA, GPS, etc. The security design flow here is when developer mistakenly declares permission(s) on the manifest file that are not necessary for the application functioning. Malware could gain these permissions to create harmful actions.
- **A1.V5**. Permission Enforce: If an application uses the method *enforcePermission* to grant permission at run-time, it won't throw a SecurityException when another component in the same process was granted permission earlier. Thus a malicious application could get privilege permission. The exploit of the attack scenario related to this vulnerability does not work on Android 12, 11 and 10 (API 29..33) but works on Android 9 (API 28).
- **A1.V6**. EnfroceCallingOrSelfPermission: When the method enforceCallingOrSelfPermission is used to grant permission, it will not throw further SecurityException from the moment it has already granted one permission to another application, meaning a malicious app could ask a permission after a benign app, and get that permission granted.

2) **A2**. Data injection: Data injection is a type of attack where malicious actors are able to inject malicious data into a software system. It has serious consequences on sensitive data, access control, and quality of service. In Android ecosystem, this can occur through the exploitation of various vulnerabilities, such as:

   - **A2.V1**. Ordered broadcasts: As mentioned in Section II, a broadcast is a component that allows an application to send and/or receive messages (intents) to/from the applications and/or the system. Each component could be registered to a broadcast to be notified whenever this broadcast is generated. On its configuration, a broadcast receiver can be declared as either ordered or non-ordered. In non-ordered mode, Android will deliver all broadcasts to all receivers at the same time. On the other hand, ordered broadcast receivers define a priority of transmission. The receiver with higher priority responds first and forwards it to lower priority receivers. Hence, a malicious receiver with high priority can intercept the broadcast, change its content, and forward the malicious payload to the receivers with low priority.
   - **A2.V2**. Sticky broadcasts: A sticky broadcast in Android is a broadcast message that is saved by the system and sent to all registered receivers. This type of broadcast can be potentially dangerous because a malicious receiver can modify the message and broadcast it again, causing all receivers to receive the updated message. Sticky broadcast could be sent using the method *sendStickyBroadcast()*, which is deprecated in the latest version of Android. Starting from Android 8.0 (Oreo), the use of sticky broadcasts is discouraged.
   - **A2.V3**. *Weak Checks On Dynamic Invocation:* to invoke any provider-defined method. Android does no permission checking on this entry into the content provider. Whenever the developer calls this method without doing its own permission checks, unauthorised components are allowed to interact with the content provider. So, apps that use the call() in the Content Provider API are vulnerable to exposing the underlying data store to unauthorized reads and writes.
   - **A2.V4**. *Uncontrolled External Storage Reads* are used to store application data with a public share. If an application reads from any file stored in *External Storage*, even if the file is in the private storage directory, then the application has no control over the file it is reading. If a malicious app changes the file being read by a victim application, then this

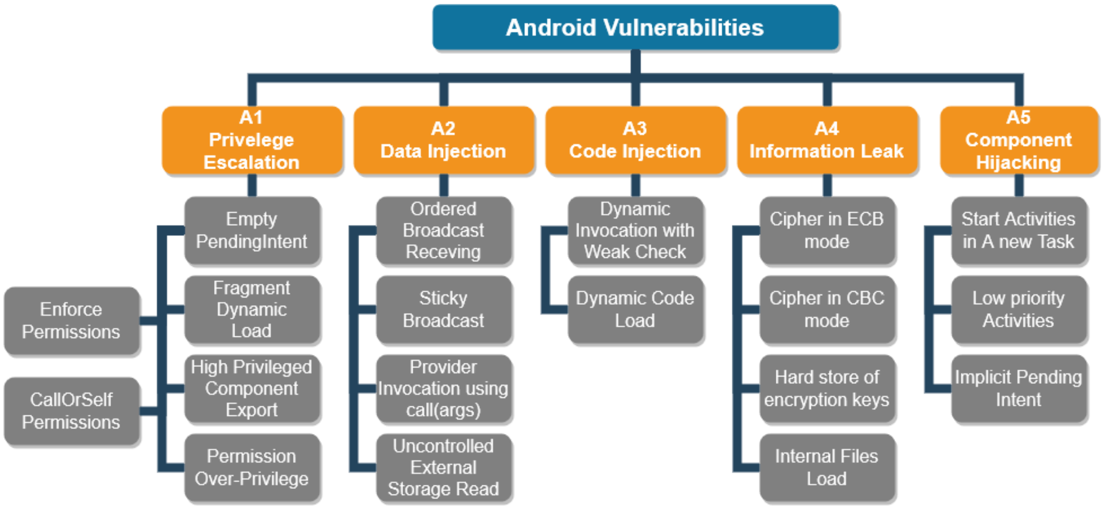


Fig. 5. A taxonomy of Android vulnerabilities

one will unknowingly read malicious content.

3) **A3**. Code injection: consists of injecting potentially malicious code that is after that interpreted/executed by the application. Situations where such an attack can occur are mainly related to:
   - **A3.V1**. Dynamic code loading: Since the apps can load classes from local archives (via PathClassLoader) or remote archives (via URLClassLoader), malicious actors can change such archives and affect the behavior of apps that use the archive. So, Android apps that rely on dynamic code loading without verifying the integrity and authenticity of the loaded code may be vulnerable to code injection.
   - **A3.V2**. Dynamic invocation with weak checks: Android allows developers to share data across apps through the Content Provider API. The Content provider API provides a method call() to call any provider-defined method. But, Android has no idea whether call will read or write data in the provider, so it cannot enforce those individual permissions. Therefore, malicious apps with read and without write permission can write data through the Content Provider.
4) **A4**. Information leak: It occurs when an application private data are accessed by unauthorised applications or when developers improperly use the cipher modes (ECB, CBC) and AEAD (Authenicated Encryption With Additional Data) cipher like GCM (Galois Counter Mode) or by exploiting weaknesses in random number generation (RNG) process. So, an attacker could be able to guess the encrypted message. Situations where such an attack can occur are mainly related to the use of:
   - **A4.V1**. Block Cipher algorithm in Electronic Codebook Block (ECB) mode: ECB mode leaks information about the plaintext because identical plaintext blocks produce identical ciphertext blocks, it does not hide data patterns well. An attacker could be able to guess the encrypted message. So, applications that use a block cipher algorithm in ECB mode for encrypting sensitive information are vulnerable to leaking sensitive data. Thus, ECB is not recommended for use in cryptography protocols.
   - **A4.V2**. A block cipher algorithm in cipher block chaining (CBC) mode: The greatest advantage CBC has over ECB is that, with CBC mode, identical blocks do not have the same cipher. This is because the initialization vector adds a random factor to each block; hence, the same blocks in different positions will have different ciphers. But, applications that use a block cipher algorithm in CBC mode with a non-random initialisation vector for encrypting sensitive information are vulnerable to leaking sensitive data.
   - **A4.V3**. Applications that store the encryption key in the source code are vulnerable to leaking encrypted information. An attacker with decompiler and static analyzer tools can identify the encryption function and find encryption key. Someone who has the assembly code will even be able to call Decrypt directly, not bothering with extracting the key.
   - **A4.V4**. Ability to load files from internal storage (private access) to external storage (public access). Android allows apps to save data in files. These files can be stored in Internal Storage or in External Storage. Files stored in Internal Storage are private to the app. External Storage can be accessed by all apps that have permission to access ExternalStorage.
5) **A5**. Android components hijack: this attack can occur in the following situations:
   - **A5.V1**. Activities that start in a new task: This situation occurs when the user can navigate from a benign activity A to a malicious activity M by pressing the back button, instead of navigating to an intended benign activity C. Therefore, malicious activity M will hijack benign activity C.
   - **A5.V2**. Applications with low priority activities: If a malicious activity has a higher priority than a benign one, then it will appear before the benign activity, making it more likely for the user to choose the malicious activity over the benign one.
   - **A5.V3**. Pending Intent with implicit base intent: If base intent is an implicit intent then, when the pending intent is performed, it can be intercepted by a malicious application. If the implicit intent has sensitive information in it, then it will be leaked.

As we can see, developing Android applications without a prior knowledge and/or a specific focus on security aspects could lead to critical security attacks. It is worth-noting that the above list of vulnerabilities is not exhaustive, and it is intended to be extended in future research.

2) Attack scenario: To clearly outline the exploits utilizing these vulnerabilities, we provided, based on Ghera repository a scenario of attack related to each vulnerability. Each scenario was documented using Gherkin, a language supported by Martin Fowler[5] for outlining functional software test scenarios. In this Section we describe as an example the attack scenario related to vulnerability *A1.V1*. The remaining scenarios can be found in the technical report [45].

