TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214
DOI: 10.26599/TST.20xx.9010xxx
Volume xx, Number x, xxxxxxx 20xx

Bridging the Compliance Gap: Effective and Efficient Detection of Non-Compliant Behaviors in Android Applications

Runqi Fan, Fan Wu, Zifeng Kang, Peng Hu, Weiting Chen, Song Li*

Abstract: As mobile applications become increasingly complex and privacy regulations continue to evolve, the task of accurately identifying app violations in compliance detection has become a major challenge. Prior works mainly relied on taint analysis and dynamic monitoring to address this issue. However, taint analysis requires specifying data sources and sinks for each type of violation, necessitating multiple rounds of analysis for a single app, which results in inefficiency. Additionally, dynamic monitoring suffers from incomplete coverage, resulting in high false negatives.

This paper introduces the Behavior Property Graph (BPG) for detecting non-compliant behaviors in Android applications. BPG integrates the features from various graph representations, including Abstract Syntax Tree (AST), Control Flow Graph (CFG), Call Graph (CG), Program Dependency Graph (PDG), and Pointer Assignment Graph (PAG), enabling comprehensive modeling of complex app behaviors. Violations are identified by querying the BPG using behavioral patterns extracted from real-world apps. We developed a prototype system called BPGEN to generate BPGs and evaluated its performance by testing seven types of non-compliant behaviors on a dataset of 200 real-world applications. Notably, BPGEN successfully identified 14 zero-day non-compliant applications. The results show that BPGEN can efficiently and effectively detect app compliance violations.

Key words: compliance detection, Android applications, behavior analysis, static analysis, graph query

1 Introduction

With the growing prevalence of mobile applications in daily life and their significant impact on users' privacy and consent rights, ensuring compliance of these apps with regulations has become increasingly essential.

Governments worldwide have introduced numerous compliance regulations to meet these demands. Popular regulations such as the General Data Protection Regulation (GDPR) [1] and the California Consumer Privacy Act (CCPA) [2] have attracted considerable interest from researchers. In China, regulations such as the Data Security Law [3] and the Personal Information Protection Law [4] have established strict standards for how apps manage data and ensure transparency. Additionally, in 2024, the Ministry of Industry and Information Technology (MIIT) increased its focus on issues such as unauthorized launches and unclosable apps [5], further broadening the scope of compliance requirements. Despite the rapid growth of these

• Runqi Fan, Fan Wu and Song Li are with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou 310027, Zhejiang, China. E-mail: {fanrunqi, fanwu01, songl}@zju.edu.cn.
• Zifeng Kang is with Johns Hopkins University, Baltimore, Maryland, United States. E-mail: zkang7@jhu.edu.
• Peng Hu is with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou 310053, Zhejiang, China. E-mail: penghu01@outlook.com.
• Weiting Chen is with Ant Group Co., Ltd., Hangzhou 310000, Zhejiang, China. E-mail: weiting.cwt@antgroup.com.
* To whom correspondence should be addressed.
Manuscript received: 2024-08-01; revised: 2024-10-10; accepted: 2024-10-27

regulations, research on compliance policies and app behavior in China remains relatively limited.

Furthermore, the expanding scope of violations and compliance requirements introduces significant challenges to app compliance oversight, particularly in behavior analysis. On the one hand, the growing variety and complexity of violations underscore the need for precise modeling of each violation type, requiring an accurate representation of execution logic and data handling processes. On the other hand, the continuous emergence of new violations and evolving regulatory requirements necessitate greater adaptability and scalability in behavior analysis techniques.

Prior research in Android app behavior analysis primarily employed taint analysis [6–10] and dynamic monitoring [11, 12]. However, these approaches are limited in the current regulatory landscape. Taint analysis necessitates the specification of sources (e.g., user information) and sinks (e.g., external transmissions), requiring multiple rounds of analysis for various types of violations. This not only results in inefficiencies but also restricts the ability to detect non-data leakage behaviors, such as unauthorized cross-app launches, and other emerging violations. Moreover, dynamic analysis relies on real-time execution monitoring—such as network traffic monitoring and system API instrumentation—but is often constrained by its inability to capture all possible violations in large and complex applications, leading to false negatives. Overall, these limitations highlight the urgent need for more comprehensive and efficient methods in app behavior analysis to address the evolving landscape of regulatory compliance.

To overcome these limitations, we introduce a novel graph structure for app compliance detection called Behavior Property Graph (BPG). BPG incorporates various components from multiple graph representations, including the Abstract Syntax Tree (AST), Control Flow Graph (CFG), Call Graph (CG), Program Dependency Graph (PDG), and Pointer Assignment Graph (PAG), to effectively model the full range of app behaviors. This integrated representation demonstrates compatibility in detecting a wide range of violations, extending beyond privacy leakage issues. Once BPG for a specified app is generated, it will be stored in the graph database. By performing graph traversals on BPG, we can check for the existence of violations. In addition, when new compliance requirements arise, they can be accommodated by simply adding query statements, eliminating the need to rebuild BPG. This approach provides a more scalable and flexible solution for compliance detection.

We implemented a prototype system called BPGEN (BPG Generator) to generate BPGs for Android applications and evaluated the system on a dataset of 200 Android applications, focusing on seven types of non-compliant behaviors. As shown by the experimental results, BPGEN completed the analysis for over 90% of apps in under 10 minutes and, compared to existing approaches, can detect a broader range of violations with lower False Positive Rate (FPR) and False Negative Rate (FNR), highlighting its efficiency and effectiveness.

In summary, we make the following contributions in the paper:

- We designed the Behavior Property Graph (BPG) for effective and efficient app behavior analysis, which also exhibits high scalability for detecting compliance violations.
- We analyzed real-world applications to extract behavioral patterns, guiding the creation of graph queries for detecting non-compliant behaviors.
- We developed a prototype system, BPGEN, which identified zero-day compliance violations in 14 real-world applications during evaluation.

2 Overview

In this section, we first present a motivating example and then delineate the threat model pertinent to app compliance detection.

2.1 A Motivating Example

We demonstrate the use of Behavior Property Graph (BPG) in detecting non-compliant behaviors through an example that checks whether an app provides a privacy policy to users in a regulated manner.

The regulations mandate that there must be a hyperlink or button for the privacy policy prominently displayed within the popup, login/registration page, or settings section [13]. Upon clicking this hyperlink or button, the full text of the privacy policy must be readily accessible and displayed to the user. Therefore, developers are required to present the privacy policy text to users in its entirety, adhering to the regulatory requirements.

