JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

Detecting Android App Collusion: A Comparative Study of Dynamic Analysis and Machine Learning Approaches

Anshpreet Singh[1], Karandeep Singh[2]
[1]Research Scholar, Department of Computer Science and Engineering, Punjabi University Patiala, India
[2]Assistant Professor, Department of Computer Science and Engineering, Punjabi University Patiala, India

Abstract
The paper reviews the detection of app collusion in Android, where the menace involves two or more apps working together to circumvent the permission-based security model of the system. Simple security tools, which consider operating alone, would thus miss identifying such coordinated acts of malice. The paper discusses dynamic analysis and machine learning (ML) techniques as potent candidates for the detection of app collusion. Among dynamic analysis techniques, inter-app communication monitoring, runtime taint tracking, and sensor signal analysis are elaborated for their capability to acquire system metadata in real-time and detect overt and covert channels of collusion. The paper presents an overview of machine-learning techniques that rely upon features based on permission sets, communication graphs, and behavioral logs for detection, with some techniques reporting an accuracy of up to 97.5%. This analysis of different approaches will enable their capabilities and limitations to be distinguished so that the potentiality of hybrid approaches that extend dynamic analysis by ML is highlighted. The review also states open challenges, such potential problems as the scarcity of data, resource efficiency, and adaptive attacks, as well as explaining AI, and future research proposals to improve Android app collusion detection.

Keywords: Android security, App collusion, Dynamic analysis, Machine learning, Inter-app communication

1. INTRODUCTION

1.1 Android's Permission-Based Model

An Android operating system, working across billions of devices, adopts a permission-based security model aimed at protecting users' privacy and limiting the ability of applications to infringe upon device integrity. In this model, applications request permissions in their manifest files to access certain sensitive resources, which include location information, contacts, or networking capabilities. These permissions are then notified to the user as the application installs, thereby allowing the user to make decisions on what powers the app has. Nevertheless, the model assumes that every single app will work independently and can be exploited in the form of app collusion. App collusion occurs when two or more applications coordinate to carry out malicious activities that would otherwise be prohibited if only performed by a single application by circumventing Android's permission constraints [1]. For example, the first application accesses some sensitive data, such as location of the user, and shares the same with the second comes along and more with the permission of network access, thereby exfiltrating the data without actually asking user's consent.

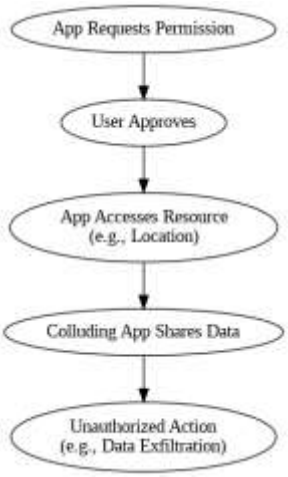

Figure 1: Flowchart of Android's Permission-Based Model and App Collusion

A flowchart depicting how Android's permission-based model operates, including steps like app permission requests, user approval, and how colluding apps bypass these restrictions by sharing data (e.g., one app accessing location data and another transmitting it).

1.2 App Collusion

If turning the tables, by generating co-existing privileges from two apps, would act as a flood gate to stronger malware intentions, then app collusion is truly a serious threat to Android security. Collaboration could progress by way of explicatory inter-app communication mechanisms such as Intents, Content Providers, or Binder calls; on the otherwise, a covert channel could be plotted using sensor signals or timing-based communications [2]. An exemplary collusion case-the Soundcomber bait-where it used audio signals from a mic of the device to leak sensitive information like call tones-just goes to show how stealthy these attacks can get [7]. Additionally, cross-update privacy leaks, where new methods of leaking private information are made possible by app updates, can be made easier by collusion [26]. Colluding apps, on the other hand, are often analyzed along separately and found to be benign, as their evil deeds show up only in their concerted activities, and therein lies their challenge. Hence, it becomes a very grave issue of going after activities, thus making collusion dangerous indeed, since traditional methods look at single-app behavior, isolate it and treat it accordingly [2].

1.3 Problem Statement

Detecting application collusion poses an extraordinary challenge simply because the majority of current security realized tools work by analyzing applications in isolation-they miss out on inter-app interactions that characterize collusion [28]. According to Abro et al., commercial malware detectors today work by evaluating only one app and so miss collaborative or synergistic actions that characterize collusion [2]. Static analysis argues with dynamic behaviors like code loading during runtime or obfuscation and has notorious false-positive characterization when searching for collusion. For instance, Elish et al., discovered the 55% false-positive rate by porting falsely positive benign app pairs through static analysis tools such as XManDroid, bringing into question the ability of static approach methods to detect collusion [3]. These difficulties present the crucial gap for solutions that monitor runtime interaction as well as view complicated behavior patterns that signify collusion.

1.4 Scope of This Review

This review deals exclusively with dynamic analysis- and machine learning (ML)-based techniques for app collusion detection on Android since they run better to circumvent static analysis. Dynamic analysis simply watches the app behavior at runtime, enabling the capture of their interactions such as inter-app communication or covert channels, which static analysis may overlook [3]. For example, it is paramount to accurately classify inter-component communications (ICC) to recognize collusive behavior since static techniques usually cannot inspect these interactions at such a granular level [3]. Alternatively, ML methods are excellent at spotting collusion since they can learn complex patterns from data, be it app permission usage, communication graphs, or behavior logging [5]. These methods promise detection of all kinds of Android malware, including those that work through collusion, using both static and dynamic features [5, 20].

Table 1: Key Challenges in App Collusion Detection

Challenge	Description	Relevant References
Isolated Analysis	Traditional tools analyze apps individually, missing inter-app interactions.	[2]
Dynamic Behaviors	Static analysis struggles with runtime code loading and obfuscation.	[3]
High False Positives	Static methods like XManDroid produce high false-positive rates (e.g., 55%).	[3]
Stealthy Communication Channels	Collusion often uses covert channels (e.g., sensors, timing) that are hard to detect.	[2, 7]

2. ANDROID APP COLLUSION OVERVIEW

2.1 Inter-App Communication (IPC) Mechanisms

To fully grasp the concept of the collusion, it is essential to understand how the Android architecture provides mechanisms for processes to communicate among various applications for legitimate purposes of data sharing and functionality. Two or more malicious apps are said to be in collusion if they undertake activities that confer privileges on them that are beyond their own set of permissions, thereby circumventing Android's permission-based security model [1]. Generally speaking, these IPCs involved in collusion types are:

Intents: Intents are messaging objects through which the requesting application can prompt an action from other components, such as starting activities, broadcasting messages, or calling services. Colluding apps may misuse Intents to transfer sensitive information or data; one app could, for instance, send location information to another with internet access [2].

Content Providers: These allow one app to share structured data with another. Malicious apps can write sensitive data to a Content Provider, whereas the colluding app possessing different permissions may retrieve and exfiltrate the data [6].

Binder: The Binder framework is useful for app processes to directly communicate for more complex interactions. Colluding apps may exchange data clandestinely through Binder calls, sidestepping permission checks [6].

Services: Applications can keep bound to services offered by other applications, so as to compel them to do a task. The possibility for maleficent acts arises here when one application forces the leakage of private data by means of a service offered by another [2].

These IPC mechanisms exist as a prerequisite to Android functionality; however, they provide another path for colluding apps to share data and coordinate at times when they would indeed be prevented. Marforio et al. proved that these communication channels do exist on the current-day smartphones and can be turned for collusion; the demonstration showed how apps can exchange data with each other via Intents and Content Providers at the very low cost of overhead [6]. In their thorough analysis of the risks, fixes, and difficulties related to Android Inter-App Communication, Wang and Wu [27] emphasize how they contribute to collusion.

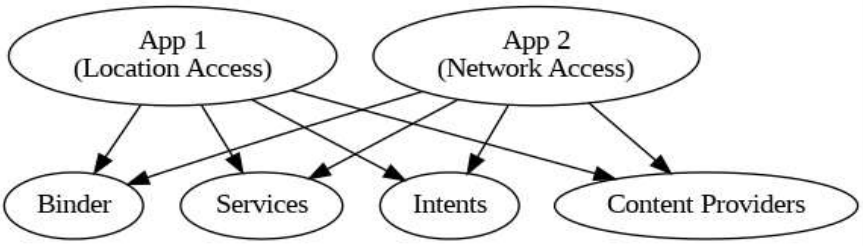

Figure 2: Diagram of Inter-App Communication (IPC) Mechanisms

A diagram illustrating Android's IPC mechanisms (Intents, Content Providers, Binder, Services) and how they can be exploited for collusion, showing data flow between apps with complementary permissions.

2.2 Types of Communication Channels

Colluding apps abuse overt and covert channels of communications to enable unauthorized data flows. Such channels differ in their visible level, each posing peculiar challenges for security analysis.

Overt Channels

In overt channels, Android IPC mechanisms are used-for legitimate inter-App interaction but may be utilized for collusion. For instance, the following shows:

Shared files: An application can write data into shared storage, i.e., external storage, which can be then read by any other application. This method is simple to implement, yet quite obvious if an investigator uses file access monitoring [2].

Intents: Intents are said to facilitate interactions where apps send messages to each other or pass data. For example, the contacts-app can access the data and send it to a colluding app that has network permissions [6].

Content providers: These allow sharing information in a more structured way- say, through a shared database. Colluding pairs of apps may resort to Content Providers to exchange sensitive information like user credentials in the absence of direct permission complementation [6].

Being overt makes these channels relatively easier to monitor because they are well-defined Android APIs. Yet their legitimate use in benign apps makes it difficult to separate malicious behavior from normal behavior [2].

