ApkDiffer: Accurate and Scalable Cross-Version Diffing Analysis for Android Applications

JIARUN DAI, Fudan University, China
MINGYUAN LUO, Fudan University, China
YUAN ZHANG, Fudan University, China
MIN YANG, Fudan University, China
MINGHUI YANG, OPPO, China

Software diffing (a.k.a., code alignment) is a fundamental technique to differentiate similar and dissimilar code pieces between two given software products. It can enable various kinds of critical security analysis, e.g., n-day bug localization, software plagiarism detection, etc. To date, many diffing tools have been proposed dedicated to aligning binaries. However, few research efforts have elaborated on cross-version Android app diffing, largely hindering the security assessment of wild apps. To sum up, existing diffing works usually establish scalability-oriented alignment algorithms, and suffer from significant alignment errors when handling the large codebases of modern apps.

To fill this gap, we propose ApkDiffer, a method-level (i.e., function-level) diffing tool dedicated to aligning versions of the same closed-source Android app. ApkDiffer achieves a good balance between scalability and effectiveness, by featuring a two-stage decomposition-based alignment solution. It first decomposes the codebase of each app version, respectively, into multiple functionality units; then tries to precisely align methods that serve equivalent app functionalities across versions. In evaluation, the results show that ApkDiffer noticeably outperforms existing alignment algorithms in precision and recall, while still having a satisfactory time cost. In addition, we used ApkDiffer to track the one-year evolution of 100 popular Google Play apps. By pinpointing the detailed code locations where app versions deviate in privacy collection, we convincingly revealed that app updates may pose ever-evolving privacy threats to end-users.

CCS Concepts: • **Security and privacy** → **Software reverse engineering**.

Additional Key Words and Phrases: Software Diffing, Android Applications, Evolution Analysis

ACM Reference Format:
Jiarun Dai, Mingyuan Luo, Yuan Zhang, Min Yang, and Minghui Yang. 2025. ApkDiffer: Accurate and Scalable Cross-Version Diffing Analysis for Android Applications. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 363 (October 2025), 28 pages. https://doi.org/10.1145/3763141

1 Introduction

Currently, end-users or third-party security analysts, who have no access to the app source code, are in severe need of reliable app diffing tools to help tease out the security or privacy risks of those ever-evolving wild apps. To be specific, Android apps are frequently updated over time to maintain usability, e.g., about 25% of popular apps [48] from Google Play release new versions on a weekly basis. However, those newly deployed app versions are found to leak more privacy information [55], introduce new bugs [28] or even silently launch malicious backdoors [4, 5]. To

Authors' Contact Information: Jiarun Dai, Fudan University, China, jrdai@fudan.edu.cn; Mingyuan Luo, Fudan University, China, myluo24@m.fudan.edu.cn; Yuan Zhang, Fudan University, China, yuanxzhang@fudan.edu.cn; Min Yang, Fudan University, China, m_yang@fudan.edu.cn; Minghui Yang, OPPO, China, yangminghui@oppo.com.

ACM 2475-1421/2025/10-ART363
https://doi.org/10.1145/3763141

thoroughly audit whether an app update brings these veiled issues, a fundamental step is to first infer the code changes caused by the update, that is to diff the code of pre-update version and post-update version in a fine-grained way. In practice, such a diffing task is usually given at the method-level (i.e., function-level). With the method mapping in hand, one can concentrate on examining the added/removed/modified methods to ease the auditing of app updates. Meanwhile, the alignment information also eases the reverse engineering of the updated app, by migrating the reversing knowledge (e.g., method profiles) of an old version to the new one.

To date, various diffing tools [6, 13, 24, 27, 34, 42] (e.g., BinDiff [24]) have been proposed dedicated for aligning binaries. Comparatively, for Android apps, existing research [16, 25, 41, 58, 61, 69] mostly put an emphasis on repackaged/cloned app detection. They can quantitatively measure how similar are given apps, while failing to further pinpoint the detailed code changes between similar apps. There do exist some app diffing works [20, 39], but either with overly coarse granularity (e.g., class-level [20]) in diffing or merely adopting straightforward code-structure-based diffing strategy (e.g., match packages and their belonging methods [39]), which are not resilient to cross-version obfuscation deviations, code structure refactoring and implementation updates.

Problem Description. In essence, the method alignment has long been formulated as graph matching problem [24]. It aims to find a solution of method correspondences between two call graphs that minimizes both node (i.e., methods) discrepancies and edge (i.e., caller-callee relations) disagreements. However, due to the combinatorial nature of node-to-node comparison and edge-to-edge comparison, graph matching is theoretically NP-hard [43] and an exact optimal matching can only be applied to small graphs (e.g., graphs with dozens of nodes). Considering that an Android app usually contains tens of thousands of methods [47], it is obviously infeasible to scale an optimal graph matching algorithm to cross-version method alignment of Android apps.

Existing Method Alignment Algorithms. In the field of binary diffing, although the past decade has witnessed rapid advances in method similarity calculation [15, 23, 29, 31, 42, 44, 59, 60, 67] (i.e., to measure the similarity between two single methods), less light is shed on the NP-hard whole-software method alignment problem (i.e., call graph matching problem). To alleviate the scalability issue of call graph matching, previous binary diffing works [6, 13, 24, 27, 34, 42] usually establish kinds of approximation algorithms, by neglecting either node-to-node comparison or edge-to-edge comparison. For instance, the Greedy Search algorithm (e.g., Diaphora [6]) and the Linear Assignment algorithm (e.g., BinHunt [27]) eliminate edge information. They reduce the graph matching problem to a bipartite matching problem, which aims to correlate two sets of discrete nodes rather than connected ones. Alternatively, the Neighbor Search algorithm (e.g., Bindiff [24]) simplifies the node-to-node comparison by only trying to align neighbors of already aligned nodes. Although these approximations help solve the call graph matching problem in feasible time, significant alignment errors would occur. Besides, in the field of app diffing, some existing works [39] attemp to reduce the complexity of whole-app method alignment task, by first matching high-level code structures (i.e., packages) and then matching their belonging methods. However, the widely existing code-structure-related obfuscation and refactoring would easily cause error propagation of method alignment.

Moreover, existing works indiscreetly adopt a simplified 1-to-1 method alignment mechanism (i.e., one method in a version can only be aligned with one method in another version). However, method extract/inline, which is one of the most popular types of code refactoring [49] and compiler optimization [32], would probably derive complex 1-to-n or n-to-n method alignment relations (examples see Figure 1). As highlighted in [32], suppose that buggy or sensitive methods are extracted or inlined, existing 1-to-1 alignment tools cannot accurately correlate them across versions.