Figure 1 illustrates a simplified exemplary code snippet from `bubei.tingshu`, a real-world Android

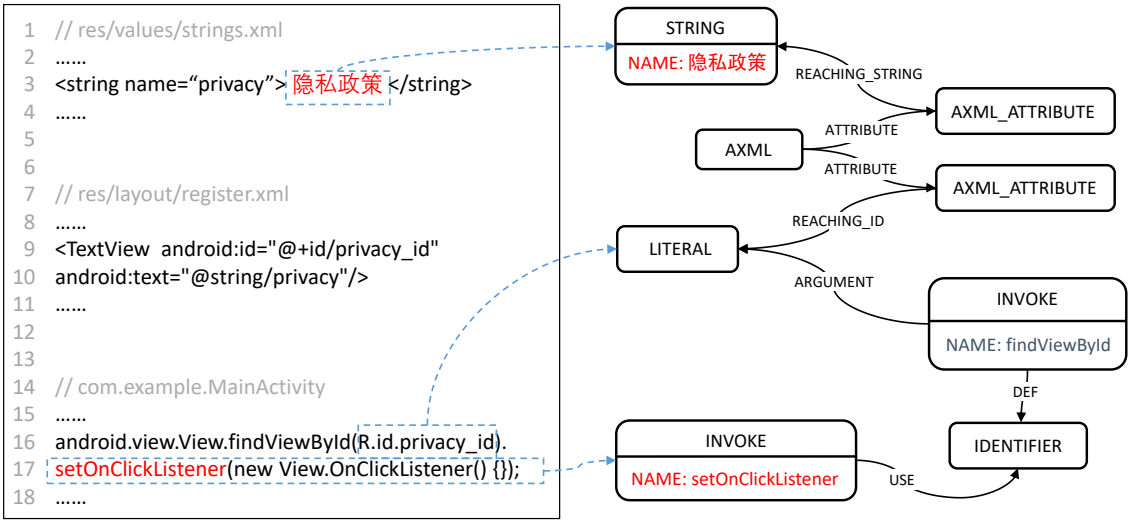


Fig. 1 Example code and the corresponding partial Behavior Property Graph (BPG).

application providing audio digital listening service, to showcase one of the primary approaches adopted by apps to provide privacy policies.

When the user interacts with the app, the predefined privacy policy text from the string resource file is seamlessly integrated into a layout file and displayed to the user via a `TextView`. To allow the user to access the full policy, a click listener is attached to this `TextView` in the `Activity`. Upon clicking, the listener executes a predefined action, such as navigating the user to a new screen within the app that displays the full policy or opening a web page in the user's default browser where the policy is hosted, thereby providing the user with easy access to the privacy policy information. The absence of a readily accessible privacy policy text to users signifies a violation of compliance requirements.

The illustrative case motivates the design of BPG in two fundamental ways. Firstly, the extensive manipulation of resource files drives the need to model these files and operations as nodes and edges within BPG. Secondly, the use of resource IDs in `findViewById` within `MainActivity`, coupled with user click monitoring via `setOnClickListener`, underscores the requirement for BPG to model invocation relationships and data dependencies, thereby enabling a comprehensive depiction of various behaviors.

Figure 1 illustrates how the behavior of having a clickable privacy policy within an app is modeled in BPG. The developer locates the resource ID corresponding to the privacy policy using `findViewById` and binds it to a click event listener `setOnClickListener`.

In the following sections, we will provide a more detailed introduction to the structure of BPG.

2.2 Threat Model

The threat model of our study refers to misconduct introduced by app developers, whether deliberate or inadvertent, which leads to violations of regulatory requirements.

We categorized non-compliant behaviors into three types:

- **The existence of behaviors** indicates that the app fails to provide or avoid certain behaviors as required by regulations. Examples of violations in this category include *No Privacy Policy, No Account Deletion, Unauthorized Cross-App Launch, App Exits Upon Permission Denial*, and *Unclosable Targeted Push*. These behaviors either violate transparency requirements or undermine user choices, which can be perceived as coercive and in violation of informed consent principles.
- **The content of behaviors** refers to the specific data or information accessed or transmitted during their execution, which may violate regulatory requirements. For instance, violations like *Collecting Non-Modifiable Unique Device Identifiers* fall under this category, as they often exceed the scope necessary for providing services and violate the principle of data minimization.

- **The frequency of behaviors** indicates that the occurrence of behaviors exceeds the required threshold. A notable example is *Looping Permission Pop-ups After Denial*, which demonstrates this frequency issue as it repeatedly requests permissions even after users have declined. This behavior disrupts the normal user experience and potentially violates consent norms.

In Section 4.2, we will discuss how we utilize graph traversal on BPG to identify these three types of non-compliant behaviors. Subsequently, in Section 5, we will analyze the occurrence of the aforementioned violations in real-world apps.

3 Behaviour Property Graph

In this section, we describe the definition of BPG and the procedure of constructing BPG.

3.1 Definition

We define Behavior Property Graph (BPG) as a graph representation, which integrates features from Abstract Syntax Tree (AST), Control Flow Graph (CFG), Call Graph (CG), Program Dependency Graph (PDG), and Pointer Assignment Graph (PAG) to a joint graph using the concept of property graphs. The app's BPG is constructed based on its resource and code files, and stored in a graph database using graph structures comprising nodes, edges, and properties. Next, we will introduce these components within BPG.

3.1.1 Nodes

App behavior is shaped by execution logic, data handling, and interactions with resources. To model these aspects effectively, BPG incorporates two types of nodes: resource nodes and code nodes.

Resource nodes represent the configurations and various resources of an app, which can be extracted from files such as `AndroidManifest.xml`, `strings.xml`, and `register.xml`. For instance, when parsing `strings.xml`, BPGEN traverses each string resource tag and creates corresponding nodes, with properties that store the resource identifier and value.

Structuring resource nodes enables a comprehensive analysis of how various resources interact with app functionalities. When detecting app violations, examining the resources linked to specific behaviors is often essential. For example, to determine whether an app provides a privacy policy to users, one critical step involves analyzing the string resources to check for the presence of specific privacy policy-related strings.

Code nodes, extracted from .dex files, represent the essential building blocks of an app's logic and structure, encompassing classes, methods, statements, fields, literals and identifiers. These nodes facilitate a comprehensive analysis of how various code elements influence app behavior. For example, fields are essential for evaluating data manipulation within an app, as they may store sensitive user information. If an app improperly exposes these fields—perhaps through unprotected API—it could result in a data breach.

3.1.2 Edges

BPG utilizes the relationships between nodes as edges, which are extracted from various graph representations and then integrated to form a complete graph. Specifically, BPG includes syntax relations from the Abstract Syntax Tree (AST), which capture structural connections between nodes, and calling relations from the Call Graph (CG) represent the connections between invocation statements and their corresponding method nodes. Additionally, it outlines control flow from the Control Flow Graph (CFG) and data dependencies from the Program Dependency Graph (PDG), representing execution paths and def-use chains. Finally, point-to relations from the Pointer Assignment Graph (PAG) capture how pointers reference objects in memory, aiding in tracking inter-procedural data flow.

Syntax Relations are extracted from Abstract Syntax Tree (AST), which can describe structural relationships between elements within the code of an application. These relations are fundamental in understanding how different syntactic components such as classes, methods, variables, and statements are interconnected.

In BPG, the edges used to describe syntax relations are categorized into three types: CONTAIN edges, ARGUMENT edges, and PARAMETER edges.

CONTAIN edges represent the hierarchical structure, showing how elements like classes, methods, and statements are nested within each other. Firstly, its importance lies in enabling a clear understanding of the scope and context in which operations are performed. For example, in an Android app, permission management is typically implemented within a specific class that includes methods for checking permissions, requesting permissions from the user, and handling the user's response. When analyzing violations

like improper permission handling, CONTAIN edges help trace the permission request back to the class and method it belongs to, allowing verification of whether related methods, such as those handling the permission result, follow proper procedures. If not, issues like unauthorized app exit after permission denial or repeated permission requests may arise. Secondly, CONTAIN edges streamline the graph traversal process. When searching for an invoke statement within a method, CONTAIN edges allow traversal of the method's internal structure directly, without needing to check each statement one by one.