Covert Channels

Covert channels operate by exploiting avenues never intended to act as means of communications, and thus become a lot trickier to detect by system administrators. Such channels operate with low bandwidth, which nevertheless adequately serves the leaking of crucial data from inside. Some examples include:

Sensor Signals: An application may encode some data into some sensor outputs such as audio signals picked up via a microphone or accelerometer readings. Attacks like Soundcomber used audio signals to encode and transmit sensitive information, for instance, call tones, thus lending a new way of clandestine data leakage [7].

Timing channels: Apps may encode messages in the times at which certain events occur, in deliberate delays in processing or generation of network requests, for example. One app might modulate CPU usage to signal data to another app referencing system resources [6].

Power Consumption: Covert channels utilize changes in power consumption to transmit the data. Heavy resource operations will be turned on and off by the apps in a manner detectable to and colluding with each other [2].

A covert channel is difficult to detect because it does not use standard APIs and rarely leaves any noticeable trace. If Marforio et al. are to be believed, establishing, discovering, or monitoring covert channels is beyond the scope of current advanced analysis tools, which restrict themselves to observing established flows of information: traditional data loss channels, like sensor output or system resource usage [6].

Table 2: Comparison of Overt and Covert Channels

Channel Type	Examples	Ease of Detection	Throughput	Relevant References
Overt Channels	Intents, Content Providers, Shared Files	Moderate (API-based monitoring possible)	High	[2, 6]
Covert Channels	Sensor Signals, Timing, Power Consumption	Difficult (non-standard data flows)	Low	[6, 7]

2.3 Real World Examples

Two real-world instances of scenarios underline the seriousness and stealthiness of the application collusion. The Soundcomber attack became very well-known for demonstrating how two colluding applications might leak sensitive information over audio covert channels [7]. In those attacks, an application with access to the microphone would be listening to audio signals such as call tones or ambient sounds and would encode sensitive data-for instance, phone numbers or credit card numbers-within those signals. A colluding application, with network access, would then decode the data and transmit them towards a remote server. The stealth nature of the attack is provided by the use of the microphone, which is most commonly disregarded being associated with data exfiltration by the traditional security tools [7]. Through accurate Inter-App analysis, Wang et al. [28] further demystify collusion attacks by exposing sophisticated techniques employed by colluding apps.

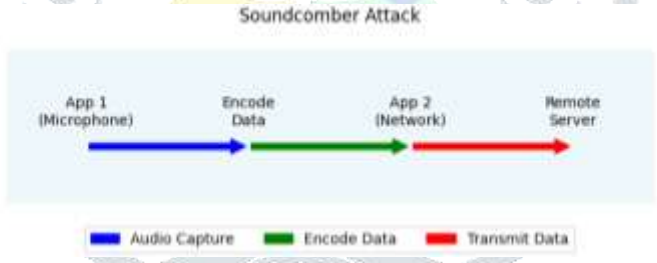

Figure 3: Illustration of the Soundcomber Attack

A diagram showing the Soundcomber attack, where one app captures audio signals (e.g., call tones) and encodes sensitive data, which a colluding app with network access decodes and transmits.

Another example involves the use of Content Providers for sharing sensitive data. Abro et al. describe instances in which one app writes user data to a shared Content Provider such as location or contacts. A colluding app having network permissions retrieves this data and sends it outside [2]. These examples show that colluding apps may be able to exploit both overt and covert channels to achieve malicious intentions when their individual behaviors look innocuous.

2.4 Challenges in Detecting App Collusion

Detecting app collusion presents more challenges that call for specialized analysis techniques:

Benign Appearance of Individual Apps: Colluding apps are designed to appear benign when considered in isolation because each app operates within the permissions it declares. Their maliciousness only reveals itself when the apps interact in coordination, ergo any attempts of traditional single-app analysis fail [2].

Stealthy Communication: Collusion sometimes employs covert communication channels with a very low throughput, like sensor signals or timing information-share such minimal information that they leak-sensitive data-but are quite difficult to observe since they are subtle in nature. For instance, the Soundcomber attack uses the audio channel to produce very little observable activity, thus evading the standard monitoring tools [7].

Complexity of Interplay: There are so many different IPCs and covert channels that detection is made very difficult. Application programs may use several channels in parallel and require monitoring of conventional APIs along with non-standard flows of information [6].

Evasion Techniques: Sophisticated colluding apps may employ evasion techniques like delaying communication or detecting the analysis environment to avoid getting caught between limited monitoring windows [3].

These challenges imply the need for advanced countermeasures that make use of dynamic analysis and machine learning to track run-time interactions and tease out complex patterns that suggest collusion [3,5]. The following sections examine these methods in detail with a focus on how they can help battle the unique challenges posed by app collusion.

3. DYNAMIC ANALYSIS TECHNIQUES FOR COLLUSION DETECTION

One cannot underestimate the importance of dynamic analysis for collusion detection in Android apps, whereas it relies on running the apps to capture their real-time behavior and interactions, often missed by static ones. This technique is immensely suitable for detecting collusion as the colluding apps typically seem benign when analyzed individually-the evil really is in their coordinated interaction. Dynamic analysis checks run-time activities to detect behaviors that static analysis is unable to pinpoint, be it dynamically loaded code, covert communication channels, and so on [3, 5]. However, the greatest challenges to dynamic analysis arise from a heavy runtime overhead, inability to cover all possible execution paths, and a skillful malware that could perhaps evade by changing its behavior in the analysis environment [3, 5, 6].

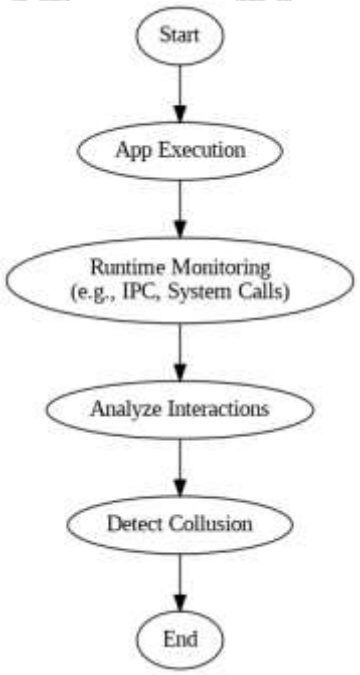

Figure 4: High-Level Flowchart of Dynamic Analysis Process

A flowchart outlining the dynamic analysis process for collusion detection, including app execution, runtime monitoring (e.g., IPC, system calls), and analysis of interactions.

3.1 Inter-App Communication Monitoring

The observation of inter-app communications is a basic step in performing dynamic analysis for app collusion since these communications serve as the method of coordinating malfeasance by colluding apps. IPC mechanisms of Android such as Intents, Binder calls, and Content Providers are the most common methods exploited for app collusion. By understanding what is in the communications and the frequency and manner in which interactions take place, unusual behaviors may be flagged and hence the unusual correlation may result in the identification of collusion. For example, one app may have the right to access location data and then sends it out via an Intent to a second application that has network access, thereby enabling data exfiltration that is not authorized [3]. Guan et al. [29] suggested a framework especially made to identify hidden partners in app collusion in order to overcome these difficulties and improve the precision of collusion detection.

Figure 5: Sequence Diagram for Inter-App Communication Monitoring

This sequence diagram is a runtime monitoring aspect of communication channels between apps (Intents, Binder calls) and interaction among Apps and the Monitoring system.

Tools like XManDroid usually use permission-based policies to monitor communication channels at runtime, and they more often than not, generate many false positives. According to Elish et al., XManDroid misclassified 55% of benign app pairs from colluding, exemplifying how hard it is for permission-based approaches to distinguish between legitimate and illegal communications [3]. Other more sophisticated tools such as TaintDroid monitor data flows between apps by dynamically tracking taints, but Marforio et al. established that TaintDroid does not detect all communication channels- especially the covert ones- so the collusion threat may no longer be adequately addressed by today's solutions [6]. Their experiments further showed that communication channels- both overt (Intents, Content Providers) and covert (sensor signals)- could achieve throughputs ranging from 0.47 bps to 4.22 kbps, sufficient enough to leak extremely sensitive information such as GPS coordinates or contact lists [6].

The best way to increase detection accuracy is by monitoring suspicious behavior, such as permission-complementary acts, where an app accesses a resource (for example, GPS), and the other performs an action (for instance, network transmission) for it to be malicious [2]. Such monitoring should cover all the different ways colluding apps could communicate, including the traditional IPC mechanisms and the "non-standard" ones.

Table 3: Inter-App Communication Monitoring Techniques

Technique	Description	Strengths	Limitations	References
Intent Monitoring	Tracks Intent broadcasts to detect data sharing between apps.	Captures explicit communications; API-based monitoring is feasible.	High false positives; misses covert channels.	[3, 6]
Binder and Content Provider Monitoring	Observes Binder calls and Content Provider access for unauthorized data flows.	Detects structured data sharing; integrates with system logs.	Limited to API-based interactions; complex to differentiate benign from malicious.	[6]
Permission-Complementary Analysis	Identifies apps with complementary permissions enabling malicious actions.	Targets collusion-specific patterns; reduces false negatives.	Requires precise behavioral modeling; computationally intensive.	[2]

3.2 Runtime Taint and System Behavior Monitoring

Detection of data leaks in Android applications is one of the special fields in which dynamic taint analysis grows to be foremost, especially in cases of collusion between the applications. Such systems label tainted objects in memory and propagate them via API calls across apps, finding when data flow occurs wrongly between colluding apps [1, 6]. Assume TaintDroid is running so that it can detect when an app that has access to contacts data actually shares that data with an app that sends it elsewhere. However, Marforio et al. remarked that such sophisticated taint tracking systems may end up missing covert channels of communication, like those based on sensor signals or timing-based ones, which in turn may call for other monitoring solutions [6]. Additionally, Alhanahnah et al. [25] highlighted the significance of taking such situations into account in collusion detection by concentrating on identifying vulnerabilities in Android Inter-App communication, particularly in dynamically loaded code.

