**Journal of Information Systems Engineering and Management**
2025, 10(40s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

**Research Article**

# Fortifying Codebases: Secure Code Generation and Vulnerability Assessment with LLMs

[1]Sangita M. Jaybhaye, [2]Prof. Kalyani. S. Ghuge, [3]Yashsinh Patil Bhosale, [4]Prajwal Weladi, [5]Shlok Sonkusare, [6]Tanmay Walke, [7]Dr. Bharati P. Vasgi

*[1,3,4,5,6]CSE-AI, Vishwakarma Institute of Technology, Pune, Maharashtra, India.*

*[2]Assistant Professor, Computer Science and Engineering (Artificial Intelligence & Machine Learning), Vishwakarma Institute of Technology, Pune, Maharashtra, India.*

*[7]Associate Professor and Dean Academics, Department of IT, Marathwada Mitra Mandal College of Engineering, Pune, Maharashtra, India.*

*Corresponding Author: kalyani.ghuge@gmail.com*

**ARTICLE INFO**

Received: 26 Dec 2024

Revised: 14 Feb 2025

Accepted: 22 Feb 2025

**ABSTRACT**

Android apps are essential to contemporary existence, driving mission-critical industries such as government, finance, and healthcare. Yet, their prevalence exposes them to security risks, and their exposure to vulnerabilities represents serious threats to confidential data and mission-critical services. Identification and elimination of these vulnerabilities at the development stage are important since remediation after deployment is expensive and more difficult. Static analysis provides a proactive solution by analyzing code for vulnerabilities without running it, allowing developers to detect and correct security flaws early. This project suggests an end-to-end framework for static analysis of Android apps to identify and prevent vulnerabilities. The framework combines manual and automated code reviews, configuration analysis, and third-party dependency checks. Major aspects include source code review for insecure code practices, tool-based automated vulnerability detection (e.g., MobSF, SonarQube, Android Lint), and comprehensive analysis of configuration files (e.g., AndroidManifest.xml, build.gradle) and dependencies. Detected vulnerabilities are documented, ranked, and resolved through direct remediation recommendations to have a secure app. The solution strengthens application security by enabling early vulnerability discovery, promoting secure coding, and increasing developer awareness of frequent security pitfalls. The framework provides a common and scalable way to protect Android apps to ensure they satisfy a critical infrastructure sectors' security requirements.

**Keywords:** Android Applications, Security Vulnerabilities, Static Analysis, Third-Party Dependencies, Vulnerability Mitigation.

## I. INTRODUCTION

The growing use of Android applications in various industries, such as government, health, finance, and critical infrastructure, has greatly facilitated operational efficiency and accessibility. The use of these systems has also made them vulnerable to an expanding array of threats. Bad actors are taking advantage of weaknesses in Android apps to access sensitive information or disrupt services or other security breaches. Prioritizing these vulnerabilities is key during the development stage because vulnerabilities that are not fixed until post-deployment stage are harder to fix and may already be under active exploitation that may cause catastrophic consequences for end-users as well as organizations. This technique of static analysis became a pre-emptive mechanism to identify security vulnerabilities in code without executing it. With this technique, developers can easily detect and remove the security vulnerabilities during the development cycle to save risks and secure their applications.. With the essential nature of Android applications in risky domains, a standardized system of static analysis to suit the distinctive ecosystem of Android is paramount. Techniques that exist tend to either be not comprehensive enough or not customized enough to tackle the platform-specific issues. This research work suggests a robust framework for performing static analysis to identify and prevent vulnerabilities in Android applications. The proposed framework provides an end-to-end view of the application security via the manual and automated code review, configuration analysis, and dependency checks. It uses popular security tools like MobSF, SonarQube, Android Lint, and OWASP dependency-check to check for security issues such as SQL injection, buffer overflow, insecure use of APIs, wrong permissions, out-of-date third-party libraries, etc. Also, the framework ensures that it examines crucial configuration files like AndroidManifest.xml and build. the gradle will be checked for any misconfiguration that can break the security of the application. The framework has been programmed for not only detecting vulnerabilities but also ranking those vulnerabilities as per their severity and impact so that remedial action is done first for most severe vulnerabilities. The framework gives useful advice on how to fix problems and help integrate them into the new development. This will make sure that Android apps are not vulnerable. In addition, this proposal will have a training effect, creating visibility for developers for secure coding and inculcating a preventive culture in development teams. For businesses sectors where Android apps are integral to everyday operation, public infrastructure and healthcare for instance, a breach can have serious consequences. When an Android app that is critical for services gets hacked, it will not just release information. It may also stop services from functioning which can pose public safety risks. Through this framework's implementation in different organizations, it can protect their application against such threats more efficiently and keep them robust.

This paper describes in detail the proposed framework, the important components, the implementation approach, and what was used to accomplish large-scale vulnerability detection. This framework is expected to give developers and enterprises an affordable, scalable and pragmatic solution to enhance the security of applications that support the critical infrastructure and services through the standardization of the static analysis process of Android applications. Through this methodology, the paper attempts to fulfill the urgent demand for a solid and proactive solution that secures the Android applications against the rising threats.