Our Work. In light of this, we propose ApkDiffer, a scalable, effective and extract/inline-aware tool dedicated for aligning methods between versions of the same closed-source Android app. Different from existing alignment algorithms, ApkDiffer tackles the scalability issues of call graph matching via a novel decomposition-based solution, which can achieve a good balance between scalability and effectiveness. To be specific, it decomposes the large call graphs of two app versions into matched pairs of sub-graphs, by clustering methods that serve equivalent app functionalities across versions. In such a manner, ApkDiffer can avoid computation-heavy large-scale graph matching and achieve satisfactory scalability by only aligning methods among decent-scaled sub-graphs. To further ensure high effectiveness, ApkDiffer brings ideas from advanced graph matching algorithms [18, 38, 68] to comprehensively consider both node and edge consistencies when deciding 1-to-1 alignment results. Finally, ApkDiffer identifies overlooked n-to-n method correspondences, by verifying whether callers and callees of already aligned methods are potentially extracted methods. The challenges and key insights to realize the above ideas are elaborated in §3.

Evaluation Results. We conducted a series of experiments to demonstrate the advantages of ApkDiffer. Due to the lack of public benchmark datasets for app diffing, we first constructed a large benchmark dataset from F-Droid [7] apps, containing 150 app versions and over 300,000 ground truth 1-to-1 method mappings. Experimentally, ApkDiffer shows high effectiveness on this benchmark dataset, having 96.06%/92.76% precision and 86.70%/81.79% recall when identifying 1-to-1 method mappings between app versions with 1-/5-version-gap. As comparison, ApkDiffer is noticeably superior to all existing method alignment algorithms in effectiveness, achieving 3.69% ~ 34.86% precision improvement and 16.12% ~ 64.30% recall improvement. Besides, ApkDiffer can reliably identify n-to-n method correspondences which are overlooked by existing alignment algorithms, with 84.0% precision and 82.8% recall. Moreover, ApkDiffer has satisfactory efficiency due to decomposition-based design and can practically scale to complex apps with large codebase. We also conducted thorough ablation studies to emphasize the importance of each design choice.

Utility of ApkDiffer. To further demonstrate the utility of our work, we utilized ApkDiffer to assist the privacy assessment of real-world app updates. Notably, previous studies [55, 64] have already confirmed that Android apps are collecting more user-sensitive privacy information during the version update process, technically by observing cross-version changes in permission usage [64] or network traffic [55]. However, without the capability of fine-grained app diffing, these studies cannot pinpoint the exact code locations where app versions deviate in privacy collection, consequently hard to achieve an in-depth diagnosis of app privacy issues (e.g., to understand the intention or threats of evolved privacy-collection behaviors through code-level analysis). To fill this gap, we utilized ApkDiffer to enumerate all method-level code changes between app versions, and then automatically checked whether each code change causes a deviation in privacy-collection behavior (i.e., sensitive API invocation that accesses user privacy).

Following the above idea, we tracked the one-year evolution of 100 popular Google Play apps (each with over 50 million downloads). Results show that each quarterly update of these apps may introduce about 11.0% previously unknown privacy-collection behaviors. After classifying the code locations of these privacy-collection behaviors, we found that about 70% of them are caused by app functionality updates and the remaining ones occur in version-shared functionalities. With manual code-level diagnosis, we confirmed that some evolved privacy-collection behaviors are ill-intended and may stealthily bring new threats (e.g., abuse of user location or identity theft) to user privacy.

Contributions. This paper makes the following contributions:

- We propose ApkDiffer, an end-to-end alignment tool that can precisely and efficiently identify both 1-to-1 and n-to-n method correspondences across Android app versions.

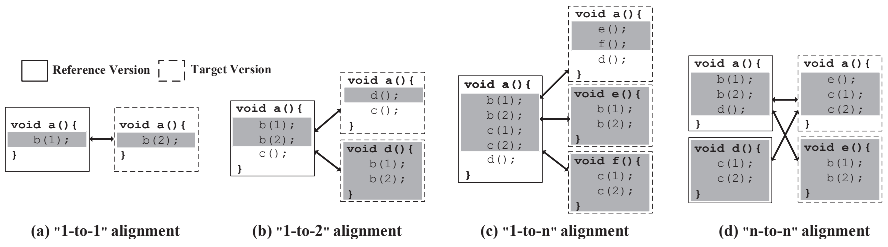


Fig. 1. Examples for different types of method alignment relations. Method extract/inline causes 1-to-2 alignment relations and nested method extract/inline probably derives more complex 1-to-n (n>2) or n-to-n (n ≥ 2) relations.

- We conducted extensive experiments to demonstrate the effectiveness and efficiency of ApkDiffer. Results on the benchmark dataset show that ApkDiffer is superior to existing alignment methodologies in both precision and recall, meanwhile having satisfactory time cost.
- We utilized ApkDiffer to track the evolution of real-world popular apps and conclusively uncovered the privacy threats of app updates.

2 Background

In this section, we first define the necessary symbols, and introduce the problem formulation. Then, we review existing alignment algorithms and pinpoint their limitations.

Symbol Definition. The call graph G_x of app version V_x can be defined as $G_x(N_x, E_x)$. The node set N_x is a finite set that contains all the app methods, and the edge set $E_x \subset N_x \times N_x$ represents the caller-callee relations between methods.

Problem Formulation. Given two app versions V_r and V_t, the method alignment can be formulated as a graph matching problem, which aims to identify a solution of node correspondences between $G_r(N_r, E_r)$ and $G_t(N_t, E_t)$. For ease of expression, the solution of method alignment can be presented as a matrix $P \in \{0,1\}^{|N_r|\times|N_t|}$, where $P[i, j] = 1$ only if the method $n_r^i \in N_r$ is aligned with the method $n_t^j \in N_t$ (otherwise, $P[i, j] = 0$). Considering the existence of method extract/inline, there might be different types of method alignment relations across versions, including 1-to-1, 1-to-n or even n-to-n (examples see Figure 1).

According to the graph theory [56], searching for an optimal solution of node correspondences across graphs is to find one that maximizes the sum of the node-to-node and edge-to-edge similarities. In the problem context of method alignment, the node-to-node similarity reflects the semantic and syntactic consistency between two given methods, technically calculated via code similarity measures [23, 29, 31, 42, 59, 67]. The edge-to-edge similarity further evaluates the consistency of the caller-callee relations between aligned methods.

