National College of Ireland

A MASVS-Aligned Evaluation of Android Static Analysis Tools Using the OWApp Benchmark

MSc Research Project
MSc in Cybersecurity

Jasdev Singh
Student ID: 23423340

School of Computing
National College of Ireland

Supervisor: Dr. Arghir Nicolae Moldovan

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Jasdev Singh……
Student ID:	23423340……
Programme:	MSc in cybersecurity…… **Year:** 2025-26……..
Module:	Research Practicum Part-2……
Supervisor:	Dr. Arghir Nicolae Moldovan……
Submission Due Date:	11/12/2025……
Project Title:	A MASVS-aligned evaluation of Android Static Analysis Tools Using the OWApp Benchmark ……
Word Count:	6236…… **Page Count:** 22……

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Jasdev Singh……

Date: 10/12/25……

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	✓
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	✓
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	✓

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

A MASVS-Aligned Evaluation of Android Static Analysis Tools Using the OWApp Benchmark

Jasdev Singh
23423340

Abstract

This study evaluates the effectiveness of four widely used Android Static Application Security Testing (SAST) tools like MobSF, SEBASTiAN, Trueseeing and APKHunt, using the MASVS-aligned OWApp Benchmark. Although SAST tools are essential for identifying mobile vulnerabilities, their outputs differ significantly, making cross-tool comparison difficult. To address this, the OWApp automated framework was used to scan 40 benchmark applications, and custom Python scripts were developed to parse, normalise and consolidate results into a unified severity model.

The analysis reveals some inconsistencies in detection outputs, with tools using different patterns and identifiers. The research delivers a reproducible evaluation pipeline and a comparative assessment of tool performance in terms of severity of vulnerabilities, providing practical insights for improving mobile security testing workflows and offering MASVS category coverage.

1 Introduction

Mobile applications have become central to modern digital ecosystems, enabling users to perform banking, communication, healthcare access and other essential services through handheld devices. As smartphones have proliferated, so too have security threats, targeting Android as a platform which shares biggest market shares in terms of users. Most of the time Android applications are often vulnerable to poor security of data storage, slack components, ineffective cryptography and unsafe network settings, which can be exploited by unauthorized intruders or data leakage.(Enck et al., 2009; Faruki et al., 2015)

As a response to these issues, industry and research communities turn toward structured security guidelines like the OWASP Mobile Application Security Verification Standard (MASVS), offering formalized set requirements when it comes to evaluating mobile application security(*OWASP Mobile Application Security | OWASP Foundation*, n.d.) MASVS groups vulnerabilities into themes such as Authentication, Cryptography, Platform Interaction, Resilience, Storage and others enabling security analysts and application developers to measure applications in a consistent, systematic way.

SAST tools are very significant in the process of detecting such vulnerabilities without the need to run the application. They decompose the code or decompiled artworks in order to identify security holes at a young stage during the development lifecycle. Nevertheless, scholarly research has also demonstrated the weaknesses of SAST instruments such as the inconsistent detection rates, the difference in the range of rules detected, the high false-positive rates, and the inability to standardize the interpretation of(Li et al., 2017; Wadhams et al., 2024; Zhu et al., 2024). Consequently, it is still difficult to make reliable comparisons of SAST tools.

1.2 Research Gap

While there are several SAST tools which are there for detection of vulnerabilities in Android apps such as MobSF, SEBASTiAN, Trueseeing and APKHunt(Cyber-Buddy, 2023/2025; *MobSF/Mobile-Security-Framework-MobSF*, 2015/2025; *Talos-Security/SEBASTiAn*, 2022/2025; Yoshimura, 2015/2025), there is no standardized approach to assess their performance in the challenge of performance in relation to MASVS-aligned vulnerabilities. Previous comparative research is more likely to utilize various datasets, which are not given the same severity scale or unorganized evaluation systems, which cannot be compared with other tools.

The newly created OWApp Benchmark solves a portion of this problem by offering a curated list of deliberately vulnerable Android applications to MASVS categories (Ferrari et al., 2025). However, their work did not have further analysis like the following:

- A Severity Standardization model
- A cross – tool comparison analysis
- MASVS-based vulnerability coverage metrics.

1.3 Research Question

This research investigated the following problems:

"How effectively do widely used Android SAST tools detect MASVS-aligned vulnerabilities in the OWApp benchmark and how can their heterogeneous outputs be normalized to get some meaningful analysis"

1.4 Objective

- Evaluate Android SAST tools (MobSF, SEBASTiAN, Trueseeing and APKHunt) using the OWApp Benchmark.
- Parsing of raw outputs like JSON, TXT, HTML.
- Develop a unified severity model to align severity levels across tools.
- Perform MASVS category analysis, identifying the weightage of app dataset of the benchmark against each MASVS category.
- Generate consolidated visualizations and metrics enabling cross tool comparison.
- Analyse tool limitation in terms of detection behaviour, severity distribution and category coverage.

1.5 Contributions

- **Unified Parsing Framework:** Designed and implemented python based parsing scripts to convert heterogeneous SAST outputs into structured CSV datasets.
- **Unified Severity Model:** Development of standardized approach mapping tool specific severity labels to a common scale (low, medium, high).
- **MASVS coverage analysis:** Analyzing the weightage of each category in accordance with each app assigned to each category of MASVS.

- **Cross tool comparative evaluation:** Analyzed the detection behavior of MobSF, SEBASTiAN, Trueseeing and APKHunt, providing insights.
- **Reproducible benchmark execution environment:** Executed the OWApp automated pipeline and documented a replicable configuration workflow.

1.6 Summary

This chapter introduced the motivation behind Android application security assessment, highlighted limitations of the work done in the past and the research gap addressed in this study. It presented the research question, objective and key contributions.

The next chapter provides a formal literature review based on Android architecture, MASVS/MASTG standards, vulnerability benchmarks, and previous study of the effectiveness of static analysis tools. This gives the basis to the methodology and so on.

2. Related Work

The rapid adoption of smartphones and the increased handling of sensitive data on mobile platform have led to increase in research in mobile security in the past decade. Android, as the dominant mobile operating system globally, has therefore became a frequent target for threat actors and attackers exploiting the weaknesses and vulnerabilities(Enck et al., 2009). To mitigate these risks there have been variety of tools and frameworks developed over time.
So, one of the most recognised communities, OWASP came up with Mobile Application Security Verification Standard, known as MASVS, which is a widely recognised framework providing a structured set of security requirements for mobile applications organised into categories like Authentication, Platform Interaction, Code Quality, Storage, Cryptography, Network Communication and Resilience(*OWASP Mobile Application Security | OWASP Foundation*, n.d.)