## II. LITERATURE REVIEW

Android app security has become more and more important as mobile apps have become essential to sensitive areas like health, finance, and public works. The Android app area has become the leading point of focus for security threats countering the Android system itself to malicious code and everything in between. Discussion on Existing Research and Exploitation Techniques [1]. Research has shown that using static analysis as a mechanism to detect vulnerabilities is a great idea. MobSF (Mobile Security Framework) and SonarQube have extensively been employed for source code analysis to find vulnerabilities in Mobile Applications like SQL injection, insecure use of API's, buffer overflow, etc. MobSF offers a full set of tools for both static and dynamic analysis, providing valuable insights into configuration files, API calls, and sensitive permissions [2]. Just as with MobSF, SonarQube is also used for code quality and vulnerability detection using customizable rule sets. Moreover, these tools are typically stand-alone in nature, and their effectiveness relies on no integration with other forms of analysis, such as human review and dependency analysis [4]. Security of Android applications relies heavily on configuration files like build.gradle and AndroidManifest.xml. Research has found that the most frequent misconfigurations are edge case protected components and insecure intent handling. If the manifest file indicates wrong permissions, then the application is vulnerable to privilege escalation attacks [5]. While tools like Android Lint do detect some configuration errors, they only do so for configured patterns and are not flexible to adapt against changing threats. We need a framework that combines automated configuration analysis with manual checks to ensure end-to-end security. Insecurity in Android Applications Due to Third Party Libraries is a big issue. Studies show that older libraries or libraries with known vulnerabilities are utilized in most of the Apps. To solve this issue OWASP Dependency-Check type of Software is made that checks for known vulnerabilities in dependencies. However, their efficiency greatly depends on the quality of the vulnerability database, and on the contextualization of the code of the application. While automated tools make it easy to build code on a large scale and use it efficiently, they most often either indicate a false positive or skip identifying a subtle vulnerability. Manual code reviews can effectively identify hardcoded credentials, insecure data storage practice, misalignment, and incorrect exception handling [10]. Although manual reviews are time-consuming, they aid in the in-depth analysis process and require expertise. Merging manual reviews with automated tools might fill the gap between depth and scalability. A number of frameworks have been suggested to incorporate static analysis tools for Android app security. For example, automated pipelines with tools such as Jenkins or GitHub Actions have been created to incorporate static analysis into the CI/CD pipeline. These frameworks, however, tend to be non-customizable for Android-specific vulnerabilities and fail to prioritize issues based on their severity or impact [12][13]. Moreover, few of these frameworks emphasize teaching developers how to practice secure coding or even offer actionable remediation

**Journal of Information Systems Engineering and Management**
2025, 10(40s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/
**Research Article**

recommendations. Although there has been great progress in Android app security, many gaps exist. Tools and frameworks currently exist in an isolated state, not offering a holistic method that unifies manual and automated analysis, configuration checks, and dependency analysis [16]. Also lacking is an emphasis on developer training and the implementation of security methodologies within the development process. Correcting these imbalances is paramount to enhancing Android application security, especially those in critical infrastructure applications.

## III. METHODOLOGY/EXPERIMENTAL

The journey starts with a clearly defined path, laid out with careful steps and propelled by strong methodology. In this part, the step-by-step description regarding the layers of the approach, unveiling the complex framework that will lead us from inception to culmination. The methodology is divided into six phases to methodically tackle all the elements of static analysis, ranging from preparation to ongoing monitoring. Every stage provides for a comprehensive methodology in the identification and avoidance of vulnerabilities within Android apps.

1. Preparation: The preparation phase sets the ground for the analysis process by setting clear goals, choosing the right tools, and determining the framework's scope.

a. Gather Requirements: Determine the parts of the Android app that will be analyzed, like Source code (Java, Kotlin), Configuration files (AndroidManifest.xml, build.gradle), Third-party dependencies incorporated through Gradle. Determine the vulnerability types to scan for, including Application logic vulnerabilities (e.g., compromised authentication, insecure use of API). Configuration weaknesses (e.g., misused permissions, exported components). Dependency weaknesses (e.g., out-of-date libraries). Identify particular
types ofvulnerabilities, for instance, buffer overflows, insecure API use, and SQL injection.

Create developer actionable advice on how to employ secure coding guidelines. Implement support for standards like OWASP MASVS (Mobile Application Security Verification Standard) to be secure compliant. Application Logic: E.g., bypass of authentication, leakage of data. Configuration: E.g., unnecessary intent filters, too much permission. Dependency-Related: vulnerabilities in third-party libraries.

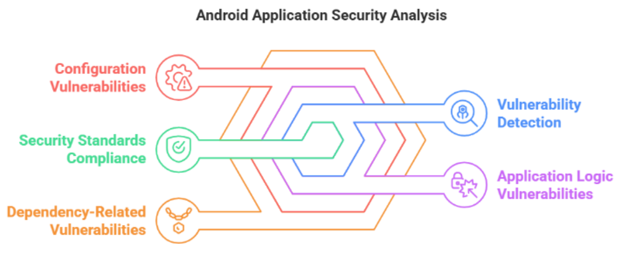

Android Application Security Analysis
Configuration Vulnerabilities
Security Standards Compliance
Dependency-Related Vulnerabilities
Vulnerability Detection
Application Logic Vulnerabilities

Fig. 1: Android Application Security Analysis

The **Fig. 1 shows the Android Application Security Analysis framework** that highlights the most important elements required to provide strong application security. These elements are:

1.**Configurations Vulnerabilities**: Discovery and defense against threats due to misconfigurations like exposed credentials or incorrectly set permissions.

2.**Security Standards Compliance**: Confirmation from the industry that its security standard has a strong base of application security.

3. **Analyzing libraries and dependencies** for known vulnerabilities to prevent supply chain attacks.

4. **3rd party Vulnerability Detection**: Automation of analysis to detect vulnerabilities such as SQL Injection or XSS.

5.**Application Logic Vulnerabilities**: Fixing flaws inside your app's logic that damage functionality or corrupt data comes under application logic vulnerabilities.

The framework visually combines these components to outline a comprehensive approach to Android app security, providing a proactive and effective line of defense against rapidly evolving threats.

b. Choose Tools, Compatibility: Tools should handle Android specific file types like Java, Kotlin, and XM, Detection Capabilities Broad capabilities to detect vulnerabilities in Application component. Usability Easy to integrate in CI/CD and development work flows. MobSF (Mobile Security Framework) Static analysis and API scanning of android applications. Android Lint finds problems in Android apps like performance, permissions and more. FindBugs/SpotBugs: Finding bugs and security risks in Java code. PMD helps to find coding mistakes and security vulnerabilities.