Existing Method Alignment Algorithms. Due to the combinatorial nature of node-to-node comparison and edge-to-edge comparison, the graph matching problem is known to be NP-hard and an exact optimal algorithm can only work on very small graphs (e.g., < 30 nodes). Therefore, existing works [13, 22, 24, 27, 34, 42] in method alignment mainly devise algorithms to solve a solution approximately, sacrificing effectiveness for efficiency. Even worse, they all miss the n-to-n method correspondences caused by method extract/inline. In the following, we will briefly introduce these alignment algorithms and their limitations, with symbols defined at the beginning of this section.

- *Greedy Search* (*GS*) (e.g., Diaphora [6]) For each $n_t \in N_t$, the *GS* algorithm determines its aligned method $n_r \in N_r$, which shares the highest code similarity score with n_t among all methods in N_r. First, it ignores the consistency of caller-callee relations between aligned methods, consequently producing biased alignment results. Second, it only accepts the locally best alignment solution even when there exists a globally better one.

- *Linear Assignment* (*LA*) (e.g., BinHunt [27]) The *LA* algorithm reduces the complex graph matching problem to a bipartite matching problem, which treats the methods of N_t and N_r as discrete nodes. Such problem reduction enables one to solve optimal alignment results in polynomial time (e.g., using the Hungarian algorithm [36]). However, similar to the *GS* algorithm, since the topology consistency is ignored, it would probably produce imprecise results.
- *Neighbour Search* (*NS*) (e.g., BinDiff [24]) The *NS* algorithm first identifies an initial set of aligned methods between N_r and N_t following specific heuristics (e.g., align methods that refer to the same constant string variable). Then, it attempts to only align methods that are neighbors (i.e., callers or callees) of already aligned ones in a 1-to-1 manner. Although this scheme can largely eliminate edge disagreements between aligned methods, it suffers from error propagation that previous alignment errors affect the alignment of remaining methods.

Notably, built on top of the above approximated alignment algorithms, existing app diffing works (e.g., PEDroid [39]) would additionally leverage code-structure-based codebase decomposition to further reduce the complexity of the alignment task. For example, PEDroid would first try to align packages or classes, and then their belonging methods. It assumes that most obfuscators and cross-version code updates will not affect the internal code structures. Obviously, this strong assumption easily fails in real-world scenarios.

3 Design Overview

In this section, we will introduce the overall idea of ApkDiffer (see §3.1), the challenges to implement this idea (see §3.2), our key insights to overcome these challenges (see §3.3), and finally the workflow of ApkDiffer (see §3.4).

3.1 Overall Idea

The pain point of cross-version method alignment for Android apps is that, released apps usually contain a large number of methods (e.g., over 60k [47]) due to an unclear boundary between app codebase and library codebase, making graph-based alignment face severe scalability issues. Existing works mitigate the scalability issue via approximation algorithms (see §2) at the cost of noticeable effectiveness loss. To ease such effectiveness-efficiency conflicts, our overall idea is problem decomposition with two key stages:

Stage-I: Graph Decomposition. Based on a partition metric $\mathbb{K}$, G_r and G_t are first partitioned into sets of sub-graphs, denoted as $\mathbb{K}(G_r) = \{G_r^1, ..., G_r^p\}$ and $\mathbb{K}(G_t) = \{G_t^1...., G_t^q\}$. Furthermore, a sub-graph mapping function $\mathbb{F} : \mathbb{K}(G_r) \rightarrow \mathbb{K}(G_t)$ is proposed to identify matched pairs of sub-graphs.

Stage-II: Method Alignment. For each matched pair of sub-graphs G_r^x and G_t^y (i.e., $G_r^x \in \mathbb{K}(G_r)$, $G_t^y \in \mathbb{K}(G_t)$, $\mathbb{F}(G_r^x) = G_t^y$), we perform method alignment between G_r^x and G_t^y to identify both the 1-to-1 and n-to-n method correspondences.

The decomposition-based solution can improve scalability, because it helps avoid the large-scale graph matching and reduce the problem scale by only aligning methods among sub-graphs. When handling limited-scale sub-graphs, we have a chance to deploy a more cautious alignment algorithm. Hence, it can potentially achieve a good compromise between effectiveness and efficiency.

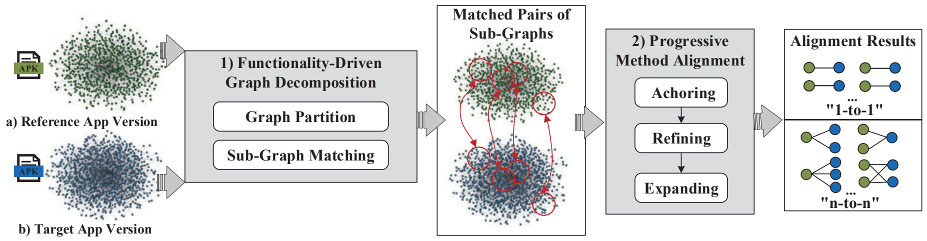


Fig. 2. Workflow of ApkDiffer.

3.2 Challenges

However, the two stages (i.e., *Graph Decomposition* and *Method Alignment*) of this decomposition-based solution, if not carefully designed, would harm both the efficiency and effectiveness of ApkDiffer. Here, we discuss the non-trivial challenges for the design of these two stages.

Challenges for Designing Graph Decomposition.

- (C1) The scale of sub-graphs got by partition should be neither too large nor too small to achieve a good performance. When the scale is too large, ApkDiffer would suffer from heavy-weight computation when aligning methods among sub-graphs. When the scale is too small, the sub-graph matching process would introduce unacceptable overhead.
- (C2) The decomposition procedure should not introduce significant effectiveness loss of method alignment. That is, we should try to ensure that two aligned methods, can always be classified into two matched sub-graphs, which requires a careful design of graph partition metric and sub-graph matching methodology.

Challenges for Designing Method Alignment.

- (C3) According to the problem formulation (see §2), the call graph matching should comprehensively consider both node similarities (i.e., code similarities between methods) and edge similarities (i.e., consistencies of caller/callee relations between aligned methods), rather than primarily focusing on one of them like existing alignment algorithms.
- (C4) To support the identification of method extract/inline, the method alignment should be built on top of the complex n-to-n alignment mechanism, which causes exponentially larger search space than the simple 1-to-1 mechanism. It intuitively poses greater challenges to design a reliable while computationally feasible alignment solution.

