International Journal of Information Security (2026) 25:17
https://doi.org/10.1007/s10207-025-01190-1

REGULAR CONTRIBUTION

From code to security: machine learning approaches in android vulnerability detection

Kaya Emre Arikan[1] · Sait Melih Doğan[1] · Ercan Nurcan Yilmaz[2] · Serkan Gönen[3]

Received: 2 April 2025 / Accepted: 8 December 2025 / Published online: 27 December 2025

Abstract

In today's technology-driven society, an increasing number of individuals rely on mobile devices, leading to a surge in the availability of applications. Smartphone users constantly search for apps that meet their needs, resulting in a flood of options in the marketplace. However, there is a growing concern regarding the security of Android applications, as many have shortcomings in addressing critical security aspects. One reason behind this issue often lies in the lack of automated mechanisms during the design and development stages to identify, test, and rectify vulnerabilities in the source code. It is crucial to address these issues proactively rather than relying solely on updates and patches for already published apps. In response to this challenge, researchers have proposed machine learning techniques to enhance application security by detecting vulnerabilities and malicious code within source code. This systematic literature review delves into this domain by examining 85 carefully selected technical studies published between 2017 and 2024. It aims to shed light on the strengths, weaknesses, and practical applicability of these techniques, while also identifying areas for further improvement. Moreover, the growing focus on advanced approaches—such as Large Language Models (LLMs) and Explainable AI (XAI)—indicates a trend toward more transparent and context-aware vulnerability detection. By synthesizing key insights from the current literature, this review enhances our understanding of Android security approaches, identifies promising directions for future research, and ultimately contributes to the advancement of more secure mobile applications through machine learning-based vulnerability detection.

Keywords Vulnerability analysis · Machine learning · Mobile security · Intrusion detection · Vulnerability detection

1 Introduction

Technological advancements in mobile devices have been remarkable, with the introduction of foldable displays offering more screen space in a portable form factor. 5G connectivity has brought faster speeds and low latency, enhancing internet experiences and enabling real-time applications [1]. Advanced camera systems equipped with multiple lenses and image processing capabilities have significantly enhanced photography and videography. Biometrics, such as facial recognition and fingerprint sensors, offer secure and convenient device unlocking. Powerful processors, AI integration, and longer-lasting batteries have enhanced overall performance and user experience. Augmented and virtual reality technologies [2] have bridged the gap between the physical and virtual worlds. At the same time, mobile payment systems and a thriving app ecosystem have revolutionized transactions and user functionalities.

Software systems in domains face numerous vulnerabilities due to their accompanying source code. As of July 2023, there are over 192,000 recorded instances of vulnerabilities in the Common Vulnerabilities and Exposures (CVE) repository [3]. In each of the 5 years, an average of over 17,000 manifestations of vulnerabilities have been identified, which represents a doubling of the vulnerability count compared to a decade ago. These vulnerabilities are caused by factors

✉ Kaya Emre Arikan
kayaemrearikan@gmail.com

[1] Department of Information Security Engineering, Graduate School of Natural and Applied Sciences, Gazi University, Tunahan, Dumlupinar 30 Agustos Street, No:2 D8, Ankara 06560, Turkey

[2] Department of Electrical and Electronics Engineering, Faculty of Technology, Gazi University, Ankara 06560, Turkey

[3] Department of Software Engineering, Faculty of Engineering and Architecture, Istanbul Gelisim University, Istanbul, Turkey

related to the source code itself, including reuses, implementation, dependencies, changes, and mistakes made by developers [4–6]. Additionally, online question-and-answer forums like Stack Overflow can unintentionally introduce software vulnerabilities when users unknowingly replicate responses as code snippets [7–9].

Mitigation of software vulnerabilities typically entails either preemptive or retrospective scrutiny through code reviews conducted by the original developers or other members within a collaborative development environment, whether it is a team or a broader community context [10]. Nevertheless, it is plausible that vulnerabilities could persist across temporal intervals, whether acknowledged or not [11]. While code reviews serve as a valuable mechanism, their undertaking can entail substantial temporal and financial investments for developers, necessitating the exploration of potential avenues of exploitation and the identification of commensurate remedial measures. In extensive code bases, the exhaustive examination and rectification of every source code file associated with a vulnerability may prove impracticable [12].

Researchers have enhanced conventional analytical approaches by incorporating machine learning methodologies, thereby augmenting the efficacy of software vulnerability identification. Nonetheless, a recent investigation of the pervasiveness of security vulnerabilities inherent in code snippets, as manifested within the collaborative question-and-answer platform Stack Overflow, has ascertained that a substantial 36% of the entirety of Common Weakness Enumeration (CWE) vulnerability classifications were discerned among an aggregate of over 600,000 instances of C/C++ code snippets. Remedial amendments aimed at alleviating these vulnerabilities were proffered for the code snippet responses, yet only half of these recommendations were integrated into practice [9].

Android is a target for security breaches because it is incredibly popular and widely used on devices such as desktops and laptops. Cybercriminals are particularly interested in exploiting apps and devices to access sensitive information, use the device's processing power, collect data through sensors, and exploit network connections. Over the past decade, the number of vulnerabilities discovered in the Android operating system has significantly increased, with 6251 reported incidents as shown in Fig. 1 [13].

A shift towards mobility has brought significant changes to various industries and our everyday lives. Thanks to advancements, we are experiencing a connected, flexible, and efficient world. The widespread use of smartphones and tablets has empowered individuals to stay connected while accessing information, communication channels, and services from anywhere [14]. This growing trend towards mobility has led businesses to prioritize interfaces and applications as part of their strategies for increasing user engagement. Moreover, the integration of 5G networks has further accelerated this transition by offering connectivity. It has also opened up experiences such as augmented reality, smart devices powered by the Internet of Things (IoT), and hassle-free mobile payments [15]. As this shift towards mobility continues, it shapes our future by driving innovation and paving the way for a world centered around movement.

Cybersecurity encompasses a combination of technologies and methods carefully designed to protect information, software applications, computer systems, and interconnected networks from infiltrations, adversities, or unauthorized access [16]. Machine Learning (ML) paradigms have proven highly effective in cybersecurity. In this context, ML applications become crucial because they can analyze datasets with the help of advanced hardware resources and improved algorithms [17]. In particular, ML has three dimensions: the reasons behind its deployment (the "why" aspect), the objectives it serves (the "what" aspect), and the methods it employs to conduct security assessments (the "how" aspect).

The initial dimension, addressing the "why," pertains to elucidating the motives behind the execution of cybersecurity undertakings. Adhering to the taxonomy posited by the Gartner model, these rationales are categorized into five classes: Prediction, Prevention, Detection, Response, and Monitoring [18].

The second dimension, encapsulating the "what," delves into the technical strata subject to vigilant scrutiny. Hierarchically organized layers within this dimension encompass the network, endpoint, application, user, and process domains.

The third dimension, encompassing the "how," delineates the modalities by which security mechanisms are scrutinized. This examination may be rendered historically, during the quiescent state, or in real-time transit. A comprehensive consolidation of these facets is succinctly presented in Table 1, summarizing the information.

Furthermore, integrating technology with the IoT has led to advancements like smart homes and cities that optimize resource management efforts while promoting sustainability. Beyond this scope, mobility has revolutionized healthcare, education, and agriculture industries by facilitating telemedicine services, remote learning opportunities, and precision farming techniques [20]. Lastly, mobile communication platforms and social media have effectively bridged geographical distances, fostering interconnectedness among communities worldwide. Overall, it is undeniable that the shift toward mobility has fueled innovation, driven growth, and advanced societal progress, thereby establishing itself as an integral part of modern life.

While organizations often prioritize securing their networks, endpoints, and critical data, they may unintentionally overlook aspects that can leave vulnerabilities. These overlooked components could include systems, outdated devices, or frequently used applications that might not receive regular

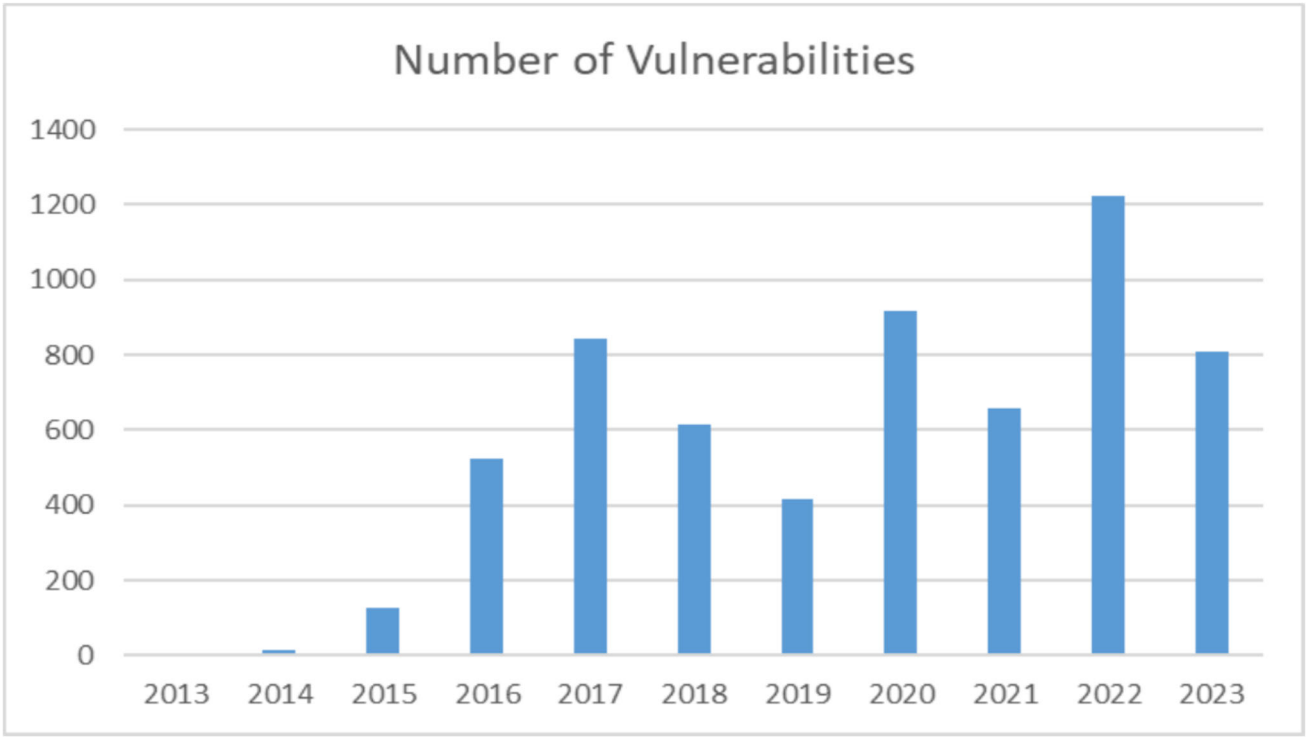

Fig. 1 Android vulnerability trends between the years 2014 and 2023

Table 1 ML dimensions in cyber security areas [19]

Protection layer (What?)	Task (Why?)	Role of ML (How?)
Network (SCADA systems, Ethernet, Virtual networks)	Prediction and Detection	Prediction of network packet parameters, Detection of varied network attacks
Endpoint (IoT device, Mobile, Server)	Prediction and Detection	Prediction of the next system call, Categorizing programs into malware, spyware, adware, etc
Application security	Detection and Prevention	Anomaly detection in HTTP requests, Detection of known attacks
User behavior	Detection, Prevention, and Monitoring	Anomaly detection in User actions, Peer-group analysis based on different users
Process behavior	Prediction and Detection	Prediction of the next user action, Detection of known fraud

security updates or patches. Moreover, third-party integrations and supply chain security might be disregarded, creating entry points for attackers. Organizations must identify and address these overlooked components to ensure cybersecurity by implementing security measures, regular updates, and ongoing monitoring to mitigate potential risks and establish a robust and comprehensive security strategy.

In this paper, we focus on studies using ML and DL techniques in the vulnerability analysis of applications on the Android operating system. In recent years, there has been a notable increase in the use of ML methods for vulnerability detection [21]. Consequently, a comprehensive understanding of the ML processes becomes imperative for thoroughly comprehending ML-based studies focusing on source code vulnerability detection. The ML lifecycle includes several essential steps: data extraction, preprocessing, feature selection, model training, evaluation, and deployment [22].

This review contributes to the field in the following ways:

It provides a comprehensive overview of prior research endeavors in the vulnerability analysis of Android Applications, focusing on the application of machine learning techniques.

This study presents a systematic literature review for vulnerability researchers, focusing solely on vulnerability detection methods that utilize machine learning. Unlike previous

studies, which mainly emphasized malware detection, this research stands out as unique.
From 2017 to 2024, we examined research on the topic, highlighting challenges and open issues to provide insights into addressing problems within the Android application vulnerability detection around ML techniques.
It offers insights into their detection accuracy and effectiveness in identifying vulnerabilities.
It gathers and presents valuable resources, including available datasets, relevant to conducting future studies in this area.
It synthesizes and summarizes the existing open challenges that require the attention and investigation of researchers. Additionally, it discusses future research directions based on recent innovative techniques such as Explainable AI (XAI), Large Language Models, and context-aware models.
Recognizing and identifying open issues and challenges in the design of ML/DL-based vulnerability detection mechanisms around Android applications, aiming to provide direction for future research efforts.

Unless explicitly specified otherwise, the term "ML techniques" is employed throughout this paper as an umbrella term encompassing both ML and DL techniques and models.

The subsequent sections of this paper are organized as follows. Section 2 describes the methodology used for this systematic literature review. Section 3 thoroughly examines the background, terms and definitions, and relevant literature. The experimental studies analyzed in this research are divided into three main sections: vulnerability analysis, available datasets, and challenges. Section 4 offers a comprehensive review of studies focused on vulnerability analysis using machine learning for Android applications, while Sect. 5 is dedicated to the review of available datasets. Section 6 assesses the challenges and limitations encountered in the current state of research in the field. Section 7 addresses the threats to the validity of the review. Finally, Sect. 8 concludes the article by summarizing the findings and final remarks from the study, and the last section outlines future directions based on our findings.

2 Methodology

To address methodological rigor and ensure the transparency and reproducibility of our literature review, we adopted the PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) framework [23]. This framework is widely acknowledged within the scholarly community as the gold standard for ensuring methodological precision in systematic reviews. Implementing PRISMA enhances the validity of our findings and facilitates critical peer assessment. Below, we outline our systematic review protocol, which includes the search strategy, selection criteria, and analysis methods employed.

2.1 Databases and search terms

Our review systematically explored several reputable databases to ensure comprehensive coverage and address the epistemological gaps in the research domain. The rigorous selection of multiple databases allowed for the triangulation of findings and minimized potential selection bias. The databases included:

IEEE Xplore: A premier repository for computer science and engineering literature
ACM Digital Library: Essential for accessing high-impact computing research
ScienceDirect: Offering extensive interdisciplinary scientific publications
SpringerLink: Provides access to journals with robust peer-review processes
Web of Science: Selected for its rigorous indexing criteria and citation metrics
Scopus: Valued for its comprehensive abstract and citation database
Google Scholar: Utilized as a supplementary source to capture grey literature and emerging scholarship

We formulated specific search queries using Boolean operators and controlled vocabulary to optimize recall and precision. The temporal parameters were deliberately set from 2017 to 2024 to capture contemporary developments in the field while ensuring sufficient longitudinal perspective. The search terms used are meticulously detailed in Table 2, refined through iterative pilot searches to achieve optimal sensitivity and specificity.

2.2 Inclusion and exclusion criteria

To ensure methodological integrity, relevance, and quality of our corpus, we developed comprehensive selection criteria through collaborative deliberation among research team members. We then systematically applied these criteria throughout our screening procedure to mitigate selection bias and strengthen the validity of our synthesized findings.

2.3 Inclusion criteria

Published between 2017 and 2024, capturing recent advancements while providing sufficient temporal scope to observe evolutionary trends in the field.