Given A simple application called BenignApp
And BenignApp has a module called BenignAppPartner
And BenignAppPartner implements a broadcast receiver
When BenignApp sends a pending intent with empty action to the BenignPartner
Then BenignAppPartner receives this Pending Intent, then launches a remote service that resides on the benign app (until now, it is a normal behavior)
When MalicousApp intercepts through its broadcast receiver the Pending Intent sent to the BenignPartnerApp
Then MaliciousApp manipulates the empty action of this Pending Intent
And MaliciousApp executes the action it wants into the Benign app' service by escalating its own permission

In order to test **A1.V1**, we use three apps: Benign, BenignPartner, and Malicious. Benign is a benign app that sends an empty pending intent to BenignPartner, which has a broadcast receiver MyReceiver.java that takes the pending intent and starts a service MyService.java in Benign app. Malicious app has a broadcast receiver MyReceiver.java that intercepts the pending intent sent from Benign app and starts an internal service in Benign that performs some sensitive operation. This service is not exported and is meant for internal use within the benign app. However, because of the empty pending intent intercepted by the malicious broadcast receiver, Malicious app can escalate its own privilege and can start MySensitiveService eventhough it is not supposed to.

C. Security analysis Approaches and Tooling (how)

Determining the effectiveness of each security analysis tool in detecting known vulnerabilities is closely related to the analysis method used by the tool. Hence, in order to examine the studied IDE plugins effectiveness based on their analysis techniques, we describe in this section common analysis approaches used in software engineering. As summarized in Table I, these approaches are generally classified into 3 groups: Static, Dynamic, and Hybrid analysis.

- **Static analysis approaches**. Static analysis inspects the program without running it. It examines source or compiled binary code against coding flaws. Generally, it is used to identify potential bugs, vulnerabilities, or other security issues. Many automated tools are able to perform static code analysis to report security problems related to Android applications and systems,

[5] https://martinfowler.com/bliki/GivenWhenThen.html

such as: (1) ComDroid [27], which statically analyses DEX code of third party Android applications to detect inter application communication vulnerabilities: Broadcast theft, Activity and Service hijacking, and Broadcast injection; (2) Lintent [46], which is based on formal verification techniques (formal calculus) to reason about the application behaviour and prevent security attack surfaces from privilege escalation and communications; (3) FlowDroid [47], which tracks the flow of data between different components of an Android app. Starting from an Android application, it constructs a full control flow graph. This graph can then be used to identify potential security issues based on data tainting; (4) Arcade [48], NatiDroid [49] and Pscout [50] that perform static analysis on the Android framework to build a permission mapping used for detecting over-privileged applications.

Several techniques could be used to perform static analysis in Android application security analysis, such as formal verification and symbolic execution:

- **Symbolic Execution** can be used to simultaneously explore multiple paths that a program could take under different inputs. This allows sound analyses that can give strong guarantees on the checked property. Rather than taking on fully specified input values, the technique abstractly represents them as symbols, resorting to constraint solvers to construct actual instances that would cause property violations [51]. These symbols will be used along the execution path to determine the path condition. A path condition is a first order logic formula that describes the conditions satisfied by the branches taken along that path. The mapping between variables and symbolic expressions or values will be stored to determine if the property is satisfied at the end of the the execution path. Klee [52] is a symbolic execution tool built on the LLVM compilation framework that automatically generates test cases for high coverage of complex and environmentally intensive programs. *Symbolic Execution* can be conducted on the basis of some pre-extracted models to ensure the reachability of some branches [53]. Many approaches, such as: Scandroid [54], Cassandra [55], etc., analyse data-flow to formally check various system level information-flow properties.
- **Formal Verification**. Formal methods are among the most used techniques in Android security analysis. They cover many application areas such as: security protocols, the implementation of access control policies, intrusion detection and even source or binary code security. Generally, they are categorized in three families as presented in [56]: 1) Specification and Process algebra where the approach that deals with the system behavior is algebraic and axiomatic. 2) Model checking or Property checking, which is a model-based specification technique that aims to develop visual models for the specified system and analyzing their properties. 3) Theorem proving that is an axiomatic technique, in which the system is expressed as a set of axioms and a set of inference rules, and the desired property is expressed as a theorem to be proved. For Android security, this technique has been widely adopted such as in [57] for mobile malware analysis based on model checking. Theorem proving techniques provide a high level of code coverage. However, they suffer from over-approximation results.

Static analysis approaches have some limitations due to undecidability problems [58]. It is impossible to determine if a program will terminate for a given input. Another limitation of static analysis tools is the fact that they report problems that are not really bugs in the code [59] i.e. they identify incorrect behaviors that cannot occur during any run of the program (false positives).

- **Dynamic analysis approaches**. In contrast to static analysis, dynamic analysis is performed at run-time. The goal is to identify problems that cannot be detected by static analysis, such as: race condition, performance, and concurrency problems. In comparison to static analysis, dynamic analysis provides sound results. It does not require an access to the source code, it traces a binary code to understand the system behaviour.

  Many automated tools have been proposed to perform dynamic security analysis on Android applications and system, such as: TaintCheck [60], LIFT [61], Valgrind [62], just to name a few. They instrument the bytecode to control the information flows in the program and detect security attacks. Thus, they suffer from significant performance overhead that does not encourage their use in real-time applications.

  Several techniques could be used to perform dynamic analysis in Android application security analysis, such as fuzzing and instrumentation:

  - **Fuzzing** is an automated technique used for software testing and security. The main idea behind fuzzing is to use randomly generated inputs to fuzz the software under analysis in order to find bugs or vulnerabilities. The set of inputs are called *Oracle*. In software testing, fuzzing could be performed based on different methods, such as: (1) generation-based fuzzing that involves generating inputs from scratch; (2) mutational-based fuzzing (called also feedback-based fuzzing) that modifies the existing inputs during the testing process, in order to redirect the execution paths; and (3) model-based fuzzing that constructs a model for the input data, then generates inputs that conform to that model.

    Fuzzing could be conducted dynamically. It has been used in significant works analysing security vulnerabilities and attacks in Android applications. The results showed that it could cover numerous boundary cases using invalid data as application input to better ensure the absence of exploitable vulnerabilities [63]. Among the existing tools allowing to apply fuzzing for Android security analysis we found: (1) Jazzer, which is a coverage-guided fuzzer for the Java Virtual Machine (JVM). It works on the bytecode level and can be applied directly to compiled Java applications and to targets in other JVM-based languages such as Kotlin or Scala. It is composed by a native binary that links in libFuzzer and runs a Java fuzz target through the Java Native Interface (JNI) and by a Java agent that runs in the same JVM as the fuzz target and applies instrumentation at run–time. These fuzzers can often find out previously unknown vulnerabilities [64]; (2) Dynamo [65] uses mutational based fuzzing to provide inputs that fuzz the Android framework code related to checking the API access control. This process helps to construct a permission-mapping allowing developers to identify over-privileged applications.

    Despite the advantages that fuzzing can offer it suffers from some limitations such as: 1) Low code coverage due to the fact that the executed paths are related the selected inputs; 2) The set of inputs could be computationally intensive; and 3) Inadequate inputs generation, which can result in a high number of false negatives.
  - **Instrumentation.** Generally, fuzzing is combined with instrumentation techniques to perform dynamic analysis. Instrumentation provides the possibility to modify a program under analysis at run-time, and add some functionalities such as: logging messages or new instructions in order to understand the software behaviour.

    There are several toolkits, e.g., Frida [66], ARTIST [67], DaVinci [68] that are used for dynamic instrumentation in Android system. Frida allows developers, reverse-engineers, and security engineers to inject snippets of JavaScript into

TABLE I
SOFTWARE ANALYSIS APPROACHES: STRENGTHS, WEAKNESSES, AND TOOLS

Approach	Technique	Pros	Cons	Tools
Static Analysis	Formal Verification, Symbolic Execution	Efficient, Scalable, High Code coverage	Decidability, Low Soundness	ComDroid, Lintent, FlowDroid, NatiDroid
Dynamic Analysis	Fuzzing, Instrumentation, Debugging	Race Condition, Concurrency, Performance	Time Consuming, Resource-Intensive	Frida, Dynamo, stowaway

native Android apps to monitor and debug running processes. ARTIST is an open source instrumentation framework for Android's apps and Java middleware. It is based on the new ART runtime and the on-device dex2oat compiler to monitor the execution of Java and native code. ARTIST modifies code during on-device compilation. In contrast to existing instrumentation frameworks, it preserves the application's original signature and operates on the instruction level. DaVinci is an Android kernel module for dynamic system call analysis. It provides pre-configured high-level profiles to easily analyze the low level system calls.

The instrumentation adoption faces also some challenges. First, adding some functionalities to a software system will add significant complexity, which may cause bugs and crashes. In addition, there is a high risk of execution interference between the real and the added code. Finally, instrumentation can have a significant impact on performance because the added code can slowdown execution time.

The dynamic analysis limitation is that it consumes resources and has difficulty covering all execution paths, so it generates false negatives.