3.3 Key Techniques

To tackle these challenges, we propose two key techniques, namely *Functionality-driven Graph Decomposition* and *Progressive Method Alignment*, and our insights behind are presented as follows:

Functionality-driven Graph Decomposition. Here, our key observation is that, Android apps are event-driven software, where the app runtime behaviors are triggered through different types of events, including GUI events, lifecycle events and system events. Each event-handling can be regarded as a functionality unit of the app. This observation inspires us to propose a functionality-driven graph decomposition, which can potentially tackle the aforementioned challenges.

- (Tackle C1) To make code reusable and maintainable, developers commonly prefer code modularization [8] to implement each functionality unit with a limited portion of code. Therefore, by clustering methods that serve the same functionality unit, we can probably achieve a balanced call graph partition, which produces decent-scaled sub-graphs.

- (Tackle C2) The aligned methods across versions should expose identical or similar code behaviors, thus probably serving equivalent functionality units. As such, we can match sub-graphs by verifying the equivalence of corresponding functionality units across versions, and after that, only need to align methods among matched sub-graphs. This functionality-driven decomposition strategy should be more stable than those code-structure-based ones (e.g., package-hierarchy-based decomposition [39]).

Progressive Method Alignment. According to recent advances [18, 38, 68] in graph matching, progressive algorithms have attracted a lot of attention due to their high effectiveness and efficiency. They first identify a sub-optimal solution, and refine it incrementally in adaptive search space, so as to return a satisfactory solution with acceptable time cost. Borrowing this idea, we design a 3-step progressive alignment algorithm, including *Anchoring*, *Refining* and *Expanding*, to overcome the challenges mentioned in §3.2.

- (Tackle C3) The *Anchoring* step solves an initial set of 1-to-1 alignment results by maximizing the sum of only node similarities. After that, the *Refining* stage switches the objective function of alignment to the sum of node similarities and edge similarities, and incrementally mutates the alignment status of selective methods to see whether it helps boost the objective. In this way, we can gradually correct previous alignment errors that are caused by ignorance of edge comparison.
- (Tackle C4) Finally, the *Expanding* step works by thoroughly verifying whether any neighbors of 1-to-1 aligned methods are actually extracted methods. The key rationale behind is that, two aligned methods across versions, even with certain code pieces extracted or inlined, should still share a high similarity score with each other and thus can possibly be aligned in previous steps. Besides, the extracted/inlined methods should always be neighbors of the original method on the call graph.

3.4 Workflow

As illustrated in Figure 2, the general workflow of ApkDiffer consists of following steps:
Stage-I: Functionality-Driven Graph Decomposition.

- *Graph Partition:* ApkDiffer identifies all functionality units in each app version, and performs graph partition by grouping methods that serve the same functionality unit.
- *Sub-graph Matching:* ApkDiffer matches sub-graphs across versions by measuring the equivalence between functionality units.

Stage-II: Progressive Method Alignment.

- *Anchoring:* ApkDiffer identifies an initial set of 1-to-1 method mappings among each matched pair of sub-graphs.
- *Refining:* ApkDiffer refines the 1-to-1 mappings by incrementally correcting alignment errors.
- *Expanding:* ApkDiffer identifies n-to-n method correspondences by searching overlooked extracted methods.

4 Approach

4.1 Functionality-Driven Graph Decomposition

As introduced in §3.3, ApkDiffer utilizes functionality unit as the key metric for graph decomposition. Here, we first introduce the definition of functionality units, followed by the functionality-driven graph partition and sub-graph matching.

Definition of Functionality Unit. Due to the framework-based model and the event-driven nature of Android apps, developers can accordingly customize different framework callbacks to

Table 1. The Whitelist of Framework Callback Classes.

Event-Type	# of Callback Classes
GUI-Event	82
Lifecycle-Event	7
System-Event	381

handle various runtime events. Hence. all executed methods to handle a specific event, grouped together, can be viewed as a basic functionality unit of the app. More formally, given the call graph $G_x(N_x, E_x)$, a functionality unit can be represented as a sub-graph $G_x^u(N_x^u, E_x^u)$ of G_x, where $N_x^u \subsetneq N_x$. Each sub-graph $G_x^u(N_x^u, E_x^u)$ has only one entry point method $entry(G_x^u)$, which refers to a customized framework callback. For any method $m \in N_x^u$, m is always reachable starting from $entry(G_x^u)$ on the call graph.

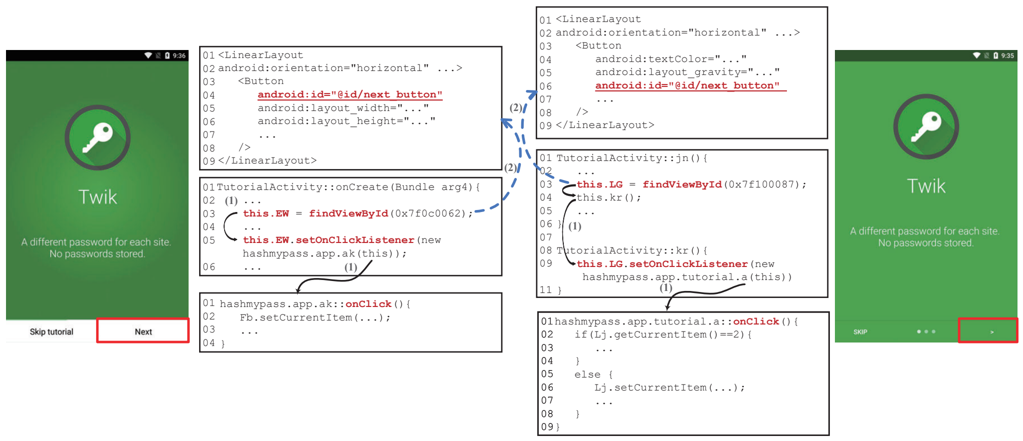
Similar to existing works [40, 46], we classify the functionality units into three categories, according to the type of events to be handled: ❶ *GUI-Event Handlers* respond to user actions that originate from the graphical interface (e.g., button click and text edition); ❷ *Lifecycle-Event Handlers* manage lifecycle state transitions of app or UI components (e.g., creation of activity or fragment); ❸ *System-Event Handlers* are executed when the app receives system notifications (e.g., updated GPS locations).

Functionality-driven Graph Partition. Existing graph partition techniques are usually designed based on the modularity maximization principle, which aims to group densely interconnected nodes as a sub-graph. These kinds of partition techniques are widely used in community detection among social networks [21], object localization in computer vision [62] and so on. However, the graph partition in ApkDiffer acts as a pre-processing step of method alignment between Android app versions. It should not only split the call graph into limited-sized sub-graphs with low time cost, but also ease the precise matching between sub-graphs. As thoroughly discussed in §3.2, the functionality unit should be a proper partition metric that meets these requirements.