Table 2 Search terms used in the literature review

Search term	Keywords
Android Vulnerability Detection	"Android vulnerability detection", "Android security flaws", "Android exploit detection."
Machine Learning in Android Security	"Machine learning Android security", "ML Android vulnerability", "AI in Android security"
Deep Learning for Android Vulnerabilities	"Deep learning Android vulnerabilities", "DL Android vulnerability analysis", "Neural networks Android security"
Mobile Application Security	"Mobile application security", "Mobile app vulnerability detection", "Smartphone security threats"
Security in Mobile Operating Systems	"Security in mobile operating systems", "Mobile OS vulnerability assessment", "Smartphone OS security"

Note: Each search term was combined with relevant keywords to ensure a comprehensive literature search

English language publications to ensure accessibility and accurate interpretation by the research team
Studies specifically focused on ML or DL applications in Android vulnerability detection represent the core intersection of our research domains.
Peer-reviewed journal articles or conference proceedings to ensure scientific quality and scholarly validation
Studies presenting empirical evidence, methodological innovations, or substantial theoretical contributions to the field.

2.4 Exclusion criteria

Non-English publications, due to translation resource constraints and potential interpretation inaccuracies
Studies not directly related to Android or security vulnerabilities, ensuring domain specificity
Publications that did not employ ML or DL methodologies as their primary analytical framework
Duplicated or incomplete records are prevented from redundancy and ensure data integrity.
Review papers without original contributions (though these were cataloged separately for background context)
Studies lacking methodological transparency or sufficient technical details for reproducibility
Publications with inadequate evaluation metrics or validation procedures
Studies that focus on malware detection mainly

These criteria were applied in a two-phase screening process—first to titles and abstracts, followed by full-text assessment—with disagreements resolved through consensus discussions among multiple researchers to minimize subjectivity.

2.5 Study selection process

A detailed PRISMA Flow Diagram (Fig. 2) visually represented the study selection process. Initially, 300 studies were identified through database searches. After removing duplicates, 222 studies remained. These were screened based on titles and abstracts, resulting in 127 studies for full-text evaluation. Ultimately, 85 studies met our inclusion criteria and were included in this review.

During the title-and-abstract screening stage, we discovered that some of the records returned by the string "vulnerability detection" concentrated on malware detection rather than on identifying software vulnerabilities. Although the two research areas occasionally overlap, they differ fundamentally in scope, threat model, and analytical methodology: vulnerability detection seeks to uncover latent flaws in seemingly benign code, whereas malware detection aims to distinguish malicious artifacts from benign ones [24–28]. As a result, studies primarily focused on malware classification, analysis, or signature generation were considered beyond the scope of our review and were manually excluded at this point.

3 Background and related literature reviews

3.1 Background

Android applications have become ubiquitous in modern society, yet they often present significant security vulnerabilities that can compromise user privacy and data integrity. A primary concern is insecure data storage practices, where applications store sensitive information without proper encryption, rendering it susceptible to unauthorized access if the device's security is compromised. Equally concerning is inadequate user input validation, which creates opportunities for malicious actors to execute code injection attacks, including SQL injection and cross-site scripting (XSS). These attacks potentially grant attackers control over application functionality or unauthorized access to sensitive data [22]. Communication vulnerabilities represent another critical security concern, as unencrypted or improperly secured transmission channels can expose user data

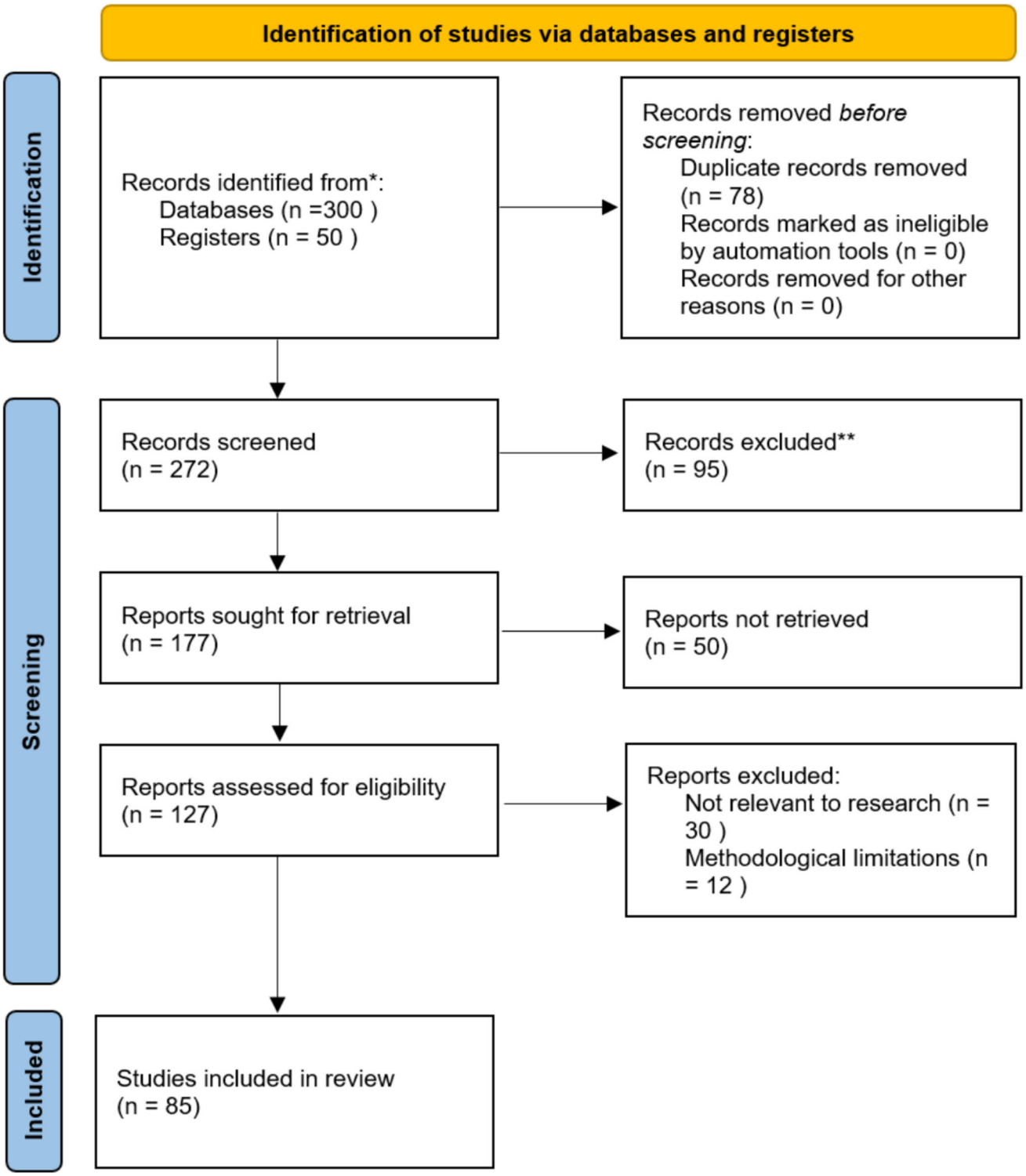

Fig. 2 Study selection process using PRISMA

to man-in-the-middle attacks. In these scenarios, attackers intercept data transmitted between the application and server, compromising confidentiality and potentially modifying information in transit.

Using outdated libraries and components introduces additional security risks, as these elements frequently contain known vulnerabilities that attackers can exploit to gain control of applications or devices. Furthermore, privilege escalation vulnerabilities arise when applications request excessive permissions beyond their functional requirements, creating potential vectors for attackers to access sensitive information or misuse device features and capabilities [29]. Application repackaging and code manipulation techniques have emerged as sophisticated attack vectors, where malicious actors modify legitimate applications and distribute them to unsuspecting users through unofficial channels. Additionally, insufficient authentication mechanisms frequently lead to account compromise or identity impersonation attacks, fundamentally undermining the security architecture of Android applications.

To address these vulnerabilities effectively [30], developers must implement comprehensive security measures, including secure coding practices, regular and rigorous security assessments, and timely updates to application components and libraries. Concurrently, end users must exercise vigilance when granting application permissions and refrain from installing applications from unverified sources to minimize exposure to potentially malicious software. The literature [31] outlines several approaches to mitigating vulnerabilities in Android applications. Secure coding practices are a foundational strategy during application development, including proper input validation through established APIs and adherence to coding standards to minimize security weaknesses. Implementing these practices significantly lowers the likelihood of introducing vulnerabilities during development.

Automated security testing tools provide another effective approach to securing Android applications. These sophisticated tools analyze application code to identify vulnerabilities related to data storage, communication protocols, and authentication mechanisms [32]. Integrating these tools into the development lifecycle enables early identification and remediation of security flaws, resulting in more robust and secure applications.

Research emphasizes the critical importance of continuous monitoring and timely updates [33]. Post-deployment surveillance of Android applications is essential for identifying emerging threats and vulnerabilities. Regular updates tackle known vulnerabilities and fortify applications against evolving attack methods. Moreover, academic studies support the adoption of specialized security frameworks and libraries [34] that enable the implementation of strong security features and reduce common vulnerabilities in Android application development.

Recent advancements in ML have introduced innovative approaches to vulnerability detection in Android applications. ML-based solutions analyze application source code and binaries to identify security weaknesses using algorithms trained on datasets containing known vulnerabilities. These models can recognize patterns and characteristics of insecure code, effectively identifying vulnerabilities, including input validation issues, insecure data storage practices, and authentication weaknesses.

The primary advantages of ML-based approaches lie in their scalability and adaptability to emerging threats. These systems continuously refine their detection capabilities through ongoing learning, improving accuracy. However, significant challenges exist in implementing ML-based security solutions, including the requirement for comprehensive labeled datasets for model training—a process that can be resource-intensive and costly. Additionally, the effectiveness of these approaches is highly dependent on the quality and comprehensiveness of training data, feature selection methodologies, and the appropriateness of the selected ML algorithms for security analysis applications.

3.2 Terms and definitions

3.2.1 Artificial Intelligence (AI)

AI is a field within computer science that enables computers to simulate intelligent behaviors typically associated with human intelligence, including reasoning, problem-solving, and decision-making [35].

3.2.2 Machine Learning (ML)

ML is a subfield of AI that equips systems with the capability to autonomously learn and enhance their performance based on experience without requiring explicit programming. This learning process begins by examining data (such as examples, instructions, or experiences) to identify patterns, thereby facilitating improved decision-making in future tasks. The ultimate objective is to enable automated learning with minimal human intervention [36].

3.2.3 Supervised machine learning algorithms

Supervised algorithms utilize a known training dataset to develop a predictive model. By analyzing the provided inputs and comparing the model's predictions to known outputs, these algorithms iteratively refine themselves, reducing errors and improving their predictive accuracy on new, unseen data [36].

3.2.4 Unsupervised machine learning algorithms

Unlike supervised learning, unsupervised algorithms seek to uncover hidden structures or relationships within unlabeled datasets. Instead of relying on predefined outputs, these algorithms explore and interpret the data, identifying inherent patterns without explicit guidance or labeled examples [36].

3.2.5 Semi-supervised machine learning algorithms

Positioned between supervised and unsupervised methods, semi-supervised algorithms leverage both labeled and unlabeled data during training. Typically, a relatively small set of labeled data is combined with a larger amount of unlabeled data, significantly enhancing the accuracy and efficiency of the learning process [36].

3.2.6 Reinforcement Learning (RL)

RL algorithms enable systems to interact dynamically with their environments, learning from actions through trial and error. Unlike other machine learning methods that depend on explicit instructions, RL algorithms independently discover optimal behaviors by aiming to maximize rewards or minimize penalties within specific contexts, an approach inspired by behaviorist psychology [36].

3.2.7 Deep learning

Deep learning is an advanced subset of machine learning, inspired by the neural networks of the human brain, that focuses on processing and extracting complex patterns from large, unstructured, or unlabeled datasets. This approach facilitates decision-making and predictive modeling by autonomously identifying deep structural insights from data [37].

3.2.8 Explainable Artificial Intelligence (XAI)

XAI is a concept emphasizing transparency and comprehensibility in AI systems. It involves creating AI models with clear and understandable decision-making logic and processes. XAI aims to mitigate the opacity of traditional "black-box" models, thereby promoting the ethical use of AI by enhancing trust, reducing bias, and ensuring accountability in automated decisions [38].

3.3 Related literature reviews

Before presenting the detailed analysis of current studies, it is essential to acknowledge previous literature reviews that have informed the understanding of this research domain. Thus, this subsection provides a summary of relevant review works that establish the foundational context for our study.

Garg et al. [39] researched Android security evaluations, examining the evolving trends and patterns in different analysis approaches, techniques, code representation tools, and frameworks. This comprehensive research, covering the period from 2013 to 2020 and drawing insights from around 200 studies, highlighted privacy breaches, cryptographic concerns, app duplication, permission misappropriation, code validation, malware identification, test case formulation, and energy consumption. The study explored static analysis techniques, focusing on sensitivity analysis, data structures, and code representations, based on a literature review. Additionally, dynamic analysis techniques were examined at kernel, application, and emulator levels using taint analysis and anomaly-based approaches. The review revealed an amount of research dedicated to uncovering vulnerabilities and leaks in Android systems. Furthermore, this comprehensive inquiry examined numerous Android assessment methodologies, with a focus on call graphs and control flow graphs as the primary data structures used. However, it is important to note that studies related to vulnerability mitigation were not addressed in this discussion.

The analysis presented in Ref. [40] encompassed an assessment of 124 distinct research studies spanning the years 2011–2015, with the overarching goal of delineating static analysis mechanisms deployed within the domain of Android applications. The investigation effectively identified the prevalent use of static analysis techniques in many research endeavors, predominantly those concerned with privacy and security matters. Among these techniques, taint analysis emerged as the most widely employed approach. In this context, it was established that prominent tools and formats for conducting such analyses included Soot, a comprehensive framework facilitating the analysis, instrumentation, optimization, transformation, and visualization of Java and Android applications [41], alongside Jimple, an intermediate representation conducive to the streamlined analysis and transformation of Java bytecode [42]. Notably, certain studies also showed a tendency toward path-sensitivity considerations. After a careful examination, the review identified leaks and vulnerabilities as the primary concerns addressed by the combined research efforts. Additionally, the investigation revealed a range of further focal points, including issues such as permission misuse, energy consumption, detection of app clones, generation of test cases, code verification, and implementation of cryptographic techniques.

In their studies, Senanayake et al. [43] reviewed 118 studies between the years 2016 and 2022 related to Android vulnerability detection using either conventional or ML/DL-based models. They discussed various methods for detecting vulnerabilities, including static, dynamic, and hybrid analyses. The authors also discussed the use of ML and DL

methods for detecting vulnerabilities. The advantages of using ML-based methods include automatic early detection of security issues and vulnerabilities. However, the performance of ML-based methods can be affected by the quality and quantity of the training data, the choice of features, and the selection of the ML algorithm. The authors also discuss the importance of analyzing the source code of Android applications to detect vulnerabilities. Study selection, data extraction, and synthesis were conducted to identify studies aiming to answer the formulated research questions. The results were cross-validated by performing a peer-verification process with all the authors.

Sharma et al. [44] provide a comprehensive and meticulous exposition of the confluence between source code analysis and machine learning. A meticulously curated assemblage of 479 principal research studies, spanning the period from 2011 to 2021 (with partial extension into 2022), has been scrutinized, encompassing a diverse spectrum of 12 distinct software engineering domains. The culmination of this endeavor is the presentation of a synthesized compendium of the chosen investigations, methodically categorized and subcategorized, further elucidated through the detailed description of the intricate steps involved.

In addition to this review, the survey aims to provide various valuable resources, including datasets and tools, thereby creating a helpful platform for future research efforts. The survey concludes by highlighting the challenges and opportunities in this field. It encourages practitioners and researchers to contribute methods, tools, and techniques. The ultimate goal is to create an environment that seamlessly integrates machine learning techniques into software engineering applications in a flexible manner.

Though the existing reviews provide in-depth details of the related studies, the review in Ref. [40] did not cover the recent works conducted in this area. The review in Ref. [43] thoroughly reviewed the studies on Android-specific vulnerability detection/prevention using conventional and ML methods performed in source code analysis. However, it seems they missed some notable research in this field, as well as the new techniques and approaches developed through the analysis period using ML techniques. Sharma et al. [44] provide a detailed survey of source code analysis with machine learning mechanisms without focusing on particular programming languages in the time range between 2011 and 2021. Therefore, a comprehensive review of recent studies on Android vulnerability detection using ML methods and prevention is necessary. In this work, we summarize the currently available datasets along with their strengths and weaknesses, illustrating the existing status from an Android-specific perspective. Additionally, this study presents a comprehensive overview of the latest trends and challenges related to the topic. Table 3 summarizes and compares the related reviews with this work's contribution.