Static analysis is one of the early detection techniques to identify weaknesses in software development phase. Mobile Security Framework MobSF is among the most widely used tool in its category which provides automated static and dynamic analysis capabilities which includes permission checks, API misuse identification, cryptographic weakness detection and insecure data storage practices(*MobSF/Mobile-Security-Framework-MobSF*, 2015/2025). Another SAST tools known as SEBASTiAN, a more lightweight, static analysis tool, specializes in pattern-based detection of common android weaknesses and is designed for CI/CD automation(*Talos-Security/SEBASTiAn*, 2022/2025). While Trueseeing offers a rule based engine for analyzing decompiled APKs, specially identifying suspicious API calls and on the other hand APKHunt takes a different approach by aligning its analysis with MASVS requirements, making it primarily a MASVS compliance checker rather than vulnerability severity classifier(Ferrari et al., 2025). This diversity among these tools in terms of methodology, output and detection philosophy makes it difficult to for evaluation or comparison.

Benchmarking SAST tools remain challenging due to absence of reliable datasets for Android applications. Some existing benchmarks such as DroidBench and Ghera focus on specific categories such as data leaks and are also not being updated over time as vulnerabilities keep on evolving with upgrades in OS and its features. Moreover, they don't provide MASVS coverage(Mitra & Ranganath, 2017). The OWApp benchmark addresses this limitation by

providing a modern MASVS compliant suite for vulnerable applications. The benchmark is packaged with multiple insecure APK's covering core MASVS categories, offering a realistic and standard oriented foundation for evaluating the coverage and behavior of security tools(Ferrari et al., 2025).

Previous work comparing Android SAST tools often reports variations in detection rates and false positives between tools(Faruki et al., 2015). Research also indicates that formats of tool output are highly inconsistent, including JSON and plain text, up to graphical reports, which makes comparative analysis difficult. Additionally, the severity rating by the tools is not standardized. These inconsistencies make it harder to perform unified analysis to get some meaningful benchmarking results.

The existing literature also points out challenges in getting a balance between coverage and precision. MobSF and similar tools deliver many findings, enhancing coverage but have more noise, while others like SEBASTiAN have narrower, more focused results. Comparisons conducted highlight that no tool performs consistently better than the others across every vulnerability class, reinforcing that the consolidated results from multi-tool analysis are of great value, which has also been asserted(Zhu et al., 2024)

In summary, literature supports that SAST evaluation of Android-focused tools requires structured datasets like OWApp and a well-designed methodology normalizing the tool outputs. This research follows that route by applying a unified severity taxonomy and performing MASVS-based cross-tool comparison, increasing the interpretability and relevance of SAST evaluation results.

2.1 Android Security Architecture

Android is the most widely used operating system. It has become the backbone of billions of smartphones, tablets and embedded devices. Its design comprises of stack of layers which includes the Linux kernel, hardware abstraction, native libraries, the Android runtime, application framework APIs, and the application layer*(Architecture Overview, 2023.)*. The Linux kernel Provides core security primitives such as process isolation, user-based permissions, file system controls and memory management. These mechanisms form the Android's sandboxing model, where each application is isolated from each other as each one of them within their own UID-isolated process which prevents direct access to the memory and storage of other applications.

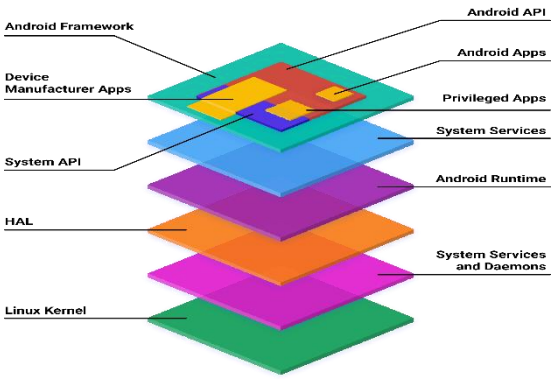

Figure 2.1: Android System Architecture
(*Architecture Overview*, 2023.)

As per fig. 2.1, the kernel, the Hardware Abstraction Layer (HAL) mediates interaction between device hardware and system components by providing standardized interfaces for the hardware drivers. The native layer comprises various system libraries, like SQLite, WebKit, and OpenGL ES, commonly involved in security attacks when badly configured or exposed through vulnerable APIs(Negi et al., 2021). The Android Runtime, ART, was introduced into Android 5.0 as a replacement for the Dalvik VM and performs execution of application bytecode compiled into. Dex format with ahead-of-time compilation, garbage collection, and runtime verification integrated into it, enhancing performance and additionally providing more security guarantees.

The Application Framework provides higher-level services, including telephony, permissions, location, notifications, and activity management, which can interact through binder-based IPC mechanisms. This framework provides the component model that is the core of Android applications: Activities, Services, Broadcast Receivers, and Content Providers. Each component exposes entry points which, if not properly protect may present attack surfaces such as exported components, privilege escalation, and unintended data disclosures(Negi et al., 2021).

Android security model takes much reliance on the permissions, which are declared in manifest of an application and provide access to the protected resources in the system. Android 6.0 also implements runtime permissions, providing the user with a superior level of control since he/she is asked to explicitly grant his/her consent to sensitive operations. Enforcement of SELinux, verified boot, code-signing, and application sandboxing are other platform-level defenses(*SELinuxProject/Selinux*, 2014/2025)

2.2 OWASP Mobile Application Security Verification Standard (MASVS)

MASVS (Mobile Application Security Verification Standard) of OWASP is an internationally recognized framework that specifies mobile app security requirements. It sets a uniform standard for the security evaluation, development, and testing of Android and iOS applications(*OWASP Mobile Application Security | OWASP Foundation*, n.d.). MASVS serves as a platform that facilitates secure mobile development and security assessments making it a central component in the design of vulnerability datasets like the OWApp Benchmark(Ferrari et al., 2025)

MASVS defines security requirements across multiple domains to ensure a comprehensive assessment of application behaviour, data handling and platform interactions. The standard is divided into distinct categories, each targeting a specific dimension of mobile security which MASVS-STORAGE, MASVS-CRYPTO, MASVS-AUTH, MASVS-NETWORK, MASVS-PLATFORM, MASVS-CODE and MASVS-RESILIENCE.

Recent revisions of MASVS (2023) have streamlined the standard into a unified model, replacing older multi-level scoring (L1, L2, R) and introducing a requirement based that highlights practical verification tasks and testing guidance through accompanying MASTG handbook(*OWASP Mobile Application Security | OWASP Foundation*, n.d.) . Each MASVS control is mapped to practical testing techniques which enable us to align the tools and benchmarks with industry accepted standards. MASVS is important for static analysis because it offers a clear and standard way to organize different types of mobile security vulnerabilities. This helps ensure that findings from different SAST tools can be compared consistently.