In complement of taint tracking, monitoring system-level behaviors can reveal anomalies of collusion. For example, weird patterns of network activities or resource sharing between apps can be signs of colluded malicious behaviors [13, 14]. Han et al. proposed a real-time malware detection system based on analyses of network traffic, which could be exploited in collusion scenarios to detect coordinated network

activities between apps [13]. Similarly, Kim and Choi used multivariate time-series approaches to check system metrics, allowing them to speculate that such techniques could detect anomalies in resource usage indicative of collusion [14].

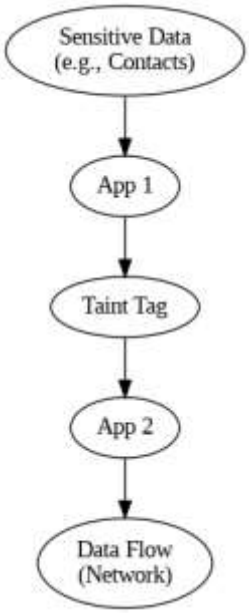

Figure 6: Diagram of Runtime Taint Tracking

A diagram illustrating how taint tracking monitors data flows across apps, showing how sensitive data (e.g., contacts) is tagged and tracked to detect unauthorized sharing.

Casolare et al. use advanced techniques in which model checking is performed on system traces to verify whether apps conform to expected behavior. Modeling apps as automata with temporal properties allows for the detection of collusion logic, such as coordinated data flow to shared resources, thereby furnishing a formal method of locating malicious interactions [18]. Additionally, running apps in controlled settings with instrumented APIs (i.e., wrapping SMS or network APIs) via sandboxing or emulation environments enriches the detection.

3.3 Sensor and Audio Signal Analysis

Detection of collusion between apps using sensor and audio signal analysis is the field that justifies the innovation in methods by using unconventional data sources. In other words, Casolare et al. introduced a very interesting idea that consists of converting executables of Android apps into audio files and processing the latter with machine learning approaches to extract features-the eventual classification of collusive behavior [16]. This may capture some patterns that might escape the traditional analysis, thus giving a rarity to the means through which we seek to detect underground interactions between colluding apps.

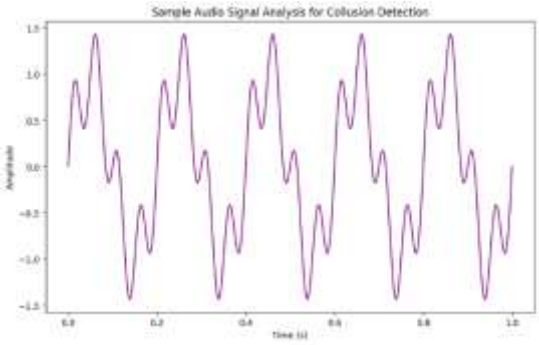

Figure 7: Graph of Sensor and Audio Signal Analysis

A graph showing how sensor or audio signals (e.g., microphone or accelerometer outputs) are analyzed to detect covert channels, possibly including signal patterns or frequency analysis.

Sensor fingerprinting techniques exploit the outputs from sensors, such as microphone or accelerometer, to construct behavioral profiles of apps. Security tools can essentially detect anomalous patterns revealing clandestine collusion, especially coordinated sensor access, or data encoding, when analyzing these profiles [2, 7]. A classic example is the Soundcomber attack, in which two colluding apps made use of audio signals captured via the microphone as a channel for encoding and transmitting sensitive information like call tone or credit card data, thereby showcasing the possibility of sensor-based channels for covert communication [7]. These methods seem promising but have to be painstakingly calibrated with due consideration for distinguishing between malicious and benign behaviors, especially in noisy environments and the variable nature of sensor characteristics.

Table 4: Sensor and Audio Signal Analysis Techniques

Technique	Description	Strengths	Limitations	References
Audio Signal Analysis	Converts app executables into audio files for feature extraction and ML-based detection.	Captures unique behavioral patterns; effective for stealthy collusion.	Requires complex preprocessing; sensitive to noise.	[16]
Sensor Fingerprinting	Uses sensor outputs (e.g., microphone, accelerometer) to profile app behavior.	Detects covert channels; non-traditional approach.	Calibration challenges; high false positives in noisy environments.	[2, 7]

3.4 Challenges in Dynamic Detection

Dynamic analysis, even with all of its strengths, is faced with several challenges, each limiting the analysis's effectiveness in detecting app collusion. First, it highly depends on the execution environment and user input. Collusion might occur only under very special situations or user actions; thus, triggering and observing all possible scenarios during the analysis might become very difficult [3]. Being conditioned on the environment may yield false negatives, where collusion remains undetected because the conditions were never met.

Second, dynamic analysis introduces significant overhead to the performance. Instrumenting the apps or even the system to track some interactions might reduce execution speed and consume plenty of resources, including battery power and memory. Such overheads would discourage methods aimed at real-time detection on resource-constrained mobile devices [5], but this substantially limits their potential for mass deployment.

The third kind of malfunction could be an evasion technique employed by sophisticated malware. Colluding apps may detect analysis environments, such as an emulator or a debugger, and when this occurs, they have the capacity to change behavior, e.g., delay performing malicious actions or even communicate maliciously only when they are not under observation. Marforio et al. underline that such evasion abilities hinder the collusion detection since applications may appear to be innocent under the analysis but may actually perform some mischievous activities when in normal operation [6].

The resolution of these issues lies in the design of more effective dynamic analysis techniques or in improving the dynamic method by incorporating the static one and/or machine learning to increase the coverage and reliability of app collusion detection methods.

Table 5: Challenges in Dynamic Detection

Challenge	Description	Impact	References
Environment Dependence	Results vary with device state or user input; hard to trigger all collusion scenarios.	Leads to false negatives; incomplete coverage.	[3]
Performance Overhead	Instrumentation slows apps and consumes resources.	Limits real-time detection on devices.	[5]
Evasion Techniques	Malware detects analysis environments and alters behavior.	Reduces detection accuracy; increases false negatives.	[6]

4. MACHINE LEARNING APPROACHES TO COLLUSION DETECTION

ML methods have recently emerged as effective tools enabling detection of Android app collusion by learning intricate patterns of inter-app communication that rule-based systems often cannot identify. In short, ML can generalize detection of unfamiliar collusion scenarios, bringing forth subtle relations amongst apps that could point to their malicious coordination [20]. They can utilize both static features, such as permission declarations, or dynamic features like API calls at runtime or sensor usages to model app behavior. This survey reviews supervised and unsupervised ML approaches that were found able to tackle the challenges posed by collusion detection, where apps appear to be benign individually but act maliciously when combined [2, 5]. However, promising these approaches, ML faces challenges such as the at-times scarce labeled collusion data sets, an expensive cost of feature extraction, and the interpretability of complex models [20]. The use of machine learning techniques for Android malware detection has been extensively investigated. While Sk and Anu [24] presented a hybrid deep learning model, Kaur et al. [23] suggested a system utilizing machine learning techniques. Aldhafferi [31] used support vector regression for dynamic feature analysis, and Zhang et al. [30] created a multimodal pre-training technique.

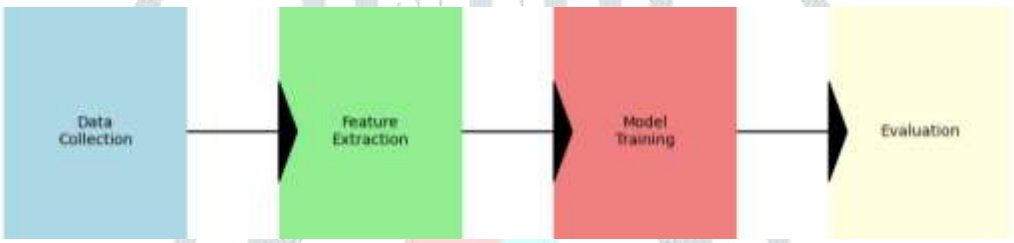

Figure 8: Block Diagram of ML Pipeline for Collusion Detection

A block diagram showing the machine learning pipeline, from data collection (e.g., app logs, permissions) and feature extraction (e.g., permission-based, ICC graphs) to model training and evaluation.

4.1 Feature Engineering for Collusion

Feature engineering is the most crucial phase of ML-based collusion detection since it transforms raw app data into meaningful inputs for the ML models. Good features should embody the characteristics of colluding apps, such as communication or complementary permission usages. The following are commonly used feature categories:

Permission-Based Features

Usually, colluding apps request complementary dangerous permissions so they can combine abilities which an app on its own could never achieve. For instance, one app might request ACCESS_FINE_LOCATION for precise GPS data, and another might ask for INTERNET to send that data outside the device, thereby facilitating unauthorized data leaks [2]. An ML model uses features based on permissions including the number of permissions, number of dangerous permissions, or binary vectors representing the sets of permissions to detect suspicious app pairs. These features are all static; they are extracted from the app's manifest file and are outright effective because a single set of permissions suggests the potential for collusion without carrying out runtime analysis [20]. However, solely depending on permissions may gather misdetections, as benign apps might also share permissions for legitimate purposes [3].

Intent/ICC Graph Features

Inter-App Communication (IAC), a feature provided by Android mechanisms like Intents, serves as a standard channel for collusive behavior. ML models characterized the communication graph representing which apps sent or received Intents and what kinds of actions were involved [3]. Features it considered included Intents sent or received, Intent action categories, or graph metrics such as node centrality or edge density. These features may help expose suspicious communication patterns that might be collusive, such as the frequent exchange of data between apps with complementary permissions [2]. The extraction of ICC features is often computationally expensive since static analysis must be performed to identify potential Intent flows and dynamic monitoring to observe actual communications [6].