ARGUMENT edges illustrate the relationship between a method and its actual arguments, capturing the contextual information of a method invocation. In contrast, PARAMETER edges focus on understanding data flow within a method by linking it to its formal parameters. These edges are fundamental to inter-procedural data flow analysis, enabling precise tracking of data flow across various methods, which will be further elaborated in subsequent sections.

Calling Relations, extracted from Call Graph (CG), depict how methods within an application invoke one another. These relations are crucial for understanding inter-procedural information flow and dependencies.

In BPG, calling relations are represented by CALL edges, which connect call statement nodes to the corresponding method nodes. In establishing these calling relations, we primarily address two challenges: asynchronous calls and native calls.

The first challenge is handling **asynchronous calls**. Android applications often use asynchronous programming techniques, such as listeners and callbacks, to manage events and background tasks. These asynchronous mechanisms can disrupt the direct invocation and execution flow of methods, creating challenges for precise modeling in calling relations.

BPG addresses these challenges through the following approaches.

When app starts a new `Activity`, an `Intent` object is created to specify the current `Activity` and the target `Activity`. This `Intent` is passed as a parameter to the `startActivity` method. When `startActivity` is invoked, it automatically triggers the `onCreate` method of the target `Activity`. BPG captures this potential relations through graph traversal using chain operations, rather than directly modeling callbacks between methods. Specifically, BPG retains the name of parameters. When an `Intent` is instantiated, it preserves the name of the target `Activity`, and this `Intent` is passed as a parameter to the `startActivity` method. The ARGUMENT edges and USE edges (to be discussed later) in the graph connect these elements. Additionally, each `Activity` CLASS node retains its NAME property, allowing for matching the callback during the graph traversal phase.

Additionally, developers frequently set click listeners using anonymous inner classes, such as `setOnClickListener`, which registers a listener for click events on UI components. For these cases, BPG addresses this challenge by accounting for the implicit calls to `init` (i.e., instance initializer) and `clinit` (i.e., class constructors). When an anonymous class is instantiated, the `init` method initializes the listener, and the `clinit` method handles class-level initialization. Since anonymous classes are instantiated as parameters in the `setOnClickListener` method, BPG tracks these initialization methods to link the listener setup (`i.e., setOnClickListener`) with the corresponding callback method (`i.e., onClick`) using ARGUMENT edges, CONTAIN edges, and CALL edges. Consequently, the asynchronous callbacks are effectively modeled within the graph.

By employing these techniques, BPG enables precise modeling of asynchronous calls within apps.

The second challenge is managing **native calls**. Developers leverage both Java and Android libraries to invoke their APIs, which are native calls written in languages such as C/C++. In Java-based analysis, these native calls function as black boxes, where corresponding method details are not available, leading to a break in data flow analysis. Therefore, it becomes essential to model these native calls to understand how information propagates through them.

We categorize native methods into three types based on how data is propagated through these functions:

- Methods that directly return a value via return statements. For example, `android.os.Build.getSerial` is used to retrieve the device's serial number in Android.
- Methods that assign values from other parameters to the first parameter without returning anything (e.g., `java.lang.StringBuilder.append`, used to append the string representation of some argument to the sequence).

- Methods that return the invoking object (e.g., `java.lang.String.toString` returns the String representation of the object).

To address this challenge, we define a shadow class that mirrors the expected behavior of native methods. Based on data flow patterns, the types of parameters and return values are determined and shadow methods are defined accordingly. Each type of native call is then redirected to its corresponding shadow method. This approach allows us to simulate the behavior of native calls and maintain data flow analysis continuity, ensuring that the flow of information through these native calls can be effectively modeled and analyzed.

Control flow edges represent the flow of execution within a program. In Behavior Property Graph (BPG), control flow is modeled with specific edges to capture both intra- and inter-procedural flows.

For intra-procedural control flow, BPG establishes CF (control flow) edges to connect individual statements within a method, preserving their sequential execution. This structure allows for flow-sensitive analysis, where the order of operations is critical for accurately interpreting program behavior. For instance, the app is expected to collect the user's explicit consent before sending any sensitive data. In this case, the order of operations matters: the app must first ask for consent, then collect the data, and only after receiving consent, send it over the network. Flow-sensitive analysis ensures that the correct sequence is followed.

Inter-procedural control flow edges represent transitions between different methods and are vital for elucidating how control transitions between methods, allowing for a comprehensive analysis of program behavior. To enable inter-procedural control flow analysis in BPG, edges are added between the invoking statements and the invoked methods, capturing the control flow initiated by method calls. Additionally, return edges are included to signify the return points in the invoked methods, allowing the analysis to track how control flows back to the calling method after the invoked method has completed execution. This comprehensive representation facilitates a deeper understanding of program behavior, enabling accurate detection of compliance violations and enhancing data flow analysis.

BPG incorporates two types of edges to model data transfer within apps: data dependency edges and point-to edges.

Data dependency edges, drawing inspiration from the Program Dependency Graph (PDG), employ a def-use chain to map the lifecycle of variables and fields within a method. On the other hand, **Point-to** edges, originating from the Pointer Assignment Graph (PAG), delineate how pointers—encompassing variables and fields—reference objects in memory. Such insights are essential for mapping inter-procedural data dependencies, offering a clear view of the interactions between different methods and their shared data.

In BPG, we define DEF and USE edges to model intra-procedural data flow, while point-to edges enable the tracking of inter-procedural data transfer. By integrating these relations, BPG provides a comprehensive view of data interactions, making it easier to identify potential compliance violations. For instance, if an app retrieves sensitive information, such as the IMEI (International Mobile Equipment Identity) number, by invoking a system API, it typically stores this information in a variable or field. This sensitive data may then be passed between various methods. BPG's data dependency and point-to relations can effectively track this flow. If the app subsequently transmits this data without obtaining proper user consent, BPG's analysis would highlight this as a compliance violation.

3.2 BPG Constrcution

BPGEN first constructs an app's BPG according to its resource files and `classes.dex` files and then stores it in a graph database, which uses graph structures (including nodes, edges, and properties) to represent and store data.

Specifically, BPGEN first utilizes the libraries provided by FlowDroid [6] to parse all resource files, including `AndroidManifest.xml`, and establishes corresponding resource nodes. Then, it uses Soot [14] to disassemble the `classes.dex` into Jimple IR code. The construction of a BPG for code begins by traversing all classes, fields, methods, and statements and creating the corresponding code nodes. Based on the structure of the AST, BPGEN identifies the containment relationships among these elements and adds CONTAIN edges between the corresponding nodes. Similarly, BPGEN creates edges based on the program's CG, CFG and PAG. When encountering call statements, variable definition and usage statements, and conditional statements, BPGEN creates new nodes

for the variables and establishes corresponding edges (such as ARGUMENT, DEF, and USE edges) between the variable nodes and the relevant statements to represent syntactic relations or the propagation of data values.

The constructed BPG is then stored in a Neo4j [15] graph database, which effectively handles the graph structures and facilitates efficient querying and analysis of the app's behavior and potential security issues.