Further, the datasets that are in line with MASVS, such as the OWApp Benchmark, allow the assessment of SAST tools based on not only their detection rates but also the particular MASVS categories that they cover(Ferrari et al., 2025)

2.3 Summary Table

Work	Type	Primary focus	Scale	Strengths	Limitations
(Zhu et al., 2024)	Survey/ Comparative Evaluation	Analysis of few Android's SAST tools & comparison of capabilities	Medium-scale survey of multiple tools, no large dataset	Good classification of SAST features	Does not use benchmark datasets, lacks MASVS alignment
(Enck et al., 2009)	Foundational Security Study	Empirical analysis of 1,100 Android apps to understand security posture	Large-scale real-world dataset	Highly influential, reveals systemic permission misuse, rigorous empirical method	Outdated, no modern vulnerabilities
(Faruki et al., 2015)	Security Survey	Malware analysis, defenses, threat landscape, high-level mobile security overview	Broad coverage of Android ecosystem	Good foundational overview of malware & defenses	Not focused on SAST tools; lacks dataset benchmarking
(Mitra & Ranganath, 2017)	Benchmark repository	Tool-agnostic, well-documented vulnerability benchmarks	25 benchmark pairs across ICC, Storage, System,	Ready-to-use, reproducible benchmarks	covers 4 areas only, not comprehensive for all Android APIs
(Ferrari et al., 2025)	Benchmark dataset / MASVS-aligned suite	MASVS-aligned vulnerable apps + scripts for automating SAST evaluation	40 + apps (MASVS categories)	MASVS mapping, automation scripts, modern coverage	No metric analysis available in Benchmark, limited to 4 SAST tools only

3. Research Methodology

3.1 Research Design and Workflow

The methodology of this study is built upon a well-organized, multi-phase structure that is intended to assess the performance and detection capabilities of four Android Static Application Security Testing (SAST) tools when used on a benchmark compliant with the Mobile Application Security Verification Standard (MASVS). The main aim of the research design is to guarantee the application of a consistent, reproducible, and standardized evaluation approach that is in line with contemporary mobile security demands. The process begins by selecting a representative benchmark dataset, then systematically collecting data and using the SAST tools selected, normalizing the outputs from the heterogeneous sources, and finally, conducting the analysis according to the OWASP MASVS categories.

The study employs the OWApp Benchmark, a modern suite of intentionally vulnerable Android applications designed according to OWASP Mobile Application Security Verification Standard (MASVS) requirements(Ferrari et al., 2025).

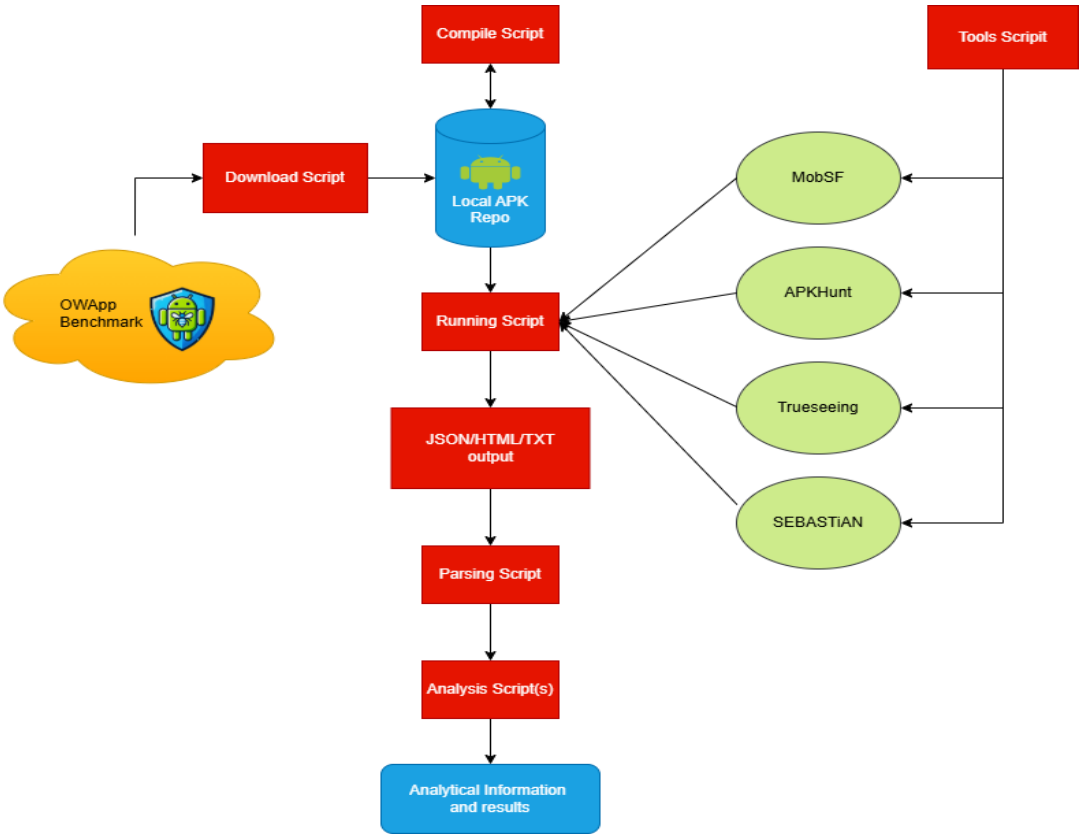

Fig. 3.1: Workflow Overview

3.2 Selection of Platforms and Tools

The OWApp Benchmark and the four SAST tools used in the evaluation are introduced in this subsection. This study examines four Android Static Application Security Testing (SAST)

tools with the help of a MASVS-aligned benchmark of purposely vulnerable applications. The tool selection and dataset were influenced by the factors of relevance, accessibility, diversity of detection methods, and alignment with current research trends in Android security analysis(Li et al., 2017). As per **Fig3.1**, let's discuss the platform and the tools associated with it.

3.2.1 OWApp Benchmark Suite

The OWApp benchmark suite is an automated framework for evaluating mobile applications security using multiple SAST tools. It is structured into three core scripts i.e. Download, tool and running script along with an optional compile script for building APk's. (*Mobile-IoT-Security-Lab/OWApp-Benchmarking-Suite: The OWApp Benchmark: An OWASP-Compliant Vulnerable Android App Dataset*, n.d.)

- **Download Script:** This Script automates retrieval of all OWApp applications from GitHub and prepares them for analysis. It installs prerequisites such as curl, fetches the dataset, and optionally compiles all apps using a user-specified SDK version via the –*compileAll* flag.

- **Tools Script:** The Tools Script prepares the analysis environment by installing and configuring all required SAST tools (MobSF, SEBASTiAN, Trueseeing, APKHunt). It sets up dependencies including GCC, Python, OpenJDK, Go, Jadx, Dex2Jar, Docker and a Python virtual environment, ensuring all tools are ready for execution.

  Running Script: The Running Script performs the actual vulnerability analysis by executing selected SAST tools on APKs stored in a working directory. It scans each app, detects potential security issues and generates detailed JSON reports for downstream parsing and evaluation.

- **Compile Script:** Compile Script automates Android SDK installation and application compilation. It configures SDK components, resolves dependencies and builds each app via gradlew, producing debug-mode APKs for use in security scanning or further analysis.