Behavioral Features

The behavioral features are dynamic analysis derived; they characterize runtime activities such as API calls, network traffic, or sensor usage. Typical features include n-grams of API calls, standing for sequences of function calls (e.g., accessing contacts followed by network operations), and sequences of network packets, which describe communication patterns [10, 13]. For instance, Canfora et al. used opcode n-

grams for Android malware detection-the same technique could be used to detect collusion by identifying coordinated behavior across apps [10]. On the other hand, Han et al. put forth the network traffic analysis to detect synchronized activity that would be used to identify collusive app pairs [13]. Such features are great at capturing events of dynamic interaction, but they require monitoring of the apps during runtime for a considerable length of time, which creates an added overhead on computation [20].

Audio/Signal Features

With newer approaches, Casolare et al. prepared app executables or runtime behaviors into audio signals and then proceeded to extract numerical features for ML analysis [16]. Converting app binaries into sounds facilitates capturing a behavioral pattern that might otherwise remain hidden from a traditional perspective. Features from these audio signals, such as frequency components or signal amplitude, are used as training data for the ML-based collusion model. The method is especially apt for recognizing the covert channels employed, for instance, in the Soundcomber attack, whereby apps encode data within audio signals [7, 16]. Nonetheless, the approach similarly calls for complex preprocessing and degrees of sensitivity might arise from environmental noises or a variation in devices.

Dataset Sources

The lack of collusion datasets in the real world is a major problem faced by ML-based detection techniques. To tackle this, researchers tend to blend colluding app pairs that are known with benign pairs of apps or synthesize colluding pairs. Blasco et al. came up with a tool to synthesize colluding apps by taking benign ones and facilitating them with malicious interactions-in essence, for example, a benign program accesses sensitive data and shares it with another through some inter-process communication mechanism [21]. Though such synthetic datasets can be really useful for conducting well-controlled experiments, their design might not consider all aspects of real-world collusion. Public datasets are very few in number, and hence, studies resorting to either proprietary datasets or small-scale datasets may not stand good for model generalizability [20].

Table 6: Feature Types for Collusion Detection

Feature Type	Description	Strengths	Limitations	References
Permission-Based	Number or vectors of permissions, focusing on dangerous ones.	Easy to extract; reflects collusion potential.	High false positives; static only.	[2, 20]
Intent/ICC Graph	Communication graph metrics (e.g., Intent frequency, graph density).	Captures inter-app interactions; specific to collusion.	Requires static and dynamic analysis; complex to compute.	[3, 6]
Behavioral	API call n-grams, network traffic, sensor usage patterns.	Reflects runtime behavior; detects dynamic collusion.	High computational cost; needs runtime logs.	[10, 13, 20]
Audio/Signal	Features from app-converted audio signals.	Detects covert channels; innovative approach.	Complex preprocessing; noise sensitivity.	[16]
Dataset Sources	Synthetic or real colluding/benign app pairs.	Enables training; controlled experiments.	Limited real-world data; synthetic data may lack realism.	[21]

4.2 Supervised Learning Methods

Supervised learning methods aim to train a machine learning model using labeled data sets, wherein pairs of apps were labeled as colluding or as benign, information that is then fed into the network detecting algorithms. Normally, other classification algorithms used are SVM, Random Forests, Decision Trees, Naive Bayes, and Neural Networks, all of which balancing trade-offs between accuracy and computational time [20].

Then, Faiz et al. considered a second-step classification procedure in collusion detection. First, Naive Bayes is used over likelihood ratios to treat single apps as either benign or malicious, while, second, the prediction of collusive app pairs was obtained by using logistic regression and perceptrons over features, including permission overlaps and ICC patterns.

This two-stage approach helps decrease false positives: first conducts the filtering to eliminate clearly benign apps and then studies pair-wise interactions [19]. Similarly, Magsi et al. proposed a dynamic framework with two predictive functions to detect collusion and reached an outstanding accuracy of around 97.2% on a set of 800 apps [17]. Their design uses dynamic behavioral properties, such as sequences of API calls, with static permission features to recognize collusive behaviors.

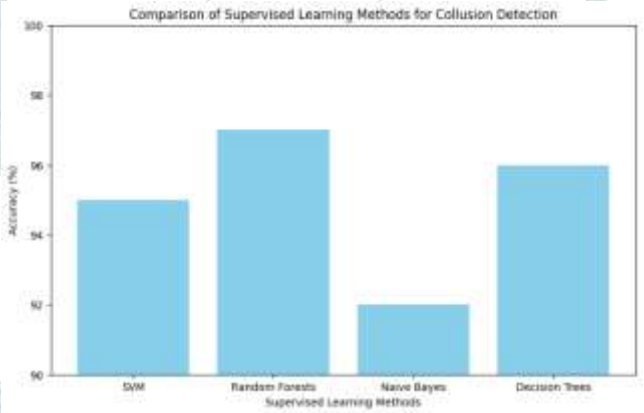

Figure 9: Bar Graph of Supervised Learning Method Performance

A bar graph comparing performance metrics (e.g., accuracy, precision, recall) of supervised learning methods (e.g., SVM, Random Forests, Naive Bayes) for collusion detection [17, 19].

Hybrid classification methods merge clustering with supervised learning to handle partially labeled data. Faiz et al. attempted k-means clustering of apps by behavior, then followed it with SVM classification to pick out colluding pairs from suspicious clusters [19]. This hybrid solution works when very little data is labeled for collusion, using unsupervised clustering to alleviate the burden of needing many labeled examples.

Table 7: Supervised Learning Methods for Collusion Detection

Method	Description	Strengths	Limitations	References
Classification Algorithms	Uses SVM, Random Forests, Naive Bayes, etc., for app pair classification.	High accuracy; well-studied algorithms.	Requires labeled data; sensitive to feature quality.	[19, 20]
Two-Stage Models	Classifies individual apps, then app pairs for collusion.	Reduces false positives; structured approach.	Complex pipeline; needs robust first-stage classifier.	[19]

Predictive Functions	Dynamic framework with predictive functions for collusion detection.	High accuracy (~97.2%); integrates dynamic features.	Dataset-specific; may not generalize.	[17]
Hybrid Classification	Combines k-means clustering with SVM for unlabeled data.	Handles scarce labeled data; flexible.	Clustering errors can affect classification.	[19]

4.3 Unsupervised and Deep Learning Approaches

An ideal situation when labeled collusion data is scarce would involve the use of unsupervised learning methods, such as clustering and anomaly detection. Clustering algorithms like DBSCAN put apps into groups based on similarities: permission usage or communication pattern. Clusters exhibiting uncommon interactions between apps, e.g., frequent Intent exchanges, might indicate potential collusion [20]. In contrast, in anomaly detection, models are trained on what is defined as benign behaviors and the deviations are flagged as possible collusive behaviors; e.g., an app pair suspended with synchronized network activity or sensor access that deviates from benign patterns could be flagged [13].

Deep learning techniques based on RNN-LSTM-like architectures could, in theory, lend themselves to the capturing of complex temporal patterns in raw data-input sequences consisting of API calls or system-level metrics. Such models are fit to account for fine-grained dependencies in the app behavior that could be very useful for a collusion scenario involving sophisticated methods. On the downside, there is a need for huge datasets and high computational power, which may be highly difficult for collusion data owing to its scarcity. So, although particular references lack enough studies on deep learning for collusion detection, Senanayake et al. state that deep learning techniques have been successfully applied to Android malware detection and hence could be applied to collusion detection with some modifications [20].

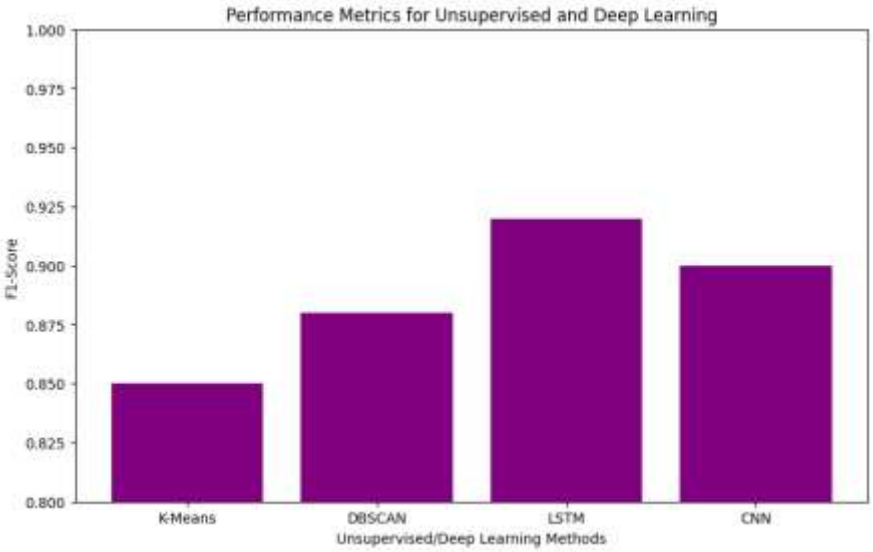

Figure 10: Performance Metrics for Unsupervised and Deep Learning

A bar or line graph showing performance metrics (e.g., precision, recall, F1-score) for unsupervised and deep learning approaches, referencing studies like [20].

4.4 Model Evaluation and Datasets

Evaluation metrics for ML models in collusion detection include accuracy, precision, recall, and F1-score, a balancing act that weighs maximizing true colluder identifications versus minimizing false positives. Casolare et al. claimed an accuracy of roughly 97.5% using an audio signal-based ML approach [16], demonstrating that innovative features can lead to strong performance. However, evaluation is hindered by data imbalances, as colluding pairs-of-apps are far rarer than benign ones, thus producing skewed metrics [20].