4 BPG Queries

In this section, we describe graph queries to BPG for non-compilant behaviors. We first present how to model queries as several types of graph traversals in Section 4.1 and then describe how to represent non-compilant behaviors via those graph traversals in Section 4.2.

4.1 Graph Traversals

Traversals denote the ways we query the graph database according to nodes, edges, and properties. By modeling non-compilant behaviors as graph traversals, we perform them over BPGs to identify violating apps.

A graph traversal is a function $T: P(V) \rightarrow P(V)$ that maps a set of nodes to another set of nodes on top of BPG where V is a set of BPG nodes and P is the power set of V. There are three operations: function composition $\circ$, function intersection $\cap$, and function union $\cup$, which allow chaining, intersecting, and combining the results of two graph traversals on V, respectively.

Through those simple operations, we can break a complex graph traversal into multiple basic traversal components. We utilize the following symbols to represent them:

- $MATCH^{p}_{type}$: represents matching nodes with type $type$ and properties p,
- $N1 \overset{[REL^{p}]^{len}}{\rightarrow} N2$: denotes a path from node $N1$ to node $N2$. The path is connected by relationship REL with property p and length len.

4.2 Behavior Patterns

In this subsection, we describe how to use graph traversals to represent three categories of non-compliant behaviors. As introduced in Section 2.2, we categorize non-compliant behaviors into three types: behavior existence, behavior content, and behavior frequency. We will then provide examples to illustrate the graph query representation for each type, as shown in the Table 1.

Table 1 An example list of graph queries for different types of violations.

Violation Type	Graph Query
Behavior Existence	$MATCH^{\{\}}_{METHOD} \xrightarrow{[CONTAINS^{\{\}}]^{1}}$ $MATCH^{\{NAME:\text{"setComponent"}\}}_{CALL} \xrightarrow{[ARGUMENT^{\{\}}]^{1}}$ $MATCH^{\{TYPE:\text{"android.content.ComponentName"}\}}_{IDENTIFIER}$ $\xleftarrow{[REACHING\_DEF^{\{\}}]^{1..5}} MATCH^{\{\}}_{LITERAL}$ (1)
Behavior Content	$MATCH^{\{NAME:\text{"getImei"}\}}_{JInvokeStmt} \cup$ $MATCH^{\{NAME:\text{"getDeviceId"}\}}_{JInvokeStmt} \cup$ $MATCH^{\{NAME:\text{"getSubscriberId"}\}}_{JInvokeStmt} \cup$ $MATCH^{\{NAME:\text{"getMeid"}\}}_{JInvokeStmt} \cup$ $MATCH^{\{NAME:\text{"getSimSerialNumber"}\}}_{JInvokeStmt} \cup$ $MATCH^{\{NAME:\text{"getSerial"}\}}_{JInvokeStmt} \cup$ $MATCH^{\{NAME:\text{"getActiveSubscriptionInfoList"}\}}_{JInvokeStmt}$ (2)
Behavior Frequency	$\Big(MATCH^{\{NAME:\text{"onClick"}\}}_{METHOD} \xrightarrow{[CONTAINS\|CALL]^{*0..10}}$ $MATCH^{\{NAME:\text{"android.permission.X"}\}}_{LITERAL}$ $\xrightarrow{[ASSIGNMENT\|ARGUMENT\|DEF\|USE\_Obj]^{*1..5}}$ $MATCH^{\{\}}_{InvokeStmt} \xrightarrow{[CALL]}$ $MATCH^{\{\}}_{METHOD} \xrightarrow{[CONTAINS]}$ $MATCH^{\{NAME:\text{"init"}\}}_{InvokeStmt} \xrightarrow{[CALL]}$ $MATCH^{\{\}}_{METHOD} \xrightarrow{[CONTAINS]}$ $MATCH^{\{\}}_{*} \xrightarrow{[USE\_FIELD]^{*2}}$ $MATCH^{\{\}}_{*} \xleftarrow{[CONTAINS]}$ $MATCH^{\{\}}_{METHOD} \xrightarrow{[CONTAINS]}$ $MATCH^{\{NAME:\text{"setNegativeButton"}\}}_{InvokeStmt}$ $\xrightarrow{[ARGUMENT\|DEF\|USE_{Obj}]^{*1..5}}$ $MATCH^{\{\}}_{*} \xrightarrow{[CALL]}$ $MATCH^{\{NAME:\text{"clinit"}\}}_{METHOD} \xleftarrow{[CONTAINS]}$ $MATCH^{\{\}}_{CLASS} \xrightarrow{[CONTAINS]}$ $MATCH^{\{NAME:\text{"onClick"}\}}_{METHOD} \xrightarrow{[CONTAINS\|CALL]^{*1..5}}$ $MATCH^{\{NAME:\text{"dismiss"}\}}_{InvokeStmt}\Big) \bigcap$ $\Big(MATCH^{\{NAME:\text{"dismiss"}\}}_{IdentityStmt} \xrightarrow{[CF]^{*}}$ $MATCH^{\{\}}_{ReturnVoidStmt}\Big)$ (3)

For behavior existence, we use *Cross-App Launching* as an example, as Equation 1 shows. *Cross-app launching* refers to the behavior in which one app automatically initiates the startup of another app, potentially without the user's consent. We define the following graph query, which identifies method calls to `setComponent` within an Android app where the argument type is `android.content.ComponentName`. The `setComponent` method is used to specify the

target component for an `Intent`, which could be an `Activity`, `Service`, or `BroadcastReceiver`, either within the same app or across different apps. The `android.content.ComponentName` class provides the fully qualified name (package and class name) of the target component, allowing precise identification of which app or component the `Intent` should address. If the package name in the `ComponentName` differs from the app's own package name, the app is targeting a component in a different app. The presence of such invocation statements within the app indicates the existence of cross-app launching behavior.

For behavior content, we take *Collect Non-Modifiable Unique Device Identifiers* as an example, as Equation 2 shows. We define the following graph query, which searches for method invocations that retrieve device identifiers, such as IMEI (International Mobile Equipment Identity), Device ID, Subscriber ID, MEID (Mobile Equipment Identifier), SIM (Subscriber Identity Module) number, and active subscription info list. When an app accesses these identifiers, it indicates the app is attempting to gather unique device information, which could be used for tracking user identification or targeting specific devices.

For behavior frequency, we take *Loop Popup After Permission Denial* as an example, considering the scenario in which an app repeatedly displays permission request dialogs despite the user having denied authorization requests. We define the following graph query, as Equation 3 shows, which identifies a flow starting from an `onClick` method containing an `android.permission` literal, traversing through constructor calls, UI setup (`setNegativeButton`), static initializers, and additional `onClick` methods, ultimately leading to a `dismiss` method invocation. It also finds a separate path from a `JIdentityStmt`, through the control flow graph (CFG), to the same `dismiss` method, which then returns `void`. The goal is to uncover a specific sequence of interactions related to permission handling and the authorization cancellation UI. The presence of such a call sequence in an app indicates a behavior pattern of repeatedly prompting the user for permissions after authorization has been denied. In contrast, a benign app would typically redirect the user to a new page following a permission denial, rather than remaining on the original page and coercively prompting for authorization to proceed with further actions.