3.3.2 SAST Tools

Four open SAST tools were bundled with the OWApp Benchmark which were picked to show different static analysis techniques and reporting styles. They were added according to the usual evaluation practices in previous research, and thus, are covering rule-based, signature-based, heuristic and standards-aligned scanning methods.

MobSF

MobSF is a widely adopted mobile application security analysis framework that performs both static and dynamic analysis. For this study, only its static analysis component was used. MobSF conducts code inspection, manifest analysis, cryptographic checks and permission verification, producing detailed JSON reports and severity-labelled findings(*MobSF/Mobile-Security-*

Framework-MobSF, 2015/2025). Its broad detection set, and rich metadata make it particularly suitable for large-scale comparative studies.

SEBASTiAN

SEBASTiAN is a rule based static scanner developed for integration into CI/CD pipelines. It analyses APK's by applying a collection of security rules to decompiled code and metadata, which gives output in plain text logs that identify potential security weakness. While more limited in scope than MobSF, SEBASTiAN provides deterministic and fast scans making it useful as a lightweight option within the evaluation(*Talos-Security/SEBASTiAn*, 2022/2025)

Trueseeing

Trueseeing is also a lightweight and quick static analysis tool which specializes in scanning API usage and decompiled bytecode patterns. It gives minimalistic reports typically generates to the point limited findings, reflecting its focus on specific high-risk indicators(Yoshimura, 2015/2025).

APKHunt

APKHunt is a MASVS-aligned static analysis tool designed specifically to verify Android applications against the MASVS and MASTG guidelines. Its findings are organized under MASVS control identifiers (e.g., MSTG-STORAGE-1, MSTG-NETWORK-3). Unlike other tools in this study, APKHunt does not attach severity levels to findings. APKHunt's standards alignment makes it an important tool for comparing detected vulnerabilities against MASVS expectations(Cyber-Buddy, 2023/2025)

3.3.3 Comparison of SAST Tools

Tool	Analysis approach	Output Format	Detection Focus
MobSF	Hybrid (rule-based + heuristic)	JSON	Broad vulnerability coverage
SEBASTiAN	Rule-based	Log file (JSON based)	Fast checks for common misconfigurations
Trueseeing	Pattern based	Plain text	Lightweight usage/Targeted scans
APKHunt	MASVS aligned, standard checks	Plain text	MASVS and MASTG requirement verification

3.4 Data Processing Pipeline

The data processing pipeline employed in this research was the one that was meant to produce the final and comparable vulnerability dataset from the diverse outputs of the four SAST tools. This is because every tool has its own unique operating procedure and also provides results in different formats, thus it was necessary to create a well-organized workflow to gather, interpret and standardize the results before any comparative analyses were done. The whole pipeline which was created using a mixture of Bash automation and personalized Python scripts stands among the main methodological contributions of the research.

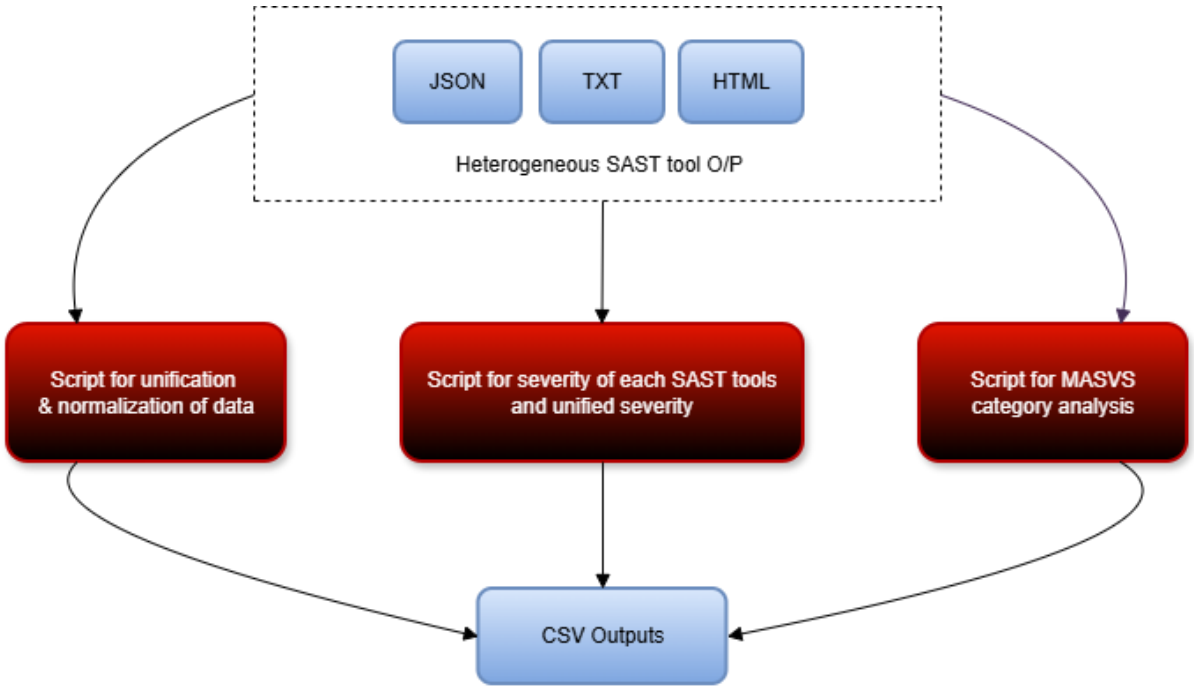

Fig. 3.2: Data Processing pipeline

All scans were executed within a controlled Kali Linux environment running on VMware Workstation Pro, ensuring consistent behaviour across tools and preventing external factors from influencing execution. The OWApp Benchmark's bundled script was used to run each tool sequentially across all target APKs. Each tool produced output artefacts in distinct formats:

- **MobSF** generated structured **JSON** reports containing vulnerability descriptions, categories, severity labels and metadata.
- **SEBASTiAN** produced **JSON log files**, where findings were printed as rule-triggered messages requiring pattern-based extraction.
- **Trueseeing** also generated **plaintext reports**, with minimal structure and limited metadata beyond API-related flags.
- **APKHunt** produced **MASVS-organised plaintext outputs**, where findings were arranged under MASVS headings such as *MSTG-STORAGE-1*. Severity levels were not provided.

As per fig 3.2, the heterogeneous output formats necessitated the creation of a set of Python scripts for the extraction of some common outcome and analyzing combined outcomes. The vulnerability names, descriptions, affected code components, and severity levels were retrieved from MobSF's JSON reports that were parsed through standard dictionary extraction. The outputs of SEBASTiAN and Trueseeing had to be filtered through regular expressions to pinpoint valid findings, eliminate noise, and connect the results with the right APK. A custom parser for APKHunt divided section headers of MASVS from their related findings, thus maintaining the MASVS category structure while making the data ready for subsequent analysis.