4 Vulnerability analysis

Vulnerability analysis encompasses a broad spectrum of techniques for identifying, understanding, and mitigating security weaknesses in software systems. Traditional static and dynamic analysis methods have been widely used for this purpose; however, the increasing complexity of software architectures and the growing sophistication of cyber threats necessitate more advanced and scalable solutions.

Recent advancements in artificial intelligence, particularly in ML and DL, have significantly contributed to the automation and efficiency of vulnerability detection. Additionally, integrating Large Language Models (LLMs) and Explainable AI (XAI) has opened new possibilities for improving detection accuracy, interpretability, and developer adoption. This section explores various approaches to vulnerability analysis, including ML-based detection methods, taint analysis, the role of LLMs, cross-language detection techniques, and the application of XAI to enhance transparency in security assessments.

4.1 Vulnerability detection with machine learning methods

In vulnerability analysis, machine learning techniques play a role by helping us identify patterns that indicate vulnerabilities. A crucial aspect is carefully constructing datasets [21] that encompass a range of vulnerabilities. We also need to extract and select features from the underlying code or binaries to improve the accuracy and relevance of vulnerability detection. Choosing the anomaly detection or pattern recognition algorithms is equally important in ensuring the model's effectiveness. Additionally, domain expertise is vital to understanding vulnerabilities in their context and defining learning objectives. Combining all these elements enables us to develop machine learning techniques [80] that effectively identify vulnerabilities, thereby strengthening cybersecurity efforts.

The above figure (Fig. 3) presents an overview of a typical pipeline involved in code representation. Many repositories undergo processing to train a model during the training phase, which is subsequently employed in the inference phase.

The source code is prepared and transformed into a model, such as an Abstract Syntax Tree (AST) or a sequence of tokens. This model is then passed to a feature extractor, which extracts important features such as AST paths and tree-based embeddings. Subsequently, a ML model is trained using the extracted features. The model generates a numerical representation, i.e., a vector, which can be further utilized for specific software engineering applications, including defect prediction, vulnerability detection, and code smell detection.

Table 3 Summary of Related reviews (including this SLR) and key research papers

Literature review	The focus of the review	Period	ML-based vulnerability detection	Challenges/Limitations	New research areas	Android dataset availability[39]
Garg et al. [39]	Android security assessments and Application analysis methods	2013–2020	✓	×	×	×
	Paper	ML/DL Technique used	Classifier	Dataset/Apps/Tools	Features	Accuracy
	AdDroid [45]	SL	Adaboost with DT	Benign = 510 Malicious = 910	Permissions, API calls, functions, code features	99.11%
	Appice et al.[46]	UL	k-means + +	Benign = 1,23,453 Malicious = 5560	Permissions, API calls, Network Addresses	96.60%
	ATMPA [47]	DL	Rd, SVM, CNNs	Benign = 1000, Malware samples = 10,867	N/A	88.70%
	DroidDetector [48]	DL	DBN	Benign = 20,000 Malicious = 1760	Permissions, Sensitive API, Dynamic Behaviors	96.76%
	EASEAndroid [49]	SSL	K-NN	1.3 million audit logs	N/A	Learns 2518 benign & malicious access patterns & generates 331 policy rules
	Fadadu et al.[50]	DL	RF-DT, XGBoost, NN, CNN, RNN	Benign = 10,000 Malicious = 10,000	API call sequences	91.79%
	Garg et al.[27]	SL	MLP, SVM, PART, RIDOR, Ensemble	Benign = 60,000 Malicious = 24,000	Permissions, API Calls, Libraries, Broadcast receivers	98.37%
	Han et al. [51]	SL	SVM	Benign = 28,489 Malicious = 30,113	API calls	99.75%
Garg et al. [30]	Chandra et al. [52]	SSL	LLGC	3,00,000 apks	Permissions, API calls	93.78%
	Mantoo et al.[53]	SL	k-NN, LR	Benign = 300 Malicious = 300	Static-SMS, Phone, Storage, Contacts, Location, Camera, etc	97.50%
	Martín et al.[54]	SL	LR, RF	82,866 suspicious apps, 259,608 malware	Clock get time, Mprotect, Epoll_pwait, Receive from, Send to, Ioctl, Write, Getuid Intrinsic-Size Antivirus	F1-score = 0.84
	Nguyen-Vu et al. [55]	DL	DNN	Benign = 1901 Malicious = 1600	API calls	91.70%

Table 3 (continued)

Literature review	The focus of the review	Period	ML-based vulnerability detection	Challenges/Limitations	New research areas	Android dataset availability[39]
	Sharmeen et al.[56]	DL, SSL	DNN	Benign = 10,000 Malicious = 8560	Permissions, API calls	99.02%
Senanayake et al. [43]	Android source code vulnerability detection and prevention	2016–2022	✓	×	✓	✓*
	Paper	ML/DL Technique used	Classifier	Dataset/Tools/apps	Features	Accuracy
	Pang et al. [57]	DL	DNN algorithm	Features based on mining source code using N-gram, APKs from F-Droid	Source Code	92.87
	Wu et al. [58]	DL	CNN, LSTM, CNN-LSTM	Function call sequences (9,872), VDiscover tool [33]	Function call collection	83.6
	Zhuo et al. [59]	SL	AB and DT	From major sources	Permissions, Intents	77
	Garg et al. [27]	SL	MLP, SVM, PART, RIDOR	From major sources	System calls	98.27
	Bilgin et al. [60]	SL	MLP and a customized model	ML-based analysis to differentiate vulnerable and non-vulnerable source code	Source Code	70,1
	Gupta et al. [61]	ML	J48 and JRip	Efficient human-readable vulnerability detection rules, tenfold cross-validation	functions	96
	Kim et al., [62]	DL	CNN	PE data extraction, image generation, ML algorithms judging maliciousness	PE	98,77
Sharma et al. [44]	Machine learning methods for source code analysis	2011–2021	✓	✓	✓*	×
	Paper	ML/DL Technique used	Classifier	Dataset/Tools/apps	Features	Accuracy
	Narayanan et al.[17]	SL	MKL-SVM	60,561 apps (malicious and benign)	Tokenized part of the source code	97.30%
	Shiqi et al., [63]	DL	Deep Belief Networks	5560 malicious, 123,453 benign apps	Image texture and API calls	94%

Table 3 (continued)

Literature review	The focus of the review	Period	ML-based vulnerability detection	Challenges/Limitations	New research areas	Android dataset availability[39]
	Alatwi et al., [64]	SL	SVM	5063 Android apps, 1000 benign rest are malware	Static and dynamic behavior	89.47%
	Zang et al., [65]	SL	DT RF	1406 apps with 4416 different versions	20 static code metrics	91.38%
	Ma et al., [66]	DL	DNN LSTM	10 k benign and vulnerable apps	API calls, APIs freqs	98.98%
	Gupta et al.. [61]	ML	J48 JRip	2 Android projects	Static code metrics	96%
	Wei et al., [67]	SL	DT and Naïve Bayesian	219 malicious apps	System functions	92%
This work	Android vulnerability detection using ML/DL techniques	2017–2024	✓	✓	✓	✓
	Paper	ML/DL Technique used	Classifier	Dataset/Tools/apps	Features	Accuracy
	Zhang et al. [65]	SL	RF	Fdroid [68]	Static code metrics	91.38%
	Rahman et al. [69]	SL	r-SVM	Fdroid [68]	Static code metrics	83%
	Piskachev et al., [70]	SL	SVM	235 methods from 10 popular Java frameworks	Functions	82.60%
	Zhuo et al. [59]	SL	AB and DT	300 APKs from major sources	Dynamic and static analysis	84%
	Pang et al. [57]	DL	DNN	Fdroid [68]	Static analysis/source code with code metrics	92.87%
	Gupta et al. [61]	ML	J48 JRip	2 Android projects	Functions	96%
	Malik et al. [71]	ML	KNN	Four apps with known issues	System calls	85%
	Senanayake et al. [72]	AutoML	XGBoost	46,333 vulnerable code samples	Code chunks	94%
	Defendroid [73]	Neural Networks	Vanilla	Lvdandro [72]	Code chunks	96%
	Llbezpeky [74]	DL	LLM (GPT-4)	Ghera benchmark [75]	Functions	91.67%
	Acved [76]	ML	XGBoost	Lvdandro [72]	Code chunks	95%
	Yim et al. [77]	ML	RF	Android Open Source Project [78]	Code changes	98%
	Fedrevan [79]	Federated Neural Networks	RF	Lvdandro [72]	Code chunks	96%

✓*, Partially

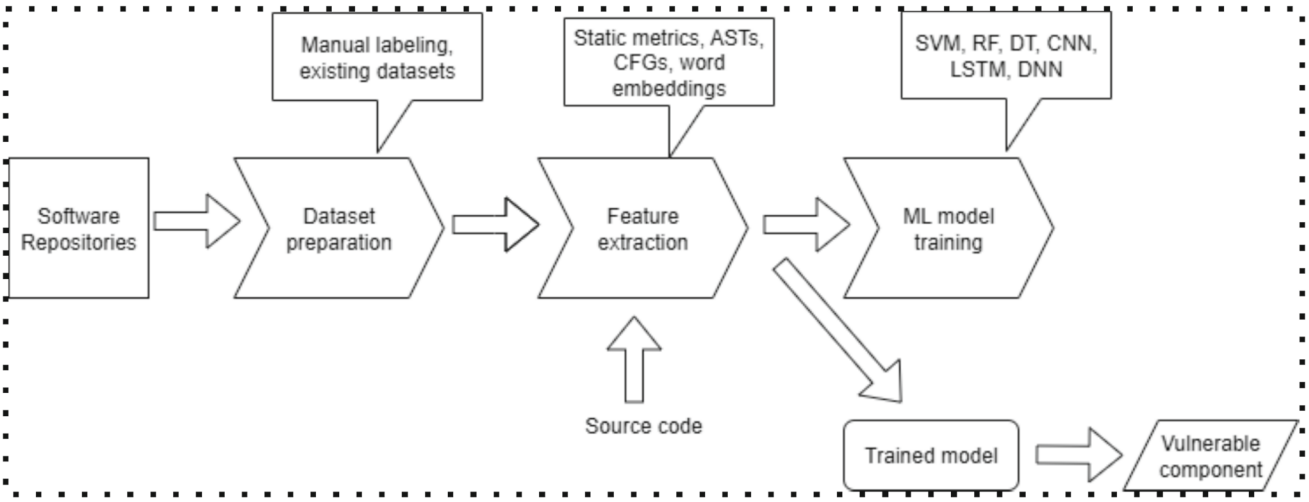

Fig. 3 Vulnerability detection pipeline [44]

Building upon these fundamental machine learning approaches for code analysis, recent advancements in vulnerability detection for Android applications have demonstrated significant promise across multiple methodologies. Studies employing traditional machine learning algorithms have shown considerable efficacy. For instance, Zhang et al. [65] highlighted the superior performance of Random Forest across various risk levels in IoT applications, while Rahman et al. [69] achieved a precision of 0.83 using radial-based support vector machines (r-SVM) for security risk prediction. Several specialized tools and frameworks have emerged to address specific vulnerability detection challenges, including SWAN [70], which generalizes detection to Java applications with 0.826 precision; VuRLE [81], which focuses on automated vulnerability repair through template learning; and Vulvet [82], which achieved an impressive 95.23% precision across 3700 applications using multi-tier static analysis. Research on neural network applications has also progressed, as evidenced by studies that utilize deep neural networks for predicting vulnerable components [53] and specialized algorithms for detecting vulnerability in the Intent mechanism [59]. However, the latter achieved a moderate accuracy of 77%. Dataset innovations represent another crucial advancement, the studies [73, 76, 79, 83] with the LVDAndro dataset [72] providing over 20 million annotated code samples and enabling 94% accuracy in vulnerability classification despite limitations inherited from its underlying scanning tools, MobSF [84] and Qark [85]. Despite these advances, recurring limitations persist across studies, including insufficient dataset size and diversity, challenges in accurately labeling data for supervised learning approaches, and difficulties in detecting Android-specific vulnerabilities within varied application contexts, highlighting areas that require further investigation to develop comprehensive security solutions for Android applications. They only protect code during development as plugins, which still leaves millions of currently developed and used applications in the app stores needing additional approaches.

Studies such as [86–89], and [90] explore techniques for detecting security vulnerabilities in Android applications. As evaluated in [86], traditional static and dynamic analysis methods exhibit significant limitations, as they fail to detect many known vulnerabilities. Large Language Models (LLMs) like GPT-4, assessed in [87], outperform conventional analyzers by identifying four times more vulnerabilities with lower false positives. Similarly, [88] compares multiple LLMs in detecting Android vulnerabilities, revealing considerable discrepancies in performance. The work in [89] highlights the effectiveness of a hybrid approach that combines static and dynamic analysis, demonstrating superior detection capabilities compared to traditional tools. Finally, [90] presents VulsTotal, a unified platform for benchmarking static application security testing (SAST) tools, addressing the inconsistency and incompleteness in prior vulnerability detection frameworks. Predicting future security vulnerabilities is critical for proactive defense strategies. Study [91] employs time-series and deep learning techniques to predict vulnerabilities in Android OS versions, utilizing data from the National Vulnerability Database (NVD). The results demonstrate that LSTM models can achieve competitive accuracy, though classical models like ARIMA still outperform in some cases. The findings suggest that hybrid approaches yield better long-term predictions.

Study in this reference [77] explores a preventive approach to vulnerability detection. In this approach, a security review bot automatically flags potentially insecure code changes before they are submitted. This classifier-based framework achieves a high detection rate, offering a promising method

for integrating security checks into the software development lifecycle. The study suggests that integrating machine learning-based analysis into code review pipelines can significantly enhance secure coding practices.

While many efforts focus on detection, the study [92] investigates the automated repair of vulnerabilities using pretrained models. It highlights that transfer learning from bug fixing can significantly improve prediction accuracy, emphasizing the potential for AI-driven vulnerability patching in reducing manual debugging efforts. However, challenges remain in systematically evaluating the effectiveness and limitations of these models in real-world applications.

Recent studies in ML-based vulnerability detection have explicitly addressed hyperparameter tuning to optimize model performance. For example, researchers varied neural network depth and perceptron counts, applying systematic grid search procedures to identify optimal configurations and validate parameter suitability [73, 79]. These efforts were often complemented by magnitude-based pruning, where insignificant weights were gradually removed to increase throughput while preserving accuracy; pruning levels were carefully tuned using the TensorFlow optimization toolbox to balance sparsity against predictive precision [73, 79]. Beyond deep learning models, parameter tuning has also been explored in selective security testing frameworks, where configuration parameters—such as maximum test duration and function-level targeting—were dynamically adjusted based on source code changes and project-specific vulnerability statistics [77]. Collectively, these approaches highlight the diversity of hyperparameter optimization strategies across the literature, ranging from grid search and pruning in neural networks to adaptive configuration in security testing pipelines, underscoring the importance of tuning for both performance and efficiency.

While vulnerability detection and malware detection are frequently confused as being synonymous, they serve distinct purposes and operate under different principles. Vulnerability detection identifies weaknesses within systems that could be exploited, whereas malware detection identifies harmful software or behaviors. Recognizing these differences is essential to understanding the contribution of this study, which seeks to enhance the field of vulnerability detection by addressing critical gaps in real-world dataset accessibility and reproducibility.

Several significant studies [93–96] have examined challenges associated with Android malware detection, offering valuable insights that could contribute to improving vulnerability detection techniques. A persistent issue in this domain is the dependence on outdated or small-scale datasets, such as MalGenome [97] and Drebin [98], which do not adequately reflect contemporary threat landscapes. This shortcoming limits the ability to train effective machine learning models, often leading to overfitting. Integrating static and dynamic analysis has been proposed to mitigate this—static features are prone to obfuscation. In contrast, dynamic features are more challenging to collect despite being more resistant to such techniques. Another key challenge is the variability of in-app behavior across different devices, which complicates detection accuracy, particularly in large-scale frameworks such as federated learning. These obstacles underscore the necessity of employing up-to-date datasets, leveraging adaptive modeling techniques, and addressing the complexities of evolving security threats. Insights gained from these studies could also inform future advancements, making vulnerability detection methods more robust and applicable in real-world scenarios. In a related study [99], researchers introduced a supervised learning approach for detecting Android malware, utilizing a labeled dataset comprising over 18,000 samples across five categories. Their methodology, validated against multiple benchmark datasets, yielded competitive results compared to state-of-the-art techniques, offering valuable insights into malware classification and detection enhancements. In addition, Wajahat et al. [100] directly addressed the challenge of dataset imbalance by proposing a preprocessing framework that removes duplicates, balances class distributions, and applies feature selection techniques such as Gain Ratio and Chi-square tests. Their evaluation on the Drebin and Tuandromd datasets demonstrated that classifiers like Random Forest and SVM can achieve accuracy rates approaching 99% when supported by these data-centric strategies. While this study focuses on Android malware detection, its emphasis on class rebalancing and feature optimization provides a transferable perspective for enhancing the robustness of vulnerability detection models, which are often limited by imbalanced datasets.