The detailed procedure of *Functionality-driven Graph Partition* is presented as follows: Firstly, we identify all app callbacks that are implemented to handle GUI events, lifecycle events or system events. To achieve this goal, we manually construct a whitelist of framework classes (summarized in Table 1) where the callback interfaces are defined, and identify app methods that override one of these framework callbacks. Secondly, for each identified app callback (i.e., the entry point of a functionality unit), a unique sub-graph is constructed by grouping all methods that are reachable from this callback method on the call graph.

Functionality-driven Sub-Graph Matching. After the graph partition, we can get a list of sub-graphs for each app version, i.e., $\mathbb{K}(G_r) = \{G_r^1, G_r^2 ... G_r^p\}$ and $\mathbb{K}(G_t) = \{G_t^1, G_t^2 ... G_t^q\}$. Subsequently, the sub-graph matching aims to find a correspondence between $\mathbb{K}(G_r)$ and $\mathbb{K}(G_t)$, i.e., $\mathbb{F} : \mathbb{K}(G_r) \rightarrow \mathbb{K}(G_t)$. Each sub-graph got by partition uniquely represents a functionality unit and has only one entry point method. Thus, we choose to match functionality units (i.e., sub-graphs) by correlating these entry point methods.

- *Match GUI-Event Callbacks.* The GUI-event callbacks are registered and invoked to handle user operations on interactive UI elements. Our key insight is that, conceptually same user interfaces should trigger consistent app behaviors. Hence, to match GUI-event callbacks, one can verify whether these two callbacks are bound with identical UI elements across versions. To this end, recent studies [26, 70] have validated that the identifier name (i.e., $ViewIdResourceName$) can be a very reliable clue to determine whether two UI elements across versions are identical. For



```
<LinearLayout
android:orientation="horizontal" ...>
   <Button
      android:id="@id/next_button"
      android:layout_width="..."
      android:layout_height="..."
      ...
   />
</LinearLayout>
```

```
TutorialActivity::onCreate(Bundle arg4){
    ...
    this.EW = findViewById(0x7f0c0062);
    ...
    this.EW.setOnClickListener(new
    hashmypass.app.ak(this));
    ...
```

```
hashmypass.app.ak::onClick(){
   Fb.setCurrentItem(...);
   ...
}
```

```
<LinearLayout
android:orientation="horizontal" ...>
   <Button
      android:textColor="..."
      android:layout_gravity="..."
      android:id="@id/next_button"
      ...
   />
</LinearLayout>
```

```
TutorialActivity::jn(){
     ...
     this.LG = findViewById(0x7f100087);
     this.kr();
     ...
}

TutorialActivity::kr(){
     this.LG.setOnClickListener(new
      hashmypass.app.tutorial.a(this))
}
```

```
hashmypass.app.tutorial.a::onClick(){
   if(Lj.getCurrentItem()==2){
      ...
   }
   else {
      Lj.setCurrentItem(...);
      ...
   }
}
```

Fig. 3. Example for matching GUI-event callbacks between versions of a F-Droid app (Twik-1.3.1 vs Twik-1.3.6). We follow two steps to determine that two *onClick*() callbacks can be matched. Firstly, we utilize data-flow analysis (marked as (1) in the figure) to identify the UI elements (i.e., *this.EW* and *this.LG*) that the callback methods are bound with. Secondly, we find that two UI elements share the same identifier name "next_button" (marked as (2) in the figure).

instance, [70] has extensively evaluated the effectiveness of this ID-based heuristic with real-world apps, showing only 2.28% false negative rate and 3.1% false positive rate on 42,504 pairs of cross-version GUI elements. Therefore, we choose to use this heuristic to match GUI-event callbacks: Firstly, for each callback, we identify the corresponding UI element by tracking the framework-based callback registration procedure with data-flow analysis. Secondly, we verify whether two UI elements share the same identifier name to match two given callbacks. For ease of understanding, Figure 3 shows a running example.

- *Match Lifecycle-Event Callbacks.* The Android framework allows developers to build lifecycle-aware app components and UI components (e.g. activities and fragments). The lifecycle-event callbacks, which manage the lifecycle state transition, are declared in corresponding component classes. Thus, we can match these callbacks by verifying whether they belong to identical component classes across versions. To accomplish this task, we resort to existing research efforts [71] in class matching.
- *Match System-Event Callbacks.* Similar to how we match lifecycle-event callbacks, we match system-event callbacks by matching their declaring classes.

Handling Sub-Graph Overlapping. After matching all these three types of event-handling callbacks, we can determine the matching relations between two sets of sub-graphs, i.e., $\mathbb{F} : \mathbb{K}(G_r) \rightarrow \mathbb{K}(G_t)$. But notably, the sub-graphs got by partition commonly have overlaps. The overlapping parts (example see Figure 4) are usually utility modules or library modules, and the methods within belong to more than one sub-graph. In practice, we will gradually merge these methods with caller-callee relationships to determine the overlapping parts. Since we need to align methods among each matched pair of sub-graphs (see §4.2), we should try best to avoid repeatedly aligning methods within the overlapping parts.

To alleviate this issue, we first extract the list of overlapping parts for app version V_r and V_t respectively, denoted as $\{O_r\}$ and $\{O_t\}$. Here, each $o_r \in \{O_r\}$ or $o_t \in \{O_t\}$ is a weakly-connected component on the call graph of V_r or V_t, i.e., given any two methods $m_i, m_j \in o_r$, there always exists a call path from m_i to m_j, or from m_j to m_i. Then, we try to align the overlapping parts across versions. For each $o_r \in \{O_r\}$ and $o_t \in \{O_t\}$, o_r can be aligned with o_t only when they share exactly the same entry points (i.e., o_r and o_t can be reached from aligned callbacks on the call graph of V_r and V_t). Finally, for each aligned pair (o_r, o_t), we remove their containing methods from other sub-graphs, and mark (o_r, o_t) as a new matched pair of sub-graphs.

Here, we set a strict policy to match overlapping parts since mismatched overlapped parts would cause error propagation to method alignment procedure. For matching overlapped parts, we expect to boost the efficiency of ApkDiffer and avoid introducing alignment errors, thus favoring precision over recall.

4.2 Progressive Method Alignment

Based on the results of graph decomposition, we seek to identify both the 1-to-1 and n-to-n method alignment relations among each matched pair of sub-graphs. In this section, we first give a global picture of the alignment algorithm, then present the algorithm details.