One methodological challenge in cross-tool comparison is the inconsistent severity terminology used by different scanners. MobSF labels findings across six levels (e.g., *info*, *warning*, *low*, *medium*, *high*, *critical*), while SEBASTiAN applies severity labels inconsistently across rule hits, and Trueseeing provides no severity classification at all. APKHunt does not include severity metadata since it is designed primarily for MASVS compliance evaluation rather than vulnerability prioritisation. To address this, the study implemented a **unified severity model** applied to the three tools that provide severity information (MobSF, SEBASTiAN and partially Trueseeing, where mapping was inferred when possible). Severity terms were consolidated into four harmonised categories:

- **High** — *high*, *critical*
- **Medium** — *medium*
- **Low** — *info*, *warning*, *low*
- **Unknown** — missing, tool-omitted or unclassifiable severity

Since the OWApp Benchmark is explicitly aligned to the OWASP MASVS, the final stage of the processing pipeline involved assigning MASVS categories to each finding. APKHunt results already contain MASVS identifiers, enabling direct extraction. For MobSF, SEBASTiAN and Trueseeing, MASVS categories were inferred based on vulnerability descriptions, keywords and known mappings between detection rules and MASVS controls(*OWASP Mobile Application Security | OWASP Foundation*, n.d.)

While the underlying mapping of apps to MASVS categories is provided within the benchmark itself, the study expanded this through additional analysis of category distribution, which reflects how vulnerabilities are spread across MASVS domains. This analysis is presented in Results. Section

4. Design Specification

The proposed system's architecture is mainly determined by the necessity for a consistent approach to heterogeneous SAST tool output analysis. Due to the differences in scanning methodology, depth, output format, and coverage between SAST tools, the system is designed to have a single processing pipeline that first conducts the raw tool execution and then separates data transformation and analysis. This provides flexibility of scaling, reproducibility and maintainability, while also permitting the addition of new tools or benchmark datasets with only slight changes. In Fig 4.1 we demonstrate a high-level overview of design architecture layers. At a high level, the architecture consists of four conceptual layers i.e. Input Layer

(OWApp benchmark suite), SAST execution layer, Parsing and Normalization layer and Analysis layer.

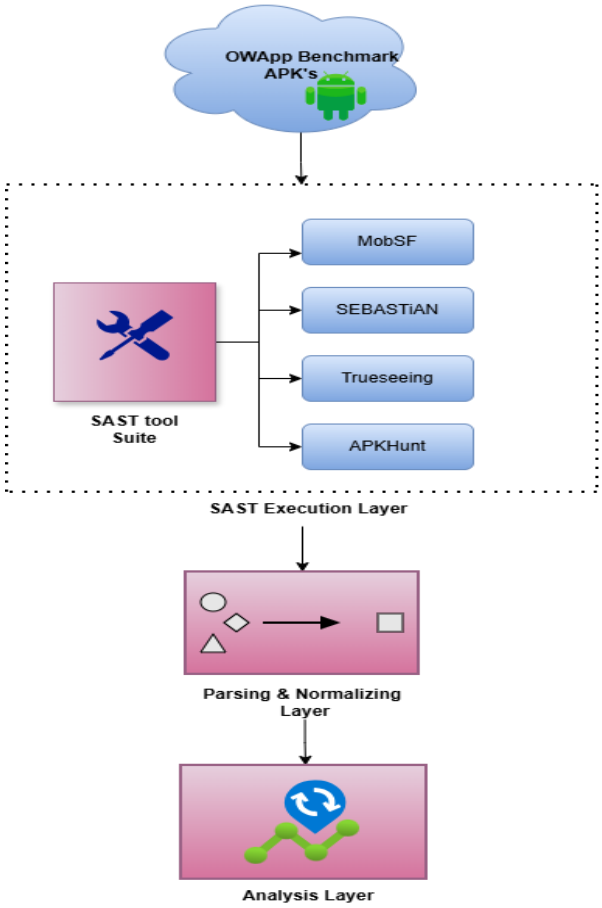

Fig. 4.1: Design Architecture

- **Input Layer:** The OWApp Benchmark provides a controlled set of MASVS-aligned vulnerable android apps for scanning.
- **SAST execution Layer:** MobSF, SEBASTiAN, Trueseeing and APKHunt are executed sequentially against each APK on a controlled isolated VM over Kali Linux environment.
- **Parsing and Normalizing layer:** Custom python scripts transform diverse outputs like JSON, text log files and MASVS-tagged plaintext into a unified format.
- **Analysis Layer:** The normalized dataset is processed to commute severity distribution, MASVS category coverage and vulnerability insights

This layered approach ensures that the system is able to carry out data acquisition, processing and analyzing logic. It also shows design principles commonly found in static analysis research, which argue that raw tool outputs seldom yield direct comparison without the mediation of an abstraction layer(Li et al., 2017).

5. Implementation

The Implementation phase focuses on operationalizing the design pipeline as described in

section 4, by developing a functional approach capable of executing SAST tools, collecting output, parsing them and producing unified data for analysis. The entire infrastructure was set up in a controlled environment of a Kali Linux virtual machine with VMware Workstation Pro, ensuring consistency, isolation, and reproducibility across all benchmark scans, which is a common methodological approach recommended in mobile security research.(Li et al., 2017)

The SAST tool execution is automated using OWApp Benchmark's bundled scripts, which sequentially executes all the scanning of the SAST tools i.e. MobSF, SEBASTiAN, Trueseeing and APKHunt to analyse each APK sequentially. The selection of these tools was based on their respective differences in methodology, detection capabilities, and current relevance in Android security testing. For instance, one of the notable features of MobSF is the provision of detailed static analysis reports in structured JSON format (MobSF, 2024), while APKHunt offers a direct alignment with the OWASP MASVS requirements and delivers MASVS-tagged discoveries as output(Wadhams et al., 2024). The employment of various SAST tools is also in harmony with the generalization found in the Android security literature that not a single static analysis tool provides full coverage, thereby necessitating the use of multi-tool pipelines for realistic benchmarking

Custom python scripts were developed to process heterogeneous outputs. On top level the following core tasks were performed.

- Converting raw outputs to structured intermediate objects.
- Unifying data to a common schema like vulnerability descriptions, severity etc.
- Merging severity levels, apps associated to each MASVS category

5.1 Tools, libraries and script implementation

The system implementation relied on a combination of command line tools, SAST scanners, and custom Python parsing utilities. All these components were run in a Kali Linux virtual machine on VMware Workstation Pro to make sure that the behavior of the system was controlled and repeatable across all the benchmark scans. Following tools were directly used for the implementation of the research.

- **MobSF (Mobile Security Framework)** - performed static analysis and produced structured JSON reports.

- **SEBASTiAN** - a static scanner based on rules generating logs in JSON.

- **Trueseeing** - a lightweight static analysis tool providing very minimal plaintext findings.

- **APKHunt** - a MASVS-aligned scanner giving structured plaintext sections grouped by MASVS identifiers.

- **Bash scripts** - they were used for scanning orchestration and raw output storage per APK.

- **Python scripts** - they were used for parsing, normalization and dataset generation.

The tools were all run offline in the VM which guaranteed a uniform environment for the research.