4.2 Taint analysis with machine learning

In analyzing taint, source and sink functions are vital in monitoring how information moves within a program. Taint analysis is a security technique to detect how data, known as "taint," spreads throughout the code. Interacting with different functions and recognizing the significance of sources and sinks is crucial for identifying security weaknesses and safeguarding sensitive data [101].

Sources refer to functions or methods in the code where sensitive data originates. These can include user inputs such as passwords, credit card numbers, or personally identifiable information (PII). A source labels data as "tainted," indicating that it should be tracked throughout the program's execution. Identifying sources is crucial, as they establish entry points for potential data breaches or security vulnerabilities. By tracing data to its sources, security analysts can comprehend how confidential information enters the application and monitor its movement. Sinks are functions or methods that interact with the outside world, such as network

communication, file I/O, or database queries. These functions are potential destinations for tainted data to leak outside the application's intended scope. Security analysts can identify potential data leaks or vulnerabilities that expose sensitive information to unauthorized users or attackers by monitoring tainted data as it reaches these sinks. Properly identifying and securing sinks prevents data exfiltration and protects user privacy.

Source and sink functions are crucial for data propagation, as they help taint analysis algorithms track how tainted information moves through code. Developers can implement effective security measures by understanding how sensitive data enters an application and where it might be leaked [102]. Taint analysis also ensures that sensitive information is handled securely during application execution, minimizing the risks of data breaches and privacy violations. Additionally, integrating taint analysis with machine learning techniques shows promise for detecting vulnerabilities in source code [103].

Over the years, various tools have been developed to conduct taint analysis of Android applications. However, comparing these tools is challenging due to different evaluation targets and methodologies. Some studies fail to describe the datasets used for evaluation, while others rely on benchmarks but only partially cover them. Furthermore, discrepancies exist in the types of data leaks considered, and many tools lack a ground truth for accurate validation. These inconsistencies make it difficult to compare the effectiveness of different tools.

To address these challenges, ReproDroid [104] was introduced as a framework for benchmarking Android taint analysis tools in a reproducible manner. It enables researchers to infer the ground truth for data leaks, apply tools automatically to benchmarks, and systematically evaluate their results. Using ReproDroid, six major taint analysis tools—Amandroid [105], DIALDroid [106], DidFail [107], and FlowDroid [108]—were evaluated under uniform conditions. While the overall results were positive, the study revealed that four of these tools did not fully meet their claimed capabilities in terms of features and accuracy. Additionally, the study contributed an improved version of the DroidBench test suite to enhance unbiased benchmarking in future research.

Among the most prominent taint analysis tools, SuSi and FlowDroid adopt distinct approaches. SuSi [109] leverages machine learning techniques to automatically classify Android source and sink functions. It extracts semantic and syntactic features from Android source code, training a model that can categorize unknown methods as sources, sinks, or neither. While SuSi achieves high precision and recall, discrepancies exist between testing and validation precision scores, leading to misclassifications, particularly in differentiating source and sink classes. Sas et al. [110] reported that although SuSi achieved 86.2% accuracy with an average recall of 85%, the model exhibited a high false positive rate for source and sink classifications, suggesting the need for improved semantic analysis.

FlowDroid [108], on the other hand, employs a static taint analysis approach that precisely tracks data flows within applications. It detects a high fraction of data leaks while maintaining a low false positive rate, outperforming commercial tools such as IBM AppScan Source and Fortify SCA. FlowDroid successfully identifies leaks in large-scale datasets, including those from Google Play apps and malware samples on VirusShare. However, like other static analysis tools, FlowDroid may struggle with scalability when handling complex, obfuscated applications and dynamic code execution.

When compared, SuSi provides an automated method for classifying sources and sinks, making it highly efficient for identifying taint entry and exit points. In contrast, FlowDroid excels in precise taint propagation analysis. However, Sas et al. [110] highlighted that SuSi's classification accuracy can be compromised by misidentifications. In contrast, FlowDroid's reliance on static analysis may limit its performance in real-world scenarios where dynamic execution plays a crucial role. Combining these approaches—leveraging SuSi's classification for source-sink identification and FlowDroid's propagation analysis—could lead to a more comprehensive taint analysis framework. Future research could integrate machine learning-based classification with advanced static and dynamic analysis techniques to enhance accuracy and scalability.

Furthermore, Ausera [111] introduced a comprehensive vulnerability taxonomy to address the shortcomings of functionality, limited vulnerability coverage, and the restricted availability of benchmark datasets. This taxonomy encompasses 50 different vulnerability types to enhance coverage and reduce false negatives and false positives. Additionally, it provides a robust benchmark dataset that may be utilized in future machine learning methods for further improvement.

4.3 Leveraging Large Language Models (LLMs)

The emergence of LLMs with billions of parameters has significantly expanded the capabilities of artificial intelligence in various domains, including software security. These models are believed to capture intricate patterns related to syntax, semantics, and the structural properties of human language [112]. Interestingly, their proficiency extends to programming languages, as these often exhibit well-defined grammar and semantic structures [113]. Recent research has explored the potential of LLMs in identifying security vulnerabilities in code, yielding promising but varied results [87]. For instance, studies evaluating GPT-based models for vulnerability detection have shown mixed effectiveness—Cheskov et al. [114] reported that their approach did not perform well.

However, their methodology was relatively simplistic. On the other hand, more advanced techniques, such as extended prompting strategies and hybrid approaches that integrate LLM-driven analysis with complementary security mechanisms, have demonstrated improved accuracy in detecting CWEs within code [115, 116].

Additionally, some studies have taken a more comprehensive approach, investigating multiple facets of LLM-based security analysis. These in-depth evaluations suggest that LLMs can offer valuable insights into software vulnerabilities when appropriately fine-tuned and guided, making them a promising avenue for future security research and practical applications [117].

4.4 Cross-language vulnerability detection approaches

This review has explored methodologies and patterns relevant to identifying vulnerabilities in Android applications. However, considering the need for up-to-date insights, it is wise to examine the need for up-to-date insight fields. This investigation can be useful for evaluating Android applications and providing insights for research directions. To the best of our knowledge, we would like to share a study conducted on various languages, though they may not directly align with our specific interests.

In a study, a DL-based framework was presented that offers user-friendly Python scripts for developing and evaluating vulnerability detection mechanisms. This framework undertakes a dual-pronged evaluation using distinct datasets to assess its efficacy and the proficiency of the embedded neural networks [67]. The initial dataset, SARD, encompasses synthetic vulnerabilities, while the second dataset is meticulously curated, comprising over 1300 instances of vulnerable files and functions extracted from real-world sources. Leveraging the framework, it conducts three empirical case studies utilizing the Deep Neural Network (DNN), Bidirectional Long Short-Term Memory (Bi-LSTM) network, and Text Convolutional Neural Network (text-CNN). The outcomes of these experiments indicate a uniform performance trend across the synthetic SARD dataset, implying that the structural configurations of the networks play a subordinate role in influencing performance within the context of synthetic vulnerability samples. However, the performance diverges when confronted with an authentic, real-world dataset. Notably, the contextually aware network models, namely the text-CNN and Bi-LSTM networks, exhibit enhanced capabilities in detecting real-world vulnerabilities. This highlights the effectiveness of context-aware network architectures in identifying genuine vulnerabilities in real-world scenarios. However, the proposed real-world vulnerability dataset in this study is still in its early stages and could be improved; further effort is required. They acknowledge that they will focus on collecting vulnerable components at the binary level and that it is essential to verify vulnerabilities after developers claim to patch them and assess the effectiveness of their approach in this manner.

Most ML approaches aim to identify parts in source code, whether a line or a function. We believe that a code segment vulnerability depends on its context, specifically in terms of control flow and data flow considerations. Therefore, we argue that context awareness is crucial for vulnerability detection. In a study [118], the authors introduced a framework designed to generate code embeddings specifically for detecting vulnerabilities at the function level. The proposed framework focuses on capturing source-sink data flows without getting caught up in validation complexities, which relies on a model that helps uncover meaning hidden within the source code. By transforming code sequences into rich semantic association vectors, it feeds these vectors into Bi LSTM layer to better understand long-term dependencies. As a result, their presented network has become skilled at identifying code segments. To ensure the compatibility of the extracted code embeddings with machine learning classifiers, a max pooling layer transforms the acquired embeddings into vector representations. This approach avoids code analysis by taking the source code as input. Empirical validation confirms that the code embeddings generated by their method serve as feature sets for detecting vulnerabilities. Interestingly, when applied to our real-world software projects, using these embeddings in conjunction with the forest classifier yields better performance than four other baseline systems.

When locating vulnerable codes, the role of Natural Language Processing (NLP) is crucial in enhancing the detection of sections in source code repositories. By utilizing NLP techniques, we can bridge the gap between human explanations of vulnerabilities and the complex syntax of source code. This enables the development of models that can comprehend and analyze code snippets, much like humans understand language. NLP helps create algorithms that not only identify vulnerabilities but also understand nuances, leading to a more comprehensive understanding of potential security threats. Consequently, integrating NLP into vulnerability detection processes significantly enhances the accuracy and efficiency of identifying and mitigating security risks within software systems. The researchers [119] propose using a learning approach to identify vulnerabilities by analyzing their LLVM IR representations. They draw inspiration from techniques commonly used in natural language processing. First, they identify source codes that contain vulnerabilities. Then, they pinpoint the lines of code responsible for those vulnerabilities. This two-step approach effectively reduces the number of alarms when detecting lines. Through experimentation with world and synthetic code datasets obtained

from NVD and SARD, they achieved an impressive accuracy rate of approximately 98% in detecting vulnerabilities in source code.

This research [120] aims to address the issue of the "start" problem in machine learning. The main concern is acquiring features that can be generalized across projects. To strike a balance between having features and being able to generalize, we have developed a data-centric approach incorporating several innovative ideas. Primarily, the semantic intricacies of code are unveiled through serialized abstract syntax trees (ASTs), with tokens encoded via Continuous Bag-of-Words neural embeddings. Subsequently, these serialized ASTs undergo analysis via a sequential deep learning classifier, specifically a bidirectional LSTM network. This cognitive process culminates in generating a distinctive representation indicative of software vulnerability. Importantly, the neural representation acquired from pre-existing software projects is transposed onto new projects, facilitating early-stage vulnerability detection even without an extensive corpus of training labels. To assess the effectiveness of this vulnerability detection approach, they conducted a labeling process involving 457 instances of vulnerabilities. They curated a dataset containing over 30,000 vulnerable functions from six open-source projects. The empirical findings confirm that the model can accurately generate representations indicating vulnerabilities. It demonstrates adaptability across project scenarios. Compared to code metrics, the transfer learning-based representations effectively predict vulnerable functions within individual projects and across multiple diverse projects. Advancements in DL have sparked interest in applying DL methodologies to automated vulnerability detection. Several recent studies have shown results achieving up to 95% accuracy in vulnerability detection. In this research contribution, we investigate the performance of state-of-the-art DL-driven techniques in real-world vulnerability prediction.

Unexpectedly, this analysis [121] reveals a performance drop of over 50% when transitioning from controlled environments to real-world scenarios. A thorough investigation into the reasons behind this decline in performance highlights the challenges present in vulnerability prediction methods based on machine learning. These challenges arise from both the makeup of the training data, which suffers from issues such as data quality and imbalanced class distributions, and the selection of model architectures, which often prioritize token-based approaches. These methods often fail to identify the underlying causes of vulnerabilities, instead relying on superficial knowledge extracted from the dataset, such as specific variables or function names. Building on these insights, a more methodologically sound approach is advocated for acquiring data and designing model architectures that accurately capture the contexts of vulnerability prediction. By adopting this approach, the resulting tools demonstrate improvements compared to established baseline methodologies—achieving a precision increase of up to 33.57% and a recall boost of 128.38% compared to the successful model documented in the existing literature. In conclusion, the study examines the underlying issues in vulnerability prediction systems, with a focus on learning. It provides a roadmap for research efforts to fully utilize the capabilities of learning methods in predicting vulnerabilities.

While our focus remains primarily on Android applications, it is important to note that the underlying methodologies, particularly those based on semantic embeddings, control/data flow analysis, and transfer learning strategies, are not restricted to a single programming language or platform. In principle, these approaches can be adapted to other ecosystems, enabling broader applicability across diverse codebases. Future research should therefore place greater emphasis on cross-project and cross-language validation to ensure that models trained in one environment can generalize effectively to others.

4.5 Behavioral pattern analysis and compact model design

Behavioral pattern analysis refers to approaches that capture how software components or code segments behave over time—whether through runtime traces, control-flow patterns, or function-level interactions—rather than relying solely on static code tokens. Compact model design, on the other hand, focuses on reducing the size, complexity, and resource demands of ML models so that they remain deployable in practical environments such as CI/CD pipelines or mobile devices.

Recent contributions have emphasized that improving vulnerability detection requires moving beyond static token-level features toward behavioral pattern analysis. Interpretable dynamic-analysis pipelines for Android demonstrate how fine-grained runtime traces (e.g. system-call sequences) can be linked to human-readable rationales, thereby improving trust in model outputs [122]. Similarly, temporal behavior modeling approaches explicitly capture how execution patterns evolve, which enhances the robustness of classification in security-sensitive contexts [123]. At the source-code level, researchers have introduced frameworks that model abstract behavioral semantics and function interactions, allowing detectors to generalize across projects instead of memorizing surface-level features [124]. These studies collectively highlight that context-aware and behavioral perspectives substantially improve both accuracy and interpretability in ML-based vulnerability analysis.

Meanwhile, another key area of research focuses on making models more efficient for real-world applications. Surveys on model compression techniques—including quantization, pruning, and knowledge distillation—show that

compact neural networks can significantly cut down memory use and inference time while still delivering competitive accuracy [125]. Such techniques are particularly relevant for continuous integration pipelines and resource-constrained environments, where vulnerability detection models must operate under strict efficiency requirements. Finally, to integrate code-based vulnerability detectors into operational security practices, researchers have proposed threat modeling frameworks specifically tailored to ML-intensive systems. These frameworks emphasize the systematic identification of assets, attack surfaces, and controls during deployment [126]. Incorporating such perspectives bridges the gap between academic prototypes and real-world secure development workflows, strengthening both the reliability and the deployability of ML-based detection systems.

4.6 Explainable AI (XAI)

At the start of 2017, the Defense Advanced Research Projects Agency (DARPA) funded the "Explainable AI Program" [127]. The primary goal of this initiative is to create AI models that are both highly interpretable and capable of maintaining robust predictive performance. Furthermore, the program seeks to ensure that human users can easily understand, trust, and effectively manage upcoming generations of AI technologies. In 2020, the National Institute of Standards and Technology (NIST) complemented this effort by outlining four core principles for XAI—Explanation, Meaningful, Explanation Accuracy, and Knowledge Limits, as shown in Fig. 4. Together, these initiatives highlight the dual need for accuracy and transparency in AI-driven decision-making [128].

In the field of vulnerability detection, however, interpretability remains a crucial challenge. While ML-based methods have achieved significant accuracy, many models still fail to understand the semantic meaning of code. They cannot clearly explain why a segment is flagged as vulnerable. This undermines trust and hinders adoption in real-world settings. Recent efforts [73, 129–131] have explored model-agnostic methods such as feature attribution and visualization to improve interpretability, yet balancing accuracy with transparency continues to be an open problem. Future research must focus on context-aware explainability—embedding functional and structural code semantics into explanations—to make automated vulnerability detection both reliable and developer-friendly. Techniques such as LIME and SHAP exemplify how interpretability can be enhanced by revealing which features or code segments contribute most to predictions. Although they introduce computational overhead, these mechanisms provide actionable insights that build developer trust and support more effective mitigation. Given the high-stakes nature of Android vulnerability detection, combining predictive performance with explainability should be considered an essential requirement for future research.