- **Hybrid analysis approaches.** Hybrid analysis combines and benefits from the advantages of static and dynamic analyses. Regarding the provided tools performing static or dynamic analysis, fewer tools combine both of them due to the difficulty of combination. For instance, AM Detector [69] is an automatic malware detection prototype system based on attack tree model. This approach employs a hybrid static-dynamic analysis method. Static analysis tags attack tree nodes based on application capability. It filters the obviously benign applications and highlights the potential attacks in suspicious ones. Dynamic analysis selects rules corresponding to the capability and conducts detection according to run-time behaviours. In dynamic analysis, events are simulated to trigger behaviours based on application components, and hence it achieves high code coverage.

Overall, static and dynamic analysis are complementary approaches. We presented in this section the commonly used techniques for detecting security issues related to Android applications. Starting from the presented strengths and weaknesses of each technique, we investigate in the coming section the based-analysis approach of the selected IDE plugins in order to study their effectiveness in detecting the selected vulnerabilities.

VI. SELECTED TOOLS (RQ1)

The result of the *Tools search & selection* (phase 1) of the research methodology is a set of 16 IDE plugins that match the selection criteria provided in Section IV-C. This set is an answer to **RQ1**. The list of selected tools is summarised in column 1 of TABLE II. In the reminder of the section, we overview these plugins.

- **Curbing** [5] is the first proposed tool assisting developers in utilizing least privilege principle when developing Android applications. It notifies developers if one or more unnecessary permission(s) is (are) declared unintentionally by the developer. At the time when Curbing was developed in 2011, there was no ready data-set of Android application source code. So, the tool was analysed empirically,
- **SonarLint** [18] is a popular linter that is largely used in industrial IT projects. It contains a large set of code smells for many programming languages, including those for Android security. The tool can also provide suggestions for remediation. SonarLint performs static code analysis (SAST) by whether inspecting the program AST, or using data taint propagation. It also performs dynamic analysis (DAST) even if it is still limited to few dynamic functionalities such as: Memory Error Detection, Invariant Inference, Concurrency Error, which may cause race-condition, resource/memory leak, etc. SonarLint can also be used in many ways: IDE integration (Eclipse, IntelliJ, Android Studio, etc.), or through DevOps pipeline (github, jenkins, etc.),
- **FindBugs** [19] provides static code analysis to look for bugs in Java code from within IntelliJ IDE. For more than 200 bug patterns related to different topics, such as performance, correctness, bad practices, FindBugs also provides the opportunity to investigate the code security, and detect the malicious code vulnerability. As a code review tool, it investigates the AST of the program against predefined rule patterns. The last available version (2016) of FindBugs is not compatible with the final version of IntelliJ. Since 2016, FindBugs is integrated into FindSecBugs [70] tool.
- **Snyk** [21] is one of the leading industrial vulnerability analysis tools. It has a large open-source database of the common vulnerabilities related to different programming languages including Java and Kotlin for Android application development.
- **Android-Lint** [20] (also called Lint) is the official analysis tool provided by Google for Android application development. It checks common issues in an Android project's source code and provides suggestions for improving the code's quality and security. Lint offers a robust API that can be utilized to create custom Android lint checks for additional analysis rules. Several tools based on this API have been found in the literature.
  - **Lintent** [22] is a security analysis plug-in integrated with the Android development tools suite. Based on a static analysis approach, it implements a static formal calculus based analysis to reason on the Android inter-component communication API. Lintent analyzes Java source code through reasoning about types. The goal is to statically prevent privilege escalation attacks on well-typed components. Lintent also detects over-privileged application based on the permission mapping described on the corresponding readme description file of the tool in github [6]. To analyse the program elements, Lintent is built on top of Android–Lint library,
  - **icc-lint** [23] called also AndroidICC. Like Lintent and 9Fix its security checker is based on Lint Checker. It serves to analyse Android application source code and reveals vulnerabilities related to inter-component-communications. These vulnerabilities are extracted from *Ghera* repository. This tool is open-source and ready for use,
  - **9Fix** [34] is another recent plugin supporting secure program-

[6] https://github.com/alvisespano/Lintent

ming. It inspects the vulnerable code and instantly suggests an alternate secure code for developers. The security properties covered by 9Fix are classified as general security code smells related to different topics such as password stealing, the use of vulnerable cryptography algorithms, SSL and TLS communications, just to name a few. 9Fix is integrated as an Android Studio plugin.

- **PerHelper** [33] is an IDE plugin that guides developers to declare permissions automatically and correctly and identify permission-related mistakes, such as over-privilege and under-privilege permissions, unprotected APIs, etc. By inferring the candidate permission sets, PerHelper helps to set permissions more effectively and accurately.
- **VanDroid** [35] is an Analysis tool based on Model Driven Reverse Engineering approach (MDRE). In a first step, Vandroid analyzes the code source of the application based on its Graphical Abstract Syntax Tree (GAST), then generates a corresponding security model that facilitates and improves accuracy of the analysis. As a second step, it launches a formal verification process that identifies security risks related to the Android communication model. The formal verification in Vandroid checks the satisfaction of specified security properties in the generated model. Vandroid is provided as an Eclipse plugin, which limits its adoption on the current Android development projects. We contacted the corresponding authors of Vandroid by email in order to gain access for experimenting the tool. The tool cannot be used in new IDE supporting Android development such as IntelliJ and Android Studio.
- **Sema** [36] is a new methodology for designing secure Android applications from an app's storyboards. Storyboards are used as state-transition diagrams. The states specify the screens of the application, and the transitions present the operations to be launched to transit from one screen (state) to another one. The goal of Sema is to help application designers to care about security at the design stage. While specifying the application functioning and designing storyboards, Sema has the ability to check security properties at this stage based on formal analysis. Finally, Sema is able to generate a secure source code starting from the designed storyboards. The tool is open-source and can be integrated in Idea IDEs such as IntelliJ and Android Studio.
- **PoliDroid-As** [37] is a tool allowing Android developers to check if the created code adheres to privacy security policies. These policies are not created by developers but by legal experts using natural language. The main contribution of PoliDroid−As is to automatically align the code source of the application to the specified security policies. By using natural language processing algorithms, PoliDroid-As maps key phrases existing in the documentation of the used APIs to low-level technical terminology in the API.
- **Page** [38] is a tool that supports developers creating natural language privacy policies during the development process. It is proposed as an Eclipse plugin to document privacy tasks and notes during the construction and testing of the application. There is no automatic alignment between the specified textual policies and the source code of the application. The goal of the tool is just to make an internal IDE repository to remind the developers checking the validity of these policies after each code evolution.
- **PermitMe** [71] has the same goal as Curbing, Perhelper and Lintent. PermitMe is a code review tool (CR) used to notify developers when they intentionally include extra permissions in their apps. By statically investigating the program Abstract Syntax Tree (AST). PermitMe decides whether a declared permission is used in the program or not. If at least one permission is not used, then the application is flagged as over-privileged.
- **Coconut** [72] has been developed on 2018 as an IDE plugin for Android Studio. It helps developers to handle privacy based on the concepts of annotation. Through heuristics (H), Coconut detects the code that handles personal data, and asks the developer to add an annotation that describes how and why the personal data is used. To simplify the handling of annotations, Coconut proposes a quickfix to automatically generate an annotation skeleton with several fields (i.e., 'dataType', 'frequency'). These fields could be filled automatically if they could be inferred from the code, or manually in the opposite case. Filling out the annotation makes the developers think to the use of private data, such as examining the collections manipulating the private data, checking that there is a legitimate reason to collect the private data, etc. Coconut is available and open-source.
- **FixDroid** [73] is an Android Studio plugin (it is not available on IntelliJ). It provides helpful security alerts, explanations and quickfixes whenever possible. It is updated periodically to add new features and fix software bugs.

TABLE II
IDE Security Analysis Tools for Android applications

Tool Name	Year	SDLC	Focus	Approach	Method	Availability	AV
Curbing	2011	Dev	Permission Over-privilege	Static, Manual	AST	No	2.2
Lintent	2013	Dev	Communication	Static	FM	Yes	4.x
PermitMe	2014	Dev	Permission Over-privilege	Static	AST	No	5.0
Page	2014	Spec	Privacy policies	Static	NLP	No	-
Vandroid	2018	Design, Dev	ICC	Static	FM	No	9.0
AndroidLint	2019	Dev	General Code Smells	Static	AST	Yes	all
Sema	2019	Design	General Security Properties	Static	FM	Yes	10
PerHelper	2019	Dev	Permission Over-privilege	Static	AST	No	10
PoliDroid−As	2017	Spec	Privacy security policies	Static	NLP	No	8
9Fix	2021	Dev	General code smells	Static	AST	No	12
SonarLint	2021	Dev	General code smells	Hybrid	DAST	Yes	12
Find Bugs	2016	Dev	General code smells	Static	AST	Yes	7
Cocunut	2018	Spec	Privacy policies	Static	H	No	-
FixDroid	2017	Dev	General Code smells	Static	AST	Yes	7
icc−lint	2019	Dev	ICC	Static	AST	Yes	10
Snyk	2014	Dev	General code smells	Hybrid	DAST	Yes	all

[1] AST: Abstract Syntax Tree; CR: Code Review; FM: Formal Methods; Spec: Specification;

[2] SD Stage: Software Development Stage; AV: Android Version; NLP: Natural Language Processing;

[3] DAST: Dynamic Application Security Testing

The following section presents the results of the analysis process.