5 Evaluation

In this section, we evaluate BPGEN by answering the following research questions.

- [RQ1-Coverage]: What are the primary types of non-compliant behaviours, and is BPG capable of modeling them?
- [RQ2-Effectiveness]: How effective is BPGEN in detecting compliance violations?
- [RQ3-Performance]: What is the performance overhead of BPGEN?

We performed our experiments on a server with 256GB=8*32GB 3200MHz DDR4, Intel(R) Corporation MontageJintide(R) C6248R 3000MHz 24 cores 48 threads 1536/24576/36608 KB cache, and 2 * 400GB 6Gbps SATA HWE62ST3480L003N Hard Drive and 10TB remote network disk.

The test set comprises 200 real-world apps, with half consisting of officially reported non-compliant apps, and the other half randomly selected from app markets such as TapTap [16], Huawei AppGallery [17] and Tencent MyApp [18]. Figure 2 illustrates the distribution of the size of APK (Android Package Kit) files within the test set.

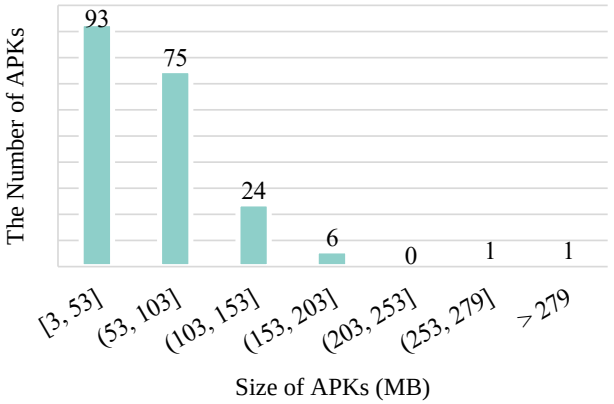


Fig. 2 The distribution of the number of APKs based on size in the dataset of 200 APKs.

5.1 RQ1: Coverage Analysis

In this subsection, we answer the research question on the capability of BPG in modeling non-compliant behaviors.

We start by summarizing seven prevalent non-compliant behaviors of mobile apps that frequently appear in official reports, as illustrated in Table 2.

First, we compare the violation detection scopes of various methods, including static taint analysis,

Table 2 [RQ1-Coverage] Coverage of methods: BPG, taint-Based, and dynamic-Based methods for behavior analysis, and coverage of systems: BPGEN and three popular compliance detection platforms for seven types of violations. "Rounds" refers to the number of analysis needed by the method to detect n types of violations in a given application.

Non-Compliant Behaviours		Method			System			
		Taint Analysis	Dynamic Monitoring	BPG	OPPO	VIVO	Alibaba	BPGEN
1	No Privacy Policy	✓		✓				✓
2	No Account Deletion Function	✓		✓				✓
3	Unauthorized Cross-App Launch		✓	✓				✓
4	Loop Popup After Permission Denial		✓	✓	✓	✓		✓
5	App Exits Upon Permission Denial		✓	✓	✓	✓		✓
6	Collect Non-Modifiable Unique Device Identifiers	✓	✓	✓				✓
7	Unclosable Targeted Push	✓		✓				✓
Rounds		n	1	1				

dynamic analysis, and BPG.

Taint analysis excels at detecting personal privacy data leaks, such as unauthorized collection of unique device identifiers. Based on the behavior patterns of real-world apps, string resources in resource files—such as "privacy policy', "account deletion', and "targeted push'—can be considered as sources, while interactive components (e.g., buttons) serve as sinks, thereby enabling taint analysis to check for the presence of these behaviors. However, taint analysis is unable to detect behaviors such as auto-launching of apps or unexpected app exits, which do not directly involve the flow of tainted data from sources to sinks. Dynamic analysis refers to the process of analyzing a program by executing it and observing its behavior in real-time, but it is limited to identifying actions that should not occur, rather than confirming the presence of required behaviors. For example, if an app does not provide a privacy policy or an account deletion function, dynamic analysis may struggle to determine whether these actions were never triggered or were simply not provided by the app.

Additionally, the Table 2 presents the number of analysis rounds required by different methods to detect n types of violations for a specific app. Taint analysis necessitates the specification of sources (e.g., sensitive data) and sinks (e.g., network transmission interfaces), often requiring multiple rounds of analysis for various types of violations. In contrast, dynamic analysis and BPG can analyze an app in a single round, resulting in greater detection efficiency.

Second, we introduce several advanced compliance detection platforms for the purpose of comparing with BPGEN in Table 2. OPPO and VIVO's compliance detection platforms [19, 20] predominantly rely on dynamic analysis, leveraging their advantage as smartphone manufacturers to conduct real-device debugging. In contrast, Alibaba's detection platform [21] primarily employs static analysis and manual review methods.

As shown in the Table 2, BPG supports a broader range of behavior detection methods compared to the two most mainstream behavior analysis methods, and the corresponding system, BPGEN, also exhibits greater detection coverage compared to popular platforms.

5.2 RQ2: Effectiveness

In this subsection, we address the research question of effectiveness evaluation by examining two metrics: False Positive Rate (FPR), i.e., FP/(TP+FP), and False Negative Rate (FNR), i.e., FN/(TP+FN). In this context, discovering violations is considered positive data.

We compare BPGEN with other platforms on these two metrics based on seven types of non-compliant behaviours. Alibaba platform does not support these detection items so it is excluded in the comparison.

We established the benchmark for false positives (FP) and false negatives (FN) through manual analysis of APK files. Table 4 shows the FPR and FNR of BPGEN and existing platforms in detecting non-compliant behaviours. It is evident that BPGEN has the lowest FPR and FNR, outperforming other compliance detection platforms. This superior performance is due to its precise modeling of the app's internal logic including control flow and data flow.

The analysis below explores the causes of false negatives (FN) and false positives (FP). We have defined the occurrence of non-compliant behaviors as positive. Some behaviors are considered compliant

Table 3 [RQ2-Effectiveness] A selective list of non-compliant apps found by BPGEN. "Violations", "# Downloads" and "Source" columns represent violations detected in the app, the number of downloads at the source – which reflects the popularity of the app – and origins of the tested APK, respectively.

APK Name	Version	Violations	# Downloads	Source
App A	2.1.9	No Privacy Policy	1.17 million	TapTap
App B	2.19.18	No Account Deletion Function	0.56 million	Tencent MyApp
App C	14.7.2	No Account Deletion Function Loop Popup After Permission Denial	12.48 million	Huawei AppGallery
App D	10.10.56	No Account Deletion Function	20.46 million	Huawei AppGallery
App E	6.4.6.4	No Account Deletion Function	62.04 million	Huawei AppGallery
App F	7.5.1	No Account Deletion Function	500 million	Huawei AppGallery
App G	6.7.3	No Account Deletion Function	23.99 million	Tencent MyApp
App H	6.6.9	No Account Deletion Function	13.87 million	Huawei AppGallery
App I	1.6.9	App Exits Upon Permission Denial	0.23 million	Huawei AppGallery
App J	3.5.2.1014	No Account Deletion Function	100 million	Huawei AppGallery

Table 4 [RQ2-Effectiveness] False Positive Rate (FPR), i.e., FP/(TP+FP), and False Negative Rate (FNR), i.e., FN/(TP+FN), of BPGEN compared to two popular detection platforms across seven types of violations in the test set. "N/A (Not Applicable)" signifies that the platform lacks the capability to detect this violation.

