# The OWApp Benchmark: an OWASP-compliant Vulnerable Android App Dataset [*]

Luca Ferrari[1] [†], Francesco Pagano[1], Luca Verderame[1][‡], Andrea Romdhana[2], Davide Caputo[2], and Alessio Merlo[3]

[1] DIBRIS - University of Genoa, Genoa, Italy
{luca.ferrari, francesco.pagano, luca.verderame}@dibris.unige.it
[2] Talos srls, Genoa, Italy
{andrea.romdhana, davide.caputo}@talos-sec.com
[3] CASD - School of Advanced Defense Studies, Rome, Italy
alessio.merlo@unicasd.it

**Abstract**

Static Application Security Testing (SAST) tools play a critical role in detecting vulnerabilities in mobile apps by analyzing the source code without executing the app. These tools are essential in modern development workflows, particularly in Continuous Integration/Continuous Deployment (CI/CD) pipelines. However, SAST tools face significant limitations due to their dependence on predefined rules and patterns. As a result, many vulnerabilities, particularly complex ones, such as cryptographic flaws or multi-component interactions, go undetected.

Despite the introduction of unified frameworks such as the OWASP Mobile Application Security Verification Standard (MASVS) and the Mobile Application Security Testing Guide (MASTG), which standardize mobile application security assessments, the data sets used to train and evaluate SAST tools of Android Apps cover only specific categories of vulnerabilities. In addition, most of the available datasets do not contain recent vulnerability patterns.

To fill this gap, this paper introduces a novel and comprehensive vulnerable app dataset for Android Apps (i.e., the OWApp Benchmark) that maps the OWASP MASVS. This benchmark improves the evaluation of Android SAST tools by providing broader coverage of security controls, thus enhancing detection of a more comprehensive range of vulnerabilities. Each dataset entry includes source code, APK, vulnerability details, and relevant artifacts to support accurate evaluations.

**Keywords**— Android Security, Vulnerability Assessment, OWASP, SAST, App Benchmark

## 1 Introduction

The proliferation of mobile devices has revolutionized various aspects of daily life, providing unprecedented convenience and connectivity. According to recent statistics, more than 4.8 billion smartphone users worldwide continue to increase annually [32]. This widespread adoption has catalyzed the development and deployment of millions of mobile applications (hereafter, apps) across diverse domains, including social networks, banking, health, and entertainment. These apps routinely access and utilize large amounts of user and device data.

The increased complexity and data-centric nature of modern mobile apps underscore the critical importance of security. The presence of vulnerabilities or security misconfigurations in apps could lead

[*]This Paper will be published in Mobisec 2024 Proceedings
[†]The authors have been listed in order of contribution.
[‡]Corresponding author.

to severe implications ranging from denial of services to unauthorized data access or significant financial losses for users and developers.

Notable examples include the Autospill vulnerability [17], which impacted Android password managers during autofill operations on login pages embedded within apps, enabling attackers to capture the auto-filled credentials. Another recent finding revealed that eight out of nine popular apps used for inputting Chinese characters into mobile devices have a vulnerability that allows passive eavesdroppers to collect keystroke data [9].

Therefore, ensuring robust security assessment methodologies for mobile apps is not a technical challenge but a foundational requirement to deliver secure and reliable apps. A recent study [39] reported that mobile apps released in the first quarter of 2021 across 18 of the most popular categories had, on average, 39 vulnerabilities per app that could compromise users' security and privacy.

The Open Source Foundation for Application Security (OWASP) maintains two key frameworks to standardize the vulnerability assessment process for mobile apps, intending to establish a unified approach: the Mobile Application Security Verification Standard (MASVS) [16] and the Mobile Application Security Testing Guide (MASTG) [15]. OWASP MASVS provides a set of security requirements that mobile apps must meet to ensure a baseline level of security, establishing a clear standard for evaluating apps and promoting consistency in assessments. OWASP MASTG offers comprehensive guidelines and best practices for performing security tests on mobile apps, detailing specific technical testing procedures.

Both industry and the research community have developed a range of security assessment methodologies and tools aimed at detecting vulnerabilities. These include Static Application Security Testing (SAST) techniques [11, 31, 40], Dynamic Application Security Testing (DAST) techniques [3, 18], and hybrid approaches that combine both methodologies [5, 30, 33].

SAST tools, in particular, detect security hazards in programs relying on white-box or black-box analysis techniques. White-box techniques analyze the program by accessing its source code, offering more accurate analyses than black-box techniques that analyze compiled code without having access to the source. Unlike DAST, SAST techniques do not require the program to be executed and tested in a dedicated environment, which speeds up the analysis process. Consequently, SAST tools are often preferred in CI/CD pipelines, as vulnerability assessments can be executed when new modifications are available during the build and production process.

One key limitation of SAST tools is their reliance on predefined rules, patterns, and signatures to detect vulnerabilities in the code. These rules are crafted based on known vulnerability types, coding practices, and threat models.

In this context, datasets of vulnerable mobile apps are essential. Developers of SAST tools frequently use these datasets to identify key vulnerability classes, which serve as the foundation for building detection rules within their tools. Additionally, these datasets, often sourced from intentionally vulnerable apps, are used to test and validate the effectiveness of the tools detection capabilities.

However, this practice introduces a significant bias, as the tool's detection capability becomes heavily dependent on the dataset's representativeness. If detection rules or patterns are derived from incomplete or outdated vulnerability datasets that fail to capture the full scope of real-world security threats, the tool may overlook newer or less common vulnerabilities [23]. This limitation is further highlighted when the datasets used during the testing phase lack certain vulnerability classes, making it challenging to identify gaps in the tool's ability to cover the full spectrum of the attack surface.

The current state-of-the-art comprises several datasets for training and evaluating SAST tools. For example, DroidBench [4] and Ghera [22] are widely used datasets designed to assess the effectiveness of SAST tools in detecting vulnerabilities within Android apps. They include samples containing data leaks, information flow vulnerabilities, and flaws in inter-component communication.

Despite their usefulness, these datasets focus on specific categories of vulnerabilities and fail to encompass the full range of security controls outlined by standards such as the OWASP MASVS. For example, while Ghera offers vulnerable apps based on common coding errors, it does not address more advanced security issues, such as complex cryptographic vulnerabilities or multicomponent interactions, which are critical in modern mobile app ecosystems. This misalignment poses a significant challenge: SAST tools trained and configured on these datasets may not develop the capability to detect or assess the broader range of security controls that MASVS and MASTG demand.