VII. Analysis and Evaluation Results

We analysed the selected tools according to the classification framework of section V. The results are summarised in Fig. 2, and the technical report [45]. provides a detailed description for the performed analysis and results. As mentioned in Section IV-C, we proceeded in two steps, which results are exposed below.

A. Shallow analysis results

This section presents the analysis results deduced from reading available documentation, communications and research articles. A global overview is presented in Table II.

For each studied tool, we identify the software development stage (SDLC stage), the type of covered security vulnerabilities (Focus), the analysis approach (Approach), the analysis method (Method) and its availability (Availability). We also identify the Android Version (AV) attribute that represents the version upon which the tool is constructed. This will help to measure the tools updates.

1) Considered development phases for security analysis `(RQ2)`*:* It is broadly admitted that security concerns should be handled as early as possible during the application development lifecycle. Secure development life-cycle (SDLC) methodologies have been adopted by many software organisations, e.g., Microsoft through their Microsoft Security Development Lifecycle (SDL) [74], OWASP with their SDLC and Software Assurance Maturity Model (SAMM) processes [75], etc. The following outcome of our work aims at clearly identifying development phases during which studied tools could be used, thus helping developers in hardening developed applications with regards to security requirements. In our study, most identified tools can be used at one or several of the following development phases: specification, design, coding and testing, as described below.

- **Specification**: during this phase, security policies are acquired usually from natural language descriptions, stating mainly how private data should be managed. Such descriptions could be legal documents. As a consequence, application developers need to clearly state how applications collect, use, and share personal information, introducing relevant security policies in the design of the application. As shown in Table II, PoliDoid−AS, Page, Cocunut are used in the specification phase.
- **Design**: threat modelling should be part of the application design in order to allow designers to analyse the designed application architecture for potential security issues, that could be then mitigated. During this phase, issues related to the use of third-party components or libraries could also be handled. As shown in Table II, Sema through storyboards, and Vandroid through modelling are used in the design phase.
- **Coding & Testing**: analysis approaches could be used during this phase to assess the application code with regards to security vulnerabilities. As shown in Table II, Curbing, Lintent , PermitMe, Vandroid, 9Fix, AndroidLint, PerHelper, SonarLint, FindBugs, FixDroid, icc−lint , Snyk are used in the coding and testing phases. We combined the coding and testing phases here because we consider only solutions integrated in the IDE. Although coding and testing are two separate sub-processes in the software development life-cycle, in the context of our study, testing solutions serve only as a means of code review to identify coding flaws within the IDE. This classification does not include penetration tests or application post-deployment tests.

Referring to the results displayed in Table II, we classified the existing plugins by development activity. Fig. 6 maps identified tools with design phases where they can be used advantageously with regards to security enhancement during the development life cycle.

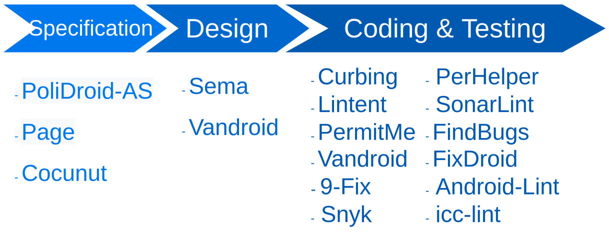


Fig. 6. Classification Per Design Level

On the one hand, we found that most of IDE plugins are considered at the *coding* phase of the development life cycle. They act as code review tools notifying developers about their "unconscious" security issues. On the other hand, few works allowing security checks at *specification*, *design* and *testing* phases have been proposed. As a consequence, efforts towards filling this gap are expected and could allow significant enhancement in tackling security issues.

2) Used security analysis approaches `(RQ3)`*:* We found that 88% of the adopted analysis approaches are static. AST analysis and formal methods are among the most used analysis methods by our sample of plugins.

- **Static AST Analysis**: Most of IDE plugins investigate statically the program Abstract Syntax Tree (AST) provided by the IDE such as SonarQube [18], Findbugs [19] and AndroidLint [20]. The goal is to extract information that enables to check the validity of predefined security properties patterns. Other tools, such as PerHelper [33], PermitMe [71] and Curbing [5] also investigate the **AST** to find the declared permissions in the application and the list of API calls requiring those permissions. The goal is to detect extra declared permissions that are not associated to any API call.
- **Formal Methods**: Other tools, such as Lintent [46] analyse the data-flow to formally check information flow with regards to security properties. Lintent [46] uses the **formal calculus** for reasoning on the Android inter-component communication API, and **type and effect** to statically prevent privilege escalation attacks on well typed components. In the same line, Sema [36] uses **formal verification** of security properties in order to generate a secure code.

As a consequence, when comparing our observations with the security analysis methods presented in Section V-C, we found that only some static ones are adopted by studied plugins. Dynamic and hybrid approaches are not referred despite their advantages. We underline this point in detail in Section VIII.

3) Considered security vulnerabilities `(RQ4)`*:* The visual matrix presented in Fig. VII-A2 summarises the plugins capabilities in analysing the considered security vulnerabilities (*cf.* Section V-B). The final results of the analysis (Fig. VII-A2) cover all the vulnerabilities of all the selected IDE plugins. For each category of attacks, we present which associated vulnerabilities are covered (or not) by the tools. We used dark colors to specify the results obtained following an experimental evaluations, while light colors were used to indicate results obtained from a deep analysis of the corresponding tool's documentation. Green colors are associated to the True Positive (TP) cases. This means that the tool has detected the vulnerability that actually exists in the application. Red colors are associated to the False Negative (FN) cases. This means that the tool has not detected the vulnerability that actually exists in the application.

In this first analysis iteration, we classify the analysis difficulty in three levels:

- Tools that are specialised in a specific and unique security concern were *easy* to investigate. Based on the corresponding published papers for the plugins: Curbing [5], PermitMe [71] and PerHelper [33]. They are clearly specialised in detecting privilege escalation attacks **(A1)** resulting from the extra use of permissions **(V4)** in the application. For other tools such as 9Fix [34], Fixdroid [73], and icc−lint [23], the list of covered vulnerabilities was explicitly declared in the related paper. So, it was easy to know that these tools detect **A3.V1** vulnerability.

- Tools specialised in detecting a specific type of attacks but the number of covered vulnerabilities is too large, are *less easy* to investigate. As an example, Lintent [46] could theoretically detect a large number of vulnerabilities as it formalises a notion of safety against privilege escalation. Based on the related published paper, it was not easy to decide whether the tool detects the vulnerability or not as the described formal model was too general. Fortunately, we found the list of covered vulnerabilities mentioned in the corresponding git repository [22]. Thus, we found that the tool covers:
  - EmptyPendingIntent **A1.V1** because the authors declares explicitly that the tool covers secrecy of pending intents,
  - Over-privilege application **A1.V4** because it contains the permission mapping for some Android APIs,
  - The remaining vulnerabilities related to privilege escalation **A1.V2**, **A1.V3**, **A1.V5**, and **A1.V6** are also covered following the github readme file where it is declared that the tool covers attack surfaces for privilege escalation,
  - It also detects **A5.V3** with Vandroid related to component hijacking.