	BPGEN		OPPO		VIVO	
	FNR	FPR	FNR	FPR	FNR	FPR
1	0.00%	5.03%	N/A	N/A	N/A	N/A
2	2.86%	44.85%	N/A	N/A	N/A	N/A
3	0.00%	0.00%	N/A	N/A	N/A	N/A
4	0.00%	0.00%	100%	0	100%	0
5	0.00%	0.00%	100%	0	100%	0.50%
6	0.00%	0.00%	N/A	N/A	N/A	N/A
7	2.86%	20.00%	N/A	N/A	N/A	N/A

when present (e.g., the existence of a privacy policy), while others are compliant when absent (e.g., unclosable targeted push). As a result, both FN and FP can arise from similar causes. Therefore, our primary focus is to figure out why certain non-compliant behaviors were not detected in apps where they actually exist.

The primary cause of detection errors in BPGen arises from limitations in processing certain Chinese characters. Specifically, when assessing whether an app provides privacy policy, account deletion functions, or targeted push notifications, BPGEN relies on matching specific strings or resource IDs. However, problems may arise if Chinese characters are not correctly parsed by Soot [14] or if these features are presented through web pages or customer service. In such cases, the absence of these keywords in the analyzed data can lead BPGEN to incorrectly conclude that the app lacks these functionalities. This results in a false positive, where the app is wrongly deemed non-compliant with policy requirements.

To summarize our findings, BPGEN identified a total of 37 non-compliant apps within the test set. Table 3 presents a subset of these apps along with the detected violations. Since the findings have not yet been communicated to the company, we have to hide the full package name. We will reach out to the company as soon as possible. Notably, we discovered zero-day violations in 14 of the apps. These apps exhibit high download volumes, indicating that the violations may have caused extensive impact.

5.3 RQ3: Performance Overhead

In this subsection, we answer the research question of the performance overhead of BPGEN in generating BPG for real-world Android applicaions.

Our methodology is as follows: We ran BPGEN on all the APKs in the test set until the analysis was complete or until it timed out. We recorded the time taken to generate the graph through static analysis and the time required to store the graph in the graph database, as the Figure 3 shows. It can be observed that the duration of static analysis, which encompasses both

Soot analysis and graph generation, generally remain below 200 seconds, while storage time, representing the time to save the graph to the graph database, do not exceed 300 seconds. The largest APK in our test set is 848 MB, and it can be analyzed within 180 seconds. The fluctuation in analysis time can be primarily attributed to the analysis of resource files within the APKs, like large images, which significantly prolongs the duration.

Additionally, we analyzed the distribution of APK analysis time, which highlights the efficiency of BPG construction. Figure 4 shows that 90% of APKs can be analyzed within 10 minutes. The results indicate that BPGEN is efficient, with the majority of APKs processed within a reasonable time.

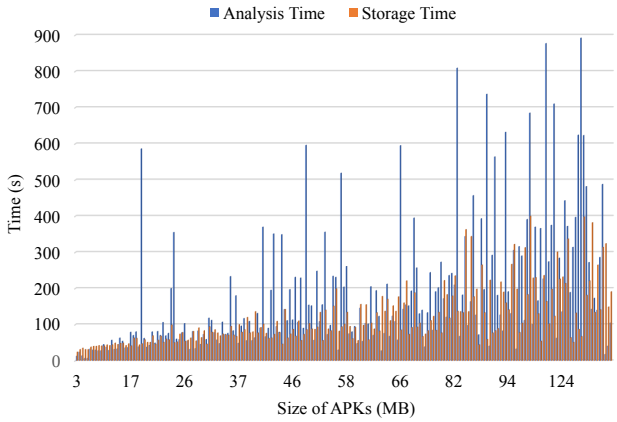


Fig. 3 [RQ3-Performance] Analysis time and storage time for APKs of varying sizes. detection.

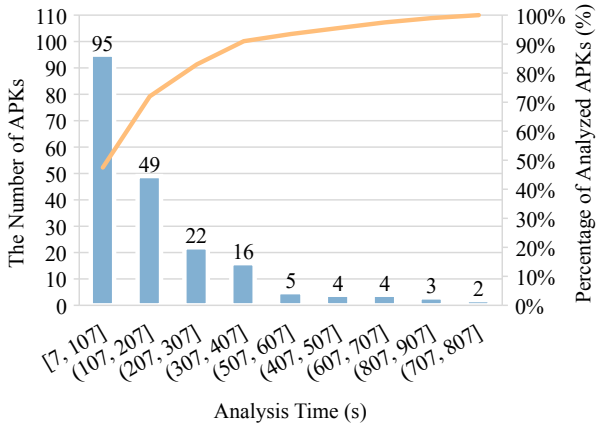


Fig. 4 [RQ3-Performance] Analysis time distribution.

Finally, we measured the execution time of different graph queries separately. As illustrated in Figure 5, the majority of queries are executed within 200 seconds. Notably, a subset of queries encountered timeouts.

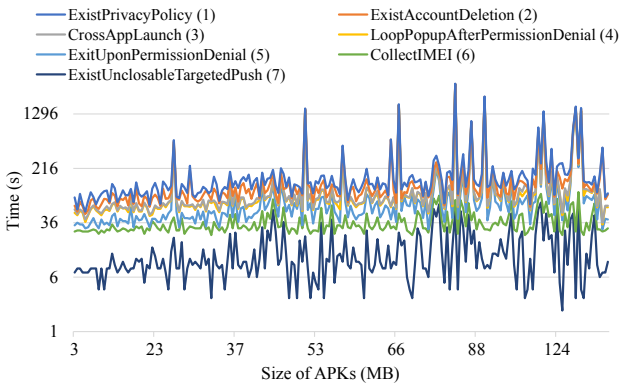


Fig. 5 [RQ3-Performance] The time required to execute different graph queries.

That's because the graph queries were unable to detect the specified behavior within BPGs after a long time (over 20 minutes). However, this often indicates the presence of compliance violations. For example, during the execution of the *ExistPrivacyPolicy* query, if the app does not provide a privacy policy, the relevant string resources information will not be recorded in BPG. This absence prevents the graph query from detecting the expected behavior, leading to a timeout. Nevertheless, it also indicates that the app has violated regulatory requirements for not having a privacy policy available to the user.

6 Related Work

Behavioral Analysis of Mobile Applications. Static and dynamic analysis methods have garnered extensive research attention in the realms of program optimization, security vulnerabilities, and compliance detection [6–12, 22–27]. To analyze the behavior of mobile applications, particularly to identify data leakage, numerous static and dynamic analysis methods have been developed. Static analysis methods mostly employ techniques such as taint analysis and property graph [6–9, 22]. Dynamic analysis methods execute the app in a controlled environment to observe its behavior in real-time. These techniques can include dynamic taint tracking, sandboxing, and system call monitoring [10–12].