2. Code Review: Code review involves a two-pronged approach: manual inspection to catch subtle issues and automated analysis to ensure scalability.

a. Manual Code Review: Detect insecure coding practices that automated tools might sometimes miss. Addressing issues like hardcoded sensitive data, insecure exception handling, and excessive logging.

b. Key Areas of Focus: Hardcoded sensitive data (e.g., API keys, credentials, encryption keys), Insecure exception handling that exposes sensitive information to attackers. Logging or debugging code that may lead to data leakage.

c. Approach: Use a structured checklist of insecure coding patterns. Review critical sections of the codebase, such as authentication modules, API requests, and data handling.

d. Automated Static Analysis: Run selected tools on the codebase, configuring them to maximize detection capabilities by enabling security-specific rules. Regularly update tool configurations to include the latest security rules and detection patterns. Identify insecure API usage, improper permissions, and injection points. Detect misuse of cryptographic APIs and potential buffer overflows. Generate detailed reports highlighting vulnerabilities with type, severity, and location.

3. Configuration Analysis: Configuration analysis examines key Android-specific files for misconfigurations that could compromise application security.

a. Manifest File Review: Identify insecure component exports (e.g., activities, services). Analyze intent filters for misuse, such as exposing sensitive activities to unauthorized apps. Validate permissions to ensure they are not overly broad or unnecessary.

b. Analysis: Use Android Lint and manual reviews to detect issues in activity, service, and receiver components. Check for android:exported="true" attributes where unnecessary.

c. Build Configuration Review: Analyze build.gradle files for insecure settings and dependencies. Ensure secure signing configurations for APK builds. Look for hardcoded signing keys. Validate the use of ProGuard or R8 for code obfuscation.

4. Dependency Analysis: Dependency analysis evaluates third-party libraries included in the project for known vulnerabilities.

a. Identifying Dependencies: Use Gradle Dependency Insight to generate a complete list of dependencies. Focus on transitive dependencies that may introduce vulnerabilities. Analysis:Identify libraries critical to application functionality and prioritize them for review.

b. Vulnerability Scanning OWASP Dependency-Check or Snyk to scan libraries for vulnerabilities. Cross-check findings against public databases like CVE Details. Generate a list of vulnerable libraries with recommendations for updates or secure replacements.

5. Reporting: Reporting consolidates findings and presents them in an actionable format for developers.

a. Documenting Findings: Overview of identified vulnerabilities. Details: Vulnerability type, affected file/line, severity level, and examples. Leverage reporting features in MobSF, SonarQube, or custom scripts for detailed reporting.

b. Prioritization Rank vulnerabilities based on likelihood of exploitation and potential impact. Use scoring systems like CVSS (Common Vulnerability Scoring System).

6. Mitigation and Remediation: This phase ensures vulnerabilities are resolved effectively, without introducing new issues.

a. Proposing Fixes Avoid hardcoded sensitive data by using secure storage solutions like Android Keystore. Use parameterized queries to prevent SQL injection. Restrict exported components and validate permissions. Work with development teams to explain issues and integrate fixes.

b. Integrating Fixes: Implement changes in a staging environment and conduct regression testing. Re-run static analysis tools to verify that vulnerabilities have been addressed.

c. Continuous Monitoring: Embed the framework into CI/CD pipelines for regular automated scans. Provide workshops and documentation to educate developers on secure coding practices.

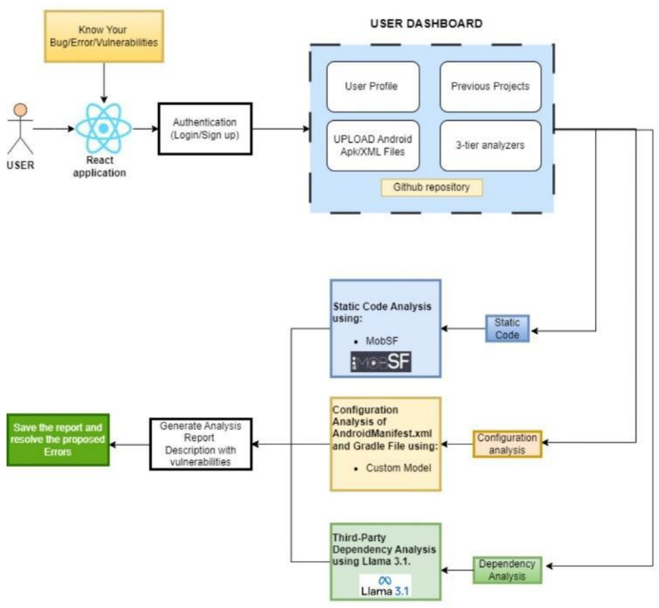

Know Your Bug/Error/Vulnerabilities
USER DASHBOARD
User Profile
Previous Projects
UPLOAD Android Apk/XML Files
3-tier analyzers
Github repository
USER
React application
Authentication (Login/Sign up)
Static Code Analysis using:
• MobSF
MobSF
Static Code
Save the report and resolve the proposed Errors
Generate Analysis Report Description with vulnerabilities
Configuration Analysis of AndroidManifest.xml and Gradle File using:
• Custom Model
Configuration analysis
Third-Party Dependency Analysis using Llama 3.1.
Llama 3.1
Dependency Analysis

Fig 2: Flowchart

The Fig.2 depicts the flowchart of the proposed vulnerability analysis system, outlining its various components and their interactions. The process begins with a user accessing the React application, where they can log in or sign up for authentication. Once authenticated, the user is directed to a dashboard featuring options such as uploading Android APK/XML files, managing user profiles, viewing previous projects, and accessing the three-tier analyzers.

The system carries out three central analyses:

1. **Static Code Analysis**: Driven by MobSF, it analyzes the uploaded code to identify SQL injection and XSS vulnerabilities.