Progressive Alignment Algorithm. Inspired by recent advances [18, 38, 68] in graph matching, ApkDiffer tries to align methods among sub-graphs in a progressive manner, including 3 key steps (listed in Algorithm 1): *Anchoring*, *Refining* and *Expanding*. ❶ The *Anchoring* step computes the initial 1-to-1 alignment results with Linear Assignment algorithm, which may produce significant alignment errors without comprehensively considering the topology consistency. ❷ The *Refining* step is designed to correct the 1-to-1 alignment errors incrementally and iteratively. The key idea is to design a topology-sensitive objective function to re-evaluate each previously aligned pair, and seek better alignment solutions in adaptive search space. ❸ The *Expanding* step finally tries to expand the 1-to-1 alignment results to n-to-n ones, by identifying overlooked extracted methods.

For ease of understanding, Figure 5 shows a running example of the above procedure. In the following, we will introduce the algorithm details of these 3 steps respectively, with symbols defined in §2 (e.g., alignment results are expressed via a (0,1)-matrix P).

Step-I Anchoring. During *Anchoring*, we compute initial 1-to-1 alignment results by leveraging Linear Assignment algorithm (e.g., Hungarian algorithm [36]). To be specific, it searches for an optimal (0,1)-matrix P^*_{anchor} by only maximizing the sum of node similarities (i.e., code similarities between methods, denoted as Sim_{node}), under the 1-to-1 alignment mechanism.

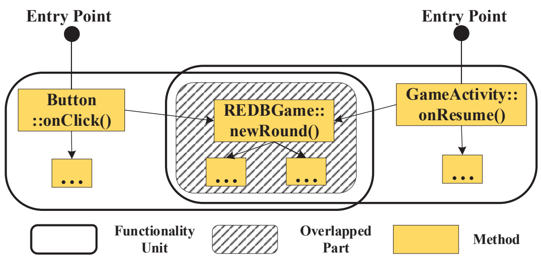


Fig. 4. Example for sub-graph overlapping in a F-Droid app com.phikal.regex-v1.2. The method *newRound*() and its callees play the role of game initialization and might be executed by different functionality units.

Algorithm 2 Progressive Alignment Algorithm.

```
Input:  Sub-graphs of $V_r$, $\mathbb{K}(G_r) = \{G_r^1, G_r^2 ... G_r^p\}$;
        Sub-graphs of $V_t$, $\mathbb{K}(G_t) = \{G_t^1, G_t^2 ... G_r^q\}$;
        Sub-graph Correspondence, $\mathbb{F} : \mathbb{K}(G_r) \to \mathbb{K}(G_t)$;
Output: 1-to-1 Method Alignment Results: $M_1$;
        n-to-n Method Alignment Results: $M_N$;
 1: $M_1 \leftarrow \varnothing$; $M_N \leftarrow \varnothing$;
 2: $tmp \leftarrow \varnothing$; // tmp results of 1-to-1 alignment relations
 3: /* Step-I Anchoring */
 4: for $f = (G_r^i, G_t^j)$ in $\mathbb{F}$ do
 5:     $tmp$.put($f$, $Anchor(G_r^i, G_t^j)$);
 6: /* Step-II Refining */
 7: for $Iteration$=1 to K do
 8:     for $f = (G_r^i, G_t^j)$ in $\mathbb{F}$ do
 9:         $Refine(tmp.get(f), G_r^i, G_t^j)$;
10: /* Step-III Expanding */
11: for $f = (G_r^i, G_t^j)$ in $\mathbb{F}$ do
12:     $M_1^f, M_N^f = Expand(tmp.get(f), G_r^i, G_j^t)$;
13:     $M_1 = M_1 \cup M_1^f$; $M_N = M_N \cup M_N^f$;
14: return $M_1, M_N$;
```

$$P^*_{anchor} = arg\max_P \sum_{P[i,j]=1} Sim_{node}(m_r^i, m_t^j)$$

Step-II Refining. Since the *Anchoring* step would produce significant alignment errors without considering edge similarities between aligned methods, the *Refining* step aims to incrementally correct this kind of 1-to-1 alignment errors. To enable this capability, we design a new objective function which integrates both node similarities (Sim_{node}) and edge similarities (Sim_{edge}), to re-evaluate the previous alignment solution and seek better ones. We implement Sim_{edge} based on *neighborhood consensus measure* [72]. That is, to calculate what percent of the local neighbors of two aligned methods are also correctly aligned with each other. Here, ApkDiffer still works under the 1-to-1 alignment mechanism.

$$P^*_{refine} = arg\max_P \sum_{P[i,j]=1} Sim_{fusion}(m_r^i, m_t^j)$$

$$Sim_{fusion}(m_r^i, m_t^j) = \frac{Sim_{node}(m_r^i, m_t^j) + Sim_{edge}(m_r^i, m_t^j)}{2}$$

To get a satisfactory solution under the above objective function, we propose an iterative *Decide-Search* mechanism (i.e., Decide potential misaligned methods and Search for better alignment solutions) as the optimal policy for this non-linear objective function. For ease of expression, we denote the (0,1)-matrix during k-th iteration as $P^{(k)}_{refine}$ and we have $P^{(0)}_{refine}$=P^*_{anchor} as the starting point. In the following, we introduce the *Decide-Search* mechanism which boosts the objective score when transforming $P^{(k-1)}_{refine}$ to $P^{(k)}_{refine}$.

- *Decide.* Given a previously aligned method pair (m_r^i, m_t^j) in the k-1 iteration, the *Decide* phase of k-th iteration mainly aims to verify whether there exist any m_r^x or m_t^y that $Sim^{(k)}_{fusion}(m_r^x, m_t^j)$

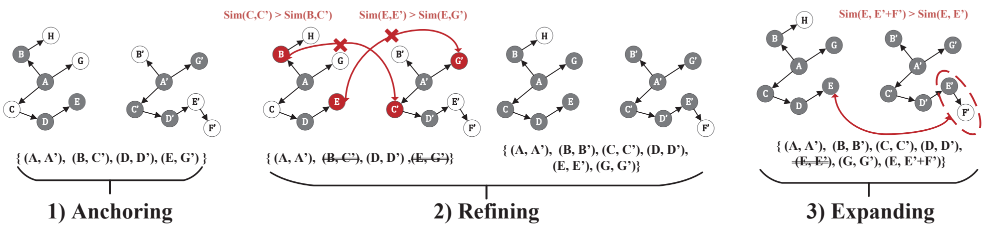

Sim(C,C') > Sim(B,C')
Sim(E,E') > Sim(E,G')
Sim(E, E'+F') > Sim(E, E')
{ (A, A'), (B, C'), (D, D'), (E, G') }
{ (A, A'), ~~(B, C')~~, (D, D') ,~~(E, G')~~}
{ (A, A'), (B, B'), (C, C'), (D, D'), (E, E'), (G, G')}
{ (A, A'), (B, B'), (C, C'), (D, D'), ~~(E, E')~~, (G, G'), (E, E'+F')}
1) Anchoring
2) Refining
3) Expanding