5.2 Python Libs and Environment

Library	Purpose
Json	Reading and extracting fields from MobSF JSON reports
re	Regex-based extraction for SEBASTiAN and Trueseeing log
pandas	Exporting normalized CSV's
os, glob, pathlib	File discovery and structured directory traversal
datetime	Timestamp logging and naming output files

These libs were chosen to keep the environment lightweight and portable.

6. Evaluation

6.1 Overview of Finding

The evaluation phase analysed the output generated by three SAST tools that provide severity information, MobSF, SEBASTiAN, Trueseeing alongside the MASVS aligned findings produced by APKHunt. Across the benchmark, a total of 1029 findings from 40 app of the benchmark across each MASVS categories. The findings represent combined outputs from all the tools. This dataset forms the basis for the consolidated analysis performed in this study.

MobSF reported the highest number of findings, that is around 800 issues, reflecting its broad detection capabilities that includes the identification of manifest misconfigurations, insecure API usage, cryptographic weaknesses, and permission-related risks. SEBASTiAN identified 189 findings, while Trueseeing contributing 40 findings, consistent with its lightweight and more narrowly scoped detection approach. Although APKHunt did not indicate the severity of its findings, the MASVS-tagged findings were considered in the category-based analyses that will be discussed later in this chapter.

The consolidated severity distribution, which was calculated during the course of this study, clearly indicates that the bulk of the findings were classified as Low severity, which means that they were the major issues identified through all the tools used. However, tools like Trueseeing only classify its findings with "High" and "Medium" only and SEBASTiAN classify its findings with "Low" and "High" severity ratings. Large portion of the flagged errors are low-risk misconfigurations, and the tools recognize fewer high-impact vulnerabilities.

6.2 Severity Analysis

The severity analysis provides insights into the distribution of the criticality distribution of vulnerabilities across the three SAST tools with severity metadata: MobSF, SEBASTiAN, and Trueseeing. Since APKHunt does not assign severity levels to its findings, it is excluded from this portion of the evaluation. The analysis that follows is based solely on the merged dataset created by the custom parsing and normalization pipeline developed for this study and therefore contributing beyond the scope of previous works with OWApp benchmark.

6.2.1 Per Tool Severity Distribution

MobSF Severity Profile

MobSF demonstrates the broadest detection range, reported 800 findings across all 40 benchmark applications. Its findings were characterized by lower-impact issues:

- **Low severity**: 646 findings
- **Medium severity**: 79 findings
- **High severity**: 75 findings

This trend is indicative of the great number of rules that MobSF applies, which also checks for the security of the configurations, the correctness of the metadata, and the adherence to the general best practices. The issues found are often high in volume but less in criticality, which is consistent with the findings in static analysis literature.

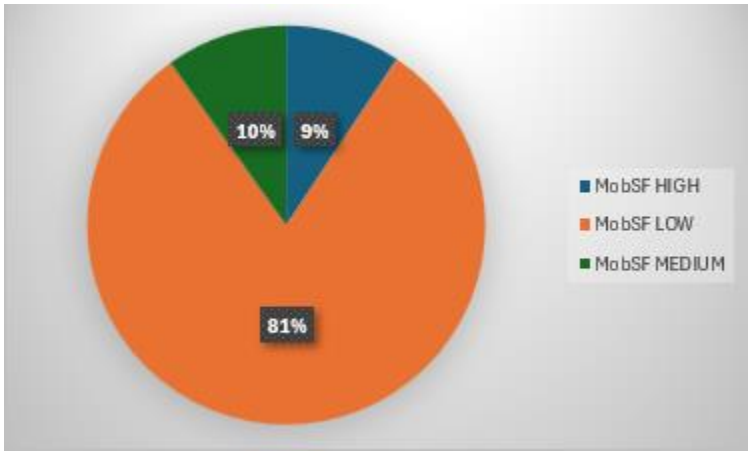

Fig 6.1: MobSF Severity Distribution

SEBASTiAN Severity Profile

SEBASTiAN produces a total of 189 findings, and every split was between low and high severity mark only.

- **Low severity**: 95 findings
- **High severity**: 94 findings

On the other hand, SEBASTiAN brought up a proportionally higher number of severe findings. This is expected given its rule-based design, where each triggered rule corresponds to well-defined violation. The equal distribution also suggests that SEBASTiAN's detection range is narrower but directed at significant violations.

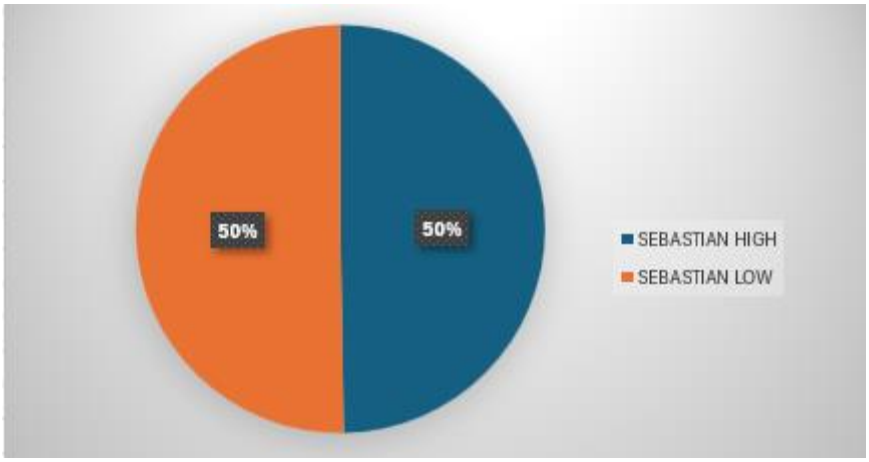

Fig 6.2: SEBASTiAN Severity Distribution

Trueseeing Severity Distribution

Trueseeing found 40 findings, with following breakdown

High Severity: 33 findings
Medium Severity: 7 findings

Trueseeing emphasis o identifying suspicious or unsafe API usage naturally results in a higher proportion of severe findings. It lacks low-severity output which reflects its design philosophy which flags patterns strongly associated with high-risk behaviours

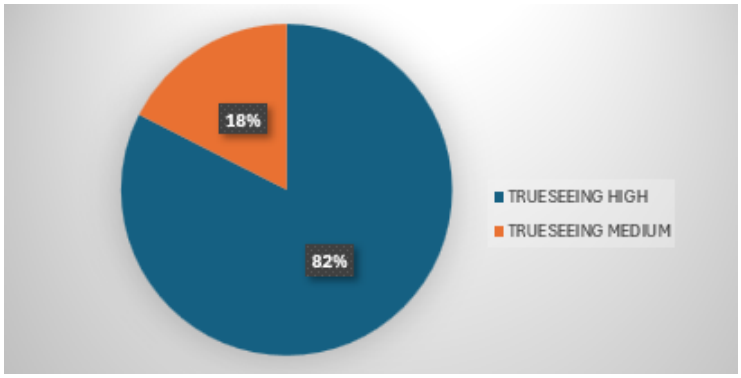

Fig6.3: Trueseeing Severity Distribution

6.2.2 Unified Severity Distribution

This section is a unified and consolidated view of vulnerability severity for the whole benchmark apps. This distribution shows a typical trend of static analysis pipelines: a very

large number of low-severity issues, a moderate number of medium-severity findings, and a small number of real high-risk vulnerabilities. The prevalence of low-severity items is also indicative of the characteristics of automated SAST tools, which often perform broad rule-matching and conduct configuration scanning.