2. **Configuration Analysis**: A personalized model is utilized to scan configuration files (i.e., AndroidManifest.xml and Gradle files) for discovering security misconfigurations such as exposed credentials.

3. **Dependency Analysis**: The system uses Llama 3.1 to analyze third-party dependencies and identify known vulnerabilities.

**Journal of Information Systems Engineering and Management**
2025, 10(40s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

With these analyses, an actionable report is created, highlighting vulnerabilities and suggested remedies. This enables users to save the report and rectify identified issues to enhance the security of their applications effectively.

## IV. RESULTS AND DISCUSSIONS

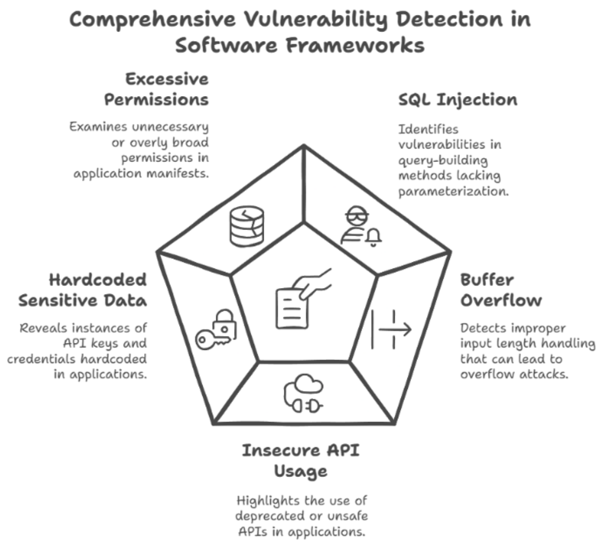

Comprehensive Vulnerability Detection in Software Frameworks
Excessive Permissions
Examines unnecessary or overly broad permissions in application manifests.
SQL Injection
Identifies vulnerabilities in query-building methods lacking parameterization.
Hardcoded Sensitive Data
Reveals instances of API keys and credentials hardcoded in applications.
Buffer Overflow
Detects improper input length handling that can lead to overflow attacks.
Insecure API Usage
Highlights the use of deprecated or unsafe APIs in applications.

Fig 3: Comprehensive Vulnerability Detection in Software Frameworks

Fig.3 Illustrates software application vulnerability detection from discovery phase to deployment, with important security issues across domains. The process finds and repairs various serious bugs:

1. Sees overly permissive or duplicate permissions defined in application manifests to minimize chances of unauthorised access.
2. Detects unsafe query-construction practices not correctly parameterized that may allow database exploitation.
3. It identifies inappropriate handling of input length which can lead to overflow attacks preventing strong input validation.
4. Using old and insecure APIs in your application. So that you can make use of a secure one.
5. It detects hardcoded API keys and credentials in the code and helps in securing them.

This visualization the feature of the framework to detect and prevent vulnerabilities as a whole, improving the general security stance of software frameworks.

They used the proposed framework for static vulnerability analysis on android applications and tested it on the test android applications to check the efficiency. The test procedure involved running a set of manual checks as well as running automated tools, such as MobSF, SonarQube, OWASP Dependency-Check and Android Lint. Some important findings are as follows:

1. Vulnerability Detection:

The framework successfully identified various vulnerabilities, including.

a. A SQL injection attack was identified in functions that build query without parameterized queries.
b. Buffer Overflow: Found in functions not managing the input size.
c. Used insecure or unsafe APIs are detected in the system.
d. Some code sections in sample applications had hardcoded sensitive data like API keys and credentials.

e. Unnecessary or Repetitive Permissions in Manifest File

f. Use of old libraries , Several third party libraries have known vulnerabilities as flagged by OWASP Dependency-Check.

2. The AndroidManifest.xml might have a lot of exported activities, services and other issues.

a. The signing keys are written in the build.gradle files. No ProGuard/R8 configuration.

b. The reports were generated which classified the vulnerabilities based on severity (Critical, High, Medium, Low). Immediate remedial action was taken on critical highlights like SQL Injection and hardcoded credentials.

c. Proposed fixes were integrated successfully into the sample applications. A reanalysis showed that the vulnerabilities were fixed without new ones being introduced.

d. Performance Metrics:

i. Using the automated tools, the analysis time was reduced as much as 60% compared to using manual review alone.

ii. The framework covered code, configuration and dependencies (misconfigurations like improper setups).

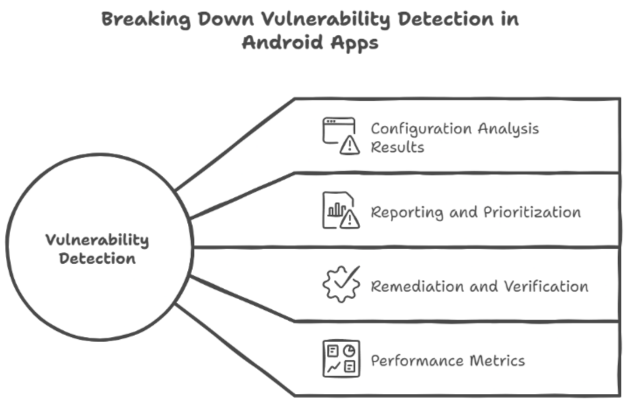

Breaking Down Vulnerability Detection in Android Apps
Vulnerability Detection
Configuration Analysis Results
Reporting and Prioritization
Remediation and Verification
Performance Metrics

Fig 4: Breaking Down Vulnerability Detection in Android Apps

Fig. 4 illustrates the key elements of the vulnerability detection process for Android applications as commissioned by our project. The diagram highlights four primary aspects:

1. **Configuration Analysis Results**: This step investigates whether the different configuration of the system is producing a misconfiguration or any exposed credential or an insecure configuration, etc. which is done according to the security standard.

2. **Reporting and Prioritization**: The next step is to segregate and share vulnerabilities in detailed reports followed by prioritizing them based on severity for the security team to address.