To address this gap, this paper proposes the creation of a restructured benchmarking suite for the configuration and evaluation of Android SAST tools that ensures comprehensive tests that map the security controls provided in the OWASP MASTG applicable to automated static app security testing.

In the first part of this work, we evaluated OWASP MASVS and MASTG to identify the security controls relevant to SAST tools that can be automated through static analysis techniques. In addition, we review existing benchmarking suites and datasets for Android apps to assess their coverage of OWASP security controls.

Subsequently, we proposed the *OWApp Benchmark Suite*[1], which includes an Android app dataset covering all the OWASP MASVS security controls that can be evaluated using SAST techniques. This suite also provides a framework for building and configuring the dataset and performing comparative evaluations of the SAST tools. Each entry in the OWApp dataset, mapped to its corresponding OWASP security control, includes the app's source code, its `.apk` file, a detailed description of the implemented vulnerability, and the necessary security check for its detection.

## 2 Basics of Android Ecosystem

**Anatomy of an Android app.** Android apps are mainly developed in Java and Kotlin and made of four kinds of components, namely, *activities*, *services*, *content providers*, *broadcast receivers*.

An **activity** is a single screen within an app, such as a page of settings or a browser window. It contains visual components that enable user interaction (for example, buttons) or display data (e.g., images). Since most activities are designed for user interaction, the `Activity` class manages the creation of a window where the User Interface (UI) is displayed, using the `setContentView(View)` method to set the layout.

A **Content Provider** is a core component of the Android app framework that manages access to a centralized repository of structured data. It encapsulates data, enforces data security, and provides a standardized interface for data sharing among different apps. For example, contact information is accessed by multiple apps and is stored within a Content Provider. Content Providers enable inter-app communication by allowing external apps to query, insert, update, or delete data in a controlled manner, supporting data sharing while maintaining data integrity and access control. They expose data to other apps through the `ContentResolver` interface. When a request is made via a `ContentResolver`, the system inspects the authority component of the provided URI. It forwards the request to the Content Provider registered with that authority. The Content Provider can interpret the remaining parts of the Uniform Resource Identifiers (URI) according to its logic, allowing data querying and manipulation.

**Broadcast Receivers** and **Services** are two main components that the Android OS leverages to perform background tasks. A Broadcast Receiver is generally used to execute short background tasks in response to specific action signals, either directed to it or broadcast system-wide. If an app needs to perform a long-running task during its lifecycle, it can instantiate a Service. When a task must continue running even when the app is inactive, the Android OS provides the Foreground Service functionality, ensuring that the task persists and remains visible through a notification. A Service can be directly bound to specific components of an app using Remote Procedure Call (RPC) mechanisms provided by Android's Inter Component Communication (ICC) framework. This binding enables interaction and data sharing between the Service and the component. Alternatively, a Service can operate as a standalone task, running independently in the background without direct interaction with other app components.

**Intents** are the main ICC mechanism in Android, allowing different components of the app, both within the same app and between different apps, to interact for data sharing and synchronization. An Intent is a messaging object that can request an action from another app component. Intents can be **explicit**, specifying the exact component to receive the message, or **implicit**, declaring a general action to take, allowing the system to determine the appropriate component based on *Intent Filters*.

[1]https://github.com/Mobile-IoT-Security-Lab/OWApp-Benchmarking-Suite

They can also be used to broadcast messages throughout the system. Intents contain an action that indicates the desired operation and can include additional data required for the task, such as URIs, data types, or additional information packaged as key-value pairs known as *extras*.

Apps are delivered as Android Packages (file extension .apk) that follow the ZIP file format and pack together the compiled code and all the app resources. The structure of an *apk* file comprises the following parts: the AndroidManifest.xml, a metadata file that describes all the app components, the required permissions to execute, and app-wide settings (like the name, the version, and the target Android OS version); (one or more) classes.dex files, which are the proprietary format for the Java VM customized for Android and contain all Java/Kotlin code compiled to Android-specific bytecode called Dalvik; the assets folder that stores the app resources, such as documents or media; the lib folder containing the precompiled native libraries written in C/C++ used by the app; the res folder storing the app resources (e.g., the layout files or the constant value files); the resources.arsc file that groups linking information between the code and the resources; and the META-INF folder that stores app verification information, e.g., the signature of the apk file.

**App distribution.** Android apps can be published in different app stores, where the Google Play Store[2] is the most widely adopted in Europe and the USA. Users access such stores to search, fetch, and install apps. In addition, there are alternative app stores, for example, Apkpure[3], APKMirror[4], and FDroid[5], where users must explicitly enable permissions on their phones to allow the installation of third-party apps. By enabling this feature, users can manually install apps as long as they possess a valid `.apk` file, even if it is downloaded outside official app stores.

# 3 OWASP Analysis

This section introduces the OWASP Mobile Application Security Project and outlines the security checks applicable to mobile apps. Then, it presents a detailed analysis of the security controls that can be applied using SAST techniques and implemented in automated tools. This evaluation lays the foundation for defining the OWApp Benchmark Suite, ensuring comprehensive coverage of Android apps' vulnerability classes.

## 3.1 OWASP Mobile Application Security Project

The OWASP Mobile Application Security Project is a comprehensive initiative that provides various resources, guidelines, and tools to enhance the security posture of mobile apps. The core part of the project consists of the Mobile Application Security Verification Standard (MASVS) [16] and the Mobile Application Security Testing Guide (MASTG) [15]. MASVS provides a set of security requirements designed to ensure that mobile apps meet a baseline level of security. MASTG, instead, provides detailed technical testing procedures for performing security tests on mobile apps to verify their compliance with the MASVS requirements.

**OWASP MASVS.** The MASVS standard is organized into groups of security requirements, each labeled as `MASVS-<id>`, that address the most critical areas of the mobile attack surface. The current version of MASVS encompasses seven primary groups. These include requirements for data management (MASVS-STORAGE), cryptographic protocols (MASVS-CRYPTO), authentication (MASVS-AUTH), network communication (MASVS-NETWORK), interactions with the platform (MASVS-PLATFORM), adherence to coding practices (MASVS-CODE) and the implementation of defense-in-depth mechanisms (MASVS-RESILIENCE). Recently, MASVS has also introduced app-specific privacy requirements (MASVS-PRIVACY) to address collecting, using, and transferring personal and device data.