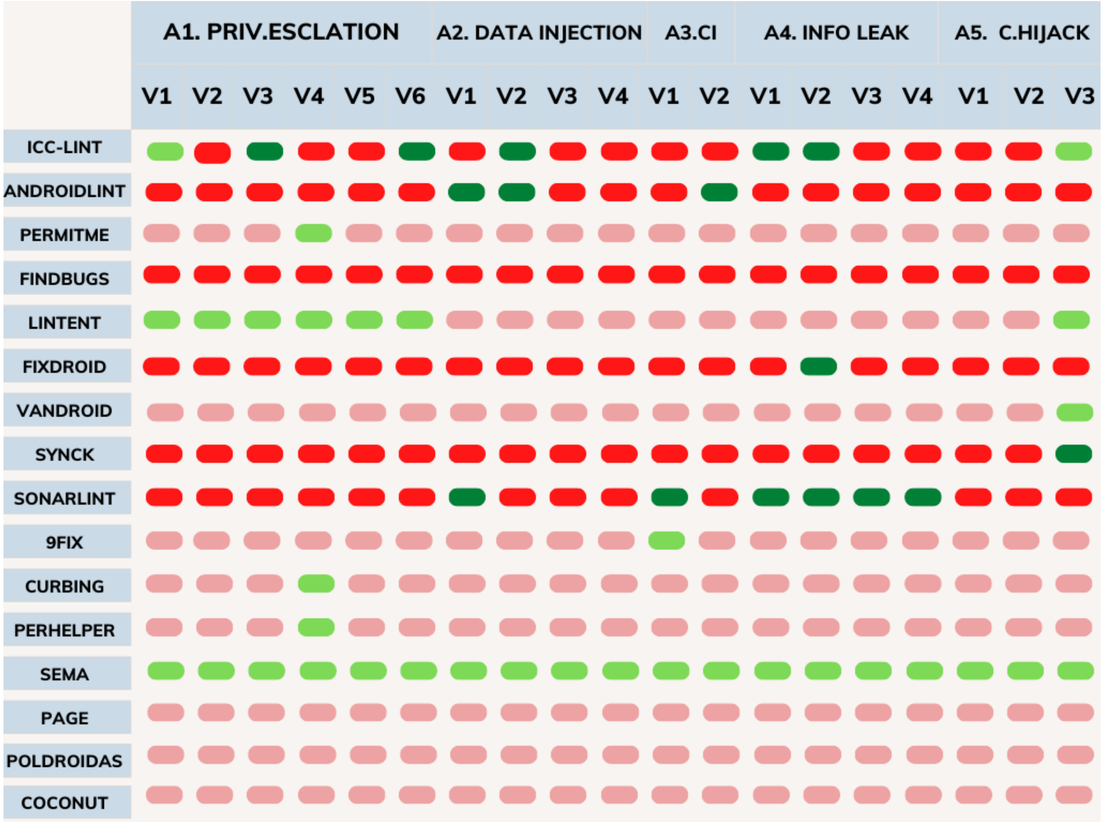
  For Page, Coconut and POLIDROID−AS, the inputs are security requirements specified by natural languages, so they are not specialized in detecting our list of vulnerabilities.

  For Sema [36] it is explicitly declared that it covers all the vulnerabilities present in *Ghera*. However, we could not experiment the tool as the inputs of Sema are graphical storyboards and not source code.
- Finally, for industrial tools such as AndroidLint [20], SonarLint [18], Snyk [21], and findbugs [19], it was hard to investigate the covered vulnerabilities based on the documentation. The scope of these tools is too general and the documentation is too large. We found that the following vulnerabilities: **V1.A2**, **A4.V1, A4.V2, A4.V3** are covered by SonarLint. For the remaining properties, we did not find any information indicating whether they are covered by these tools or not.

After performing a long analysis process of the documentation, we decided to confirm the obtained findings by a second analysis iteration, in the form of experimental work for the available tools. For the unavailable tools, a second analysis iteration of the corresponding documentation is performed by another team member. Details of this second analysis are explained in the next subsection.

B. Deep analysis results

The objective of this part of our study is to confirm shallow analysis results with an experimental evaluation using Android application benchmarks. Compared to our previous work [1], we extended our deep analysis process to cover all vulnerabilities related to the 5



	A1. PRIV.ESCLATION						A2. DATA INJECTION				A3.CI		A4. INFO LEAK				A5. C.HIJACK		
	V1	V2	V3	V4	V5	V6	V1	V2	V3	V4	V1	V2	V1	V2	V3	V4	V1	V2	V3
ICC-LINT	🟢	🟥	🟩	🟥	🟥	🟩	🟥	🟩	🟥	🟥	🟥	🟥	🟩	🟩	🟥	🟥	🟥	🟥	🟢
ANDROIDLINT	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥
PERMITME	🔴	🔴	🔴	🟢	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴
FINDBUGS	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
LINTENT	🟢	🟢	🟢	🟢	🟢	🟢	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🟢
FIXDROID	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥
VANDROID	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🟢
SYNCK	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩
SONARLINT	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟩	🟥	🟩	🟩	🟩	🟩	🟥	🟥	🟥
9FIX	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🟢	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴
CURBING	🔴	🔴	🔴	🟢	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴
PERHELPER	🔴	🔴	🔴	🟢	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴
SEMA	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢
PAGE	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴
POLDROIDAS	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴
COCONUT	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴

🟢 a true positive confirmed through the tool documentation without experimentation
🟩 a true positive confirmed through the tool documentation with experimentation
🔴 a false negative confirmed through the tool documentation without experimentation
🟥 a false negative confirmed through the tool documentation with experimentation

Fig. 7. Analysis Results - IDE plugins effectiveness in detecting known vulnerabilities

attack families: Privilege escalation, information leaks, data and code injection, and component hijacking.

We can observe that the deep evaluation confirmed that the following tools: Curbing [5], PermitMe [71], and PerHelper [33] are specifically oriented to detect over-privilege vulnerabilities (A1.V4); and not all the other vulnerabilities. Our deep evaluation also confirmed that none of the privilege escalation vulnerabilities are covered by SonarLint [18], FindBugs [19], FixDroid [32] and AndroidLint [20]. For SonarLint the documentation mentions the use of least privilege principle among the list of coding best practices to avoid A1.V4. However, during the analysis of the over-privileged application, SonarLint did not detect the vulnerability as shown in the technical report [45]. For icc−lint [23], the analysis of vulnerable apps revealed the presence of **A1.V3, A1.V6, A2.V2, A4.V1**, and **A4.V2** vulnerabilities. In the same line, **A1.V3** was also detected by icc−lint and fixDroid.

For A1.V2 and A5.V3, we marked the result because, even though the tool didn't detect them during the deep analysis phase, we found in the code two implemented rules for *EmptyPendingIntent* and *ImplicitPendingIntent*. We assume it is not a false negative, but rather a result explained by the fact that the tool needs maintenance.

For tools that are not available (Vandroid [35] and 9Fix [34]), an additional careful documentation-based analysis also confirmed that none of the privilege escalation vulnerabilities is covered.

All false negatives (FN) resulted from the shallow analysis iteration were confirmed by the deep analysis iteration, whether through experimental evaluation of tools or through a second round of documentation analysis. In order to provide a rigorous and explicit evaluation, each realized experimentation is described in details in the technical report [45].

As a conclusion of the deep analysis phase, we are surprised by the low detection rate of the selected IDE plugins against well known Android vulnerabilities. We confirmed that none of the studied IDE plugins covers all the vulnerabilities marked by the red color which are related to critical security attacks such as: privilege escalation, data and code injections, and sensitive information leaks. We consider this as a research gap to be tackled during future research.

VIII. Discussions

Section VII presented the analysis results, we now discuss on lessons learnt and key findings, mitigated by validity threats of our empirical study.

A. Key findings

Our analysis study raised some lessons:

- **Lack of maintenance.** Since the creation of the first version of Android on 2008, the system and the framework levels have shown many security improvements to protect users privacy. A new Android version is released every 6 months. As a consequence, most of the security assisting IDE plugins become outdated, and not able to deal any more with new types of application components, or new released APIs. Table III provides an overview of some open-source tools updates on git repositories, and considered IDEs. Four factors are of interest when considering outdated tools: (i) the date of the last commit (at the time of our study), (ii) the supported IDE type, (iii) information leaks, (iv) the integration of the tools within the last IDE versions. Besides observing that the date of the last commit for many tools is old, most tools are still supported by Eclipse only, which is no more used for developing Android applications.
- **Tools availability.** In order to assist developers in identifying and fixing the listed vulnerabilities, the availability of the security analysis tools is crucial for the software community. Indeed, among the proposed tools, only few are available for use in real Android development projects. Hence, among the 16 analysed tools, 8 academic tools are not available for use (See Table II). As an example, for over-privilege related vulnerability, among the 16 tools, 11 tools could not detect this vulnerability. On the other hand, the remaining 5 tools could detect it only at theoretical level (based on the related research paper). As a result, we do not find any solution that could effectively assist developers in detecting over-privileged applications.