FlowDroid [6] and VulHunter [22] have also been used in some compliance detection research [28–30], but they have certain limitations. While FlowDroid performs a flow-sensitive, context-sensitive analysis of data flows within an app to detect potential leaks of sensitive information, it does not support checks for the existence and frequency of specific behaviors. For

instance, behaviors like *Loop Popup After Permission Denial* are not data-related, and FlowDroid cannot detect them. Moreover, FlowDroid requires predefined sources and sinks for taint analysis. Since the call paths for different violations vary greatly, a unified set of sources and sinks cannot be defined. Each time a violation needs to be checked, sources and sinks must be redefined, and the app must be reanalyzed, leading to significant overhead. BPGEN, on the other hand, can analyze an app once and perform multiple queries, reducing time and resource consumption. While VulHunter adopts the concept of property graphs, it does not model resource files and primarily focuses on vulnerability detection, lacking adaptation for compliance violations detection.

Compliance Detection of Specific Behaviors. Recent works also focused on the detection of specific non-compliant behaviors. MOWCHECKER [31] detects inconsistencies in mobile apps' third-party data collection upon user withdrawal. Nguyen et al.[32] proposed a method to detect apps that send out users' personal data without prior consent. OptOutCheck [33] detects inconsistencies between the actual data practices of online trackers and their policy statements regarding user opt-out choices. These studies are based on detecting specific mechanisms of certain behaviors and lack general applicability, making it difficult to conduct broad-scale violation detection.

Privacy Policy Consistency Analysis. In compliance detection research, prior works have primarily focused on analyzing the consistency between application behaviors and their privacy policies to determine if actual behaviors align with the claims made [28, 34–44]. Researchers commonly employ natural language processing (NLP) techniques to model privacy policies, specifying the entities authorized to access user data, the types of data collected, and the purposes of data usage. Additionally, they utilize program analysis methods such as taint analysis, network traffic monitoring, and tracking of sensitive API calls to identify where and how data is accessed within the app. However, these studies primarily focus on whether personal data is misused by applications, limiting their applicability to the detection of other types of violations.

7 Discussion

In this section, we discuss the limitations of BPGEN and propose directions for future work that could enhance its applicability.

Firstly, this paper focuses on detecting compliance in Chinese apps with local regulations, addressing a gap in existing research. However, Behavior Property Graph (BPG) has the potential to be extended to other regulatory frameworks, such as the General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA). This extension would require extracting relevant clauses from these regulations, identifying specific behavioral requirements or prohibitions, and adapting the detection criteria and graph queries by analyzing differences in implementation practices across different countries.

Secondly, the range of detectable violations still needs to be expanded. Since customizing graph query statements for different violations requires manual participation, Section 5 only presents detection results for seven common types of non-compliant behaviors. Support for detecting a broader range of violations is still under development.

Additionally, BPGEN currently cannot identify third-party libraries, limiting its ability to distinguish between behaviors originating from the app itself and those from SDKs. In future work, BPGEN can support the detection of non-compliant behaviors from SDKs by integrating third-party library identification tools [45].

Lastly, as BPGEN is a static analysis tool, it cannot effectively support the detection of violations that require dynamic analysis methods. For instance, regulations may impose restrictions on aspects such as font and color usage within an app, which necessitates screen analysis techniques such as optical character recognition (OCR). Therefore, incorporating dynamic analysis will be a crucial step toward enhancing the completeness of compliance detection in future work.

8 Conclusion

In conclusion, this paper presents a universal and practical solution for compliance detection in Android apps. We introduced a novel graph structure called Behavior Property Graph (BPG), which integrates features from Abstract Syntax Tree (AST), Control Flow Graph (CFG), Call Graph (CG), Program Dependency Graph (PDG), and Pointer Assignment Graph (PAG) to effectively model an

app. By extracting behavioral patterns from real-world applications and applying graph traversal techniques, the model efficiently identifies potential privacy violations. We implemented a prototype system called BPGen to generate BPGs for Android apps and evaluated its efficiency and effectiveness on a test set comprising 200 real-world apps. The results indicate that BPGen can complete the analysis for over 90% of the apps in under 10 minutes. Moreover, when compared to existing compliance detection platforms, BPGen significantly outperforms these systems in terms of detection scope, false positive rates, and false negative rates. Notably, BPGen identified zero-day compliance violations in 14 apps, further validating its effectiveness.

Acknowledgment

This work was supported by the Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security.

References

[1] European Parliament. General Data Protection Regulation. Technical Report Regulation (EU) 2016/679, European Union, Brussels, 2016. (accessed: 2024-06-29).

[2] California Attorney General. California Consumer Privacy Act (CCPA) Regulations. Technical Report Cal. Civ. Code §§ 1798.100–1798.199, State of California, California, 2020. (accessed: 2024-06-29).

[3] Standing Committee of the National People's Congress. Data Security Law of the People's Republic of China. Technical Report Order No. 84, National People's Congress, Beijing, 2021. (accessed: 2024-06-29).

[4] Standing Committee of the National People's Congress. Personal Information Protection Law of the People's Republic of China. Technical Report Order No. 91, National People's Congress, Beijing, 2021. (accessed: 2024-06-29).

[5] Ministry of Industry and Information Technology. Notice on Apps (SDKs) Violating User Rights (2024, Batch 1, Total Batch 36). `https://www.miit.gov.cn/jgsj/xgj/fwjd/art/2024/art_615e90b52bd2458bb71b3cc0fb68355b.html`.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM sigplan notices*, 49(6):259–269, 2014.

[7] Xiang Pan, Yinzhi Cao, Xuechao Du, Boyuan He, Gan Fang, Rui Shao, and Yan Chen. {FlowCog}: Context-aware semantics extraction and analysis of information flow leaks in android apps. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1669–1685, 2018.

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

[9] Trung Tin Nguyen, Duc Cuong Nguyen, Michael Schilling, Gang Wang, and Michael Backes. Measuring user perception for detecting unexpected access to sensitive resource in mobile apps. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 578–592, 2021.

[10] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[11] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. Brahmastra: Driving apps to test the security of {Third-Party} components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1021–1036, 2014.

[12] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054, 2013.

[13] Cyberspace Administration of China Secretariat, Ministry of Industry and Information Technology Office, Public Security Bureau Office, State Administration for Market Regulation Office. Guidelines on the Identification of Illegal and Unregulated Collection and Use of Personal Information by Apps. `https://wap.miit.gov.cn/jgsj/waj/wjfb/art/2020/art_8663d2afe61b40c3beb7c65bf6ec2a64.html`, November 2019. Accessed: 2024-06-29.

[14] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, 2011.

[15] Inc. Neo4j. Neo4j. `https://neo4j.com/`, 2024. Accessed: 2024-06-29.

[16] Taptap. `https://www.taptap.com/`, 2024. Accessed: 2024-06-29.

[17] Ltd. Huawei Technologies Co. Huawei appgallery. https://developer.huawei.com/consumer/en/appgallery/, 2024. Accessed: 2024-06-29.

[18] Tencent Technology (Shenzhen) Company. Tencent myapp. https://myapp.cloud.tencent.com/, 2024. Accessed: 2024-06-29.

[19] Oppo mobile open platform. https://open.oppomobile.com/new/introduction?page_name=audit-open. Accessed: 2024-06-29.

[20] Vivo identity management system. https://id.vivo.com.cn/. Accessed: 2024-06-29.