[2] https://play.google.com/
[3] https://apkpure.com/br/
[4] https://www.apkmirror.com/
[5] https://f-droid.org/en/

**OWASP MASTG.** The MASTG is organized into 27 chapters covering Android and iOS platforms and is designed to cover the entire mobile app security testing lifecycle. The guide addresses all critical aspects of app security, ranging from fundamental concepts to advanced analysis techniques. It provides a structured step-by-step approach to testing mobile apps, including guidance on setting up a testing environment, identifying attack surfaces, selecting appropriate tools, and applying static and dynamic analysis techniques.

Additionally, the MASTG project includes a Security Checklist that links the MASTG test cases to each MASVS security requirement. This checklist, available at the OWASP site[6], consists of seven sheets detailing the specific controls to be evaluated for both the Android and iOS platforms.

## 3.2 Analysis of the Mobile Security Controls

| Category | Test Name | Static Black-Box | Static White-Box | Dynamic | Automa-table | Decision | Comments |
|---|---|---|---|---|---|---|---|
| STORAGE | Testing Local Storage for Sensitive Data | ● | - | ● | ● | ✓ | |
| | Determining Whether Sensitive Data Is Shared with Third Parties via Embedded Services | ● | ● | ● | - | ✗ | White-Box |
| | Determining Whether Sensitive Data Is Shared with Third Parties via Notifications | ● | - | ● | - | ✗ | Context-Dependent |
| | Determining Whether the Keyboard Cache Is Disabled for Text Input Fields | ● | - | ● | ● | ✓ | |
| | Testing Memory for Sensitive Data | ● | - | ● | ● | ✓ | |
| | Testing Backups for Sensitive Data | ● | - | ● | ● | ✓ | |
| | Testing Logs for Sensitive Data | - | - | ● | - | ✗ | Context-Dependent |
| | Testing the Device-Access-Security Policy | ● | - | ● | - | ✗ | Ecosystem-Dependent |
| CRYPTO | Testing Symmetric Cryptography | ● | - | ● | ● | ✓ | |
| | Testing the Configuration of Cryptographic Standard Algorithms | ● | - | ● | ● | ✓ | |
| | Testing Random Number Generation | ● | - | ● | ● | ✓ | |
| | Testing the Purposes of Keys | ● | - | ● | ● | ✓ | |
| AUTH | Testing Confirm Credentials | ● | - | ● | ● | ✓ | |
| | Testing Biometric Authentication | ● | - | ● | ● | ✓ | |
| NETWORK | Testing Data Encryption on the Network | ● | - | ● | ● | ✓ | |
| | Testing the TLS Settings | ● | - | ● | ● | ✓ | |
| | Testing Endpoint Identify Verification | ● | - | ● | ● | ✓ | |
| | Testing Custom Certificate Stores and Certificate Pinning | ● | - | ● | ● | ✓ | |
| | Testing the Security Provider | ● | - | ● | ● | ✓ | |

**Table 1:** Analysis of the OWASP MASTG Controls - Part 1.

We reviewed all documents related to the OWASP Mobile Security Project to collect the security vulnerabilities that should be included in building a comprehensive mobile apps dataset tailored to train and evaluate automated black-box SAST tools for Android apps. Specifically, our goal was to

[6]https://mas.owasp.org/checklists/

| Category | Test Name | Static Black-Box | Static White-Box | Dynamic | Automa-table | Decision | Comments |
|---|---|---|---|---|---|---|---|
| PLATFORM | Determining Whether Sensitive Stored Data Has Been Exposed via IPC Mechanisms | ● | - | ● | ● | ✓ | |
| | Testing for Sensitive Functionality Exposure Through IPC | ● | - | ● | - | ✗ | Context-Dependent |
| | Testing for Vulnerable Implementation of PendingIntent | ● | - | ● | ● | ✓ | |
| | Testing Deep Links | ● | - | ● | ● | ✓ | |
| | Testing WebViews Cleanup | ● | - | ● | ● | ✓ | |
| | Testing JavaScript Execution in WebViews | ● | - | ● | ● | ✓ | |
| | Testing for Java Objects Exposed Through WebViews | ● | - | ● | ● | ✓ | |
| | Testing WebView Protocol Handlers | ● | - | ● | ● | ✓ | |
| | Testing for Overlay Attacks | ● | - | ● | ● | ✓ | |
| | Finding Sensitive Information in Auto-Generated Screenshots | ● | - | ● | - | ✗ | Context-Dependent |
| CODE | Testing Object Persistence | ● | - | ● | - | ✗ | Context-Dependent |
| | Testing Enforced Updating | ● | - | ● | ● | ✓ | |
| | Testing for Injection Flaws | ● | - | ● | ● | ✓ | |
| | Checking for Weaknesses in Third Party Libraries | ● | ● | ● | - | ✗ | White-Box |
| | Memory Corruption Bugs | ● | - | ● | ● | ✓ | |
| | Make Sure That Free Security Features Are Activated | ● | - | - | ● | ✓ | |
| | Testing Implicit Intents | ● | - | ● | ● | ✓ | |
| | Testing for URL Loading in WebViews | ● | - | ● | ● | ✓ | |
| RESILIENCE | Testing Root Detection | ● | - | ● | ● | ✓ | |
| | Testing Anti-Debugging Detection | ● | - | ● | ● | ✓ | |
| | Testing Reverse Engineering Tools Detection | ● | - | ● | ● | ✓ | |
| | Testing Emulator Detection | ● | - | ● | ● | ✓ | |
| | Testing Obfuscation | ● | - | ● | ● | ✓ | |
| | Making Sure that the App is Properly Signed | ● | - | - | ● | ✓ | |
| | Testing File Integrity Checks | - | - | ● | - | ✗ | Dynamic Analysis |
| | Testing Runtime Integrity Checks | - | - | ● | - | ✗ | Dynamic Analysis |
| | Testing whether the App is Debuggable | ● | - | ● | ● | ✓ | |
| | Testing for Debugging Symbols | ● | - | - | ● | ✓ | |
| | Testing for Debugging Code and Verbose Error Logging | ● | - | ● | ● | ✓ | |

**Table 2:** Analysis of the OWASP MASTG Controls - Part 2.

collect vulnerabilities linked to security tests that (1) can be executed using static analysis techniques, (2) do not require access to the app's source code, and (3) can be fully automated, eliminating the need for human supervision.