4.7 Practical application: real-time deployment feasibility and tooling

Bridging research and real-world utility, one promising direction involves designing prototype web-based vulnerability analysis systems. Such systems could automatically decompile uploaded Android APKs, extract static code metrics and semantic code embeddings, and apply pre-trained ML models to provide both file-level binary vulnerability classification and an overall normalized "Application Risk Score." This type of architecture demonstrates how advanced ML methods can be transformed into lightweight, user-friendly interfaces suitable for integration into web services or continuous integration pipelines.

In parallel, real-time feasibility on mobile or edge devices requires attention to model compression [132] and inference optimization [133]. Recent studies on quantization-aware training and pruning have demonstrated that lightweight neural networks can achieve substantial reductions in memory footprint and latency while maintaining high accuracy, thereby enabling deployment on constrained Android environments [134]. In Android-specific security tasks, designing robust feature representations is equally important, as demonstrated in obfuscation-resilient approaches for malware detection [135] and recent transformer-based detection models [136]. Together, these findings suggest that quantization (e.g., 8-bit integer inference) and feature robustness strategies can make ML-driven vulnerability analysis both lightweight and resilient to adversarial conditions.

Beyond resource efficiency, practical deployment also depends on integration into modern development practices. DevSecOps pipelines are increasingly incorporating security checks directly into the build and release stages, where ML-based vulnerability detection can serve as an additional safeguard. Recent reviews highlight effective strategies for embedding static analysis security testing (SAST), dependency scanning analysis (SCA), and anomaly detection into CI/CD workflows [137]. Complementary work on ML in pipelines demonstrates how automated models can operate as pre-merge gates or nightly batch scanners, offering scalability without burdening developers [138]. Earlier studies on lightweight AI deployment also confirm that resource-constrained integration is feasible, even in edge environments [139]. Collectively, these insights suggest that ML-based vulnerability detection can be effectively integrated into both web-accessible testing services and automated CI/CD pipelines, thereby ensuring relevance in real-world secure software development lifecycles.

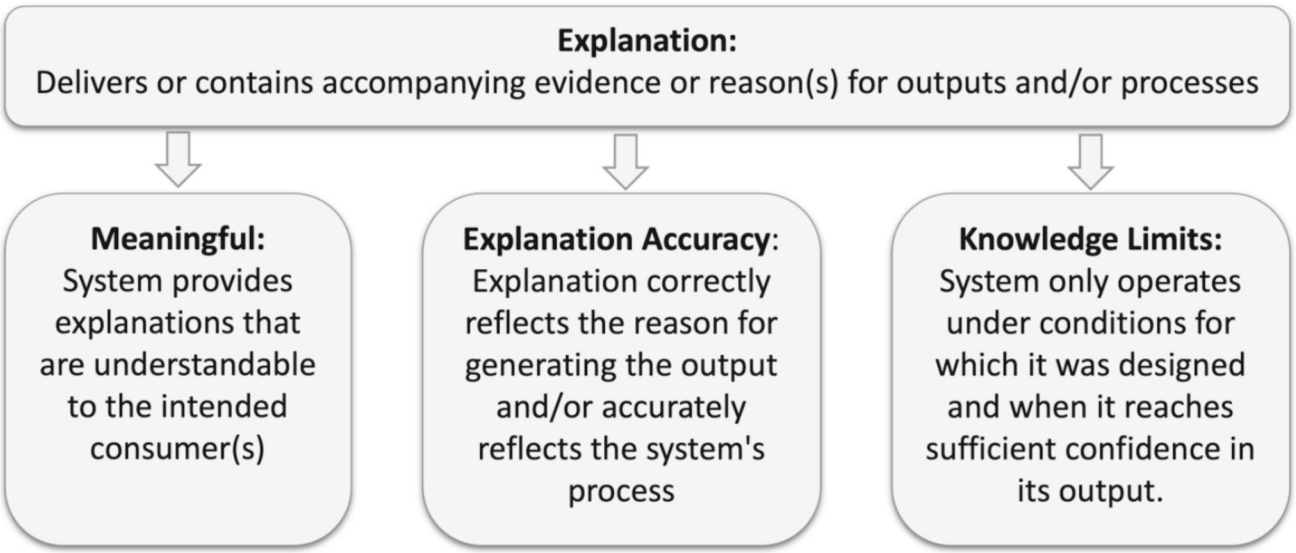

Fig. 4 Illustration of four principles of XAI by NIST [128]

5 Datasets

Data is essential in machine learning as the foundation for model development and training. They offer the raw material from which models extract patterns and relationships, allowing them to make informed predictions. The quality and diversity of a dataset significantly impact a model's ability to generalize its learning beyond the training data, thereby enhancing its overall performance and reliability.

Moreover, datasets are essential for evaluating model performance as benchmarks against which models are tested and compared. They facilitate the identification of biases and fairness issues inherent in the data, which is crucial for developing equitable and just algorithms. Additionally, datasets drive innovation by inspiring new solutions to complex problems and opening avenues for creative research.

Dataset selection and curation impact feature extraction and engineering, affecting a model's predictive accuracy. Specialized datasets tailored to specific domains imbue models with domain-specific knowledge, enhancing their contextual relevance and practical application.

The most recent review [145] regarding available datasets in the literature was conducted in 2018 and undertakes the identification and comprehensive assessment of 31 extant datasets on Android applications. Each dataset is categorized based on specific attributes, including the aggregate count of encompassed applications, the availability of commit history for the applications, the predominant focus on either the source code or the executable binaries of the applications, and the sources harnessed for the assembly of the respective dataset, among other pertinent criteria. Therefore, we decided to investigate available datasets tailored to Android for vulnerability analysis and presented the current status of available datasets for Android applications in terms of usability for further research studies. Table 4 summarizes the datasets and their useful features. Note that referenced datasets contain source codes and/or commit history of the applications generated in vulnerability analysis, meaning this does not include malware repositories or similar entities.

Geiger et al. [141] created a dataset of 8431 real Android apps whose source code is available in 8216 GitHub repositories. The data was gathered through app identification and metadata collection and stored in a graph-based database using Neo4j. This research also highlights the potential uses of the data, including investigations using algorithms from graph theory, and identifies the limitations of the dataset. However, upon examining this dataset, we noticed that it included codes in languages other than Java, which could potentially confuse future authors interested in studying Android vulnerability detection using this dataset. This also generates irrelevant data when referencing it in machine learning processes.

F-droid [68], the dataset consists of 1179 applications with 4416 different versions and 435,680 total commits. The analysis includes static analysis results obtained from various tools. The dataset also includes detailed information on each app's AndroidManifest.xml file, permissions, intents, and minimum SDK version. This dataset was found to be useful for research that requires different versions of the same application to compare the apps and determine vulnerability across the versions after adding new features to the applications. However, after a detailed investigation of this dataset,

Table 4 Available android datasets for vulnerability analysis

Paper	Year	Number of apps	Source code	Commit history	Sourced from
Krutz et al. [140]	2017	1402	Yes	Yes	F-Droid [68], GitHub
Geiger et al. [141]	2018	8431	Yes	Yes	GitHub, Google Play
Androvul [142]	2019	16,000	Yes	No	AndroZoo[143]
JEMMA[144]	2023	50,000 (from Java projects)	Yes	No	Major sources
LVDAndro[72]	2023	15,021	Yes	No	MobSF [84], Qark [85]

it becomes apparent that many are small applications with limited real-world usage.

JEMMA [144] is an extensible Java dataset designed for ML4Code applications. It is a large-scale, diverse, and high-quality dataset that comes with a considerable amount of pre-processed information such as metadata, representations (e.g., code tokens, graphs), and several properties (e.g., metrics, static analysis results) for 50,000 Java projects from the 50 K-C dataset, with over 1.2 million classes and over 8 million methods [85]. JEMMA lowers the barrier to entry in ML4Code research by providing the building blocks to experiment with source code models and tasks. With JEMMA, users can easily train and evaluate several models, conduct inference, and establish task benchmarks. The diversity of representations facilitates training on several types of model architectures, from graph-based models to models that take ASTs as input and other architectures such as code2seq, which reason over a bag of ASTs. This enables users to model source code in various formats and combinations, extracting valuable insights. ML4Code stands for Machine Learning for Source Code, a research area focusing on developing machine learning models to handle various tasks related to source code, such as code completion, code summarization, defect prediction, classification, and translation. ML4Code is important in software engineering because it enables developers to become faster and more productive by providing appropriate tool support for source code. With this dataset, developers can automate repetitive tasks, reduce errors, and improve the quality of their code. However, this proposed dataset does not include the specific routines of Android applications, such as intent security, privacy issues, and HTTP(s) implementations specific to Android-based mobile applications. Additionally, it involves some codes that Android applications do not have, such as authentication and authorization processes.

LVDAndro [72] has three different datasets but uses the one formed by scanning 15,021 APKs. This dataset includes scanned source code from open-source and closed-source applications containing 23 different CWE ID labels. It also has a 9:11 Vulnerable: non-vulnerable code sample ratio. Although collecting vulnerable Android code from real-world environments is a good approach, it relies heavily on certain open-source tools for creating these vulnerable code snippets. As a result, it may be subject to storing false positives from the tools they have used.

6 Challenges and limitations

Despite significant advancements in Android security analysis, several challenges persist, hindering the effectiveness and real-world applicability of existing approaches. These challenges primarily stem from the complexity of obfuscated applications, the lack of comprehensive real-world datasets, and the substantial computational resources required for machine-learning-based security analysis. Moreover, interpreting the decisions of machine learning models remains a crucial issue, as their black-box nature limits their transparency and usability in security-critical environments. Additionally, risks such as bias in training data and overfitting pose a threat to the reliability of automated vulnerability detection.

To address these challenges, upcoming research should focus on developing comprehensive labeled datasets and evaluation metrics, enhancing the interpretability of machine learning models, and minimizing biases that affect the generalization of detection methods. Transfer learning and domain adaptation may offer viable solutions to enhance model robustness while reducing dependence on extensive labeled datasets.

The following subsections provide an in-depth discussion of the major obstacles faced in this domain, specifically examining the impact of obfuscation on Android security testing, the necessity of real-world datasets, and the hardware constraints that limit the scalability of ML-driven code analysis techniques.

6.1 Challenges of an obfuscated android APK for conducting a security test and its impact on the analysis

Providing a percentage of Android applications that use obfuscation tools is challenging, as this information is not easily accessible to the public. However, it is quite common for developers to employ code obfuscation to protect their assets and prevent others from reverse-engineering their applications [146]. Among the options available to Android app developers, ProGuard [147] stands out due to its integration into the Android SDK and user-friendly design. Conducting a security assessment on an Android APK presents challenges for security testers and researchers. One significant obstacle arises from the increased code complexity caused by obfuscation techniques, which makes analyzing and understanding the app's functionality more difficult [146, 147]. When classes, methods, and variables are renamed, static code analysis tools can become bewildered, leading to difficulties in identifying security vulnerabilities.

Security analysis in Android applications faces two significant challenges due to obfuscation techniques:

Misleading automation—Obfuscation methods are deliberately employed to bypass automated detection mechanisms and obstruct forensic analysis. For instance, adversarial attacks can deceive machine-learning-based obfuscation detection techniques. Furthermore, straightforward approaches like repacking and manifest modifications can render traditional application fingerprinting techniques, such as hash-based identification, ineffective.
Complexity in manual analysis—Some obfuscation techniques hinder automated methods and significantly complicate manual analysis. When code is heavily encrypted or cluttered with excessive junk code, traditional static analysis methods become impractical.

These issues indicate an increasing complexity in Android security analysis, especially in automated methods that depend on highly skilled analysts and extensive forensic procedures. Various tools have been created to detect and even deobfuscate applications, with some methods yielding encouraging outcomes, particularly in managing obfuscation layers within Dalvik bytecode. However, our review noticed that most of these automated solutions are not publicly available—a finding that aligns with [148], who observed that many developed tools remain undisclosed.

Another limitation of existing literature is the insufficient focus on obfuscation techniques applied to native libraries. Multiple studies [149, 150] have emphasized that obfuscated native libraries can conceal malicious activities and evade certain detection techniques, but existing solutions still largely overlook this aspect.

Recent research has aimed to develop more resilient tools and methods less affected by obfuscation. For example, [151] introduced a tool to resist specific obfuscation strategies, such as structural transformations (e.g., modifications in the control flow graph) and dummy code insertion. However, while obfuscation-resistant techniques enhance malware detection capabilities, they do not necessarily improve reverse engineering and code comprehension. Instead of completely pulling the program's source code, these techniques depend on extracting non-code attributes to identify malware. Of special concern in this era is the emergent trend of multi-layered obfuscation, in which multiple uses of different types of obfuscation are applied iteratively to produce very intricate defense systems. Recent studies, including Mirzaei et al. [149, 152], reinforce this observation, demonstrating that layered obfuscation remains one of the most challenging barriers in Android security analysis.

However, conducting a security test on an Android APK is not impossible. Skilled security testers can effectively identify vulnerabilities by employing analysis, customized tools, and dynamic testing techniques. Nevertheless, it requires expertise and persistence compared to testing obfuscated APKs. Collaborating with app developers or gaining access to the APK version can significantly assist security testing endeavors. If researchers could discover ways to effectively test these applications without relying on source code, it would pave the way for future directions in app testing in real-world scenarios.

6.2 The importance of a dataset with real-world information

Real-world vulnerabilities exhibit a higher degree of complexity, necessitating the consideration and analysis of control flow, data flow, dominance relationships, and various other interdependencies among code elements. This paper [121] presents a systematic study focusing on various aspects of Deep Learning-based Vulnerability detection to identify real-world vulnerabilities efficiently. Through empirical analysis, the authors highlighted the deficiencies in current datasets and models that may hinder the practical applicability of these techniques. They also proposed a framework for collecting a real-world dataset for C/C++ languages. Despite these efforts, a significant challenge remains: the availability of high-quality, real-world datasets for effective ML-based vulnerability detection.

One of the primary challenges in leveraging machine learning for Android vulnerability detection is the need for large-scale, real-world datasets. The applicability of existing approaches remains limited in practical settings due to the extensive training data required to achieve reliable results. Many models rely heavily on carefully curated feature engineering techniques to approximate the complexity

of real-world applications. However, these models struggle to generalize effectively without diverse and representative datasets that capture various vulnerabilities across development practices. Furthermore, most publicly available datasets are either synthetic, outdated, or lack the contextual richness necessary to reflect real-world software vulnerabilities. This limitation is further evident when comparing real-world datasets with commonly used synthetic datasets.

Based on their experimental results, synthetic and semi-synthetic datasets like SARD [153] and Juliet [154] (composed of much simpler code snippets) result in a large number of duplicates. In contrast, real-world vulnerabilities are much more complex and have far fewer duplicates [121]. Indeed, duplicate instances within the training set can induce a DL model to acquire irrelevant features. Furthermore, common examples between the training and test sets pose challenges in fairly comparing distinct DL models designed for the vulnerability prediction task. In an ideal scenario, a DL-based model should undergo training and testing on a dataset where all examples are unique. The existence of duplications tends to inflate the overall performance of a method, as evident from the observed discrepancy in the baseline results [155]. This discrepancy suggests that evaluations based on such datasets may not accurately reflect a model's actual performance in real-world settings, leading to unreliable vulnerability detection in practical applications.

6.3 Limitations of existing benchmarks and splitting strategies

Addressing dataset coverage, train/test splitting strategies, and the implications of code duplication is crucial, as these factors directly affect the validity, generalizability, and empirical reliability of ML-based vulnerability prediction studies. In particular, the study [155] demonstrates how duplicated or recurring code fragments can artificially inflate model performance, since models may simply recognize code previously seen in the training set rather than truly generalize to unseen cases.