The unified totals are:

- **High severity:** 202
- **Low severity:** 741
- **Medium severity:** 86

This integrated severity analysis was absent from previous work based on OWApp Benchmark. It was done by parsing all the scan results of each app with all the SAST tools in benchmark enabling comparability across heterogeneous tool outputs.

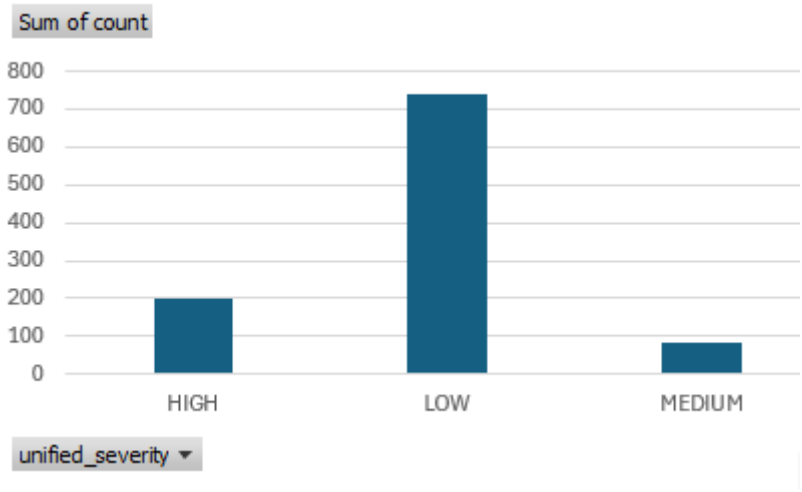

Fig 6.4: Unified Severity Distribution Across Tools

6.3 MASVS Category Analysis

In this section, the coverage of vulnerability across the OWApp Benchmark is assessed with respect to the OWASP categories of MASVS. Since the benchmark is designed in a purposeful manner to be structured by the MASVS requirements, the analysis of the category-level distribution will allow learning valuable information about the thematic focus of the dataset and the nature of vulnerabilities embodied in its design.

The analysis of MASVS categories in this paper is a new contribution to the field of study as the initial OWApp Benchmark studies does not show quantitative distribution analysis and visualisation of MASVS category representation. The outputs were obtained with the help of parsing and custom scripts that were created to work with this project.

6.3.1 MASVS Category Distribution Across Benchmark Apps

The OWApp benchmark consist of 40 vulnerable android apps, each mapped to one or more OWASP MASVS categories. Table 6.1 provides the distribution of applications across seven

MASVS domains, along with sub-category descriptions and percentage of coverage relative to full dataset.

MASVS Category	Description (Short)	#Apps	% of Total
Authentication	Login, identity handling, MFA enforcement	**2**	**6.1%**
Code Quality	Unsafe coding patterns, insecure API usage	**6**	**18.2%**
Cryptography	Key management, algorithm misuse, insecure encryption	**4**	**12.1%**
Network Communication	TLS, hostname verification, secure data transmission	**5**	**15.2%**
Platform Interaction	Exported components, permissions, IPC misuse	**10**	**30.3%**
Resilience	Anti-debugging, anti-tampering, reverse-engineering defence	**9**	**27.3%**
Storage	Sensitive data storage, file handling, insecure data-at-rest	**4**	**12.1%**

Table 6.1: Distribution of Apps among MASVS categories

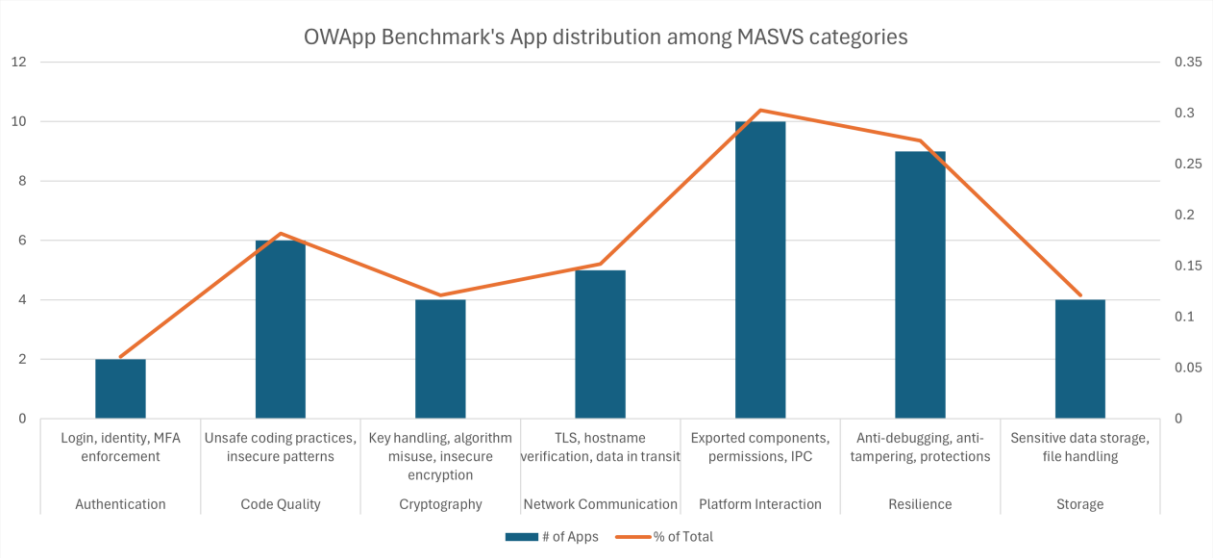

Fig 6.5: App Distribution in OWApp Benchmark

6.3.2 Interpretation of MASVS Distribution

The OWApp benchmark exhibits an intentionally uneven distribution across MASVS categories, reflecting practical mobile security priorities rather than uniform category representation.

Platform Interaction (30.3%)
It is the largest category of the benchmark, and the weaknesses identified here are exported components, misconfigured permissions and inter-process communication (IPC), which are the primary areas of problem. Their existence in the dataset may be viewed as realistic and instructive, since these issues typically occur in Android applications.

Resilience (27.3%)
The second-most represented category includes weaknesses related to tampering, dynamic analysis resistance and general hardening. This indicates that the benchmark emphasizes MASVS-R controls, supporting evaluation of how well tools detect anti-debug and anti-reversing gaps.