To do so, we gathered the OWASP MASVS version 2.0.1, released on January 18, 2024, the OWASP MASTG version 1.7.0, released on October 30, 2023, and the latest version of the OWASP Mobile Application Security Checklist.

As a starting point, we use the checklist to obtain the mapping between the security requirements and the corresponding technical controls. According to the checklist, there are 21 MASVS requirements associated with 51 MASTG security controls for Android apps, grouped into the eight MASVS categories.

Then, we review the technical documentation from OWASP MASVS and OWASP MASTG for each security test to determine whether it could be conducted using static analysis, dynamic analysis, or a combination of both approaches. Additionally, we evaluated whether each test required access to the source code (i.e., a white-box test) or could be executed directly on the app executable (i.e., a black-box test).

We supplemented our analysis with additional gray literature for tests where descriptions were unclear, e.g., overly generic instructions, or incomplete, such as lacking specific technical details. This included consulting the official Android documentation [20], browsing Stack Overflow [25], and leveraging insights from the chatbot AI ChatGPT [1].

The results of our analysis are reported in Tables 1 and Table 2 where we detailed the category and name of each security test, the type of technique required to execute the test (i.e., static black-box, static white-box, or dynamic) if the test is suitable for automation and finally if the associated vulnerability met all the requirements.

Of the 51 security controls in the MASTG, 49 were considered suitable for static analysis in white or black-box mode. In particular, only two tests (i.e., `Checking for Weaknesses in Third Party Libraries` [37] and `Determining Whether Sensitive Data Is Shared with Third Parties via Embedded Services` [34]) explicitly required access to the source code.

Then, we manually inspected the 49 static tests to assess the feasibility of using automated tools to detect the associated vulnerabilities. During this analysis, we identified a series of controls that could not be automated, as they may depend on i) context (e.g., data) or ii) ecosystem (e.g., a specific OS configuration). An example of a test that depends on the context is `Determining Whether Sensitive Data Is Shared with Third Parties via Notifications` [35], which is data dependent. Although some tools can detect the use of components like the `NotificationManager` [10], they often fail to assess whether the shared data is sensitive. This limitation arises because determining the sensitivity of the data requires a deeper understanding of the context in which the data are used. Thus, while detection mechanisms can flag the use of notifications, they typically do not recognize the sensitivity of a piece of data, leading to potential inconsistencies in the evaluation.

`Testing the Device-Access-Security Policy` [36] is an example of an ecosystem-dependent test, as it relies on the specific configuration of the mobile device and the operating system in use. Although static analysis can identify code sections where device security policies are enforced, such as PIN checks or password-protected device locking, the test outcome may vary depending on the Android version or manufacturer-specific modifications. For example, while a code review can highlight the presence of root detection mechanisms, determining whether these checks are adequate or bypassable in practice often depends on the specific device environment, making the assessment more complex. Usually, these checks can be performed using the SafetyNet API[7] but it provides the ecosystem information when the app is executed. Consequently, this check cannot be performed without acknowledging the ecosystem.

At the end of this qualitative analysis, we identified a set of security controls that can be evaluated using an automated black-box SAST tool. These controls will be used as ground truth to create the data set of apps contained in the OWApp Benchmark Suite, as described in the following sections.

## 4 Related Works

Several proposals have focused on creating open-source collections of apps that can be used for security-related tasks. These datasets are intended to provide samples to analyze and test various aspects of mobile app security, such as detecting vulnerabilities or evaluating the effectiveness of security tools. However, the scope and utility of these collections vary widely, often limiting their applicability to comprehensive security assessments.

[7] https://developer.android.com/privacy-and-security/safetynet

Proposals like DroidBench [4] and TaintBench [21] introduced test suites designed to evaluate the effectiveness of taint analysis tools for Android apps. Unfortunately, those solutions could not be used as general benchmark suites as they focused explicitly on taint-related security vulnerabilities.

Other works, instead, focused on building real-world app datasets. A notable example is AndroZoo [2], a repository containing more than 24 million Android apps, each analyzed by multiple antivirus products to detect malware. This collection supports various research projects by providing real-world Android apps from numerous app stores. However, the repository focuses on detecting malware through antivirus products rather than identifying vulnerabilities. Thus, it cannot be used as a benchmark to assess the capability of SAST tools to detect security issues, as the dataset does not provide per-app vulnerability information.

The advent of machine learning models to predict vulnerabilities has also led to specialized datasets, such as LVDAndro [38], which contain more than 20 million different source code samples labeled with CWE-IDs. Although helpful in training AI-based detection models, LVDAndro is less suitable as a benchmark because its vulnerability labeling is automated and lacks manual validation, which may result in false positives. Additionally, the dataset does not include complete app code, limiting the detection of vulnerabilities requiring multi-component analysis.

Other datasets, such as ICCBench [28] and UBCBench [29], focus on specific types of vulnerability, such as ICC-related flaws or those involving Android Shared Preferences. Unfortunately, the limited set of vulnerability classes covered in this case also does not allow the evaluation of all the detection capabilities of the generic Android SAST tools.

In July 2024, the OWASP Mobile Application Security Project released a beta version of a benchmark of sample vulnerable mobile apps, called MASTG Demos [14]. Each demo contains a brief description of the vulnerable app. This code snippet demonstrates the weakness, the specific steps to identify the weakness in the sample code and a test evaluation. Although promising, this beta version is not yet mature enough to compare directly with our work.

### 4.1 Analysis of the Ghera Benchamark

As noted in [23], the most widely used and, therefore, the most established Android app vulnerability benchmark repository is Ghera [22], an open-source vulnerability benchmark repository covering key Android app vulnerabilities.

According to the authors of the Ghera benchmark [22], vulnerabilities are classified according to specific features or capabilities that contribute to their occurrence. To determine what features to focus on, they analyzed those commonly used by Android apps and consulted various Android security resources [6], [13], [19]. The initial version of the repository included apps that exploited vulnerabilities in Component Communication (ICC), Storage, System, and Web functionality.

The current version of the Ghera repository[8] contains vulnerable Android Java apps running on OS versions 5.1.1 to 8.1. The dataset includes apps with vulnerabilities related to Crypto, ICC, Networking, NonAPI, Permission, Storage, System, and Web APIs. The vulnerabilities are organized based on the APIs they originate, such as ICC or Storage. Within each category, the benchmark contains a set of apps organized into three folders: Benign, Malicious, and Secure. The *Benign* folder contains the app with the vulnerability, the *Malicious* folder includes the app that exploits it, and the *Secure* folder contains the patched version of the Benign app.