A dearth of labeled collusion datasets emerges as another hurdle on the road. Researchers often resort to synthetic datasets, as do Blasco et al., wherein colluding pairs of apps are made by injecting malicious behaviors into benign apps [21]. Alternatively, studies might label a few pairs of known malware as exhibiting collusive behavior, but those are limited notions of collusion and may not cover the whole gamut of real collusion scenarios [20]. Evaluation is made even more difficult by the possibility of overfitting to known collusion patterns and, therefore, the absence of relevant real-world benchmarks-the models might do well on synthetic data but barely make a grade on real cases [20].

Table 8: Evaluation Metrics and Dataset Challenges

Aspect	Description	Impact	References
Evaluation Metrics	Accuracy, precision, recall, F1-score for collusion vs. benign classification.	Balances detection and false positives; high accuracy reported (~97.5%).	[16, 20]
Training Data	Synthetic colluding pairs or labeled malware; limited real-world data.	Enables training but may lack realism; affects generalizability.	[20, 21]
Challenges	Data imbalance, overfitting, lack of benchmarks.	Skews metrics; limits real-world applicability.	[20]

4.5 Advantages and Limitations of ML Methods

There are many potential benefits of ML methods when it comes to collusion detection. They are capable of generalizing to previously unseen collusion behaviors, reducing dependence on explicitly formulating collusion rules. Secondly, they can be introduced to heterogeneous features, such as static permissions or dynamic behaviors, for thorough analysis [20]. These lightweight classifiers may run on-device for real-time detection purposes [19]. Since an ML-based method has a heavy dependency on the training data and that data needs to be representative of all of the relevant collusions, this remains a challenge when collusion datasets are so scarce [20]. Deep learning is one of the powerful modeling techniques; however, they provide very little to no explanation of why the model would flag a particular pair of concerned apps as colluding [20]. Moreover, feature extraction, especially when it comes to dynamic behavioral features, is a very expensive process in terms of the computational time. That thus ruins its real-time capability on resource-constrained devices [5].

Table 9: Advantages and Limitations of ML Methods

Aspect	Description	References
Strengths	Generalizes to unseen patterns; integrates static/dynamic features; lightweight models for on-device use.	[19, 20]
Weaknesses	Relies on training data quality; deep models lack interpretability; feature extraction is costly.	[5, 20]
Real-Time Use	Lightweight classifiers feasible; feature extraction limits scalability.	[5, 19]

5. COMPARATIVE ANALYSIS OF DYNAMIC VS ML APPROACHES FOR ANDROID APP COLLUSION DETECTION

Android app collusion detection consists of analyzing multiple collaborating apps that circumvent permission restrictions and perform malicious activities together. For collusion detection, one must have robust methods capable of identifying behaviors in coordination. Two approaches most dominantly define this field: dynamic analysis, which observes apps at runtime to see how they act and interact in reality, and machine learning (ML), which uses algorithms to extract complicated patterns from a data set so that subtle inter-app relationships can be detected. We will detail the comparison of these approaches in five major dimensions: detection accuracy; overhead and scalability; coverage; adaptability; and deployment feasibility. In examining their pros and cons, we aim to stress their complementary role and the option of hybrid solutions for solving collusion-detection challenges.

5.1 Detection Accuracy

Dynamic Analysis

Some dynamic analysis techniques such as taint tracking and system call monitoring are actually very reliable when it comes to looking for runtime behavior that points at collusion-like activity. Consider, for instance, TaintDroid tracking the flow of sensitive data across app boundaries, detecting unauthorized data sharing—for example, one application accessing the contacts while another sends them away [1]. The mechanisms thus provide pinpoint detection of a known method of collusion, particularly overt channels such as Intents or Content Providers. They are not able to withstand covert communications via sensor or timing channels unless specifically adapted to do so. Marforio et al. [6] show that some channels can bypass even the most advanced interception tools like TaintDroid, giving highs enough to leak sensitive data (0.47 bps to 4.22 kbps). And, with dynamic analysis, if behaviors are not activated during the interval of analysis, these may generate false negatives. They are execution-dependent [3].

Machine Learning

ML methods are considered proficient in finding statistical anomalies and patterns that may not have been captured explicitly in rule-based systems. Hence, looking at features such as permission use, the Intent communication graph, or runtime behavioral logs, ML-based methods can then accurately pin down the suspicious pair of apps. For instance, Casolare et al. achieved around 97.5% accuracy utilizing ML on audio signal features coming from behaviors of apps, which suggests the possibility of identifying covert collusion channels [16]. Likewise, Magsi et al. claimed 97.2% accuracy with a dynamic framework using predictive functions that combine permission-based and behavioral features [17]. Machine learning generalizes from its training data to identify potential novel collusion scenarios that dynamic analysis might miss simply because it was not programmed to detect them. On the other hand, the promise of ML depends heavily on the quality of the training data and how well it represents collusion scenarios. Models may struggle with collusion patterns unseen during training and that are absent from the dataset they rely on [20].

Comparison

Dynamic analysis conveys the purpose of detecting known collusion mechanisms accurately through the direct observation of behaviors at runtime. It is, therefore, capable of identifying specific data flows or interactions. Contrarily, a newly developed behavior might end up never being triggered or entirely missed by dynamic analysis, particularly when it includes covert channels. An ML-based approach might, on the other hand, learn patterns from data and identify a wide range of statistical anomalies missed by a dynamic approach. This means that the accuracy of ML depends on having training data that covers all the base cases, whereas dynamic analysis depends on execution coverage. Another thought would be to combine the two methods in a way that dynamic analysis augments the accuracy of ML, which in turn benefits from more generalization.

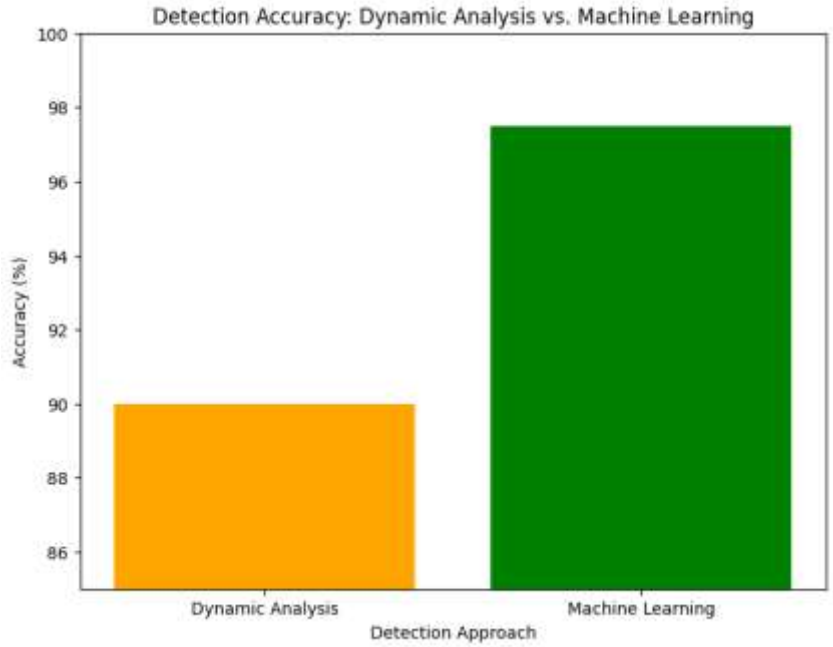

Figure 11: Bar Graph of Detection Accuracy

A bar graph comparing the detection accuracy of dynamic analysis (e.g., TaintDroid) and machine learning approaches (e.g., [16, 17]), highlighting reported accuracies like 97.5% for ML [16].

Table 10: Detection Accuracy Comparison

Approach	Strengths	Limitations	References
Dynamic Analysis	Precise detection of known data flows and interactions; effective for overt channels.	Misses covert channels; depends on execution conditions.	[1, 3, 6]
Machine Learning	Generalizes to novel patterns; high accuracy (e.g., 97.5%).	Relies on training data quality; may miss unseen patterns.	[16, 17, 20]

5.2 Overhead and Scalability

Dynamic Analysis

Due to the need to constantly monitor app activities, such as API calls, network traffic, or usage of system resources, dynamic analysis incurs significant runtime overheads. Further, during execution, techniques such as taint tracking or sandboxing can slow down the application significantly, in addition to consuming a large amount of device resources such as battery and memory, thereby rendering these techniques unusable for on-device deployment [5]. Such an analysis is not trivial to scale up to keep track of a large number of apps, as one has to consider inter-app interactions of all combinations, which grows combinatorially with the number of installed apps. An example would be monitoring interactions of all possible pairs of apps installed on a device that contains several dozens of apps, for instance, which can be computationally prohibitive [6].

Machine Learning

Immediately following training, ML models can fairly quickly perform inference, thus favoring the use of detection in the real-time on resource-constrained devices. Lightweight classifiers like Naive Bayes or Decision Trees could easily do that on the device [19]. The problem lies in training and feature extraction, especially for dynamic features such as API call sequences or network traffic. This generally demands

heavy-duty computation, often requiring offline processing on stoutly-built servers [20]. To adjust themselves to new threats, retraining happens on a periodic basis and also consumes computational effort. Nonetheless, retraining is usually carried out offline instead of being carried out on the device itself.

Comparison

Dynamic analysis comes with its own set of problems with scalability because of its very high overhead during runtime and thus, it is hard to think of dynamic analysis as a suitable option for real-time monitoring of large numbers of apps. ML in contrast does require high computational cost up front for training and feature engineering; however, once the training is done, it does afford a faster inference of models, allowing it to be more scalable when deployed on devices. While lightweight ML models help ease the resource constraint, feature extraction is still a huge bottleneck. Hybrid-based approaches where dynamic analysis is used to generate features for the ML models could be a middle ground between runtime overhead and detection capability.