Existing datasets used for Android security and ML4Code [144] studies reveal significant limitations in terms of training/testing splits and duplication control. LVDAndro [72] adopts a fixed 80:20 random split and applies a basic deduplication step at the line level; however, it does not ensure that near-duplicate code fragments across training and test sets are eliminated, thus risking inflated performance. AndroVul [142] relies on tenfold cross-validation and removes duplicate APKs, but fails to address code-level duplication, such as the reuse of third-party libraries. JEMMA [144] provides more systematic filtering through clone detection and redundant code removal, yet it delegates train/test split decisions entirely to the user, which hinders experimental reproducibility. AndroidTimeMachine [141] differs conceptually, offering a large graph-based dataset of 8431 real apps with commit histories and Google Play metadata. At the same time, repository and package deduplication are applied; the dataset does not provide any predefined ML benchmark splits, leaving researchers to design their own. Finally, Krutz et al. [140] focus on analyzing 1402 F-Droid repositories at the commit level, including all historical versions of AndroidManifest.xml; here, duplication is not treated as bias but as an inherent feature, although this would make the dataset unsuitable for direct ML benchmarking.

Overall, none of the datasets explicitly examine the negative effect of code duplication on ML performance, as shown in Table 5, which is a limitation noted in the literature. Consequently, future work should not only provide standardized and well-documented train/test partitions but also apply project-level and semantic deduplication strategies, ensuring that benchmark results genuinely reflect model generalization rather than memorization of recurring code fragments.

6.4 Hardware resources

Considering the necessity for a substantial training dataset and multiple hidden layers, machine learning model training frequently demands high-performance processing units, such as Graphics Processing Units (GPUs), and significant memory capacity [156, 157]. A user survey investigation [157] underscores the importance of specialized hardware for machine learning training. However, these hardware requirements present challenges to researchers operating within limited hardware resources.

Researchers have been actively exploring ways to optimize resource utilization in response to the challenges posed by the demanding hardware requirements for training machine learning models. Several studies have proposed techniques to improve the efficiency of training processes in hardware environments. For example, methods such as model compression and algorithmic optimizations have shown promise in reducing the burden on processing units and memory [4]-5]. Moreover, advancements in hardware technology, such as the development of energy-efficient processors, offer potential solutions for overcoming these challenges [6]. As this field progresses, it becomes crucial to find a balance between the increasing demands of machine-learning models and the practical limitations imposed by hardware resources.

7 Threats to the validity of the review

While this literature review was conducted to cover key studies on vulnerability analysis for Android applications, it is important to acknowledge that the results obtained may not

Table 5 Analysis of available datasets in terms of limitations

Dataset	Train/Test Split	Duplication handling	Duplication impact analysis	Notes
LVDAndro [72]	Fixed 80:20 random split	Basic (same line + same label level)	None	Unclear whether code duplication causes train/test overlap
AndroVul [142]	Tenfold cross-validation	APK hash-level (duplicate APKs removed)	None	Code-level duplication (e.g., SDKs/libraries) is not addressed
JEMMA [144]	User-defined, no predefined split	More comprehensive (clone detection, redundant removal)	None	Benchmarking is left to the user; lack of standardization
Geiger et al. [141]	None (commit + Google Play metadata)	Repository/Package-level deduplication	None	Graph-based commit dataset; split left to the user for ML tasks
Krutz et al. [140]	None (statistical analysis of commits)	None (all app commits intentionally included)	None	The goal is permission evolution analysis; duplication is an integral part of the methodology

include all relevant studies due to certain limitations encountered during the review process. Consequently, this section outlines potential sources of vulnerability to the study's validity, categorized into construct, internal, external, and conclusion validity, while also detailing the measures taken to address these concerns.

7.1 Construct validity

Potential threats to construct validity arise from the use of search term-based queries in repositories. It is crucial to recognize the possible existence of high-quality papers excluded from this study's scope due to their absence from key research repositories like the ACM Digital Library, IEEEXplore Digital Library, Web of Science, and Springer Link. To address this limitation, Google Scholar was utilized as an additional resource to identify studies that may have been overlooked. However, it is acknowledged that some relevant publications may still be missing from the compiled dataset. Another aspect concerns construct validity, where the possibility of minor errors during the filtration process—encompassing inclusion and exclusion criteria—can introduce bias. In response to this challenge, a comprehensive analysis of the publication list was conducted, including a detailed cross-referencing of primary studies to identify and correct potential errors and discrepancies.

7.2 Internal validity

Internal validity refers to the strength of the data extraction and analysis procedures, which is a crucial aspect of the overall review methodology. This effort involved significant work covering data extraction and subsequent analytical processes. Given this, careful cross-validation strengthened data collection, consolidating results after agreement among all authors regarding the comparative findings. Despite these careful measures, the possibility of errors in the data extraction and analytical phases remains a consideration. To address this issue, a key strategy involves engaging the original authors for verification purposes, thereby enhancing the accuracy and integrity of the extracted and analyzed information.

7.3 External validity

External validity pertains to the generalizability of the outcomes derived from the primary investigations. The analytical scrutiny, constituting an integral facet of this review, was executed on a corpus of research publications from 2017 to 2024, encompassing a comprehensive appraisal of contemporary Android applications regarding vulnerability detection and prediction methodologies. This temporal boundary was strategically selected, considering the pronounced upsurge in ML techniques harnessed for vulnerability detection during this interval, attributable to notable advancements in software security and artificial intelligence. Acknowledging that the prevailing trends and dynamics may diverge when assessed across disparate temporal epochs is vital. Consequently, it is plausible that certain seminal and exhaustive studies predating the stipulated period may not be encompassed within the present analysis.

8 Conclusion

In response to the increasing demand for Android application development, developers are constantly trying to meet the demand with new software. It is important to consider security concepts while developing new software. This comprehensive review examines the latest techniques for detecting vulnerabilities in Android applications, encompassing research from 2017 to 2024. The review examines application and code analysis methods, exploring how ML and DL techniques can be applied for vulnerability detection. Additionally, recent advances in LLMs have introduced new possibilities in this field, as they exhibit a strong ability to analyze code structures and identify potential vulnerabilities. However, while some studies report promising results, LLM-based vulnerability detection still faces challenges, particularly in handling obfuscated and complex code structures, as well as in ensuring balanced, duplication-free datasets for reliable evaluation.

The findings presented in this review serve as an invitation to practitioners and researchers, encouraging them to propose methods, tools, and techniques for further exploration and development. The ultimate objective is to facilitate the integration of ML techniques into software engineering applications while ensuring ease of use, flexibility, and maintainability. While traditional ML and DL techniques remain pivotal, incorporating LLMs offers a new avenue for security analysis. However, more research is needed to optimize these models for practical use and ensure their effectiveness in real-world Android development environments.

Furthermore, bridging the gap between technical insight and developer-friendly tools remains paramount for sustaining robust security practices. Future research should focus on developing models that identify vulnerabilities and provide meaningful explanations for their decisions, ensuring transparency and trust in automated analysis. Equally important is the evaluation of model scalability, inference speed, and integration feasibility into DevSecOps pipelines, which remain underexplored yet critical for real-world adoption. Addressing these challenges will contribute to a more effective and interpretable application of machine learning in real-world vulnerability detection scenarios.

9 Future directions

While detection mechanisms are crucial in identifying security vulnerabilities, they are insufficient for the app development community. Future research should investigate how these detection techniques can be seamlessly incorporated into Android development environments as tools or plugins. Such integration would enable developers to test security throughout the development process rather than waiting until the entire application is finished.

Another major challenge is the lack of automated methods for determining the root causes of vulnerabilities. Research should focus on integrating XAI methodologies into vulnerability detection frameworks to address this issue. Combining LLMs and XAI techniques can enhance vulnerability identification and the interpretability of results. By providing detailed insights into why certain code segments are classified as vulnerable, these approaches could bridge the gap between complex AI-driven security analysis and developer-friendly tools, improving transparency and usability.

Additionally, preserving the semantic integrity of code while applying ML-based detection techniques remains a significant challenge. Current studies often overlook the importance of interpreting code within its broader functional and structural context, which is crucial for accurate and reliable vulnerability detection. Future work should explore context-aware analysis techniques that ensure vulnerabilities are detected without introducing excessive false positives.

Future research should also prioritize dataset quality, addressing duplication, imbalance, and outdated benchmarks, to ensure empirical validity and reproducibility of findings.

Finally, strategically integrating LLMs within phased AI-driven security frameworks could unlock new capabilities. By combining ML, DL, XAI, and LLMs, future systems can deliver contextualized, explainable, and actionable insights—paving the way for secure, intelligent, and developer-oriented Android applications.

Author contribution **K.E.A**. and **E.N.Y.** conceptualized and structured the research framework, conducted the systematic literature review following the PRISMA guidelines, and authored the primary manuscript content. **K.E.A.**, **E.N.Y.**, and **S.M.D.** collectively participated in the selection, critical analysis, and synthesis of relevant publications and the design and creation of all visual representations included in the manuscript. **S.G.** provided expertise during the manuscript preparation, contributed to the interpretation of results, and supported manuscript revisions. All authors reviewed the manuscript, including **K.E.A.**, **E.N.Y.**, **S.M.D.**, and **S.G.**

Funding This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Data availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interests The authors declare no competing interests.

References

1. Kazmi, S.H.A., Qamar, F., Hassan, R., Nisar, K., Chowdhry, B.S.: Survey on joint paradigm of 5G and SDN emerging mobile

technologies: architecture, security, challenges and research directions. Wirel. Pers. Commun. **130**(4), 2753–2800 (2023). https://doi.org/10.1007/s11277-023-10402-7
2. Al-Ansi, A.M., Jaboob, M., Garad, A., Al-Ansi, A.: Analyzing augmented reality (AR) and virtual reality (VR) recent development in education. Soc Sci & Humanit Open **8**(1), 100532 (2023). https://doi.org/10.1016/j.ssaho.2023.100532
3. CVE: Common Vulnerabilities and Exposures: https://www.cve.org/about/overview. Accessed 12 March 2025
4. Dowd, M., McDonald, J., Schuh, J.: The art of software security assessment: identifying and preventing software vulnerabilities. Pearson Education (2006).
5. Pam, N.H.: Detection of recurring software vulnerabilities. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering. https://dl.acm.org/doi/abs/https://doi.org/10.1145/1858996.1859089. Accessed 12 Mar 2025. [Online].
6. Piessens, F.: A taxonomy of causes of software vulnerabilities in internet software. In: Vouk, M. (ed.), The 13th International Symposium on Software Reliability Engineering, Supplementary Proceedings of the 13th International Symposium on Software Reliability Engineering, pp. 47–52. Annapolis, Maryland, USA, November 12–15 (2002)
7. Abdalkareem, R., Shihab, E., Rilling, J.: On code reuse from StackOverflow: an exploratory study on Android apps. Inf. Softw. Technol. **88**, 148–158 (2017). https://doi.org/10.1016/j.infsof.2017.04.005
8. Fischer, F., et al.: Stack overflow considered harmful? The Impact of Copy&Paste on Android Application Security. In: 2017 IEEE Symposium on Security and Privacy (SP), May 2017, pp. 121–136. https://doi.org/10.1109/SP.2017.31.
9. Zhang, H., Wang, S., Li, H., Chen, T.-H., Hassan, A.E.: A study of C/C++ code weaknesses on Stack Overflow. IEEE Trans. Softw. Eng. **48**(7), 2359–2375 (2022). https://doi.org/10.1109/TSE.2021.3058985
10. Mäntylä, M.V., Lassenius, C.: What types of defects are really discovered in code reviews? IEEE Trans. Softw. Eng. **35**(3), 430–448 (2009). https://doi.org/10.1109/TSE.2008.71
11. McQueen, M.A., McQueen, T.A., Boyer, W.F., Chaffin, M.R.: Empirical Estimates and Observations of 0Day Vulnerabilities. In: 2009 42nd Hawaii International Conference on System Sciences, January 2009, pp. 1–12. https://doi.org/10.1109/HICSS.2009.186.
12. Czerwonka, J., Greiler, M., Tilford, J.: Code reviews do not find bugs. How the Current Code Review Best Practice Slows Us Down. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, May 2015, pp. 27–28. https://doi.org/10.1109/ICSE.2015.131.
13. Google: List of all products and related security vulnerabilities. https://www.cvedetails.com/product-list/product_type-/vendor_id-1224/firstchar-/page-1/Google.html. Accessed 12 Mar 2025. [Online].
14. Nikolaeva, A., Lin, Y.-T., Nello-Deakin, S., Rubin, O., von Schönfeld, K.C.: Living without commuting: experiences of a less mobile life under COVID-19. Mobilities **18**(1), 1–20 (2023). https://doi.org/10.1080/17450101.2022.2072231
15. Wei, Z., et al.: Integrated sensing and communication signals toward 5G-A and 6G: a survey. IEEE Internet Things J. **10**(13), 11068–11092 (2023). https://doi.org/10.1109/JIOT.2023.3235618
16. von Solms, B., von Solms, R.: Cybersecurity and information security – what goes where? Info & Comput Security **26**(1), 2–9 (2018). https://doi.org/10.1108/ICS-04-2017-0025
17. Narayanan, A., Chandramohan, M., Chen, L., Liu, Y.: A multi-view context-aware approach to Android malware detection and malicious code localization. Empirical Software Engineering. https://link.springer.com/article/https://doi.org/10.1007/s10664-017-9539-8. Accessed 12 Mar 2025. [Online].
18. Carpenter, P.: Using the Predict, Prevent, Detect, Respond Framework to Communicate Your Security Program Strategy. https://www.gartner.com/en/documents/3286317. Accessed 12 March 2025. [Online].
19. Garg, S., Baliyan, N.: Machine learning based android vulnerability detection: a roadmap. In: Kanhere, S., Patil, V.T., Sural, S., Gaur, M.S. (eds.) Information Systems Security, Springer International Publishing, Cham (2020), pp. 87–93. https://doi.org/10.1007/978-3-030-65610-2_6.
20. Lee, J., Babcock, J., Pham, T.S., Bui, T.H.: Myounggu Kang Smart city as a social transition towards inclusive development through technology: a tale of four smart cities. https://www.tandfonline.com/doi/full/https://doi.org/10.1080/12265934.2022.2074076. Accessed 12 March 2025. [Online].
21. Ghaffarian, S.M., Shahriari, H.R.: Software vulnerability analysis and discovery using machine-learning and data-mining techniques: a survey. ACM Comput Survey: **50**(4). https://dl.acm.org/doi/abs/https://doi.org/10.1145/3092566. Accessed 12 March 2025. [Online].
22. Aslan, Ö., Aktuğ, S.S., Ozkan-Okay, M., Yilmaz, A.A., Akin, E.: A comprehensive review of cyber security vulnerabilities, threats, attacks, and solutions. Electronics **12**(6), 6 (2023). https://doi.org/10.3390/electronics12061333
23. Tugwell, P., Tovey, D.: PRISMA 2020. J. Clin. Epidemiol. **134**, A5–A6 (2021). https://doi.org/10.1016/j.jclinepi.2021.04.008
24. Garg, S., Baliyan, N.: Data on vulnerability detection in Android. Data Brief **22**, 1081–1087 (2018). https://doi.org/10.1016/j.dib.2018.12.038
25. Kapitsaki, G.M.: Examining the Privacy Vulnerability Level of Android Applications. In: 15th International conference on web information systems and technologies. https://doi.org/10.5220/0007955100002366 (2019)
26. Rathod, J., Bhatti, D.: Building comprehensive dataset: android file and unstructured data collection from APKs for enhanced vulnerability detection. Int. J. Comput. Appl. Technol. Res. (2023). https://doi.org/10.7753/IJCATR1212.1004.
27. Garg, S., Baliyan, N.: A novel parallel classifier scheme for vulnerability detection in Android. Comput. Electr. Eng. **77**, 12–26 (2019). https://doi.org/10.1016/j.compeleceng.2019.04.019
28. Lochanan, A., Jaysankar, M.R., Subramanian, N.: Permissions based android vulnerability detection and classification based on severity using machine learning. In: Recent Developments in Electronics and Communication Systems, IOS Press, pp. 200–207. https://doi.org/10.3233/ATDE221258 (2023)
29. Zhan, X., et al.: Research on third-party libraries in Android apps: a taxonomy and systematic literature review. IEEE Trans. Softw. Eng. **48**(10), 4181–4213 (2022). https://doi.org/10.1109/TSE.2021.3114381
30. Wei, X., Wolf, M.: A survey on HTTPS implementation by Android apps: issues and countermeasures. Appl. Comput. Informatics **13**(2), 101–117 (2017). https://doi.org/10.1016/j.aci.2016.10.001
31. Abdullah, H., Zeebaree, S.R.M.: Android mobile applications vulnerabilities and prevention methods: a review. In: 2021 2nd Information Technology to Enhance e-learning and Other Application (IT-ELA), December 2021, pp. 148–153. https://doi.org/10.1109/IT-ELA52201.2021.9773615.
32. Automated security testing of Android applications for secure mobile development. In: IEEE Conference Publication, IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/9155681. Accessed 12 March 2025. [Online].