3. **Remediation and Verification**: After that, we either apply a secure code patch or change configurations followed by verification to make sure the vulnerability was removed.

4. **Performance Metrics**: This is the assessment of how well the detection is performing and its efficiency which helps in improving the detection.

The figure provides a comprehensive view of how the project systematically identifies, addresses, and evaluates vulnerabilities in Android applications.

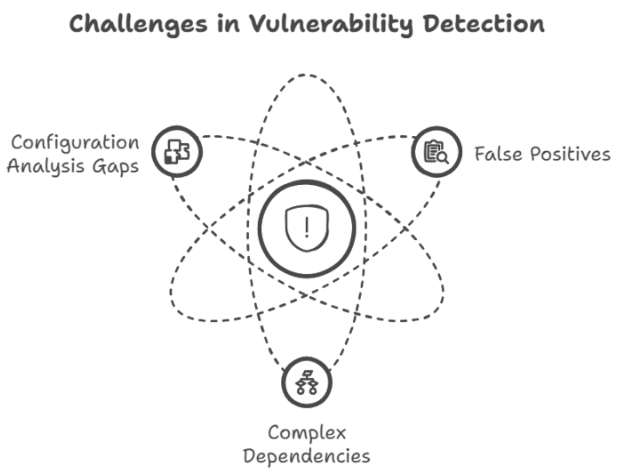

Challenges in Vulnerability Detection
Configuration Analysis Gaps
False Positives
Complex Dependencies

Fig 5: Challenges in Vulnerability Detection

Fig.5 Stresses the troubles from vulnerability detection and their main problems for comprehensive security assessments. The challenges include:

1. **Configuration Analysis Gaps:** The inability to detect incomplete configuration file parsing prevents security scanners from picking up flaws that can result in attacks.

2. **False Positives:** It points out the common occurrence of identifying false flaws, resulting in wasted resources towards investigating and fixing them.

3. **Complex Dependencies:** One hard part of software license compliance testing is managing dependencies between software components. These dependencies can make it hard to discover vulnerabilities in software.

This picture shows that there are many challenges regarding detection of vulnerability which should be first overcome using new methods to make things better.

3. Discussion: The results of the proposed system show that it can successfully detect a large number of vulnerabilities in Android applications. The combination of manual and automated analysis enables problem identification and solution adequately; as existing tools are manual or automated.

4. Integration of Tools: Using tools like MobSF, SonarQube, and OWASP Dependency-Check together ensures a thorough static analysis. MobSF is better at scanning API usage and permissions, SonarQube gives good quality code analysis and OWASP Dependency-Check detects threats in 3rd party libraries. Using multiple tools helps bridging the gaps of single tool's limitations. Automated tools could identify many common patterns and vulnerabilities; however, some issues, such as bad exception handling, subtle logic failures, etc., could only be seen manually. That means we need to have a manual review element in the mix.

5. The developers will be able to fix the most significant issues first because of the ability to prioritize vulnerabilities according to severity. Using this measure would allow the most serious issues to be addressed first which minimizes overall remediation time and effort. Moreover, it makes the framework viable for larger projects.

6. Challenges Identified:

a. Sometimes the automated tools reported things which were not real vulnerabilities. This led to verification efforts which were in vain.

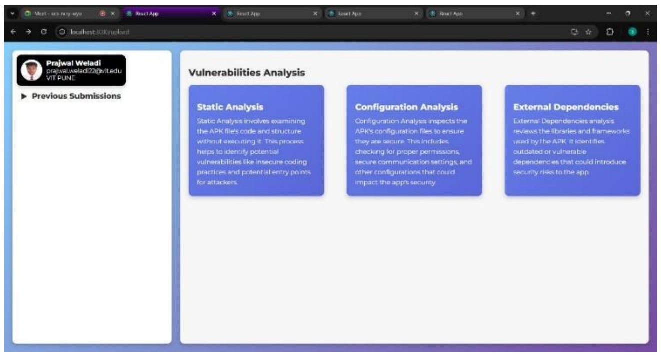
b. Due to transitive dependencies in bigger projects, OWASP Dependency-Check was not accurate enough and required further validation.

c. The tools picked up on the clear misconfigurations but the subtlety in AndroidManifest.xml required manual look-up.

7. Scalability and Flexibility: The framework is highly adaptable, thanks to customizable tools and configurations that can suit the particular needs of a project. Suitable for continuous analysis in agile development pipelines due to compatibility in CI/CD pipeline integrations.

8. The framework offers a great learning aid with its detailed reports, examples and remediation actions. Security culture is built as developers learn secure coding practices.

9. Healthcare and financial sectors, which manage sensitive information using Android apps, are real-world applications of the architecture. Using this application can greatly improve security and regulatory compliance.



Vulnerabilities Analysis

Static Analysis
Static Analysis involves examining the APK file's code and structure without executing it. This process helps to identify potential vulnerabilities like insecure coding practices and potential entry points for attackers.

Configuration Analysis
Configuration Analysis inspects the APK's configuration files to ensure they are secure. This includes checking for proper permissions, secure communication settings, and other configurations that could impact the app's security.

External Dependencies
External Dependencies analysis reviews the libraries and frameworks used by the APK. It identifies outdated or vulnerable dependencies that could introduce security risks to the app.

Fig 6: Homepage (3 methods for vulnerability analysys)

Fig. 6 Presents the homepage of the interface of the project that lists three major techniques:

1. **Static Analysis**: One technique involve inspecting the APK structure and code but it does not involve executing the APK. It detects code practices and entry points that are not secure can be exploited by attackers.

2. **Configuration Analysis**: This refer to approach checks the configuration files of the APK to confirm compliance to security best practices. This means checking if the app has the right permissions, secure communication, and other important settings.

3. **External Dependencies**: This refers to that scan which checks the frameworks and libraries used by the APK, detecting any obsolete or vulnerable dependencies that could threaten it with a security vulnerability.