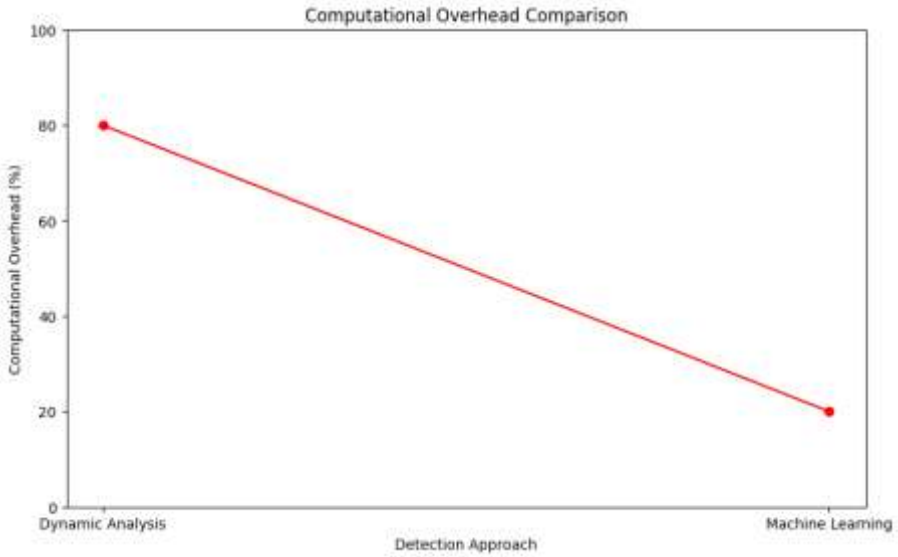

Figure 12: Line Graph of Computational Overhead

A line graph comparing computational overhead (e.g., CPU, memory usage) of dynamic analysis and ML methods [5, 20].

Table 11: Overhead and Scalability Comparison

Approach	Strengths	Limitations	References
Dynamic Analysis	Real-time monitoring; precise data capture.	High runtime overhead; poor scalability for many apps.	[5, 6]
Machine Learning	Fast inference; lightweight models for on-device use.	Expensive training and feature extraction; requires offline processing.	[19, 20]

5.3 Coverage

Dynamic Analysis

Because dynamic analysis requires specific conditions to be activated at runtime, it cannot detect all forms of collusive behavior. For instance, if colluding happens under certain user inputs or device modes and such conditions are not fed into monitoring, collusion activities may be missed [3]. Thus, in these example cases, colluding applications share data only during a few interactions and create false negatives. Besides

this, dynamic analysis has a hard time detecting collusion paths that are too rare or intricate to be reproduced easily in test environments, which means it cannot cover all potential threat scenarios.

Machine Learning

ML models are only as good as the diversity and representativeness of the training data. If certain collusion types do not find their way into the dataset, then the model will most likely fail to detect them [20]. For example, a model trained on overt channel-based collusion (e.g., Intents) could miss the covert channels (e.g., sensor signals) if the respective examples have not been provided by the training data. Overfitting to known collusion patterns can make them less effective in dealing with new and evolving threats. However, with sufficiently diverse training data, it is less feasible for ML to capture a broader range of collusion scenarios than dynamic analysis.

Comparison

Both approaches have coverage limitations: behaviors that have not been triggered or are rarely triggered elude dynamic analysis, while patterns not represented in training data are missed by ML. From the dynamic analysis perspective, its coverage is constrained per execution paths; while from the ML perspective, it is dependent on the diversity of the data. A hybrid approach where dynamic analysis generates comprehensive runtime data for ML training may improve coverage by allowing more direct observations to be generalized into patterns.

Table 12: Coverage Comparison

Approach	Strengths	Limitations	References
Dynamic Analysis	Captures observed runtime behaviors.	Misses untriggered or rare collusion paths.	[3]
Machine Learning	Broad coverage with diverse training data.	Limited by training data representativeness; risk of overfitting.	[20]

5.4 Adaptability

Dynamic Analysis

Dynamic analysis adapts to new collusion threats by changing monitoring rules, signatures, or instrumentation. For example, if a new channel arrived in the form of a novel sensor-based communication method, dynamic analysis tools should be structured so as to be able to monitor this new channel [6]. This is a cumbersome process and one that would thus lag behind a rapidly changing environment, as any changes to detection logic involved manual intervention.

Machine Learning

ML models are highly adaptable to new threats, as they train themselves whenever new data concerning the latest collusion technique are available [20]. For example, should a new collusion type using audio signals emerge, retraining an ML model with data illustrating this behavior would provide detection capabilities without any need for changes to its algorithm. This adaptability is one of the major plus points that allows ML to address new threats faster than rule-based dynamic analysis.

Comparison

ML systems are better adaptable via retraining and need just new data to deal with threats, whereas dynamic analysis may deal with threats already known but need a manual update for their detection logics, which may ultimately be a more cumbersome and resource-intensive process. ML's flexibility makes it better suited to the dynamic nature of collusion threats.

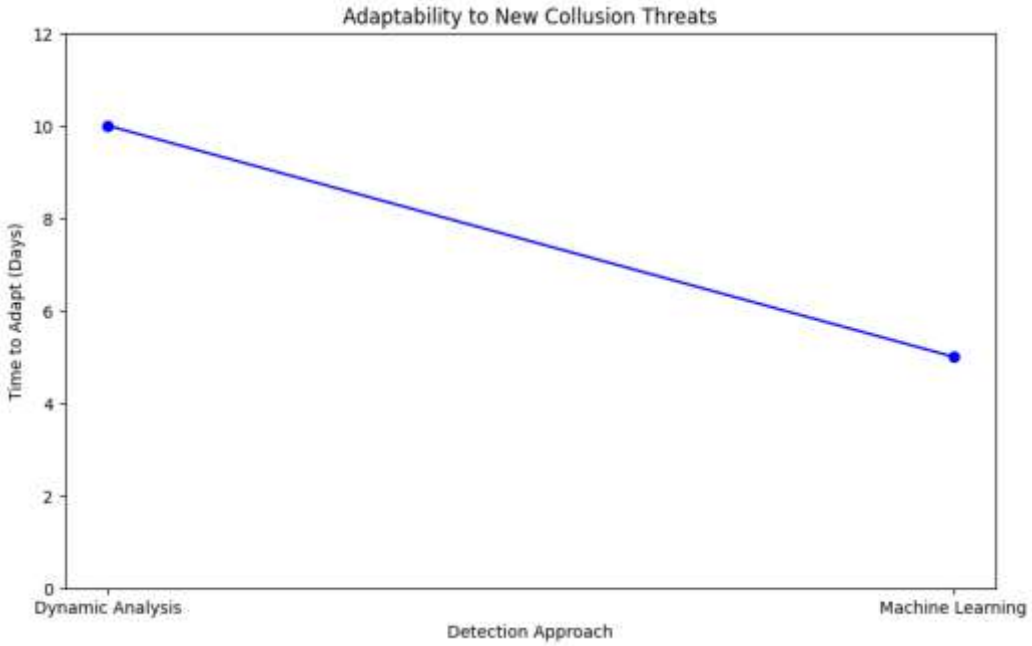

Figure 13: Timeline of Adaptability to New Threats

A timeline showing how quickly dynamic analysis and ML adapt to new collusion threats, referencing [6, 20].

Table 13: Adaptability Comparison

Approach	Strengths	Limitations	References
Dynamic Analysis	Effective for known behaviors with updated rules.	Requires manual rule updates; slow to adapt.	[6]
Machine Learning	Adapts quickly via retraining; handles new threats.	Needs new, representative data for retraining.	[20]

5.5 Deployment

Dynamic Analysis

Considering that dynamic analysis imposes a very high-performance overhead and resource demand on the device, it has always been difficult to carry out dynamic analysis on real devices. Since the continuous monitoring of app interactions slows down the performance of devices and drains their battery life, it becomes impossible to realize widespread on-device implementation [5]. However, dynamic analysis fits well in controlled environments such as sandboxes or emulators, where it may be used to supply real-time-detection for testing purposes. Such grounds allow dynamic analysis to see runtime behavior and thus come in very handy for detailed analysis.

Machine Learning

ML models, especially lightweight classifiers like Naive Bayes or Decision Trees, can be installed on the device for inference at runtime, implying that they are better suited for making end-user decisions [19]. However, periodic retraining and the cost of feature extraction hurt their capability to be really-time, most of which comes into the picture for dynamic features which require runtime monitoring [20]. Furthermore, complex ML models, such as deep neural networks, do not have interpretability; therefore, the reason for distrust on part of the user becomes difficult.

Hybrid Approaches

There is huge potential for hybrid solutions where dynamic analysis is combined with ML. Consider that dynamic analysis produces a myriad of runtime data, including communication logs and sensor outputs, which can serve as features to the ML models. For example, Casolare et al. used dynamic analysis to convert app behaviors into audio signals that are then classified by ML [16], whereas Magsi et al. employed dynamic behavioral features within their ML predictive framework [17]. Such hybrid approaches thus combine the accuracy of dynamic analysis with the generalization capabilities of ML, enabling these systems to work in carefully contrived situations and in the open field.

Comparison

Since dynamic analysis is quite cumbersome, it is best suited for controlled environments, whereby ML paradigms can be best utilized on devices where real-time inference can be accomplished by lightweight models. Hence, using hybrids takes advantage of both, i.e., dynamic analysis gives the features over which ML can perform classification as well as an equitable solution.