33. Casilari, E., Luque, R., Morón, M.-J.: Analysis of Android device-based solutions for fall detection. Sensors **15**(8), 8 (2015). https://doi.org/10.3390/s150817827
34. Secure coding practices in Java. In: Proceedings of the 40th International Conference on Software Engineering. https://dl.acm.org/doi/abs/https://doi.org/10.1145/3180155.3180201. Accessed: 12 March 2025. [Online].
35. Definition of ARTIFICIAL INTELLIGENCE. https://www.merriam-webster.com/dictionary/artificial+intelligence. Accessed 25 March 2025. [Online].
36. Mitchell, T.M.: Machine Learning. McGraw-Hill (1997).
37. Kelleher, J.D.: Deep Learning. MIT Press (2019).
38. Explainable Artificial Intelligence. DARPA. https://www.darpa.mil/research/programs/explainable-artificial-intelligence. Accessed 25 March 2025. [Online].
39. Garg, S., Baliyan, N.: Android security assessment: a review, taxonomy and research gap study. Comput. Secur. **100**, 102087 (2021). https://doi.org/10.1016/j.cose.2020.102087
40. Li, L., et al.: Static analysis of android apps: a systematic literature review. Inf. Softw. Technol. **88**, 67–95 (2017). https://doi.org/10.1016/j.infsof.2017.04.001
41. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot: a Java bytecode optimization framework. In: CASCON First Decade High Impact Papers, in CASCON '10. USA, IBM Corp., November 2010, pp. 214–224. https://doi.org/10.1145/1925805.1925818.
42. Vallée-Rai, R., Hendren, L.J.: Jimple: Simplifying Java Bytecode for Analyses and Transformations, Request PDF. https://www.researchgate.net/publication/243776080_Jimple_Simplifying_Java_Bytecode_for_Analyses_and_Transformations. Accessed 12 March 2025. [Online].
43. Senanayake, J., Kalutarage, H., Al-Kadri, M.O., Petrovski, A., Piras, L.: Android source code vulnerability detection: a systematic literature review. ACM Comput. Surv. **55**(9), 1–37 (2023). https://doi.org/10.1145/3556974
44. Sharma, T. et al.: A Survey on Machine Learning Techniques for Source Code Analysis, 13 September, arXiv: arXiv:2110.09610. https://doi.org/10.48550/arXiv.2110.09610 (2022)
45. Mehtab, A., et al.: AdDroid: rule-based machine learning framework for Android malware analysis. Mob. Networks Appl. **25**(1), 180–192 (2020). https://doi.org/10.1007/s11036-019-01248-0
46. Appice, A., Andresini, G., Malerba, D.: Clustering-aided multi-view classification: a case study on Android malware detection. J. Intell. Inf. Syst. **55**(1), 1–26 (2020). https://doi.org/10.1007/s10844-020-00598-6
47. Liu, X., Zhang, J., Lin, Y., Li, H.: ATMPA: attacking machine learning-based malware visualization detection methods via adversarial examples. In: Proceedings of the International Symposium on Quality of Service, in IWQoS '19. New York, NY, USA, June 2019. Association for Computing Machinery, pp. 1–10. https://doi.org/10.1145/3326285.3329073.
48. Yuan, Z., Lu, Y., Xue, Y.: Droiddetector: android malware characterization and detection using deep learning. Tsinghua Sci. Technol. **21**(1), 114–123 (2016). https://doi.org/10.1109/TST.2016.7399288
49. Wang, R., et al.: {EASEAndroid}: Automatic Policy Analysis and Refinement for Security Enhanced Android via {Large-Scale} {Semi-Supervised} Learning. In: Presented at the 24th USENIX Security Symposium (USENIX Security 15), pp. 351–366. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/wang-ruowen (2015). Accessed 13 March 2025. [Online].
50. Fadadu, F., Handa, A., Kumar, N., Shukla, S.K.: Evading API call sequence based malware classifiers. In: Zhou, J., Luo, X., Shen, Q., Xu, Z. (eds.) Information and Communications Security, Springer International Publishing, Cham, pp. 18–33. https://doi.org/10.1007/978-3-030-41579-2_2 (2020)
51. Han, H., Lim, S., Suh, K., Park, S., Cho, S., Park, M.: Enhanced android malware detection: an svm-based machine learning approach. In: 2020 IEEE International Conference on Big Data and Smart Computing (BigComp), February 2020, pp. 75–81. https://doi.org/10.1109/BigComp48618.2020.00-96 (2020)
52. Mahindru, A., Sangal, A.L.: Feature-based semi-supervised learning to detect malware from android. In: Satapathy, S.C., Jena, A.K., Singh, J., Bilgaiyan, S. (eds.) Automated Software Engineering: A Deep Learning-Based Approach, Springer International Publishing, Cham, pp. 93–118. https://doi.org/10.1007/978-3-030-38006-9_6 (2020)
53. Mantoo, B.A., Khurana, S.S.: Static, dynamic and intrinsic features based android malware detection using machine learning. In: Singh, P.K., Kar, A.K., Singh, Y., Kolekar, M.H., Tanwar, S. (eds.) Proceedings of ICRIC 2019, Springer International Publishing, Cham, pp. 31–45. https://doi.org/10.1007/978-3-030-29407-6_4 (2020)
54. Martín, I., Hernández, J.A., de los Santos, S.: Machine-learning based analysis and classification of Android malware signatures. Future Gener. Comput. Syst. **97**, 295–305 (2019). https://doi.org/10.1016/j.future.2019.03.006
55. Nguyen-Vu, L., Ahn, J., Jung, S.: Android fragmentation in malware detection. Comput. Secur. **87**, 101573 (2019). https://doi.org/10.1016/j.cose.2019.101573
56. Sharmeen, S., Huda, S., Abawajy, J., Hassan, M.M.: An adaptive framework against android privilege escalation threats using deep learning and semi-supervised approaches. Appl. Soft Comput. **89**, 106089 (2020). https://doi.org/10.1016/j.asoc.2020.106089
57. Pang, Y., Xue, X., Wang, H.: Predicting vulnerable software components through deep neural network. In: Proceedings of the 2017 International Conference on Deep Learning Technologies, in ICDLT '17. New York, NY, USA, June 2017. Association for Computing Machinery, pp. 6–10. https://doi.org/10.1145/3094243.3094245 (2017)
58. Wu, F., Wang, J., Liu, J., Wang, W.: Vulnerability detection with deep learning. In: 2017 3rd IEEE International Conference on Computer and Communications (ICCC), December 2017, pp. 1298–1302. https://doi.org/10.1109/CompComm.2017.8322752 (2017)
59. Zhuo, L., Zhimin, G., Cen, C.: Research on android intent security detection based on machine learning. In: 2017 4th International Conference on Information Science and Control Engineering (ICISCE), July 2017, pp. 569–574. https://doi.org/10.1109/ICISCE.2017.124 (2017)
60. Bilgin, Z., Ersoy, M.A., Soykan, E.U., Tomur, E., Çomak, P., Karaçay, L.: Vulnerability prediction from source code using machine learning. IEEE Access **8**, 150672–150684 (2020). https://doi.org/10.1109/ACCESS.2020.3016774
61. Gupta, A., Suri, B., Kumar, V., Jain, P.: Extracting rules for vulnerabilities detection with static metrics using machine learning. Int. J. Syst. Assur. Eng. Manag. **12**(1), 65–76 (2021). https://doi.org/10.1007/s13198-020-01036-0
62. Kim, S., Yeom, S., Oh, H., Shin, D., Shin, D.: Automatic malicious code classification system through static analysis using machine learning. Symmetry **13**(1), 1 (2021). https://doi.org/10.3390/sym13010035
63. Shiqi, L., Shengwei, T., Long, Y., Jiong, Y., Hua, S.: Android malicious code classification using deep belief network. KSII Trans. Internet Inf. Syst. **12**(1), 454–475 (2018). https://doi.org/10.3837/tiis.2018.01.022
64. Ali Alatwi, H., Oh, T., Fokoue, E., Stackpole, B.: Android malware detection using category-based machine learning classifiers. In: Proceedings of the 17th Annual Conference on Information Technology Education, in SIGITE '16. New York, NY,

USA, September 2016. Association for Computing Machinery, pp. 54–59. https://doi.org/10.1145/2978192.2978218 (2016)
65. Cui, J., Wang, L., Zhao, X., Zhang, H.: Towards predictive analysis of android vulnerability using statistical codes and machine learning for IoT applications. Comput. Commun. **155**, 125–131 (2020). https://doi.org/10.1016/j.comcom.2020.02.078
66. Li, Y., Ma, R., Jiao, R.: A hybrid malicious code detection method based on deep learning. Int. J. Secur. Appl. **9**(5), 205–216 (2015). https://doi.org/10.14257/ijsia.2015.9.5.21
67. Lin, G., Xiao, W., Zhang, J., Xiang, Y.: Deep learning-based vulnerable function detection: a benchmark. In: Zhou, J., Luo, X., Shen, Q., Xu, Z. (eds.) Information and Communications Security, pp. 219–232. Springer International Publishing, Cham (2020), https://doi.org/10.1007/978-3-030-41579-2_13.
68. Krutz, D.E., et al.: A dataset of open-source android applications. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, May 2015, pp. 522–525. https://doi.org/10.1109/MSR.2015.79.
69. Rahman, A., Pradhan, P., Partho, A., Williams, L.: Predicting android application security and privacy risk with static code metrics. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), May 2017, pp. 149–153. https://doi.org/10.1109/MOBILESoft.2017.14.
70. Piskachev, G., Do, L.N.Q., Bodden, E.: Codebase-adaptive detection of security-relevant methods. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, in ISSTA 2019. New York, NY, USA: Association for Computing Machinery, July 2019, pp. 181–191. https://doi.org/10.1145/3293882.3330556.
71. Malik, Y., Campos, C.R.S., Jaafar, F.: Detecting android security vulnerabilities using machine learning and system calls analysis. In: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C), July 2019, pp. 109–113. https://doi.org/10.1109/QRS-C.2019.00033.
72. Senanayake, J., Kalutarage, H., Al-Kadri, M.O., Piras, L., Petrovski, A.: Labelled Vulnerability Dataset on Android Source Code (LVDAndro) to develop AI-Based code vulnerability detection models. In: Presented at the 20th International Conference on Security and Cryptography, September 2024, pp. 659–666. https://www.scitepress.org/Link.aspx?doi=https://doi.org/10.5220/0012060400003555. Accessed 19 September 2024. [Online].
73. Senanayake, J., Kalutarage, H., Petrovski, A., Piras, L., Al-Kadri, M.O.: Defendroid: real-time Android code vulnerability detection via blockchain federated neural network with XAI. J. Inf. Secur. Appl. **82**, 103741 (2024). https://doi.org/10.1016/j.jisa.2024.103741
74. Mathews, N.S., Brus, Y., Aafer, Y., Nagappan, M., McIntosh, S.: LLbezpeky: leveraging large language models for vulnerability detection, 13 February 2024. arXiv: arXiv:2401.01269. https://doi.org/10.48550/arXiv.2401.01269.
75. Mitra, J., Ranganath, V.-P.: Ghera: a repository of android app vulnerability benchmarks. In: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, in PROMISE. New York, NY, USA: Association for Computing Machinery, November 2017, pp. 43–52. https://doi.org/10.1145/3127005.3127010.
76. Senanayake, J., Kalutarage, H., Al-Kadri, M.O., Petrovski, A., Piras, L.: Android code vulnerabilities early detection using AI-Powered ACVED plugin. In: Atluri, V., Ferrara, A.L. (eds.) Data and applications security and privacy XXXVII, pp. 339–357. Springer Nature Switzerland, Cham (2023) https://doi.org/10.1007/978-3-031-37586-6_20.
77. Yim, K.S.: Predicting likely-vulnerable code changes: machine learning-based vulnerability protections for android open source project. 26 May 2024, arXiv: arXiv:2405.16655. https://doi.org/10.48550/arXiv.2405.16655.
78. Android Open Source Project. Android Open Source Project. https://source.android.com/(2025) Accessed 14 March 2025. [Online].
79. Senanayake, J., Kalutarage, H., Petrovski, A., Al-Kadri, M.O., Piras, L.: FedREVAN: Real-time DEtection of Vulnerable Android Source Code Through Federated Neural Network with XAI. In: Katsikas, S., Abie, H., Ranise, S., Verderame, L., Cambiaso, E., Ugarelli, R., Praça, I., Li, W., Meng, W., Furnell, S., Katt, B., Pirbhulal, S., Shukla, A., Ianni, M., Dalla Preda, M., Choo, K.-K. R., Pupo Correia, M., Abhishta, A., Sileno, G., Alishahi, M., Kalutarage, H., Yanai, N. (eds.) Computer Security. ESORICS 2023 International Workshops, pp. 426–441. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-54129-2_25.
80. Nasteski, V.: An overview of the supervised machine learning methods. In: ResearchGate, December 2024, https://doi.org/10.20544/HORIZONS.B.04.1.17.P05.
81. Ma, S., Thung, F., Lo, D., Sun, C., Deng, R.H.: VuRLE: automatic vulnerability detection and repair by learning from examples. In: Foley, S.N., Gollmann, D., Snekkenes, E. (eds.) Computer security—ESORICS 2017, pp. 229–246. Springer International Publishing, Cham (2017), https://doi.org/10.1007/978-3-319-66399-9_13.
82. Gajrani, J., Tripathi, M., Laxmi, V., Somani, G., Zemmari, A., Gaur, M.S.: Vulvet: vetting of vulnerabilities in Android apps to thwart exploitation. Digit. Threats **1**(2), 1–25 (2020). https://doi.org/10.1145/3376121
83. Senanayake, J., Kalutarage, H., Al-Kadri, M.O., Petrovski, A., Piras, L.: Developing secured android applications by mitigating code vulnerabilities with machine learning. In: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, in ASIA CCS '22. New York, NY, USA: Association for Computing Machinery, May 2022, pp. 1255–1257. https://doi.org/10.1145/3488932.3527290 (2022)
84. LaMalva, G., Schmeelk, S.: MobSF: Mobile Health Care Android Applications Through The Lens of Open Source Static Analysis. In: 2020 IEEE MIT Undergraduate Research Technology Conference (URTC), October 2020, pp. 1–4. https://doi.org/10.1109/URTC51696.2020.9668870.
85. Kulkarni, K., Javaid, A.Y.: Open source android vulnerability detection tools: a survey, 31 July 2018, arXiv: arXiv:1807.11840. https://doi.org/10.48550/arXiv.1807.11840 (2018)
86. Ranganath, V.-P., Mitra, J.: Are free Android app security analysis tools effective in detecting known vulnerabilities? Empir. Softw. Eng. **25**(1), 178–219 (2020). https://doi.org/10.1007/s10664-019-09749-y
87. Noever, D.: Can large language models find and fix vulnerable software?. 20 August 2023, arXiv: arXiv:2308.10345. https://doi.org/10.48550/arXiv.2308.10345 (2023)
88. Kouliaridis, V., Karopoulos, G., Kambourakis, G.: Assessing the effectiveness of LLMs in android application vulnerability analysis, 27 June 2024, arXiv: arXiv:2406.18894. https://doi.org/10.48550/arXiv.2406.18894 (2024)
89. Amin, A., Eldessouki, A., Magdy, M.T., Abdeen, N., Hindy, H., Hegazy, I.: AndroShield: automated Android applications vulnerability detection, a hybrid static and dynamic analysis approach. Information **10**(10), 10 (2019). https://doi.org/10.3390/info10100326
90. Zhu, J., Li, K., Chen, S., Fan, L., Wang, J., Xie, X.: A comprehensive study on static application security testing (SAST) tools for Android. IEEE Trans. Softw. Eng. **50**(12), 3385–3402 (2024). https://doi.org/10.1109/TSE.2024.3488041
91. Gencer, K., Başçiftçi, F.: Time series forecast modeling of vulnerabilities in the android operating system using ARIMA and deep