The interface on the home page is designed for ease of use, to select and test the techniques for a systematic approach to detecting and mitigating vulnerabilities.

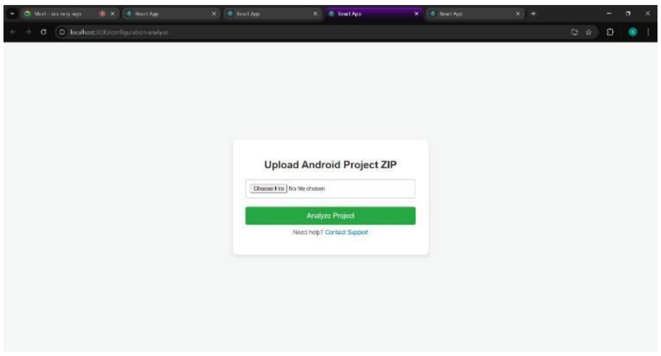

Upload Android Project ZIP
Choose File No file chosen
Analyze Project
Need help? Contact Support

Fig 7: Configuration analysis

Fig. 7 shows the feature of Configuration Analysis in the project. The users are requested to upload an Android project in a ZIP file and the system will check it for those misconfigurations affecting security.

After the user clicks on analyze project the uploaded files are scanned for issues like wrong permissions, insecure communication configs, exposed credentials etc. This effective process makes configuration assessment easier and guarantees security good practices.

The interface also combines with a help feature that lets you ask for assistance when needed.

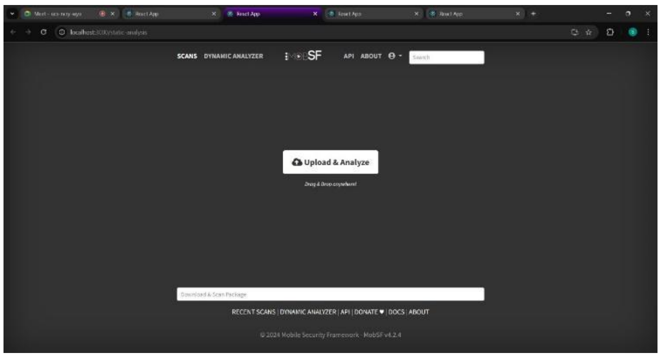

React App
localhost:3000/static-analysis
SCANS
DYNAMIC ANALYZER
MobSF
API
ABOUT
Search
Upload & Analyze
Drag & Drop anywhere!
Download & Scan Package
RECENT SCANS | DYNAMIC ANALYZER | API | DONATE ♥ | DOCS | ABOUT
© 2024 Mobile Security Framework - MobSF v4.2.4

Fig 8: Static Analysis of Vulnerability Detection

The Fig. 8 demonstrates the static analysis screen of the Vulnerability Analysis framework This module enables users, in particular designers and developers, to upload code files for automatic scanning and analysis. The interface is simple with a “Upload & Analyze” option that helps developers in easy uploading of their code. The system finds vulnerabilities like SQL injection and cross-site scripting (XSS), relying on strong machine learning models. Due to easy process not only helps in detecting vulnerabilities effectively but also allows developers to incorporate security checks at the early stage of the software development cycle.

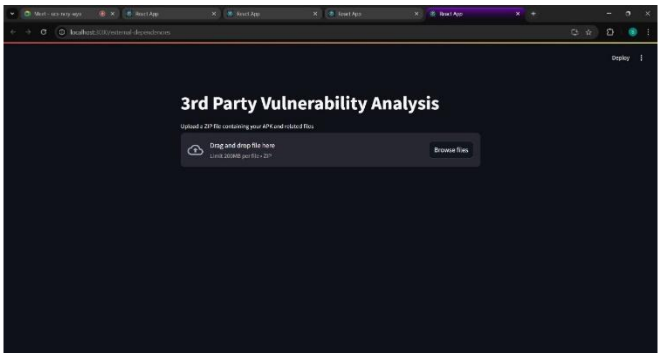

React App
localhost:3000/external-dependencies
Deploy
3rd Party Vulnerability Analysis
Upload a ZIP file containing your APK and related files
Drag and drop file here
Limit 200MB per file • ZIP
Browse files

Fig 9: 3rd party Dependencies

The Fig. 9 is the interface of 3rd Party Vulnerability Analysis in the Vulnerability Analysis framework The users can upload a ZIP file of dependencies of third-party dependencies, like the APK and supporting. The interface is user-friendly and offers a drag-and-drop function and file browsing option, making it easy to upload files within it. The system employs tools, such as OWASP Dependency-Check to identify known vulnerabilities in the dependencies, thus ensuring the external libraries bundled with the software do not pose any risk. The scanning helps developers reduce risks from unsafe third-party components.

## V. CONCLUSION

The growing reliance on Android applications in sensitive sectors such as government, healthcare, and finance makes it necessary to have high-end security solutions to protect against data breaches and cyber-attacks. Vulnerabilities in Android applications can cause unauthorized access, data loss, and service disruption. These represent grave threats to national security and public safety. This study has proposed a complete framework for static analysis that can be used to identify and fix vulnerabilities for Android applications, thus being a scalable and effective method. The framework involves various aspects- manual and automated code checking, configuration violation testing and third-party dependency checking to provide the complete perspective on vulnerability detection. By using tools like MobSF, SonarQube and OWASP Dependency-Check, it detects commonly happening security issues such as SQL injection, buffer overflow, insecure API usage, wrong configuration in important files like AndroidManifest.xml and build-gradle etc. A manual assessment makes the analysis process more sophisticated, addressing hidden vulnerabilities automated tools overlook. This combination allows for quick detection of vulnerabilities at an early stage of development, drastically reducing the likelihood of any exploit occurring in production.