Table 14: Deployment Comparison

Approach	Strengths	Limitations	References
Dynamic Analysis	Real-time detection in controlled environments.	High overhead; impractical for on-device use.	[5]
Machine Learning	Fast inference; suitable for on-device deployment.	Feature extraction costly; complex models lack interpretability.	[19, 20]
Hybrid Approaches	Combines precision and generalization; practical for deployment.	Requires integration of both systems; complex implementation.	[16, 17]

6. CHALLENGES AND FUTURE DIRECTIONS

Given the dynamic nature of collusion among Android apps, detection thereof is complex and keeps evolving, with countermeasures. This section explores five important open challenges: data scarcity, hybrid methods, resourcefulness in detection, adaptive attacks, and explainability; followed by suggestions for future avenues of research intended to advance mobile security.

6.1 Data Scarcity

One of the major challenges faced by researchers pertains to the limited availability of genuine real-world datasets portraying collusion scenarios accurately. An ML model and dynamic analysis tools require diverse and quality datasets for training and evaluation. Most researchers now resort to synthetic datasets, mainly generated by the Application Collusion Engine (ACE), which purportedly generates over 5,000 sets of colluding and non-colluding apps for experimental research [21]. Though, as Blasco et al. maintain, such synthetic datasets allow for controlled experimental conditions, they may be unable to mimic the subtleties of real-world colluding applications and thereby may restrict model generalizability [21]. Lately, the Scientific Reports, 2025, brought to attention a balanced dataset of 1,455 apps, comprising 485 colluding apps coming from ACE and DroidBench, describing in essence the effort being made toward more holistic datasets. The problem, though, is that the major focus of DroidBench is taint analysis, with a handful of test cases for inter-app communication; hence, it cannot deal with the various complexities of collusion scenarios DroidBench.

Another hurdle to the road toward strong detection systems is the absence of public, real-world datasets on collusion. Hence, it would be imperative that the research community develop into a collaborative partnering process for collecting and sharing real-world collusion datasets; this goal may be, for example, one promoted within NeurIPS 2025 Datasets & Benchmarks Track NeurIPS 2025. These datasets could contain those labeled trajectory apps of collusion spotted in the wild, with detailed metadata on communication patterns and behaviors.

Enhancing synthetic dataset generation tools, on the other hand, should be pursued so that they can simulate better real-world collusion scenarios as an interim step until enough real-world data is made available for detection training.

Table 15: Challenges in Data Scarcity

Challenge	Description	Proposed Solutions	References
Limited Real-World Data	Few real-world collusion datasets exist, limiting model training and evaluation.	Collect and share real-world datasets; enhance synthetic dataset realism.	21, Scientific Reports, 2025
Synthetic Dataset Limitations	Synthetic datasets may not capture real-world complexity.	Develop advanced generation tools like ACE; include diverse collusion scenarios.	[21]
Lack of Benchmarks	No standard benchmarks for collusion detection evaluation.	Establish community-driven benchmarks via initiatives like NeurIPS.	NeurIPS 2025

6.2 Resource-Efficient Detection

The highly resource-intensive nature of dynamic analysis and the complex ML models constitutes a significant bottleneck for on-device collusion detection, while mobile devices have limited computational power, memory, and battery life. Continuous feature monitoring or extraction from the device might hinder its performance and thus prevent real-time detection [5]. Creation of resource-efficient methods of detection is paramount to ensure protection of users from threats without impeding the usability of the device.

Promising solutions emerge with recent advances in on-device ML. Lightweight classifiers such as Naive Bayes and Decision Trees are theoretically foreseen to be suitable for on-device deployment considering their low computation requirements [19]. Faiz et al. used Naive Bayes in a two-stage classifier for collusion detection that might be further adapted for on-device execution [19]. Further, studies addressing on-device ML for Android security highlight the prowess of optimized models in ensuring a high detection rate with the fewest resource consumptions possible On-Device ML Security. Model quantization and pruning, for example, as found in tutorials for running ML models on Android devices, can further cut down resource use Running ML on Android.

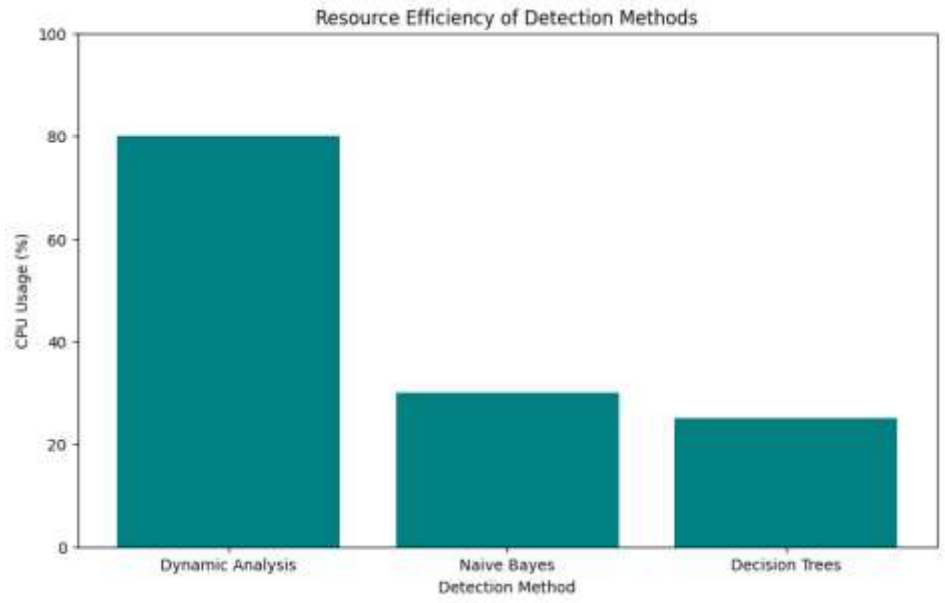

Figure 14: Graph of Resource Efficiency

A graph comparing resource efficiency (e.g., CPU, memory, battery usage) of different detection methods, referencing lightweight classifiers [19].

Table 16: Strategies for Resource-Efficient Detection

Strategy	Description	Benefits	References
Lightweight Classifiers	Use simple ML models like Naive Bayes for on-device inference.	Low resource usage; fast detection.	[19]
Model Optimization	Apply quantization and pruning to reduce model size.	Enables on-device deployment; minimal performance impact.	Running ML on Android
Selective Monitoring	Monitor only high-risk app pairs identified by static analysis.	Reduces dynamic analysis overhead.	[5]

6.3 Adaptive Attacks

With the advancement in detection methods, cybercriminals would surely work toward having adaptive attacks using more stealthy communication channels and evasive methods to bypass detection. Colluding apps may use covert channels-even through sensors or timing-which are generally not detectable through tools such as TaintDroid [6, 7]. Soundcomber introduced a covert channel that utilized audio signals to covertly encode and transmit sensitive information [7]. Also, in ML-based systems, adversarial attacks such as data poisoning or evading attacks can arrange the inputs for the model that misclassifies colluding apps as benign DroidEnemy.

Attack tools are adaptive and, accordingly, detection systems have to evolve to monitor a wide range of communication channels, including even non-traditional ones such as sensor outputs. Research into Adversarial ML for Android security, therefore, provides avenues toward building fortified detection systems. For instance, DroidEnemy proposes a malware detection approach that considers the costs of feature manipulation and is thus resistant to adversarial examples DroidEnemy. On the other hand, Evade droid offers a query-efficient attack to assess the performance of ML-based detectors and hence brings forward the need for robustness EvadeDroid. Future approaches to detection would, therefore, combine adversarial training, which balances models on adversarial examples to grant them resistance, and perhaps consider ensemble learning to combine several classifiers to improve resistance SecureDroid.

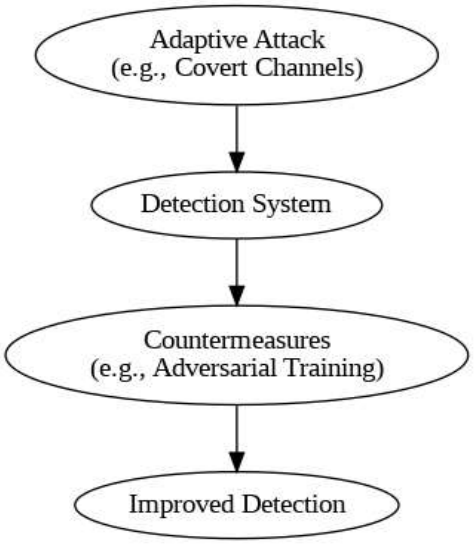

Figure 15: Flowchart of Adaptive Attacks and Countermeasures

A flowchart showing how adaptive attacks (e.g., covert channels, adversarial ML) work and how detection methods (e.g., adversarial training) counter them, referencing [6, 7].

Table 17: Countering Adaptive Attacks

Attack Type	Description	Countermeasures	References
Covert Channels	Use of sensor signals or timing for data leaks.	Monitor non-traditional channels; use advanced dynamic analysis.	[6, 7]
Adversarial ML Attacks	Manipulate ML model inputs to evade detection.	Adversarial training; ensemble learning.	[7]

7. CONCLUSION

App collusion is one threat; detection is an even bigger one in mobile security. This review has studied the role of dynamic analysis and machine learning in handling this threat, emphasizing the strengths of each approach and the possibility of hybrid approaches to exploit these strengths for detection.

Dynamic analysis brings its own power for real-time behavior and interaction observation, which otherwise would not have been caught by static analysis. While execution proceeds, channels of covert communication may get detected, for instance, in Soundcomber attacks when apps encode sensitive data in audio signals. That said, such dynamic analysis means imposing heavy computational overhead and covering only partial execution paths while at the same time falling victim to evasion techniques and that limits its scaling and effectiveness in real-world scenarios.

From the standpoint of machine-learning, recognition of collusive actions is achieved through the modeling of complex relations or searching for statistical anomalies. By using indicators such as permission sets, Intent generating graphs, and runtime logs, the ML models achieved high detection accuracies, 97.5% being reported by some studies for novel methods such as audio signal classification. However, ML performance greatly depends on the quality and contextuality of the training data, and with the paucity of real-world collusion datasets, this situation worsens.