Code Quality (18.2%) and Network Communication (15.2%)
These mid-range categories represent frequently observed vulnerability classes in practice, such as insecure API usage, improper input handling and TLS configuration mistakes. Their presence provides a balanced set of weaknesses for SAST benchmarking.

Cryptography (12.1%) and Storage (12.1%)
Both categories have moderate representation. While smaller than the platform-focused categories, they still offer enough coverage to test tools' ability to detect insecure cryptographic implementations and data-at-rest mismanagement.

Authentication (6.1%) This is the least represented domain, consistent with mobile benchmarking behavior where authentication workflows are rarely implemented fully in test applications. Nonetheless, the presence of two apps provides a minimal baseline for evaluating authentication-related detection capability.

7. Discussion

The findings indicate that there are definite variations in the way each SAST tool categorizes and ranks vulnerabilities within the OWApp Benchmark. The largest volume of findings has been generated in MobSF, with most of them being low severity, which implies that it covered a large set of issues, but tends to report numerous issues on information or configuration-level. SEBASTiAN on the other hand reported a less extreme high-low-severe result, indicating a narrower rule range to meaningful security violations. Trueseeing produced a smaller number of detections, but more issues of high severity, as it has a very specific focus on unsafe API usage and structural vulnerabilities.

Such behavioral disparities highlight one important fact: not a single SAST tool can be broad or balanced in its coverage. Rather, their respective tools have a specific detection profile, defined by its design philosophy and rule engine. This justifies the utilization of multi-tool pipeline since the integration of tools provides wider visibility about the classes of vulnerabilities.

The results can also be further contextualized with the MASVS category distribution. The benchmark has a high focus on Platform Interaction and Resilience which in turn coincides with typical high-risk areas in Android applications. As a result, devices that can scrutinize component exposure, IPC abuse as well as anti-tampering vulnerabilities operate better in these areas. Reduced coverage in other categories like Authentication and Cryptography is bound to restrict the conclusions that one would make about tool performance in the respective domains.

In general, the research confirms that although SAST instruments are useful in identifying many problem areas of low severity, the consistency of their ability to detect high-severity MASVS-compatible weaknesses varies significantly. The common severity model and MASVS mapping framework created during this work allow making a comparison in a more structured way and can deliver more insights than the previous works.

8. Conclusion and Future Work

This study evaluated four Android static analysis tools MobSF, SEBASTiAN, Trueseeing and APKHunt aligned with OWApp Benchmark catered along MASVS categories. The dataset had 40 intentionally vulnerable applications. An integrated parsing and normalization process was created to centralize heterogeneous outputs to make cross-tool comparisons possible using severity analysis and mapping of outputs to MASVS categories. The findings are shown to indicate that, though MobSF has a wide coverage and mostly low severity outcomes, SEBASTiAN and Trueseeing reveal more focused high severity problems. APKHunt brings important MASVS-linked data but does not have severity granularity. In combination, the tools have the complementary advantage and strengthen the point that no single SAST solution is comprehensive in all domains of vulnerabilities.

Future work may extend this evaluation by incorporating additional benchmarking datasets or higher precision analysis tools to explore false positives and detection accuracy. Addition of more precise SAST tool would also add up to provide more in-depth scanning capabilities of the benchmark. Expanding MASVS category coverage within the benchmark, improving automated MASVS by mapping heuristic and integrating performance metric such as runtime and resource usage.

References

Application fundamentals | App architecture. (n.d.). Android Developers. Retrieved December 9, 2025, from https://developer.android.com/guide/components/fundamentals

Architecture overview. (n.d.). Android Open Source Project. Retrieved December 9, 2025, from https://source.android.com/docs/core/architecture

Cyber-Buddy. (2025). *Cyber-Buddy/APKHunt* [Go]. https://github.com/Cyber-Buddy/APKHunt (Original work published 2023)

Enck, W., Ongtang, M., & McDaniel, P. (2009). Understanding Android Security. *IEEE Security & Privacy Magazine*, *7*(1), 50–57. https://doi.org/10.1109/MSP.2009.26

Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M. S., Conti, M., & Rajarajan, M. (2015). Android Security: A Survey of Issues, Malware Penetration, and Defenses. *IEEE Communications Surveys & Tutorials*, *17*(2), 998–1022. https://doi.org/10.1109/COMST.2014.2386139

Ferrari, L., Pagano, F., Verderame, L., Romdhana, A., Caputo, D., & Merlo, A. (2025). *The OWApp Benchmark: An OWASP-compliant Vulnerable Android App Dataset*. Preprints. https://doi.org/10.36227/techrxiv.173687667.70476692/v1

Li, L., Bissyandé, T. F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., & Traon, L. (2017). Static analysis of android apps: A systematic literature review. *Information and Software Technology*, *88*, 67–95. https://doi.org/10.1016/j.infsof.2017.04.001

Mitra, J., & Ranganath, V.-P. (2017). Ghera: A Repository of Android App Vulnerability Benchmarks. *Proceedings of the 13th International Conference on Predictive Models*

and Data Analytics in Software Engineering, 43–52. https://doi.org/10.1145/3127005.3127010

Mobile-IoT-Security-Lab/OWApp-Benchmarking-Suite: The OWApp Benchmark: An OWASP-compliant Vulnerable Android App Dataset. (n.d.). Retrieved December 9, 2025, from https://github.com/Mobile-IoT-Security-Lab/OWApp-Benchmarking-Suite/tree/main

MobSF/Mobile-Security-Framework-MobSF. (2025). [JavaScript]. Mobile Security Framework. https://github.com/MobSF/Mobile-Security-Framework-MobSF (Original work published 2015)

Negi, C., Mishra, P., Chaudhary, P., & Vardhan, H. (2021). A Review and Case Study on Android Malware: Threat Model, Attacks, Techniques and Tools. *Journal of Cyber Security and Mobility*. https://doi.org/10.13052/jcsm2245-1439.1018

OWASP Mobile Application Security | OWASP Foundation. (n.d.). Retrieved December 9, 2025, from https://owasp.org/www-project-mobile-app-security/

SELinuxProject/selinux. (2025). [C]. SELinux Project. https://github.com/SELinuxProject/selinux (Original work published 2014)

Talos-security/SEBASTiAn. (2025). [Python]. Talos Security. https://github.com/talos-security/SEBASTiAn (Original work published 2022)

Wadhams, Z. D., Izurieta, C., & Reinhold, A. M. (2024). Barriers to Using Static Application Security Testing (SAST) Tools: A Literature Review. *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*, 161–166. https://doi.org/10.1145/3691621.3694947

Yoshimura, T. (2025). *Alterakey/trueseeing* [Python]. https://github.com/alterakey/trueseeing (Original work published 2015)

Zhu, J., Li, K., Chen, S., Fan, L., Wang, J., & Xie, X. (2024). A Comprehensive Study on Static Application Security Testing (SAST) Tools for Android. *IEEE Transactions on Software Engineering*, *50*(12), 3385–3402. https://doi.org/10.1109/TSE.2024.3488041