learning methods. Sustain. Comput. Inform. Syst. **30**, 100515 (2021). https://doi.org/10.1016/j.suscom.2021.100515
92. Zhang, Q., Fang, C., Yu, B., Sun, W., Zhang, T., Chen, Z.: Pre-trained model-based automated software vulnerability repair: how far are we? IEEE Trans. Dependable Secur. Comput. **21**(4), 2507–2525 (2024). https://doi.org/10.1109/TDSC.2023.3308897
93. Wang, C., Zhang, L., Zhao, K., Ding, X., Wang, X.: AdvAndMal: adversarial training for Android malware detection and family classification. Symmetry **13**(6), 6 (2021). https://doi.org/10.3390/sym13061081
94. Bala, N., Ahmar, A., Li, W., Tovar, F., Battu, A., Bambarkar, P.: DroidEnemy: battling adversarial example attacks for Android malware detection. Digit. Commun. Networks **8**(6), 1040–1047 (2022). https://doi.org/10.1016/j.dcan.2021.11.001
95. Renjith, G., Laudanna, S., Aji, S., Visaggio, C.A., Vinod, P.: GANG-MAM: GAN based engine for modifying Android malware. SoftwareX **18**, 100977 (2022). https://doi.org/10.1016/j.softx.2022.100977
96. Rathore, H., Nikam, P., Sahay, S.K., Sewak, M.: Identification of adversarial android intents using reinforcement learning. In: 2021 International Joint Conference on Neural Networks (IJCNN), July 2021, pp. 1–8. https://doi.org/10.1109/IJCNN52387.2021.9534142.
97. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy, May 2012, pp. 95–109. https://doi.org/10.1109/SP.2012.16.
98. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.: (PDF) DREBIN: effective and explainable detection of android malware in your pocket, In: ResearchGate, https://doi.org/10.14722/ndss.2014.23247.
99. Gómez, A., Muñoz, A.: Deep learning-based attack detection and classification in Android devices. Electronics **12**(15), 15 (2023). https://doi.org/10.3390/electronics12153253
100. Wajahat, A., et al.: Outsmarting Android malware with cutting-edge feature engineering and machine learning techniques. Comput. Mater. Contin. **79**(1), 651–673 (2024). https://doi.org/10.32604/cmc.2024.047530
101. Faruki, P., et al.: Android security: a survey of issues, malware penetration, and defenses. IEEE Commun. Surv. Tutor. **17**(2), 998–1022 (2015). https://doi.org/10.1109/COMST.2014.2386139
102. Tam, K., Feizollah, A., Anuar, N.B., Salleh, R., Cavallaro, L.: The evolution of Android malware and Android analysis techniques. ACM Comput. Surv. **49**(4), 1–41 (2017). https://doi.org/10.1145/3017427
103. Gyamfi, N.K., Goranin, N., Ceponis, D., Čenys, H.A.: Automated system-level malware detection using machine learning: a comprehensive review. Appl. Sci. **13**(21), 21 (2023). https://doi.org/10.3390/app132111908
104. Pauck, F., Bodden, E., Wehrheim, H.: Do Android taint analysis tools keep their promises? In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, in ESEC/FSE 2018. Association for Computing Machinery, Oct. 2018, pp. 331–341. New York, NY, USA. https://doi.org/10.1145/3236024.3236029.
105. Wei, F., Roy, S., Ou, X., Robby.: Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. ACM Trans. Priv. Secur. **21**(3), 1–32 (2018). https://doi.org/10.1145/3183575.
106. Bosu, A., Liu, F., Wang, G.: Android Collusive Data Leaks with Flow-sensitive DIALDroid Dataset. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, pp. 71–85. ACM, Abu Dhabi United Arab Emirates (2017). https://doi.org/10.1145/3052973.3053004
107. Klieber, W., Flynn, L., Bhosale, A., Jia, L., Bauer, L.: Android taint flow analysis for app sets. In: Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis, Edinburgh United Kingdom: ACM, June 2014, pp. 1–6. https://doi.org/10.1145/2614628.2614633.
108. Arzt, S., et al.: FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. SIGPLAN Not. **49**(6), 259–269 (2014). https://doi.org/10.1145/2666356.2594299
109. Rasthofer, S., Arzt, S., Bodden, E.: A machine-learning approach for classifying and categorizing android sources and sinks. In: Proceedings 2014 Network and Distributed System Security Symposium, Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2014.23039 (2014)
110. Sas, D., Bessi, M., Arcelli Fontana, F.: Automatic Detection of Sources and Sinks in Arbitrary Java Libraries. In: 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM), September 2018, pp. 103–112. https://doi.org/10.1109/SCAM.2018.00019 (2018)
111. Chen, S., Zhang, Y., Fan, L., Li, J., Liu, Y.: AUSERA: Automated Security Vulnerability Detection for Android Apps. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, in ASE '22. New York, NY, USA: Association for Computing Machinery, January 2023, pp. 1–5. https://doi.org/10.1145/3551349.3559524 (2023)
112. Manning, C.D.: Human language understanding & reasoning. Daedalus **151**(2), 127–138 (2022). https://doi.org/10.1162/daed_a_01905
113. Hou, X., et al.: Large language models for software engineering: a systematic literature review, ACM Trans Softw Eng Methodol, December 2024, vol. 33, no. 8, pp. 1–79. https://doi.org/10.1145/3695988 (2024)
114. Cheshkov, A., Zadorozhny, P., Levichev, R.: Evaluation of ChatGPT Model for Vulnerability Detection, 12 April 2023, arXiv: arXiv:2304.07232. https://doi.org/10.48550/arXiv.2304.07232 (2023)
115. Asare, O.: Security Evaluations of GitHub's Copilot, August 2023. http://hdl.handle.net/10012/19675. Accessed 18 March 2025. [Online].
116. Wang, J., Huang, Z., Liu, H., Yang, N., Xiao, Y.: DefectHunter: a Novel LLM-Driven Boosted-Conformer-based Code Vulnerability Detection Mechanism, 27 September 2023, arXiv: arXiv:2309.15324. https://doi.org/10.48550/arXiv.2309.15324 (2023)
117. Zhang, C., Liu, H., Zeng, J., Yang, K., Li, Y., Li, H.: Prompt-enhanced software vulnerability detection using ChatGPT. In: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, in ICSE-Companion '24. New York, NY, USA (2024). Association for Computing Machinery, May 2024, pp. 276–277. https://doi.org/10.1145/3639478.3643065.
118. Wei, H., Lin, G., Li, L., Jia, H.: A context-aware neural embedding for function-level vulnerability detection. Algorithms **14**(11), 11 (2021). https://doi.org/10.3390/a14110335
119. Mahyari, A.: A hierarchical deep neural network for detecting lines of codes with vulnerabilities. In: 2022 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C), Dec. 2022, pp. 1–7. https://doi.org/10.1109/QRS-C57518.2022.00011 (2022)
120. Lin, G., et al.: Cross-project transfer representation learning for vulnerable function discovery. IEEE Trans. Ind. Inform. **14**(7), 3289–3297 (2018). https://doi.org/10.1109/TII.2018.2821768
121. Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep learning based vulnerability detection: are we there yet? IEEE Trans. Softw. Eng. **48**(9), 3280–3296 (2022). https://doi.org/10.1109/TSE.2021.3087402

122. Malware detection in android based on dynamic analysis. In: 2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security), IEEE, London, UK ResearchGate (2017). https://doi.org/10.1109/CyberSecPODS.2017.8074847.
123. Misalkar, H.D., Harshavardhanan, P.: TDBAMLA: temporal and dynamic behavior analysis in Android malware using LSTM and attention mechanisms. Comput. Stand. Interfaces **92**, 103920 (2025). https://doi.org/10.1016/j.csi.2024.103920
124. Yuan, B., et al.: Enhancing deep learning-based vulnerability detection by building behavior graph model. In: Proceedings of the 45th International Conference on Software Engineering, in ICSE '23, pp. 2262–2274. IEEE Press, Melbourne, Victoria, Australia (2023) https://doi.org/10.1109/ICSE48619.2023.00190.
125. Li, Z., Li, H., Meng, L.: Model compression for deep neural networks: a survey. Computers **12**(3), 60 (2023). https://doi.org/10.3390/computers12030060
126. Jedrzejewski, F.V.: Threat Modeling of ML-intensive Systems: Research Proposal. In: Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering—Software Engineering for AI, in CAIN '24. New York, NY, USA: Association for Computing Machinery, June 2024, pp. 264–266. https://doi.org/10.1145/3644815.3644975 (2024)
127. DARPA's Explainable Artificial Intelligence (XAI) Program. ResearchGate. https://doi.org/10.1609/aimag.v40i2.2850 (2024)
128. Phillips, P.J., et al.: Four principles of explainable artificial intelligence. NIST. https://www.nist.gov/publications/four-principles-explainable-artificial-intelligence (Sep. 2021). Accessed 25 Mar 2025 [Online].
129. Srivastava, G., et al.: XAI for Cybersecurity: State of the Art, Challenges, Open Issues and Future Directions, arXiv: arXiv:2206.03585. https://doi.org/10.48550/arXiv.2206.03585 (2022)
130. Nguyen, T.N., Choo, R.: Human-in-the-Loop XAI-enabled Vulnerability Detection, Investigation, and Mitigation. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Nov. 2021, pp. 1210–1212. https://doi.org/10.1109/ASE51524.2021.9678840 (2021)
131. Wijekoon, A., Wiratunga, N.: A user-centred evaluation of DisCERN: discovering counterfactuals for code vulnerability detection and correction. Knowl.-Based Syst. **278**, 110830 (2023). https://doi.org/10.1016/j.knosys.2023.110830
132. Francy, S., Singh, R.: Edge AI: Evaluation of Model Compression Techniques for Convolutional Neural Networks. 02 Sep 2024, arXiv: arXiv:2409.02134. https://doi.org/10.48550/arXiv.2409.02134.
133. Ngo, D., Park, H.-C., Kang, B.: Edge intelligence: a review of deep neural network inference in resource-limited environments. Electronics **14**(12), 2495 (2025). https://doi.org/10.3390/electronics14122495
134. Tan, F., et al.: MobileQuant: Mobile-friendly Quantization for On-device Language Models. In: Al-Onaizan, Y., Bansal, M., Chen, Y.-N. (eds.) Findings of the Association for Computational Linguistics: EMNLP 2024, pp. 9761–9771. Miami, Florida, USA (2024) Association for Computational Linguistics, Nov. 2024. https://doi.org/10.18653/v1/2024.findings-emnlp.570.
135. Molina-Coronado, B., Ruggia, A., Mori, U., Merlo, A., Mendiburu, A., Miguel-Alonso, J.: Light up that Droid! On the effectiveness of static analysis features against app obfuscation for Android malware detection. J. Netw. Comput. Appl. **235**, 104094 (2025). https://doi.org/10.1016/j.jnca.2024.104094
136. Sun, Y., et al.: A transformer based malicious traffic detection method in android mobile networks. In: Sheng, Q.Z., Dobbie, G., Jiang, J., Zhang, X., Zhang, W. E., Manolopoulos, Y., Wu, J., Mansoor, W., Ma, C. (eds.) Advanced Data Mining and Applications, pp. 370–385. Springer Nature, Singapore (2025) https://doi.org/10.1007/978-981-96-0821-8_25.
137. Prates, L., Pereira, R.: DevSecOps practices and tools. Int. J. Inf. Secur. **24**(1), 11 (2024). https://doi.org/10.1007/s10207-024-00914-z
138. Alugunuri, N.: AI for continuous security in DevOps (DevSecOps): integrating machine learning into CI/CD pipelines. Int. J. Intell. Syst. Appl. Eng. **12**(23s), 3048–3062 (2024)
139. Nazir, A., et al.: Evaluating energy efficiency of buildings using artificial neural networks and k-means clustering techniques. In: 2020 3rd International Conference on Computing, Mathematics and Engineering Technologies (iCoMET), Jan. 2020, pp. 1–7. https://doi.org/10.1109/iCoMET48670.2020.9073816 (2020)
140. Krutz, D.E., Munaiah, N., Peruma, A., Wiem Mkaouer, M.: Who Added That Permission to My App? An Analysis of Developer Permission Changes in Open Source Android Apps. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), May 2017, pp. 165–169. https://doi.org/10.1109/MOBILESoft.2017.5. (2017)
141. Geiger, F.-X., Malavolta, I., Pascarella, L., Palomba, F., Di Nucci, D., Bacchelli, A.: A graph-based dataset of commit history of real-world Android apps. In: Proceedings of the 15th International Conference on Mining Software Repositories, in MSR '18. Association for Computing Machinery, May 2018, pp. 30–33. New York, NY, USA (2018) https://doi.org/10.1145/3196398.3196460.
142. Namrud, Z., Kpodjedo, S., Talhi, C.: AndroVul: a repository for Android security vulnerabilities, In: Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, in CASCON '19, November 2019, pp. 64–71. USA, IBM Corp (2019)
143. Allix, K., Bissyandé, T. F., Klein, J., Le Traon, Y.: AndroZoo: collecting millions of Android apps for the research community, In: Proceedings of the 13th International Conference on Mining Software Repositories, in MSR '16, New York, NY, USA (2016). Association for Computing Machinery, May 2016, pp. 468–471. https://doi.org/10.1145/2901739.2903508.
144. JEMMA: An extensible Java dataset for ML4Code applications. |Empirical Software Engineering. https://link.springer.com/article/https://doi.org/10.1007/s10664-022-10275-7. Accessed 18 Mar 2025 [Online]
145. Geiger, F.-X., Malavolta, I.: Datasets of android applications: a literature review. (2018). arXiv:1809.10069. https://doi.org/10.48550/arXiv.1809.10069.
146. Qiu, J., Zhang, J., Luo, W., Pan, L., Nepal, S., Xiang, Y.: A survey of Android malware detection with deep neural models. ACM Comput. Surv. **53**(6), 1–36 (2020). https://doi.org/10.1145/3417978
147. Zhang, X., Breitinger, F., Luechinger, E., O'Shaughnessy, S.: Android application forensics: a survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. Forensic Sci. Int. Digit. Investig. **39**, 301285 (2021). https://doi.org/10.1016/j.fsidi.2021.301285
148. Wu, T., Breitinger, F., O'Shaughnessy, S.: Digital forensic tools: recent advances and enhancing the status quo. Forensic Sci. Int. Digit. Investig. **34**, 300999 (2020). https://doi.org/10.1016/j.fsidi.2020.300999
149. Mirzaei, O., de Fuentes, J.M., Tapiador, J., Gonzalez-Manzano, L.: AndrODet: an adaptive Android obfuscation detector. Future Gener. Comput. Syst. **90**, 240–261 (2019). https://doi.org/10.1016/j.future.2018.07.066
150. Li, Z., Sun, J., Yan, Q., Srisa-an, W., Tsutano, Y.: Obfusifier: Obfuscation-Resistant Android Malware Detection System. In: Chen, S., Choo, K.-K. R., Fu, X., Lou, W., Mohaisen, A. (eds.) Security and Privacy in Communication Networks, pp. 214–234. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-37228-6_11.

151. Pasetto, M., Marastoni, N., Preda, M. D.: Revealing similarities in android malware by dissecting their methods. In: 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), pp. 625–634. https://doi.org/10.1109/EuroSPW51379.2020.00090(2020).
152. The rise of obfuscated Android malware and impacts on detection methods [PeerJ]. https://peerj.com/articles/cs-907/ (2022). Accessed 18 Mar 2025. [Online].
153. Software Assurance Reference Dataset (SARD) Manual. NIST. https://www.nist.gov/itl/ssd/software-quality-group/software-assurance-reference-dataset-sard-manual (2021). Accessed 18 March 2025. [Online].
154. Juliet C/C++ 1.3. NIST Software Assurance Reference Dataset. https://samate.nist.gov/SARD (2011). Accessed 18 Mar 2025. [Online].
155. Allamanis, M.: The adverse effects of code duplication in machine learning models of code In: Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. https://dl.acm.org/doi/abs/https://doi.org/10.1145/3359591.3359735 (2019). Accessed 18 Mar 2025 [Online].
156. Giray, G.: A software engineering perspective on engineering machine learning systems: state of the art and challenges. J. Syst. Softw. **180**, 111031 (2021). https://doi.org/10.1016/j.jss.2021.111031
157. Wan, Z., Xia, X., Lo, D., Murphy, G.C.: How does machine learning change software development practices? IEEE Trans. Softw. Eng. **47**(9), 1857–1871 (2021). https://doi.org/10.1109/TSE.2019.2937083