To determine the Ghera dataset's representativeness, we examined its vulnerabilities and mapped them to the corresponding security requirements of the MASVS and the associated MASTG test. First, we categorized the Ghera apps according to MASVS categories. Then, we analyzed each app's source code to identify the types of vulnerabilities, associating each with the relevant MASTG test. The results of this analysis are summarized in Tables 3 and 4 in Appendix A.

This mapping revealed a significant gap: Of the 51 vulnerabilities evaluated by the MASTG controls, only 19 (that is, 37%) were included in the Ghera benchmark. Furthermore, the data set lacks coverage

[8]https://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks/src/master/

in three critical areas of the mobile attack surface, i.e., vulnerabilities related to the authentication, resilience, and privacy MASVS categories.

Finally, the benchmark contained multiple vulnerabilities associated with a single control. For example, the dataset includes eight samples associated with `MASTG-TEST0001` and seven samples of `MASTG-TEST0021`.

As a final remark, the Ghera dataset presents several technological constraints that further limit its adoption: the dataset is no longer maintained (the last commit dates back to 2019), and the apps only support Android 8.1 without accounting for newer versions or the Kotlin language.

In conclusion, the analysis revealed that even the current de facto standard dataset of vulnerable Android apps is not representative of the mobile attack surface described in the OWASP MASVS and thus cannot be used as a reference to build and assess the capabilities of automated SAST tools. This underscores the importance of having a comprehensive and detailed benchmark suite closely aligned with the security controls mandated by the OWASP project.

## 5 The OWApp Benckmark

This section introduces OWApp Benchmark, a suite designed to support evaluating SAST tools for Android apps. The OWApp Benchmark Suite consists of a comprehensive vulnerable app dataset that contains vulnerability samples mapped with the OWASP MASVS security requirements and MASTG security controls that can be evaluated using SAST techniques. In addition, the suite includes a set of auxiliary scripts to i) download and compile the dataset and ii) configure, deploy, and test SAST tools against vulnerable apps.

### 5.1 Dataset of Vulnerable Apps

We created the OWApp Benchmark Dataset, an Android app vulnerability benchmark repository to cover the seven security requirements of the OWASP MASVS and the corresponding controls defined in the OWASP MASTG that are suitable for Android SAST tools. The repository is organized in MASVS categories, and each app is labeled using the ID of the associated OWASP MASTG control. Each sample contains the app's source code, a compiled app, and a README file that describes the vulnerability. Currently, the repository contains 42 apps, providing complete coverage of the tests selected during the analysis phase in Section 3.2. The repository is publicly available at the GitHub page of the project[9].

**Test Implementation.** Each app in the dataset was implemented based on the vulnerability descriptions provided in the corresponding OWASP MASVS requirement and its associated MASTG security control. For some tests, we directly used code snippets included in the description of the OWASP requirement or control. For example, in the app `MASTG-TEST0013: Testing Symmetric Cryptography`, we included the code that encrypts and decrypts text using the Advanced Encryption Standard (AES) with a hard-coded encryption key.

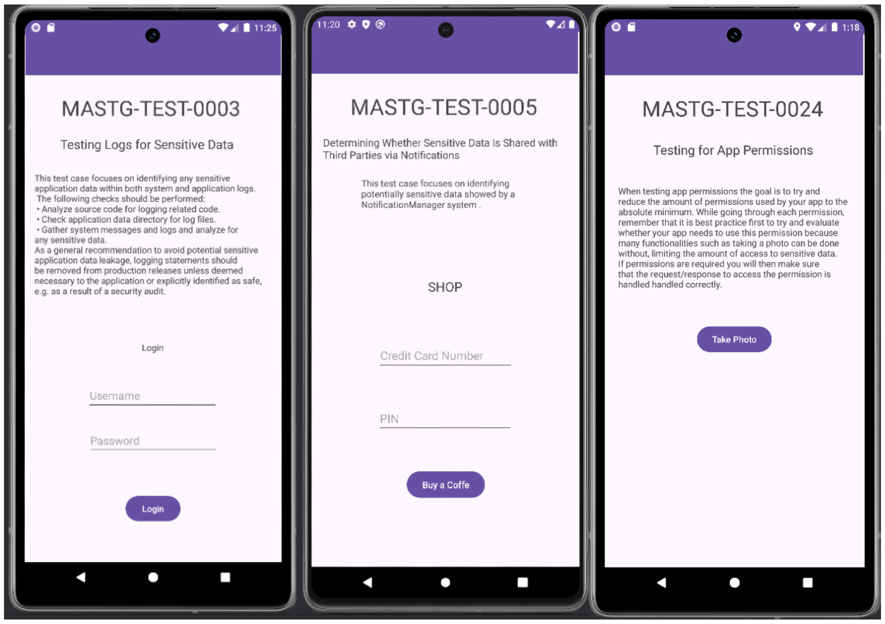
In cases where code snippets were not directly available, we explored external references in the security control descriptions and the official Android Developers' guide to collect the necessary implementation details. A relevant example is the app `MASTG-TEST0043: Memory Corruption Bugs`, where we implemented nine code snippets for timers that can cause memory leaks, as suggested in the external reference [24] cited in the description of the OWASP security control.

If the official documentation and references also do not provide sufficient details, we extended the analysis to platforms like Stack Overflow and GitHub and generated code samples using the ChatGPT AI chatbot. The information collected was then used to implement the vulnerable apps.

An example of an app implemented with this strategy is `MASTG-TEST-0025: Testing for Injection Flaws`. For this test, we revised the DIVA App code sample [7] to create an app with a login prompt that lacks input sanitization, making it vulnerable to SQL injection attacks.

[9]https://github.com/Mobile-IoT-Security-Lab/OWApp-Benchmarking-Suite

Each app in the dataset follows a common layout design, which includes a toolbar to navigate between different sections of the app, the app's ID, a description of the vulnerability, and an interactive area where users can engage with the app. Figure 1 presents various samples from the dataset, showcasing the standard interface structure across different apps.


MASTG-TEST-0003

Testing Logs for Sensitive Data

This test case focuses on identifying any sensitive application data within both system and application logs.
The following checks should be performed:
• Analyze source code for logging related code.
• Check application data directory for log files.
• Gather system messages and logs and analyze for any sensitive data.
As a general recommendation to avoid potential sensitive application data leakage, logging statements should be removed from production releases unless deemed necessary to the application or explicitly identified as safe, e.g. as a result of a security audit.