Fig. 5. Running example of *Progressive Method Alignment*. Firstly, the *Anchoring* step identifies 4 initial 1-to-1 alignment relations. Secondly, the *Refining* step determines that (B, C') and (E, G') might be wrong, by measuring the topology consistency across versions. Then it marks these methods as unaligned and re-compute new alignment relations to get 6 1-to-1 alignment results. Finally, the *Expanding* step identifies an extracted method F' and $(E, E' + F')$ forms a 1-to-2 alignment relation.

$> Sim^{(k-1)}_{fusion}(m_r^i, m_t^j)$ or $Sim^{(k)}_{fusion}(m_r^i, m_t^y) > Sim^{(k-1)}_{fusion}(m_r^i, m_t^j)$. If so, (m_r^i, m_t^j) is probably misaligned in the last $k-1$ iterations. Accordingly, we reset $P^{(k-1)}_{refine}[i, j]$ from 1 to 0.

- *Search.* After the *Decide* phase, the *Search* phase tries to seek new possible 1-to-1 alignment relations for those remaining unaligned methods.
  Here, we again leverage the Hungarian algorithm [36] to determine new alignment relations between these methods based on re-computed pairwise similarity scores $Sim^{(k)}_{fusion}$.

The *Refining* stage terminates when no new alignment relations can be found or the pre-defined limit of iteration is reached (set to 3 in the prototype of ApkDiffer).

Step-III Expanding. Finally, the *Expanding* step seeks to identify overlooked method extract/inline. Compared to previous two steps, 1) The objective function should be further adjusted to support the n-to-n alignment mechanism. For ease of expression, given two method sets $C_r = \{m_r^{i_1}, ..., m_r^{i_n}\} \subsetneq N_r$ and $C_t = \{m_t^{j_1}, ..., m_t^{j_n}\} \subsetneq N_t$, we define symbol $\phi_P(C_r, C_t)$ to denote the n-to-n alignment relation between C_r and C_t, i.e., $\forall\ m_r^i \in C_r,\ \forall\ m_t^j \in C_t,\ P[i, j] = 1$. Due to the characteristics of method extract/inline, methods in C_r or C_t should always be adjacent to the call graph. 2) New similarity metric Sim_{set} should be defined to evaluate the score between two sets of methods rather than two single methods. Basically, we can calculate $Sim_{set}(C_r, C_t)$ with $Sim_{fusion}(\pi(C_r), \pi(C_t))$, where $\pi(.)$ denotes a virtual method which is generated by merging all methods back to their corresponding callers in the given method set.

$$P^*_{expand} = arg\max_P \textstyle\sum_{\phi_P(C_r, C_t)} Sim_{set}(C_r, C_t)$$

$$Sim_{set}(C_r, C_t) = Sim_{fusion}(\pi(C_r), \pi(C_t))$$

Theoretically, the n-to-n alignment mechanism causes exponentially larger search space than the simple 1-to-1 mechanism, making it a computationally intensive task to identify method extract/inline. Based on key observations highlighted in §3.3, we figure out a reliable algorithm to solve a near optimal solution efficiently. Specifically, with the high-quality 1-to-1 alignment results $P^{(k)}_{refine}$ got from k-iteration refinement, we can search potentially extracted methods among unaligned callers or callees of those already aligned methods. For simplicity, we suppose that there exists an aligned pair of methods (m_r^i, m_t^j) and m_r^i has an unaligned callee m_r^k. The method m_r^k is viewed as an extract method only when $Sim_{fusion}\ (\pi(\{m_r^i, m_r^k\})\ , m_t^j) > Sim_{fusion}(m_r^i, m_t^j)$, implying that merging m_r^k back to m_r^i at the callsite makes m_r^i more similar to m_t^j. Furthermore, if the aligned pair $(\pi(\{m_r^i, m_r^k\}), m_t^j)$ still have unmatched callers or callees, we repeat above process to check whether there exist a nested method extract/inline.

5 Evaluation

We conducted experiments to extensively evaluate the effectiveness (RQ1, see §5.2) and the efficiency (RQ2, see §5.3) of ApkDiffer, including an ablation study to measure the contributions of its two key techniques (RQ3, see §5.4).

5.1 Experimental Setup

Prototype. We implemented a prototype of ApkDiffer with over 6,400 lines of Java code. Technically, our prototype uses FlowDroid [11] for call graph construction, and also optimizes FlowDroid to identify more app callback methods with an expanded list of callback interfaces (listed in Table 1). Moreover, our prototype uses Backstage [37] (a UI analysis framework) to associate UI elements with their corresponding callback handlers. Our implementation extends Backstage to support more types of UI elements (i.e., from 58 types to 82 types). As mentioned in §2, ApkDiffer doesn't aim to design a new code similarity function, but still relies one for code alignment. With a systematic evaluation of existing works [23, 29, 31, 42, 59, 67], our prototype uses a simple Jaccard distance-based [63] code similarity calculation method.

Baselines. As thoroughly summarized in §2, the method alignment algorithms used by existing diffing tools can be classified into three categories, namely Greedy Search (*GS*), Linear Assignment (*LA*) and Neighbor Search (*NS*). Therefore, we mainly compared ApkDiffer with these three alignment algorithms. Since there are no public implementations of these algorithms for Android apps, we re-implemented these algorithms with over 800 lines of Java code. In particular, we implemented the *LA* algorithm with the classic Hungarian algorithm [36]. Following the heuristics proposed in [24], our implementation of the *NS* algorithm first determines the initial alignment results by correlating methods that refer to the same constant string, and then gradually aligns the callers and callees of the aligned methods. To make a fair comparison, the above baseline implementations use the same code similarity function as ApkDiffer. Besides, in the field of app diffing, PEDroid [39] is a representative tool that has the end-to-end capability to align methods across different versions of the same app. Hence, we also directly compare ApkDiffer with PEDroid to demonstrate the advantages of our work.