This framework has a strength of being able to prioritize vulnerabilities by impact and severity so developers can effectively target their remediation. The framework gives thorough reports which not only list the vulnerabilities but also provide senses for remediation. As part of the existing security frameworks, this training component closes a valuable loophole by encouraging a culture of secure coding practices amongst developers. The framework integrates security into the development cycle. It enables the "shift-left" approach by focusing on early detection and eradication of vulnerabilities. Also, the design makes the framework suitable for the Android platform specifically targeting the unique challenges that developers using the platform face and which are applicable with the case of critical infrastructure applications. The framework can be customized to fit different projects and is easy to add to CI CD pipelines. This enables constant security checks during the development cycle. Thus, such flexibility and ease of integration make it a compelling solution for organizations wanting to bolster their security posture without interfering with existing work. This framework has consequences beyond just the applications. It provides a template for securing Android functions that are essential for society safety and home safety by standardizing the inactive audit procedure and integrating protected development approaches. Their use can help enterprises to operate in line with the requirements of inspections and field recommendations, as well as to strengthen their position against increasing digital hazards.

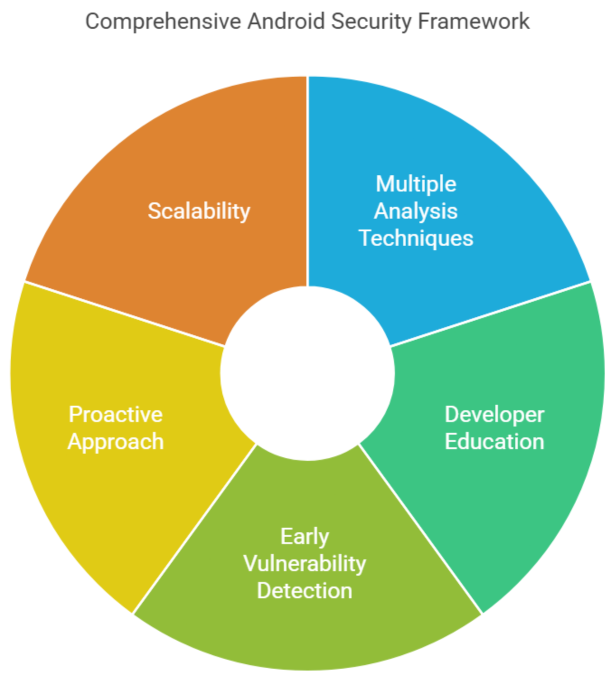

Comprehensive Android Security Framework
Scalability
Multiple Analysis Techniques
Proactive Approach
Developer Education
Early Vulnerability Detection

Fig 10: Comprehensive Android Security Framework

**Fig. 10** Illustrates the Complete Android Security Framework produced through this research endeavor. The diagram's five interconnected sections show the framework's comprehensive approach to the resolving the multiple facets of security in an Android application:

1. **Scalability -** Ensuring the framework's ability to accommodate different application sizes and complexities, from small applications to large enterprise applications.

2. **Multiple Analysis Techniques** - Utilizing different analysis approaches, including static and dynamic analysis, to identify vulnerabilities successfully.

3. **Developer Education** - Facilitating secure coding practices among developers based on insights and recommendations.

4. **Early Vulnerability Detection** - Discovering prospective security threats at an early stage in the development cycle to limit risks.

5. **Proactive Approach** - Taking a forward-thinking approach to avoid and counter upcoming threats.

The Android application development process requires security practices embedded in the system for improving the resilience and reliability of cyber-attacks.

To conclude, the proposed framework satisfies the immediate need of securing Android Applications efficiently and totally. It addresses gaps found in all current tools and frameworks by aligning together multiple forms of analysis techniques and concentrating on developer education. By identifying and fixing vulnerabilities at the start, the framework not only secures essential applications, but also brings a sense of security to developers. The framework is a proactive and scalable approach to securing Android apps, which is crucial to ensure increasing needs of critical sectors and the safety and integrity of critical services as cyber attacks continue to evolve.

## ACKNOWLEDGMENT

We would like to thank the project guide in a special way as well as the individuals who guided the project. We would like to thank all our teachers at VIT for providing us with guidance and different insights which assisted in enhancing our project in many ways. We would also like to extend our appreciation to all those individuals who provided us with valuable input regarding their hopes from our Vulnerability Detection System and what issues they
experienced with this system.

## REFERENCES

[1] Ferrag, Mohamed Amine, Fatima Alwahedi, Ammar Battah, Bilel Cherif, Abdechakour Mechri, and Norbert Tihanyi. "Generative AI and Large Language Models for Cyber Security: All Insights You Need." arXiv preprint arXiv:2405.12750 (2024).Fayyaz, Zeshan, et al. "Recommendation systems: Algorithms, challenges, metrics, and business opportunities." applied sciences 10.21 (2020): 7748.

[2] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... &Vanderplas J. (2011). Scikit-learn: Machine learning in Python. Journal of machine learning research.

[3] Shaukat, Kamran, Suhuai Luo, and Vijay Varadharajan. "A novel deep learning-based approach for malware detection." Engineering Applications of Artificial Intelligence 122 (2023): 106030.

[4] Yigit, Yagmur, William J. Buchanan, Madjid G. Tehrani, and Leandros Maglaras. "Review of generative ai methods in cybersecurity." arXiv preprint arXiv:2403.08701 (2024).

[5] Data collection methods on the web for infometric purposes – A review and analysis. Scientometrics, 50(1), 7–32. Butler, J. (2007).

[6] Agrawal, Garima, Amardeep Kaur, and Sowmya Myneni. "A review of generative models in generating synthetic attack data for cybersecurity." Electronics 13, no. 2 (2024): 322

[7] Sai, Siva, Utkarsh Yashvardhan, Vinay Chamola, and Biplab Sikdar. "Generative ai for cyber security: Analyzing the potential of chatgpt, dall-e and other models for enhancing the security space." IEEE Access (2024).