MASTG-TEST-0005

Determining Whether Sensitive Data Is Shared with Third Parties via Notifications

This test case focuses on identifying potentially sensitive data showed by a NotificationManager system .

MASTG-TEST-0024

Testing for App Permissions

When testing app permissions the goal is to try and reduce the amount of permissions used by your app to the absolute minimum. While going through each permission, remember that it is best practice first to try and evaluate whether your app needs to use this permission because many functionalities such as taking a photo can be done without, limiting the amount of access to sensitive data. If permissions are required you will then make sure that the request/response to access the permission is handled handled correctly.

**Figure 1:** Apps Layout examples of the OWApp Benchmark dataset.

## 5.2 Workflow

The suite includes three auxiliary scripts to facilitate the use of the benchmark to evaluate the Android SAST tools through a workflow depicted in Figure 2.

The initial step involves the use of the *Download Script* to download the entire dataset of apps on the local machine (Steps 1 and 2a in Figure 2). In addition, the script enables the compilation of the source code for all sample apps, allowing the generation of a local dataset targeting a specific Android version (Step 2b). The script (*Tools Script*) supports the download (Step 3) and the automatic deployment and configuration of a set of Android SAST tools (Step 4). In its current version, the script facilitates the deployment of several state-of-the-art SAST tools, including MobSF [11], Sebastian [26], TrueeSeeing [12], and APKHunt [8]. Finally, *Running Script* is designed to initiate the security analysis of configured SAST tools against apps in the OWApp dataset (Step 6). This script also collects and stores the analysis results in a report folder for further review (Step 7).

## 6 Future Work and Conclusion

This paper presented the OWApp Benchmark, a restructured benchmarking suite for configuring and evaluating Android SAST tools, ensuring comprehensive tests that map the security controls provided in the OWASP MASVS that apply to automated static application security testing. Qualitative analysis against the most established Android app vulnerability benchmark repository, Ghera, suggests that the OWApp Benchmark Dataset can cover all the security categories introduced by the OWASP standard,

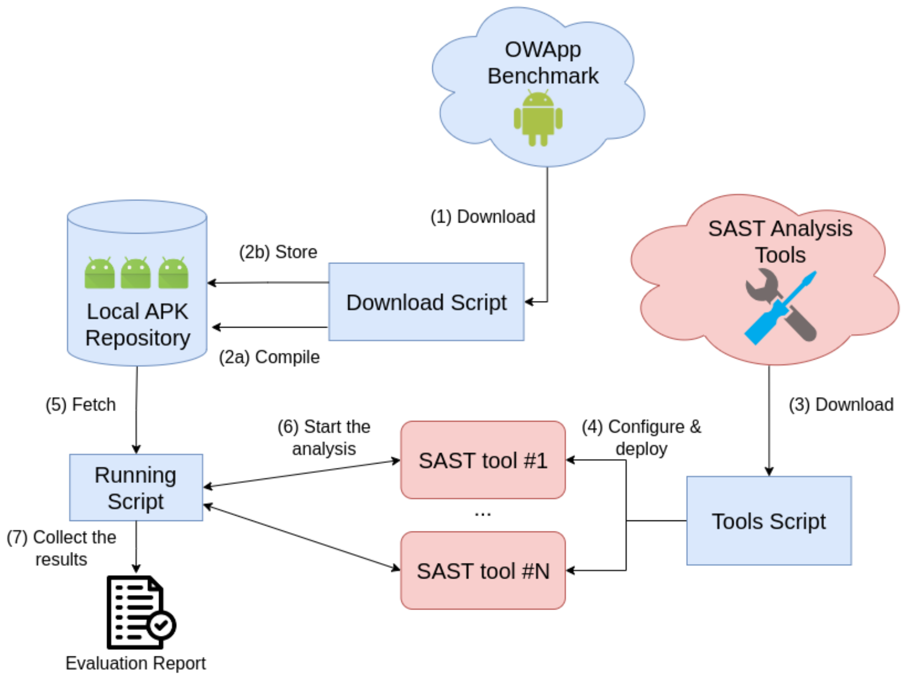


**Figure 2:** OWApp Benchmark Suite workflow for the evaluation of Android SAST tools.

thereby giving the ability to fully evaluate Android SAST tools in terms of detecting a comprehensive set of vulnerabilities affecting apps.

However, this paper is just a first step, suggesting that an extensive empirical evaluation against the leading Android SAST tools, e.g., MobSF [11] or SEBASTiAN [26], is essential to validate the applicability of the OWApp benchmark further. In addition, we advocate that future work should include studying and introducing obfuscation methodologies in the benchmark to challenge existing static analysis techniques against obfuscation [27]. As a final consideration, we plan to release the OWApp benchmark in the spirit of open science to support future research on static analysis techniques and to facilitate fair and unified benchmarking of existing and future Android SAST tools.

## Acknowledgments

This work was partially supported by the Curiosity-driven grant ”Security Assessment of Cross-domain Application Ecosystems” of the University of Genova and funded by the European Union - NextGEneration EU and by project SERICS (PE00000014) under the NRRP MUR program and funded by the EU - NGEU.

## References

[1] Open AI. Chatgpt. `https://openai.com/`, Accessed in January 7, 2025.

[2] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 468–471, New York, NY, USA, 2016. Association for Computing Machinery.

[3] Mohammed K Alzaylaee, Suleiman Y Yerima, and Sakir Sezer. Dynalog: An automated dynamic analysis framework for characterizing android applications. In *2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security)*, pages 1–8. IEEE, 2016.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, jun 2014.

[5] Wang Chao, Li Qun, Wang XiaoHu, Ren TianYu, Dong JiaHan, Guo GuangXin, and Shi EnJie. An android application vulnerability mining method based on static and dynamic analysis. In *2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC)*, pages 599–603. IEEE, 2020.

[6] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252, 2011.

[7] Payatu Security Consulting. Diva android. https://github.com/payatu/diva-android, Accessed in January 7, 2025.

[8] Cyber-Buddy. Apkhunt. https://github.com/Cyber-Buddy/APKHunt, Accessed in January 7, 2025.