[21] Alibaba cloud e-mas developer tools. https://emas.console.aliyun.com/service/devTool. Accessed: 2024-06-29.

[22] Chenxiong Qian, Xiapu Luo, Yu Le, and Guofei Gu. Vulhunter: toward discovering vulnerabilities in android applications. *IEEE Micro*, 35(1):44–53, 2015.

[23] Xin Liu, Yingli Zhang, Qingchen Yu, Jiajun Min, Jun Shen, Rui Zhou, and Qingguo Zhou. Smarteagleeye: A cloud-oriented webshell detection system based on dynamic gray-box and deep learning. *Tsinghua Science and Technology*, 29(3):766–783, 2023.

[24] Fei Yan, Rushan Wu, Liqiang Zhang, and Yue Cao. Spider: Speeding up side-channel vulnerability detection via test suite reduction. *Tsinghua Science and Technology*, 28(1):47–58, 2022.

[25] Lei Xu and Junhai Zhai. Dcvae-adv: A universal adversarial example generation method for white and black box attacks. *Tsinghua Science and Technology*, 29(2):430–446, 2023.

[26] Sohail Khan and Mohammad Nauman. Interpretable detection of malicious behavior in windows portable executables using multi-head 2d transformers. *Big Data Mining and Analytics*, 7(2):485–499, 2024.

[27] Youyang Qu, Lichuan Ma, Wenjie Ye, Xuemeng Zhai, Shui Yu, Yunfeng Li, and David Smith. Towards privacy-aware and trustworthy data sharing using blockchain for edge intelligence. *Big Data Mining and Analytics*, 6(4):443–464, 2023.

[28] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D Breaux, and Jianwei Niu. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 25–36, 2016.

[29] Le Yu, Xiapu Luo, Xule Liu, and Tao Zhang. Can we trust the privacy policies of android apps? In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 538–549. IEEE, 2016.

[30] Le Yu, Xiapu Luo, Jiachi Chen, Hao Zhou, Tao Zhang, Henry Chang, and Hareton KN Leung. Ppchecker: Towards accessing the trustworthiness of android apps' privacy policies. *IEEE Transactions on Software Engineering*, 47(2):221–242, 2018.

[31] Xiaolin Du, Zhemin Yang, Jiapeng Lin, Yinzhi Cao, and Min Yang. Withdrawing is believing? detecting inconsistencies between withdrawal choices and third-party data collections in mobile apps. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 14–14. IEEE Computer Society, 2023.

[32] Trung Tin Nguyen, Michael Backes, Ninja Marnau, and Ben Stock. Share first, ask later (or never?) studying violations of {GDPR's} explicit consent in android apps. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3667–3684, 2021.

[33] Duc Bui, Brian Tang, and Kang G Shin. Do opt-outs really opt me out? In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 425–439, 2022.

[34] Duc Bui. *Assessment of Privacy Risks in Mobile and Web Applications/Services*. PhD thesis, 2022.

[35] Benjamin Andow, Samin Yaseer Mahmud, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Serge Egelman. Actions speak louder than words:{Entity-Sensitive} privacy policy and data flow analysis with {PoliCheck}. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 985–1002, 2020.

[36] Duc Bui, Yuan Yao, Kang G Shin, Jong-Min Choi, and Junbum Shin. Consistency analysis of data-usage purposes in mobile apps. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2824–2843, 2021.

[37] Kaifa Zhao, Xian Zhan, Le Yu, Shiyao Zhou, Hao Zhou, Xiapu Luo, Haoyu Wang, and Yepang Liu. Demystifying privacy policy of third-party libraries in mobile apps. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1583–1595. IEEE, 2023.

[38] Duc Bui, Brian Tang, and Kang G Shin. Detection of inconsistencies in privacy practices of browser extensions. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2780–2798. IEEE, 2023.

[39] Yin Wang, Ming Fan, Junfeng Liu, Junjie Tao, Wuxia Jin, Qi Xiong, Yuhao Liu, Qinghua Zheng, and Ting Liu. Do as you say: Consistency detection of data practice in program code and privacy policy in mini-app. *arXiv preprint arXiv:2302.13860*, 2023.

[40] Mir Masood Ali, David G Balash, Monica Kodwani, Chris Kanich, and Adam J Aviv. Honesty is the best policy:

On the accuracy of apple privacy labels compared to apps' privacy policies. *arXiv preprint arXiv:2306.17063*, 2023.

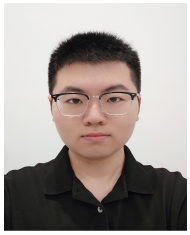
[41] Shenao Wang, Yuekang Li, Kailong Wang, Yi Liu, Chao Wang, Yanjie Zhao, Gelei Deng, Ling Shi, Hui Li, Yang Liu, et al. Miniscope: Automated ui exploration and privacy inconsistency detection of miniapps via two-phase iterative hybrid analysis. *arXiv preprint arXiv:2401.03218*, 2024.

[42] Benjamin Andow, Samin Yaseer Mahmud, Wenyu Wang, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Tao Xie. {PolicyLint}: investigating internal privacy policy contradictions on google play. In *28th USENIX security symposium (USENIX security 19)*, pages 585–602, 2019.

[43] Mafalda Ferreira, Tiago Brito, José Fragoso Santos, and Nuno Santos. Rulekeeper: Gdpr-aware personal data compliance for web frameworks. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2817–2834. IEEE, 2023.

[44] Irwin Reyes, Primal Wijesekera, Joel Reardon, Amit Elazari Bar On, Abbas Razaghpanah, Narseo Vallina-Rodriguez, Serge Egelman, et al. "won't somebody think of the children?" examining coppa compliance at scale. In *The 18th Privacy Enhancing Technologies Symposium (PETS 2018)*, 2018.

[45] Xian Zhan, Tianming Liu, Lingling Fan, Li Li, Sen Chen, Xiapu Luo, and Yang Liu. Research on third-party libraries in android apps: A taxonomy and systematic literature review. *IEEE Transactions on Software Engineering*, 48(10):4181–4213, 2021.



Runqi Fan received the Bachelor degree from Sichuan University, China, in 2023. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, Zhejiang University, China. His research interests include program analysis and mobile security.

Fan Wu received the M.S. degree from National University of Defense Technology, China, in 2013. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, Zhejiang University, China. His research interests include network security, information security and artificial intelligence.

Zifeng Kang received the Bachelor degree from Tsinghua University, China, in 2019. He is currently pursuing the Ph.D. degree with the Department of Computer Science, Johns Hopkins University, America. His research interests include security and privacy issues on the Web, such as leveraging dynamic analysis to systematically detect vulnerabilities in real-world websites.

Peng Hu received the MS degree from University of Electronic Science and Technology of China in 2023. He is currently a researcher and developer at Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security. His current research interests include Android static and dynamic analysis.

Weiting Chen, Application Security Specialist of Ant Group Co., Ltd. With over a decade of experience in mobile application security, having conducted extensive research into mobile security and the contemporary security challenges faced by applications.

Song Li received the Ph.D. degree from Johns Hopkins University, America, in 2022. He is currently a Professor at Zhejiang University. His research interests include static/dynamic analysis and privacy – trying to solve real-world challenging problems such as increasing the accuracy and efficiency of vulnerability detection, mobile APP analysis and privacy protection.