TABLE III
Measure tools updates based on the last git commit

Tool Name	Publication Year	Last Commit	Supported IDE
Curbing	2011	-	Ec
Lintent	2013	25/03/2013	Ec
PermitMe	2014	-	Ec
Page	2014	-	Ec
Vandroid	2018	-	Ec
Android−Lint	-	-	Ec, AS
Sema	2019	03/2020	AS
PerHelper	2019	-	IJ
PoliDroid−AS	2019	08/2019	AS, IJ
9Fix	2022	-	IJ
SonarLint	-	02/2022	Ec, AS, IJ
FindBugs	-	12/2018	Ec, AS, IJ
Cocunut	2018	09/2019	AS
FixDroid	2017	11/2017	IJ, AS
icc− lint	2018	07/2021	IJ, AS
Snyk	-	2023	IJ, AS

[1] AS: Android Studio: IJ: IntelliJ EC: Eclipse
[2] - : unknown

- **Analysis effectiveness**. Our study shows that none of the assessed industrial plugin covers over-privilege vulnerabilities. Furthermore, tools such as Lintent, PerHelper, PermitMe are based on Fel et al. [76] permission mapping (PM) for detecting over-privileged applications. This PM is outdated and does not consider an accurate permission set. As a result, these tools will generate more false negatives during the analysis process. To overcome this limitation, these tools need to be updated to consider a newer PM that covers more API calls such as the permission mapping proposed by Dynamo [65], which is based on dynamic analysis and provides PM for last Android versions APIs.
  On the other hand, we found that some vulnerabilities detection rules do not conform to the Android specification. As an example 9Fix, which does not allow any component to be exported by changing the value of the attribut "exported" as false, and this to prevent any other application to access that component.
  Overall, we observed a dearth of tools capable of effectively detecting the most of listed vulnerabilities. Among the 19 presented vulnerabilities, only 11 are covered by the set of IDE plugins (which is the number of vulnerabilities for which the analysis yielded at least one 'dark green' result). No one of the analysed IDE plugins is able to detect the following vulnerabilities: A1.V1, A1.V2. A1.V3, A1.V4, A1.V5, A2.V3, A2.V4, A5.V1, A5.V2. Except, for some vulnerabilities, it is only theoretically possible based on the documentation, and may not be practically detectable due to the tools unavailability.
- **Analysis approaches for security**: as observed in Section V-C, most tools are based on a static analysis approach for extracting information that enables to check the validity of predefined security properties patterns. Figure 8 displays the analysis techniques used by the sample of tools we analyzed. Among these tools, 88% are based on static analysis techniques. Natural language processing techniques are used to specify security requirements in case of Page, Coconut, PoliDroid−AS. Few other tools are based on formal verification methods. And the

remaining static based analysis tools investigate the program AST to analyse the program structure and check if it conforms to the specified security rules defined by the tools. On the other hand, only 12% of the selected tools enable dynamic application security testing, which are: sonarlint and snyk. In addition to static analysis, they enable developers to examine a running build and detect problems related to security.

As a first direction of improvement, static analysis effectiveness of IDE plugin could be improved by adopting complementary analysis techniques such as Symbolic execution, to allow sound results in case of inter-component communication analysis. We were surprised to observe that none of the investigated tools takes advantage from the integrated IDE Android simulator to perform dynamic analysis. Adopting dynamic analysis approaches could be an interesting direction to improve security IDE plugin analysis results. This makes it possible to analyse API calls performed dynamically (eg. through reflection). Furthermore, other dynamic analysis techniques could be used such as dynamic code instrumentation to exploit run-time source code, and fuzzing as a software testing technique for automatic input generation.

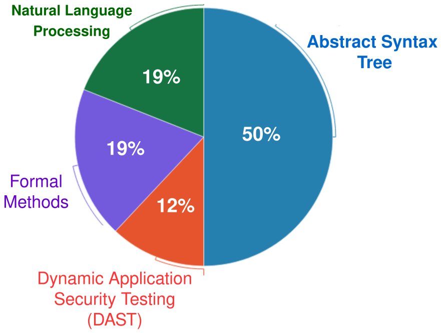


Fig. 8. Security Analysis Approach

- **The lack of native code analysis support.** We noticed a substantial shortfall in the tools' abilities to analyse native code, which is crucial for detecting vulnerabilities. For instance, overuse of permissions in Android apps cannot be accurately detected without analyzing API calls at the native code level. Currently, Per-Helper is the only plugin among the studied tools that analyzes native code, using specific regular expressions for C and C++ code. However, it only performs static analysis of the program's AST and does not provide a complete static analysis process. Additionally, it cannot detect dynamic native API calls.
- **Poor tools evaluation and validation.** After reviewing the tools' validation process, we found that most tools do not focus on the validation techniques used to confirm their ability to detect the stated vulnerabilities. Except from icc – lint that has a few unit tests for each detection rule, and Lintent, which uses formal models and rigorous validation methods to assess its analysis capabilities. None of the other tools follows a well formalised validation process, except for testing the tool on an open-source database of applications. However, there is no guarantee that the vulnerabilities are present, neither that the tool is able to detect the vulnerability.
- **Vulnerability presence per Android version** we showed through the deep analysis step that most of listed vulnerabilities (18 among 19 vulnerabilities) are prone to be exploited on last version of Android 12 (see TABLE IV) Except A2.V4 (External storage) where the vulnerability was automatically prevented by the Android system under the following versions: 12, 11, 10. However, the corresponding attack scenario of *A2.V4* is successfully launched on Android 9. This information is critical for developers of security analysis tools to target their detection rules towards specific versions of the Android API.

TABLE IV
VULNERABILITY PRESENCE PER VERSION

Vulnerability	Android version	Vulnerability	Android version
A1.V1	12	A1.V2	12
A1.V3	12	A1.V4	12
A1.V5	12	A1.V6	12
A1.V7	12		
A2.V1	12	A2.V2	12
A2.V3	12	A2.V4	9
A3.V1	12	A3.V2	12
A4.V1	12	A4.V2	12
A4.V3	12		
A5.V1	12	A5.V2	12

- **Vulnerability based vs Tool based Evaluation.** We found through deep analysis that analyzing vulnerable applications yields positive results. However, we think that a more accurate evaluation could be achieved by constructing more challenging scenarios for the tool's analysis. For certain vulnerabilities like **A1.V4**, the current scenario only considers stated permissions that have no relation to Java API calls. To properly assess the tool's detection capabilities, we need to enrich the attack scenario by adding API calls, either dynamically or through native code. This will help in evaluating the tool's effectiveness in better detecting over-privileged apps. The purpose here is to move from analyzing simple vulnerable apps to analyzing a larger attack surface sample.
- **Tools documentation.** Some tools do not mention the source of some selected vulnerabilities. In addition, no scenario validating the existence of the vulnerability was proposed. Adding a detailed description for each vulnerability, with and attack scenario demonstrating its exploitation would be a better approach to ease the use of the tool and increase the trust of end-users.
- **Benchmark availability and incompleteness**: Ghera [14] is a valuable reference for evaluating security analysis plugins that focus on open-source projects, as it implements an open-source application with common vulnerabilities. However, it lacks some vulnerabilities, such as service hijacking, and other vulnerabilities related to component hijacking. As a recommended improvement, more vulnerabilities could be found in CVE details. The goal is to enrich Ghera benchmark with new vulnerabilities. As an example, we can implement scenarios exploiting new vulnerabilities related to the manipulation of customer permissions [77]. The availability of more relevant benchmarks could lead to more comprehensive security analysis.

B. Threats to validity

In this section, we discuss various potential threats to the validity of our empirical study results, classified in the four categories of [78] according to [79].

a) Conclusion Validity: focuses on how sure we can be that the treatment we used in an experiment is really related to the actual outcome we observed [79].

- The proposed classification focuses on security tools used during: specification, design, coding and testing stages of software development. Other main stages such as: integration, deployment, and the various steps of the DevOps pipeline can also be investigated regarding the existing security tools.

- The tools are continuously evolving. Only after we conducted our evaluation, we found that a recent version of Vandroid2 has been published.
- For the included tools where all the line is in red color, e.g., POLIDROID−AS and Findbugs, this does not mean that the tool is not able to detect other vulnerabilities.

b) Internal Validity: focuses on how sure we can be that the treatment actually caused the outcome [79]. The main threat focuses on the *Tools search & selection* phase, detailed in Section IV-C.

- The set of analysis tools is not exhaustive. Despite we searched in a wide space, we may miss existing tools because the search key words differ, especially because our search target is crossing fields of the classification framework of Fig. 3. To mitigate this limitation, we crossed the references, we reviewed tools studies and related work, had a look to development forums and proceeded with snowballing to enrich the database.
- The selection (and rejection) criteria have been motivated in Section IV-C. To fix these criteria we experimented **more** plugins than those selected. All the selected tools are not dedicated to Android, some have a more general purpose but include Android vulnerability lookup. We may miss tools in that intersection field. A major weakness is that most industrial tools could not be included in the selection, and therefore some outcomes would change, surely be improved.
- The set of security vulnerabilities is not exhaustive (completeness). We based the vulnerability benchmark on the Ghera repository, referring also to CVE details because only reference repositories are sound for benchmarks. But security is a long term race and new vulnerabilities are continuously discovered.
- Besides, the selected tools target neither the same Android versions, nor the same vulnerabilities. So, we had to be very careful while extracting related information to mitigate these threats to validity. In particular, in the tool papers, the authors do not have the same reference model, and further the one we used (see also construct validity).

c) Construct Validity: focuses on the relation between the theory behind the experiment and the observation(s) [79]. While completeness is an internal validity criteria, consistency is a construct validity criteria.