[9] DarkReading. Chinese keyboard apps open 1b people to eavesdropping. https://www.darkreading.com/endpoint-security/most-chinese-keyboard-apps-vulnerable-to-eavesdropping, 2024.

[10] Android Developers. Notificationmanager. https://developer.android.com/reference/android/app/NotificationManager, Accessed in January 7, 2025.

[11] Ajin Abraham et al. Mobsf. https://github.com/MobSF/Mobile-Security-Framework-MobSF, Accessed in January 7, 2025.

[12] Takahiro Yoshimura et al. Trueseeing. https://github.com/alterakey/trueseeing, Accessed in January 7, 2025.

[13] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61, 2012.

[14] OWASP Foundation. Mastg demos. https://mas.owasp.org/MASTG/demos/, 2024.

[15] OWASP Foundation. Owasp mastg. https://mas.owasp.org/MASTG/tests/, Accessed in January 7, 2025.

[16] OWASP Foundation. Owasp masvs. https://mas.owasp.org/MASVS/, Accessed in January 7, 2025.

[17] Ankit Gangwal, Shubham Singh, and Abhijeet Srivastava. Autospill: Credential leakage from mobile password managers. In *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy*, CODASPY '23, page 39–47, New York, NY, USA, 2023. Association for Computing Machinery.

[18] Yongzhong He, Xuejun Yang, Binghui Hu, and Wei Wang. Dynamic privacy leakage analysis of android third-party libraries. *Journal of Information Security and Applications*, 46:259–270, 2019.

[19] Google Inc. Android security tips. https://developer.android.com/privacy-and-security/security-tips, Accessed in January 7, 2025.

[20] Google Inc. Mitigate security risks in your app. https://developer.android.com/privacy-and-security/risks, Accessed in January 7, 2025.

[21] Linghui Luo, Felix Pauck, Goran Piskachev, Manuel Benz, Ivan Pashchenko, Martin Mory, Eric Bodden, Ben Hermann, and Fabio Massacci. Taintbench: Automatic real-world malware benchmarking of android taint analyses. *Empirical Software Engineering*, 27:1–41, 2022.

[22] Joydeep Mitra and Venkatesh-Prasad Ranganath. Ghera: A repository of android app vulnerability benchmarks. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE, page 43–52, New York, NY, USA, 2017. Association for Computing Machinery.

[23] Joydeep Mitra, Venkatesh-Prasad Ranganath, and Aditya Narkar. Benchpress: Analyzing android app vulnerability benchmark suites. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 13–18. IEEE, 2019.

[24] Anitaa Murthy. 9 ways to avoid memory leaks in android. https://medium.com/android-news/9-ways-to-avoid-memory-leaks-in-android-b6d81648e35e, Accessed in January 7, 2025.

[25] Stack Overflow. Stack overflow. https://stackoverflow.com/, Accessed in January 7, 2025.

[26] Francesco Pagano, Andrea Romdhana, Davide Caputo, Luca Verderame, and Alessio Merlo. Sebastian: A static and extensible black-box application security testing tool for ios and android applications. *SoftwareX*, 23:101448, 2023.

[27] Francesco Pagano, Luca Verderame, and Alessio Merlo. Obfuscating code vulnerabilities against static analysis in android apps. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 381–395. Springer, 2024.

[28] Felix Pauck, Eric Bodden, and Heike Wehrheim. Do android taint analysis tools keep their promises? In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 331–341, 2018.

[29] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 176–186, 2018.

[30] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. Assessing the security of inter-app communications in android through reinforcement learning. *Computers & Security*, 131:103311, 2023.

[31] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering*, 43(6):492–530, 2017.

[32] Satista. Number of smartphone users worldwide from 2014 to 2029. https://www.statista.com/forecasts/1143723/smartphone-users-in-the-world, Accessed in January 7, 2025.

[33] Christian Schindler, Müslüm Atas, Thomas Strametz, Johannes Feiner, and Reinhard Hofer. Privacy leak identification in third-party android libraries. In *2022 Seventh International Conference On Mobile And Secure Services (MobiSecServ)*, pages 1–6. IEEE, 2022.

[34] OWASP Mobile Application Security. Mastg-test-0004: Determining whether sensitive data is shared with third parties via embedded services. https://mas.owasp.org/MASTG/tests/android/MASVS-STORAGE/MASTG-TEST-0004/, Accessed in January 7, 2025.

[35] OWASP Mobile Application Security. Mastg-test-0005: Determining whether sensitive data is shared with third parties via notifications. https://mas.owasp.org/MASTG/tests/android/MASVS-STORAGE/MASTG-TEST-0005/, Accessed in January 7, 2025.

[36] OWASP Mobile Application Security. Mastg-test-0012: Testing the device-access-security policy. https://mas.owasp.org/MASTG/tests/android/MASVS-STORAGE/MASTG-TEST-0012/, Accessed in January 7, 2025.

[37] OWASP Mobile Application Security. Mastg-test-0042: Checking for weaknesses in third party libraries. https://mas.owasp.org/MASTG/tests/android/MASVS-CODE/MASTG-TEST-0042/, Accessed in January 7, 2025.

[38] Janaka Senanayake, Harsha Kalutarage, Mhd Omar Al-Kadri, Luca Piras, and Andrei Petrovski. Labelled vulnerability dataset on android source code (lvdandro) to develop ai-based code vulnerability detection models. pages 659–666. SciTePress, 2023.

[39] TechInsideOut. Security vulnerabilities found in over 60https://www.techinsideout.co/inside/security-vulnerabilities-found-in-over-60-of-android-apps, Accessed in January 7, 2025.

[40] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)*, 21(3):1–32, 2018.