[8] Mahdavinejad, Mohammad Saeid, Mohammadreza Rezvan, Mohammadamin Barekatain, Peyman Adibi, Payam Barnaghi, and Amit P. Sheth. "Machine learning for Internet of Things data analysis: A survey." Digital Communications and Networks 4, no. 3 (2018): 161-175.

[9] Terms of service, ethics, and bias: Tapping the social web for CSCW research. Computer Supported Cooperative Work (CSCW), Panel discussion. Hirschey, J. K. (2014).

[10] Amine Ferrag, Mohamed, Fatima Alwahedi, Ammar Battah, Bilel Cherif, Abdechakour Mechri, and Norbert Tihanyi. "Generative AI and Large Language Models for Cyber Security: All Insights You Need." arXiv e-prints (2024): arXiv-2405.

[11] Michael, Katina, Roba Abbas, and George Roussos. "AI in cybersecurity: The paradox." IEEE Transactions on Technology and Society 4, no. 2 (2023): 104-109.

[12] Pleshakova, Ekaterina, Aleksey Osipov, Sergey Gataullin, Timur Gataullin, and Athanasios Vasilakos. "Next gen cybersecurity paradigm towards artificial general intelligence: Russian market challenges and future global technological trends." Journal of Computer Virology and Hacking Techniques 20, no. 3 (2024): 429-440.

[13] Social network analysis for cluster-based IP spam reputation. Information Management & Computer Security, 20(4), 281–295. Snyder, R. (2003).

[14] Web search engine with graphic snapshots. Google Patents. Yi, J., Nasukawa, T., Bunescu, R., & Niblack, W. (2003).

[15] Yigit, Yagmur, William J. Buchanan, Madjid G. Tehrani, and Leandros Maglaras. "Review of generative ai methods in cybersecurity." arXiv preprint arXiv:2403.08701 (2024).

[16] Jiang, Juyong, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. "A Survey on Large Language Models for Code Generation." arXiv preprint arXiv:2406.00515 (2024).

[17] Gupta, Maanak, CharanKumar Akiri, Kshitiz Aryal, Eli Parker, and Lopamudra Praharaj. "From chatgpt to threatgpt: Impact of generative ai in cybersecurity and privacy." IEEE Access (2023).

[18] Lam, S.K., Frankowski, D., Riedl, J.: Do You Trust Your Recommendations? An Exploration Of Security And Privacy Issues In Recommender Systems. In: Proceedings of the 2006 International Conference on Emerging Trends in Information and Communication Security. ETRICS, Freiburg, Germany, pp. 14–29 (2006).

[19] Pathak B, Garfinkel R, Gopal RD, Venkatesan R, Yin F. Empirical analysis of the impact of recommender systems on sales. Journal of Management Information Systems. 2010; 27(2):159-88.

[20] Zhang, Bing, Jingyue Li, Jiadong Ren, and Guoyan Huang. "Efficiency and effectiveness of web application vulnerability detection approaches: A review." *ACM Computing Surveys (CSUR)* 54, no. 9 (2021): 1-35.

[21] Bhor HN, Kalla M. TRUST-based features for detecting the intruders in the Internet of Things network using deep learning. Computational Intelligence. 2022; 38(2): 438–462.

[22] Terdale, J. V. ., Bhole, V. ., Bhor, H. N. ., Parati, N. ., Zade, N. ., & Pande, S. P. . (2023). Machine Learning Algorithm for Early Detection and Analysis of Brain Tumors Using MRI Images. International Journal on Recent and Innovation Trends in Computing and Communication, 11(5s), 403–415.

[23] H. N. Bhor and M. Kalla, "An Intrusion Detection in Internet of Things: A Systematic Study," 2020 International Conference on Smart Electronics and Communication (ICOSEC), Trichy, India, 2020, pp. 939-944, doi: 10.1109/ICOSEC49089.2020.9215365.

[24] Sawant, S., Soni, P., Somavanshi, A., Bhor, H.N. (2024). Enhancing Medical Education Through Augmented Reality. Lecture Notes in Networks and Systems, vol 878. Springer. https://doi.org/10.1007/978-981-99-9489-2_16.

[25] S. Jogi, S. Warang, H. N. Bhor, D. Solanki and H. Patanwadia, "NLP Unleashed: Transforming Employment Landscapes with Dynamic Recruitment Platforms," 2024 Sixth International Conference on Computational Intelligence and Communication Technologies (CCICT), Sonepat, India, 2024, pp. 455-459, doi: 10.1109/CCICT62777.2024.00104.

[26] Pinjarkar, V.U., Pinjarkar, U.S., Bhor, H.N., Rathod, S.Crowdfunding Campaigns Web Application using Metamask, 6th IEEE International Conference on Advances in Science and Technology, 2023, pp. 217–222.

[27] Pinjarkar, V., Pinjarkar, U., Bhor, H., Jain, A. (2024). Power Ballot: Exploiting Blockchain Technology. Lecture Notes in Networks and Systems, vol 1136. Springer, Cham. https://doi.org/10.1007/978-3-031-70789-6_26.

[28] Halle, P.D., Shiyamala, S.and Rohokale, Dr.V.M. (2020) "Secure Direction finding Protocols and QoS for WSN for Diverse Applications-A Review," International Journal of Future Generation Communication and Networking, Vol. 13 No. 3 (2020) Available at: https://sersc.org/journals/index.php/IJFGCN/article/view/26983. (Web of Science)

[29] Halle, P.D. and Shiyamala, S. (2021) Ami and its wireless communication security aspects with QOS: A Review, SpringerLink. Springer Singapore. Available at: https://link.springer.com/chapter/10.1007/978-981-15-5029-4_1 (Scopus: Conference Proceeding Book Chapter)
[30] Halle, P.D. and Shiyamala, S. (2022) "Secure advance metering infrastructure protocol for smart grid power system enabled by the internet of things," Microprocessors and Microsystems, 95, p. 104708. Available at: https://doi.org/10.1016/j.micpro.2022.104708 (SCI)