Since both approaches act as complements to each other, the integrated approach could be an efficient one for collusion detection by combining the accuracy of dynamic analysis and the generalization capability of ML. Hybrid approaches achieve promising results with a top-level accuracy and including advantages of dynamic analysis while unloading some of the liabilities of ML and vice versa. As an example, runtime information extraction of inter-app communications, if combined with pattern recognition based on ML, can better recognize collusion channels, overt or covert, undermining each method when used individualistically.

This very field needs further research to make a step forward in collusion detection in Android apps. For future avenues of research, one would include the following:

1) **Scalable Dynamic Instrumentation**: Designing some lightweight monitor techniques at runtime such that it maintains the performance overhead low enough so that it can suit resource-constrained mobile devices.
2) **Rich Feature Learning**: Improving feature engineering to catch delicate nuances in app behavior via deep learning or through new signal methods so as to improve performance of ML models.
3) **Comprehensive Evaluation**: Construction of concrete evaluation benchmarks and realistic datasets so as to evaluate the detection systems in a rigorous manner, instead of the synthetic data-based evaluation presently used.

Furthermore, the solutions will have to cope with data scarcity, resource-efficient detection, adaptive attacks, and explainability on their way to being truly trustworthy and deployable. Hence, joint efforts towards the development and release of collusion detection datasets from real-world applications, optimization of lightweight ML models, increasing robustness against adversarial attacks, and incorporation of explainable AI methods will definitely reinforce detection mechanisms.

It is these research lines that can help the mobile security community provide more powerful, efficient, and reliable solutions to save users from the ever-changing app collusion threat while protecting their devices and data in terms of privacy and integrity. Dynamic analysis and machine learning integration, with support from research, stands as the only hope for protecting the Android ecosystem against this malign threat.

References:

[1] Kalutarage, H. K., Nguyen, H. N., & Shaikh, S. A. (2017). Towards a threat assessment framework for apps collusion. *Telecommunication Systems*, *66*(3), 417–430. https://doi.org/10.1007/s11235-017-0296-1

[2] Abro, F. I., Rajarajan, M., Chen, T. M., & Rahulamathavan, Y. (2017). Android Application collusion demystified. In *Communications in computer and information science* (pp. 176–187). https://doi.org/10.1007/978-3-319-65548-2_14

[3] Elish, K. O., Yao, D., Ryder, B. G., & Department of Computer Science, Virginia Tech. (n.d.). On the Need of Precise Inter-App ICC Classification for Detecting Android Malware Collusions. In *Department of Computer Science, Virginia Tech* [Journal-article]. https://prolangs.cs.vt.edu/refs/docs/elish-yao-ryder-most15.pdf

[4] Elish, K. O., Shu, X., Yao, D., Ryder, B. G., & Jiang, X. (2014). Profiling user-trigger dependence for Android malware detection. *Computers & Security*, *49*, 255–273. https://doi.org/10.1016/j.cose.2014.11.001

[5] La Polla, M., Martinelli, F., & Sgandurra, D. (2012). A survey on security for mobile devices. *IEEE Communications Surveys & Tutorials*, *15*(1), 446–471. https://doi.org/10.1109/surv.2012.013012.00028

[6] Marforio, C., Ritzdorf, H., Francillon, A., & Capkun, S. (2012). Analysis of the communication between colluding applications on modern smartphones. *Association for Computing Machinery*. https://doi.org/10.1145/2420950.2420958

[7] Schlegel, R., City University of Hong Kong, Indiana University Bloomington, Zhang, K., Zhou, X., Intwala, M., Kapadia, A., & Wang, X. (n.d.). Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*. https://homes.luddy.indiana.edu/kapadia/papers/soundcomber-ndss11.pdf

[8] Enck, W., Ongtang, M., & McDaniel, P. (2009). On lightweight mobile phone application certification. *CCS '09: Proceedings of the 16th ACM Conference on Computer and Communications Security*. https://doi.org/10.1145/1653662.1653691

[9] Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., & Rieck, K. (2014). Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. *Network and Distributed System Security Symposium (NDSS)*. https://doi.org/10.14722/ndss.2014.23247

[10] Canfora, G., De Lorenzo, A., Medvet, E., Mercaldo, F., & Visaggio, C. A. (2015). Effectiveness of Opcode ngrams for Detection of Multi Family Android Malware. *International Workshop on Security of Mobile Applications, ARES 2015*. https://doi.org/10.1109/ares.2015.57

[11] Li, W., Ge, J., & Dai, G. (2015). Detecting Malware for Android Platform: An SVM-Based Approach. *The 2nd IEEE International Conference on Cyber Security and Cloud ComputingAt: New York, NY, USA*, 464–469. https://doi.org/10.1109/cscloud.2015.50

[12] Wang, Z., Li, C., Yuan, Z., Guan, Y., & Xue, Y. (2016). DroidChain: A novel Android malware detection method based on behavior chains. *Pervasive and Mobile Computing*, *32*, 3–14. https://doi.org/10.1016/j.pmcj.2016.06.018

[13] Han, H., Chen, Z., Yan, Q., Peng, L., & Zhang, L. (2015b). A real-time Android malware detection system based on network traffic analysis. In *Lecture notes in computer science* (pp. 504–516). https://doi.org/10.1007/978-3-319-27137-8_37

[14] Kim, K., & Choi, M. (2015). Android malware detection using multivariate time-series technique. *2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 198–202. https://doi.org/10.1109/apnoms.2015.7275426

[15] Beaucamps, P., Gnaedig, I., & Marion, J. (2012). Computer Security – ESORICS 2012. In *Lecture notes in computer science*. https://doi.org/10.1007/978-3-642-33167-1

[16] Casolare, R., Di Giacomo, U., Martinelli, F., Mercaldo, F., & Santone, A. (2021). Android Collusion Detection by means of Audio Signal Analysis with Machine Learning techniques. *Procedia Computer Science*, *192*, 2340–2346. https://doi.org/10.1016/j.procs.2021.08.224

[17] Magsi, A., & Soomro, A. H. (2023). Collusion Detection using Predictive Functions based on Android Applications. *Sukkur IBA Journal of Computing and Mathematical Sciences*, *6*(2), 15–26. https://doi.org/10.30537/sjcms.v6i2.953

[18] Casolare, R., Martinelli, F., Mercaldo, F., & Santone, A. (2020). Malicious Collusion Detection in Mobile Environment by means of Model Checking. *2022 International Joint Conference on Neural Networks (IJCNN)*, 1–6. https://doi.org/10.1109/ijcnn48605.2020.9207638

[19] Faiz, M. F. I., Hussain, M. A., & Marchang, N. (2018). Detection of collusive App-Pairs using machine learning. *2022 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, 206–212. https://doi.org/10.1109/icce-asia.2018.8552106

[20] Senanayake, J., Kalutarage, H., & Al-Kadri, M. O. (2021). Android Mobile Malware Detection Using Machine Learning: A Systematic Review. *Electronics*, *10*(13), 1606. https://doi.org/10.3390/electronics10131606

[21] Blasco, J., & Chen, T. M. (2017). Automated generation of colluding apps for experimental research. *Journal of Computer Virology and Hacking Techniques*, *14*(2), 127–138. https://doi.org/10.1007/s11416-017-0296-4

[22] Mawoh, R. Y., Wacka, J. B. A., Tchakounte, F., Fachkha, C., & Kolyang, N. (2025). An accurate approach to discriminate android colluded malware from single app malware using permissions intelligence. *Scientific Reports*, *15*(1). https://doi.org/10.1038/s41598-025-86568-w

[23] Kaur, A., Lal, S., Goel, S., Pandey, M., & Agarwal, A. (2024). Android Malware Detection System using Machine Learning. In *IC3 2024: 2024 Sixteenth International Conference on Contemporary Computing* (pp. 186–191). https://doi.org/10.1145/3675888.3676049

[24] Sk, H. K., & Anu, M., V. (2025). Hybrid deep learning model for accurate and efficient android malware detection using DBN-GRU. *PLoS ONE*, *20*(5), e0310230. https://doi.org/10.1371/journal.pone.0310230

[25] Alhanahnah, M., Yan, Q., Bagheri, H., Zhou, H., Tsutano, Y., Srisa-An, W., & Luo, X. (2019). Detecting vulnerable Android Inter-App communication in dynamically loaded code. *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, 550–558. https://doi.org/10.1109/infocom.2019.8737637

[26] Cho, B., Lee, S., Xu, M., Ji, S., Kim, T., & Kim, J. (2017). Prevention of cross-update privacy leaks on android. *Computer Science and Information Systems*, *15*(1), 111–137. https://doi.org/10.2298/csis170728047c

[27] Wang, J., & Wu, H. (2018). Android Inter-App Communication Threats, Solutions, and Challenges. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1803.05039

[28] Wang, B., Yang, C., & Ma, J. (2023). IAFDroid: Demystifying collusion Attacks in Android Ecosystem via Precise Inter-App Analysis. *IEEE Transactions on Information Forensics and Security*, *18*, 2883–2898. https://doi.org/10.1109/tifs.2023.3267666

[29] Guan, Q., Du, S., Wang, K., Yang, C., & Luo, X. (2024). A Framework for Detecting Hidden Partners in App Collusion. *EEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 516–523. https://doi.org/10.1109/trustcom63139.2024.00087

[30] Zhang, S., Su, H., Liu, H., & Yang, W. (2024). MPDroid: A multimodal pre-training Android malware detection method with static and dynamic features. *Computers & Security*, *150*, 104262. https://doi.org/10.1016/j.cose.2024.104262

[31] Aldhafferi, N. (2024). Android malware detection using support vector regression for dynamic feature analysis. *Information*, *15*(10), 658. https://doi.org/10.3390/info15100658