## A Appendix

| Category | Ghera Test | MASTG ID | Test Name |
|---|---|---|---|
| CRYPTO | BlockCipher-ECB-InformationExposure-Lean | 0013 | Testing Symmetric Cryptography |
| | BlockCipher-NonRandomIV-InformationExposure-Lean | 0013 | Testing Symmetric Cryptography |
| | ConstantKey-ForgeryAttack-Lean | 0013 | Testing Symmetric Cryptography |
| | ExposedCredentials-InformationExposure-Lean | 0001 | Testing Local Storage for Sensitive Data |
| | PBE-ConstantSalt-InformationExposure-Lean | 0013 | Testing Symmetric Cryptography |
| NETWORK | CheckValidity-InformationExposure-Lean | 0021 | Testing Endpoint Identify Verification |
| | IncorrectHostNameVerification-MITM-Lean | 0021 | Testing Endpoint Identify Verification |
| | InsecureSSLSocket-MITM-Lean | 0021 | Testing Endpoint Identify Verification |
| | InsecureSSLSocketFactory-MITM-Lean | 0021 | Testing Endpoint Identify Verification |
| | InvalidCertificateAuthority-MITM-Lean | 0021 | Testing Endpoint Identify Verification |
| | OpenSocket-InformationLeak-Lean | ✗ | ✗ |
| | UnEncryptedSocketComm-MITM-Lean | 0019 | Testing Data Encryption on the Network |
| | HttpConnection-MITM-Lean | 0019 | Testing Data Encryption on the Network |
| | UnpinnedCertificates-MITM-Lean | 0022 | Testing Custom Certificate Stores and Certificate Pinning |
| STORAGE | UniqueIDs-IdentityLeak-Lean | ✗ | ✗ |
| | ClipboardUse-InformationExposure-Lean | 0011 | Testing Memory for Sensitive Data |
| | ExternalStorage-DataInjection-Lean | 0001 | Testing Local Storage for Sensitive Data |
| | ExternalStorage-InformationLeak-Lean | 0001 | Testing Local Storage for Sensitive Data |
| | InternalToExternalStorage-InformationLeak-Lean | 0001 | Testing Local Storage for Sensitive Data |
| | DynamicCodeLoading-CodeInjection-Lean | 0001 | Testing Local Storage for Sensitive Data |

**Table 3:** Mapping of Ghera tests to OWASP MASTG Controls - Part 1.

| Category | Ghera Test | MASTG ID | Test Name |
|---|---|---|---|
| CODE | FragmentInjection-PrivEscalation-Lean | ✗ | ✗ |
| | NoValidityCheckOnBroadcastMsg-UnintendedInvocation-Lean | 0025 | Testing for Injection Flaws |
| | HighPriority-ActivityHijack-Lean | 0026 | Testing Implicit Intents |
| | MergeManifest-UnintendedBehavior-Lean | 0042 | Checking for Weaknesses in Third Party Libraries |
| | OutdatedLibrary-DirectoryTraversal-Lean | 0042 | Checking for Weaknesses in Third Party Libraries |
| PLATFORM | SQLite-execSQL-Lean | 0007 | Testing Whether Sensitive Stored Data Has Been Exposed via IPC Mechanisms |
| | SQLlite-SQLInjection-Lean | 0025 | Testing for Injection Flaws |
| | SQLlite-RawQuery-SQLInjection-Lean | 0025 | Testing for Injection Flaws |
| | InternalStorage-DirectoryTraversal-Lean | 0025 | Testing for Injection Flaws |
| | DynamicRegBroadcastReceiver-UnrestrictedAccess-Lean | 0029 | Testing for Sensitive Functionality Exposure Through IPC |
| | EmptyPendingIntent-PrivEscalation-Lean | 0030 | Testing for Vulnerable Implementation of PendingIntent |
| | ImplicitPendingIntent-IntentHijack-Lean | 0030 | Testing for Vulnerable Implementation of PendingIntent |
| | InadequatePathPermission-InformationExposure-Lean | 0007 | Testing Whether Sensitive Stored Data Has Been Exposed via IPC Mechanisms |
| | IncorrectHandlingImplicitIntent-UnauthorizedAccess-Lean | 0026 | Testing Implicit Intents |
| | NoValidityCheckOnBroadcastMsg-UnintendedInvocation-Lean | 0025 | Testing for Injection Flaws |
| | OrderedBroadcast-DataInjection-Lean | ✗ | ✗ |
| | StickyBroadcast-DataInjection-Lean | 0029 | Testing for Sensitive Functionality Exposure Through IPC |
| | TaskAffinity-ActivityHijack-Lean | ✗ | ✗ |
| | TaskAffinity-LauncherActivity-PhishingAttack-Lean | ✗ | ✗ |
| | TaskAffinity-PhishingAttack-Lean | ✗ | ✗ |
| | TaskAffinityAndReparenting-PhishingAndDoSAttack-Lean | ✗ | ✗ |
| | UnhandledException-DOS-Lean | ✗ | ✗ |
| | UnprotectedBroadcastRecv-PrivEscalation-Lean | 0029 | Testing for Sensitive Functionality Exposure Through IPC |
| | WeakChecksOnDynamicInvocation-DataInjection-Lean | 0024 | Testing for App Permissions |
| | UnnecesaryPerms-PrivEscalation-Lean | 0024 | Testing for App Permissions |
| | WeakPermission-UnauthorizedAccess-Lean | 0024 | Testing for App Permissions |
| | CheckCallingOrSelfPermission-PrivilegeEscalation-Lean | 0024 | Testing for App Permissions |
| | CheckPermission-PrivilegeEscalation-Lean | ✗ | ✗ |
| | EnforceCallingOrSelfPermission-PrivilegeEscalation-Lean | 0024 | Testing for App Permissions |
| | EnforcePermission-PrivilegeEscalation-Lean | ✗ | ✗ |
| | JavaScriptExecution-CodeInjection-Lean | 0031 | Testing JavaScript Execution in WebViews |
| | UnsafeIntentURLImpl-InformationExposure-Lean | ✗ | ✗ |
| | WebView-CookieOverwrite-Lean | 0037 | Testing WebViews Cleanup |
| | WebView-NoUserPermission-InformationExposure-Lean | 0033 | Testing Java Objects Exposed Through WebViews |
| | WebViewAllowContentAccess-UnauthorizedFileAccess-Lean | 0032 | Testing WebView Protocol Handlers |
| | WebViewAllowFileAccess-UnauthorizedFileAccess-Lean | 0031 | Testing JavaScript Execution in WebViews |
| | WebViewIgnoreSSLWarning-MITM-Lean | 0021 | Testing Endpoint Identify Verification |
| | WebViewInterceptRequest-MITM-Lean | 0027 | Testing Deep Links |
| | WebViewLoadDataWithBaseUrl-UnauthorizedFileAccess-Lean | 0031 | Testing JavaScript Execution in WebViews |
| | WebViewOverrideUrl-MITM-Lean | 0027 | Testing Deep Links |
| | WebViewProceed-UnauthorizedAccess-Lean | 0031 | Testing JavaScript Execution in WebViews |

**Table 4:** Mapping of Ghera tests to OWASP MASTG Controls - Part 2.