- The main threat here is the heterogeneity of the available information for each tool. We can classify along a 4-level scale. (a) The minimal information is articles or scientific publications, this is a partial information to lead the shallow analysis. (b) The information collected is better with tool documentations or author answers to our queries. (c) Available and deployable tools enable experimentation. (d) Git repositories enable deep knowledge. Unfortunately we did not have the same information on each approach.
- Identification failure is important construction threat. We based the vulnerability benchmark on the Ghera repository but all the tool providers do not use this repository. We may miss vulnerabilities detected by one tool if the tool documentation is not explicit enough to identify with the Ghera vulnerability. This is a kind of homonym/synonym lexical issue. This may affect the high rate of false negatives.
- The classification framework covers many complementary points of view but the tools are not aligned on these projection axes. This may affect some interpretation.
- We used the IntelliJ IDE to process the tests, because it is of widespread use for Android development (Android Studio). But using several IDE plugins and standalone tools would provide different results (merely better if we select the union of the passing tests).
- The tools were tested against simple scenarios specified in the repository for each vulnerability. If the vulnerability could be presented in many ways in the application code and the tool implements a security detection rule that does not match the specified manner in Gherkin, then the analysis results may be inaccurate.

d) External Validity: is concerned with whether we can generalize the results outside the scope of our study [79].

- Our observations are based on the evaluation of 16 IDE plugins against 19 known vulnerabilities. Although it is a large set, it does not represent the population of the analysis tools. The study does not include commercial IDE plugins, tools that could be used out of the IDE, etc. As a result, the above observations should be considered only for similar IDE plugins, and further exploration should be conducted before generalizing the observations to all the existing tools.
- The research protocol is generic since the selected tools, the SDLC stage and the vulnerabilities can be seen as parameters. Selecting other data set for these parameters does not change the methodology, it changes the outcomes.
- It can also be extended. Web apps are not in the scope but connecting apps will extend the scope of inter-application vulnerabilities.
- This study is evolving by nature and be replayed periodically but also contributing to the Ghera repository.

The above threats show that outcome reproducibility is related to time and facts that are true now may change. However, the process itself is to be replayed periodically, just like vulnerability repositories have to be updated continuously.

IX. Conclusion and Future Work

In this paper, we presented a detailed survey of IDE plugins used for secure Android application development. Our study was motivated by the lack of existing research studies investigating their effectiveness in preventing security attacks. We conducted a rigorous and explicit evaluation process of 16 IDE plugins against 19 known vulnerabilities. We believe that the results of the survey would be useful in many cases including:

- Assisting developers in selecting the appropriate IDE plugin to use for secure Android application development,
- Guiding researchers and security tools manufacturers in identifying the existing limits in each tool, and in conducting further research related to Android vulnerabilities,
- In addition, it could be employed as an educational framework during security training sessions.

As the studied vulnerabilities are well known and already included in the list of Common Vulnerabilities Expose (CVE), we expected to have more true positives while conducting the analysis process. However, we were surprised by the obtained number of false negatives. There remains much effort to achieve the transition from Android software development life-cycle (SDLC) to Secure SDLC.

For future works, our study highlighted many observations that could benefit from improvements. First, we aim to enrich the Ghera repository with new vulnerabilities, as an example by implementing scenarios exploiting new vulnerabilities related to the manipulation of customer permissions [77]. Second, it is necessary to add more activities to embrace the full SSDLC, especially at early specification time -by defining threats, risks and scenarios that will be entry points for the application development- and lately by considering vulnerabilities at deployment time. Third, the current study focuses on native applications. To cover the Android ecosystem, it must be extended to analyze vulnerabilities related to hybrid applications, e.g., web applications. More generally, our methodology could apply to IOS applications by revisiting the identified vulnerabilities (*what* axis). Finally, to better detect the vulnerabilities identified in this survey, it is important to develop new IDE plugins that utilize effective analysis techniques.

References

[1] M. E. A. Tebib, M. Graa, O.-E.-K. Aktouf, and P. Andre, "Ide plugins for secure android applications development: Analysis & classification study," in SECURWARE 2022: The Sixteenth International Conference on Emerging Security Information, Systems and Technologies, October 2022, pp. 48–53.

[2] Z. Ahmed and S. C. Francis, "Integrating security with devsecops: Techniques and challenges," in 2019 International Conference on Digitization (ICD). IEEE, 2019, pp. 178–182.

[3] "OWASP - Source Code Analysis Tools," Accessed: 2021-12-25. [Online]. Available: https://owasp.org/www-community/Source_Code_Analysis_Tools

[4] A. K. Jha, S. Lee, and W. J. Lee, "Developer mistakes in writing android manifests: An empirical study of configuration errors," in 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). IEEE, 2017, pp. 25–36.

[5] T. Vidas, N. Christin, and L. Cranor, "Curbing android permission creep," in Proceedings of the Web, vol. 2, no. 0, 2011.

[6] W. Ahmad, C. Kästner, J. Sunshine, and J. Aldrich, "Inter-app communication in android: Developer challenges," in 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR). IEEE, 2016, pp. 177–188.

[7] J. Li, S. Beba, and M. M. Karlsen, "Evaluation of open-source ide plugins for detecting security vulnerabilities," in Proceedings of the Evaluation and Assessment on Software Engineering, 2019, pp. 200–209.

[8] A. Z. Baset and T. Denning, "Ide plugins for detecting input-validation vulnerabilities," in 2017 IEEE Security and Privacy Workshops (SPW). IEEE, 2017, pp. 143–146.

[9] J. Mejía, P. Maciel, M. Muñoz, and Y. Quiñonez, "Frameworks to develop secure mobile applications: A systematic literature review," in World Conference on Information Systems and Technologies. Springer, 2020, pp. 137–146.

[10] J. Senanayake, H. Kalutarage, M. O. Al-Kadri, A. Petrovski, and L. Piras, "Android source code vulnerability detection: a systematic literature review," ACM Computing Surveys (CSUR), 2022.

[11] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software," IEEE Transactions on Software Engineering, vol. 43, no. 6, pp. 492–530, 2016.

[12] B. Reaves, J. Bowers, S. A. Gorski III, O. Anise, R. Bobhate, R. Cho, H. Das, S. Hussain, H. Karachiwala, N. Scaife et al., "* droid: Assessment and evaluation of android application analysis tools," ACM Computing Surveys (CSUR), vol. 49, no. 3, pp. 1–30, 2016.

[13] V.-P. Ranganath and J. Mitra, "Are free android app security analysis tools effective in detecting known vulnerabilities?" Empirical Software Engineering, vol. 25, no. 1, pp. 178–219, 2020.

[14] "Ghera," Access 2021-12-25, https://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks/. [Online]. Available: https://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks/

[15] I. Ul Haq and T. A. Khan, "Penetration frameworks and development issues in secure mobile application development: A systematic literature review," IEEE Access, 2021.

[16] "Android references," Access 2021-12-25, https://github.com/impillar/AndroidReferences.

[17] "Android security assessment tools," Access 2021-12-25, https://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks/.

[18] "Code quality and code security," no date, accessed: 2022-03-05. [Online]. Available: https://www.sonarqube.org/

[19] "Findbugs-idea," no date, accessed: 2022-03-05. [Online]. Available: https://plugins.jetbrains.com/plugin/3847-findbugs-idea

[20] "Improve your code with lint checks," Accessed: 2022-03-05. [Online]. Available: https://developer.android.com/studio/write/lint

[21] "Developer loved, security trusted." no date, accessed: 2023-01-21. [Online]. Available: https://snyk.io/

[22] "Lintent: Towards security type-checking of android applications," no date, accessed: 2021-12-25. [Online]. Available: https://github.com/alvisespano/Lintent

[23] P. Gadient, M. Ghafari, P. Frischknecht, and O. Nierstrasz, "Security code smells in android icc," Empirical Software Engineering, vol. 24, no. 5, pp. 3046–3076, 2019.

[24] M. E. A. Tebib, P. André, O.-E.-K. Aktouf, and M. Graa, "Assisting developers in preventing permissions related security issues in android applications," in Dependable Computing-EDCC 2021 Workshops: DREAMS, DSOGRI, SERENE 2021, Munich, Germany, September 13, 2021, Proceedings 17. Springer, 2021, pp. 132–143.

[25] S. Liang, A. W. Keep, M. Might, S. Lyde, T. Gilray, P. Aldous, and D. Van Horn, "Sound and precise malware analysis for android via pushdown reachability and entry-point saturation," in Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices, 2013, pp. 21–32.

[26] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the {Google-Play} scale," in 24th USENIX Security Symposium (USENIX Security 15), 2015, pp. 659–674.

[27] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, 2011, pp. 239–252.

[28] "Pmd-idea," no date, accessed: 2022-03-05. [Online]. Available: https://plugins.jetbrains.com/plugin/4596-qaplug--pmd

[29] "Checkstyle-idea," no date, accessed: 2022-03-05. [Online]. Available: https://plugins.jetbrains.com/plugin/1065-checkstyle-idea

[30] "Fortify on demand," no date, accessed: 2022-03-05. [Online]. Available: https://plugins.jetbrains.com/plugin/9943-fortify-on-demand

[31] "Checkmarx: Industry-leading application security testing," no date, accessed: 2022-03-05. [Online]. Available: https://checkmarx.com/

[32] D. C. Nguyen, D. Wermke, Y. Acar, M. Backes, C. Weir, and S. Fahl, "A stitch in time: Supporting android developers in writingsecure code," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 1065–1077.

[33] G. Xu, S. Xu, C. Gao, B. Wang, and G. Xu, "Perhelper: Helping developers make better decisions on permission uses in android apps," Applied Sciences, vol. 9, no. 18, p. 3699, 2019.

[34] A.-D. Tran, M.-Q. Nguyen, G.-H. Phan, and M.-T. Tran, "Security issues in android application development and plug-in for android studio to support secure programming," in International Conference on Future Data and Security Engineering. Springer, 2021, pp. 105–122.

[35] A. Nirumand, B. Zamani, and B. T. Ladani, "Vandroid: A framework for vulnerability analysis of android applications using a model-driven reverse engineering technique," Softw. Pract. Exp., vol. 49, no. 1, pp. 70–99, 2019. [Online]. Available: https://doi.org/10.1002/spe.2643

[36] J. Mitra, V.-P. Ranganath, T. Amtoft, and M. Higgins, "Sema: Extending and analyzi storyboards to develop secure android apps," arXiv preprint arXiv:2001.10052, 2020.

[37] R. Slavin, X. Wang, M. B. Hosseini, J. Hester, R. Krishnan, J. Bhatia, T. D. Breaux, and J. Niu, "Toward a framework for detecting privacy policy violations in android application code," in Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 25–36.

[38] M. Rowan and J. Dehlinger, "Encouraging privacy by design concepts with privacy policy auto-generation in eclipse (page)," in Proceedings of the 2014 Workshop on Eclipse Technology eXchange, 2014, pp. 9–14.

[39] "Secure software development life-cycle," accessed on 07. April 2022. [Online]. Available: https://codesigningstore.com/secure-software-development-life-cycle-sdlc

[40] R. Balebako, A. Marsh, J. Lin, J. I. Hong, and L. Cranor, "The privacy and security behaviors of smartphone app developers," Jun 2018. [Online]. Available: https://kilthub.cmu.edu/articles/journal_contribution/The_Privacy_and_Security_Behaviors_of_Smartphone_App_Developers/6470528/1

[41] G. L. Scoccia, A. Peruma, V. Pujols, I. Malavolta, and D. E. Krutz, "Permission issues in open-source android apps: An exploratory study," in 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, 2019, pp. 238–249.

[42] R. Fujdiak, P. Mlynek, P. Mrnustik, M. Barabas, P. Blazek, F. Borcik, and J. Misurec, "Managing the secure software development," in 2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS), 2019, pp. 1–4.

[43] A. Ramirez, A. Aiello, and S. J. Lincke, "A survey and comparison of secure software development standards," in 2020 13th CMI Conference on Cybersecurity and Privacy (CMI) - Digital Transformation - Potentials and Challenges(51275), 2020, pp. 1–6.

[44] R. N. Rajapakse, M. Zahedi, M. A. Babar, and H. Shen, "Challenges and solutions when adopting devsecops: A systematic review," Information and Software Technology, vol. 141, p. 106700, 2022.

[Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921001543

[45] M. Tebib et al., "IDE plugins capabilities in detecting Android vulnerabilities," 2022, https://uncloud.univ-nantes.fr/index.php/s/mzwoC44xs5xiowN.

[46] M. Bugliesi, S. Calzavara, and A. Spanò, "Lintent: Towards security type-checking of android applications," in Formal techniques for distributed systems. Springer, 2013, pp. 289–304.

[47] 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, no. 6, pp. 259–269, 2014.

[48] Y. Aafer, G. Tao, J. Huang, X. Zhang, and N. Li, "Precise android api protection mapping derivation and reasoning," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 1151–1164.

[49] C. Li, X. Chen, R. Sun, J. Xue, S. Wen, M. E. Ahmed, S. Camtepe, and Y. Xiang, "Natidroid: Cross-language android permission specification," arXiv preprint arXiv:2111.08217, 2021.

[50] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in Proceedings of the 2012 ACM conference on Computer and communications security, 2012, pp. 217–228.

[51] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," ACM Computing Surveys (CSUR), vol. 51, no. 3, pp. 1–39, 2018.

[52] C. Cadar and M. Nowack, "Klee symbolic execution engine in 2019," International Journal on Software Tools for Technology Transfer, pp. 1–4, 2020.

[53] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "Sig-droid: Automated system input generation for android applications," in 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2015, pp. 461–471.

[54] R. Spreitzer, G. Palfinger, and S. Mangard, "Scandroid: Automated side-channel analysis of android apis," in Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks, 2018, pp. 224–235.

[55] S. Lortz, H. Mantel, A. Starostin, and A. Weber, "A sound information-flow analysis for cassandra," TU Darmstadt, Tech. Rep., 2014.

[56] A. Souri, N. J. Navimipour, and A. M. Rahmani, "Formal verification approaches and standards in the cloud computing: a comprehensive and systematic review," Computer Standards & Interfaces, vol. 58, pp. 1–22, 2018.

[57] G. Iadarola, F. Martinelli, F. Mercaldo, and A. Santone, "Formal methods for android banking malware analysis and detection," in 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS). IEEE, 2019, pp. 331–336.

[58] W. Landi, "Undecidability of static analysis," ACM Letters on Programming Languages and Systems (LOPLAS), vol. 1, no. 4, pp. 323–337, 1992.

[59] B. Chess and G. McGraw, "Static analysis for security," IEEE Security & Privacy, vol. 2, no. 6, pp. 76–79, 2004.

[60] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software." in NDSS, vol. 5. Citeseer, 2005, pp. 3–4.

[61] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06). IEEE, 2006, pp. 135–148.

[62] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," ACM Sigplan notices, vol. 42, no. 6, pp. 89–100, 2007.

[63] P. Oehlert, "Violating assumptions with fuzzing," IEEE Security & Privacy, vol. 3, no. 2, pp. 58–62, 2005.

[64] J. Viide, A. Helin, M. Laakso, P. Pietikäinen, M. Seppänen, K. Halunen, R. Puuperä, and J. Röning, "Experiences with model inference assisted fuzzing." WOOT, vol. 2, pp. 1–2, 2008.

[65] A. Dawoud and S. Bugiel, "Bringing balance to the force: Dynamic analysis of the android application framework," Bringing Balance to the Force: Dynamic Analysis of the Android Application Framework, 2021.

[66] O. A. V. Ravnås, "Frida: Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers," 2019.

[67] L. Dresel, M. Protsenko, and T. Müller, "Artist: the android runtime instrumentation toolkit," in 2016 11th International Conference on Availability, Reliability and Security (ARES). IEEE, 2016, pp. 107–116.

[68] A. Druffel and K. Heid, "Davinci: Android app analysis beyond frida via dynamic system call instrumentation," in International Conference on Applied Cryptography and Network Security. Springer, 2020, pp. 473–489.

[69] S. Zhao, X. Li, G. Xu, L. Zhang, and Z. Feng, "Attack tree based android malware detection with hybrid analysis," in 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications. IEEE, 2014, pp. 380–387.

[70] "Find security bugs," no date, accessed: 2023-02-11. [Online]. Available: https://plugins.jetbrains.com/plugin/3847-findbugs-idea

[71] E. Bello-Ogunu and M. Shehab, "Permitme: integrating android permissioning support in the ide," in Proceedings of the 2014 Workshop on Eclipse Technology eXchange, 2014, pp. 15–20.

[72] T. Li, Y. Agarwal, and J. I. Hong, "Coconut: An ide plugin for developing privacy-friendly apps," Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, vol. 2, no. 4, pp. 1–35, 2018.

[73] "Fixdroid: Usable security and privacy research," accessed: 2022-03-05. [Online]. Available: https://plugins.jetbrains.com/plugin/9497-fixdroid

[74] "Explore the microsoft security sdl practices," accessed on 07. April 2022. [Online]. Available: https://www.microsoft.com/en-us/securityengineering/sdl

[75] "Software assurance maturity model," accessed on 07. April 2022. [Online]. Available: https://owasp.org/www-project-samm/

[76] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in Proceedings of the 18th ACM conference on Computer and communications security, 2011, pp. 627–638.

[77] R. Li, W. Diao, Z. Li, S. Yang, S. Li, and S. Guo, "Android custom permissions demystified: A comprehensive security evaluation," IEEE Transactions on Software Engineering, 2021.

[78] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell, Experimentation in Software Engineering - An Introduction, ser. The Kluwer International Series in Software Engineering. Kluwer, 2000, vol. 6. [Online]. Available: https://doi.org/10.1007/978-1-4615-4625-2

[79] R. Feldt and A. Magazinius, "Validity threats in empirical software engineering research - an initial survey," in Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010), Redwood City, San Francisco Bay, CA, USA, July 1 - July 3, 2010. Knowledge Systems Institute Graduate School, 2010, pp. 374–379.