App Datasets. Our evaluation requires a dataset consisting of cross-version Android apps. To the best of our knowledge, there are no such datasets that are publicly available. Hence, we decided to construct such a dataset from scratch based on apps from the F-Droid Open-source App Market [7]. During the app selection, to ensure the representativeness of the dataset, we filter those toy apps with less than 4,000 methods in the released APk file, or with less than 5 released app versions. Furthermore, as detailed in §5.2, the ground truth method mappings between APK files should be obtained from the compiling process of the app source code. To ensure the success of app compilation, we devoted great manual efforts in setting up the compilation toolchain for different app versions, fixing outdated third-party dependencies, enabling code obfuscation and compiler optimization, etc. Finally, we got 150 ready-to-compile app versions of 50 diverse apps, including Game, Video Player, Device Tool, etc. Specifically, for each app, we collected 3 released versions (V_α, V_β, V_γ) with fixed version gaps, i.e., V_β=V_α+1 and V_γ=V_α+5. From these app versions, we constructed two app datasets with different version gaps for the evaluation: ❶ $Dataset_{\Delta 1}$ contains 50 app pairs whose versions are (V_α, V_β); ❷ $Dataset_{\Delta 5}$ contains 50 app pairs whose versions are (V_α, V_γ).

Determining Code Similarity Measure. The key contribution of our work is to propose an accurate and scalable method alignment algorithm dedicated for diffing the entire codebases of app versions, while not a more reliable code similarity measure to compare two single methods. In real practice, ApkDiffer can cooperate with different similarity calculation techniques to compute method-to-method similarity scores (i.e., node-to-node similarity in call graph matching). After a systematic review of the literature, we find that existing code similarity measures [15, 23, 29, 31, 42, 59, 60, 67] mostly work on binaries, without support for Android apps. Here, we try our best to establish three method-level similarity calculation techniques for Android apps: ❶ *Sim_{T1}:* AndroSim [22] is a module implemented in AndroGuard [2]. AndroSim leverages compressor-based algorithms to compute the Normalized Compression Distance (NCD) [19] distance between two normalized methods as their similarity scores. ❷ *Sim_{T2}:* Centroid [16] encodes the structural characteristics of each method CFG into a 3-dimensional token and compute the token distance as the similarity score between methods. Since Centroid is not open source, we implemented it carefully by following the technical details presented in the paper with over 2,200 lines of Java code. ❸ *Sim_{T3}:* Existing works [59, 60] have validated that one can extract stable features from methods as the clue for similarity calculation. Following such practices, we extracted 4 types of features (listed in Table 2) for each app method. After that, we calculated the Jaccard distance [63] between feature sets of two methods as their similarity score.

Table 2. Extracted features to implement Sim_{T3}.

Feature	**Description**
Numeric Feature	Integer, Long, Float, Double with Constant Values
String Feature	References to Constant String
API Feature	Invoked Platform/System APIs
Instruction Feature	Number of *Switch*/*Throw*/*Ret* Instructions

Table 3. Effectiveness of different similarity measures.

Similarity Measure	$Dataset_{sim}$
	precision@1
Sim_{T1}	59.61%
Sim_{T2}	29.68%
Sim_{T3}	**67.53%**

From above three established similarity calculation techniques, we seek to pick up the one with the best performance to be utilized in ApkDiffer. To evaluate the performance of these similarity measures, we constructed another dataset $Dataset_{sim}$ by following the similar construction methodology of $Dataset_{\Delta 1}$ and $Dataset_{\Delta 5}$. To be more specific, we downloaded another 50 F-Droid apps (each app with two randomly selected versions) and got 66,927 true 1-to-1 method mappings among them.

Eventually, as shown in Table 3, we measured the top-1 precision (denoted as precision@1) of these three similarity calculation techniques on $Dataset_{sim}$ and Sim_{T3} has the best performance. But, it is surprising that even Sim_{T3} has a low top-1 precision (i.e., 67.53%). After thorough investigation, we find the low top-1 precision is mainly caused by numerous simple methods in Android apps (e.g., wrapper methods, getter/setter methods), which are hard for code similarity measures to differentiate. Even though, ApkDiffer achieves good results in aligning such methods, with the design of graph decomposition and progressive method alignment. Besides, to prune error alignment results during method alignment, only method pairs whose similarity scores exceed a pre-defined threshold can be aligned. We set the threshold to 0.4 since the similarity scores of 97.3% true method mappings in $Dataset_{sim}$ exceed 0.4.

Environment. All our experiments are conducted on a Ubuntu 18.04 server with 216 GB memory and 64 CPU cores (Intel Xeon Gold 5218) running at 2.30 GHz.

5.2 Effectiveness Evaluation (RQ1)

Inspired by the evaluation design of existing works on C/C++ binaries [13, 42], we first evaluated the effectiveness of ApkDiffer in 1-to-1 method alignment. In addition, different from all existing works, ApkDiffer supports n-to-n method alignment. Thus, we further designed experiments to measure its effectiveness in n-to-n method alignment.

5.2.1 Evaluating 1-to-1 Method Alignment. To evaluate the effectiveness, we need a ground truth dataset that labels the 1-to-1 method mappings between app versions. However, to the best of our knowledge, there is no such public benchmark for Android apps. Therefore, we first constructed a ground truth dataset, and then compared ApkDiffer with baselines on it.

Ground Truth Construction. Following the practice of existing works [13, 42], we labeled 1-to-1 method mappings for the collected app datasets $Dataset_{\Delta 1}$ and $Dataset_{\Delta 5}$ in two steps. First, we compiled the source code of V_α, V_β and V_γ for each app (with common code obfuscation and compiler optimization enabled) and collected the built APK files as APK_α, APK_β and APK_γ. During the compiling process, we also collected the generated ProGuard [9] mapping files, which record the original non-obfuscated method signature (i.e, package/class/method name) for each method in the built app. Second, we correlated the methods between APK_α and APK_β, and between APK_α and APK_γ, with the help of their ProGuard [9] mapping files. The principle for method correlation is simple, i.e., two methods can be mapped if they have identical original method signatures. Note that we excluded method mappings from framework packages (e.g., *java.**, *javax.**, *kotlin.**, *kotlinx.**, *android.support.** and *androidx.**) to enlarge the diversity of ground truth data. It is also notable that our dataset already includes a certain proportion of method pairs with significantly changed implementations but the same original method signature before obfuscation, especially those with a 5-version gap.

In all, we labeled 158,999 and 152,703 method mappings for $Dataset_{\Delta 1}$ and $Dataset_{\Delta 5}$, respectively. Note that the labeled method mappings do not contain n-to-n method alignment relations, which would be evaluated in §5.2.2. In the below, we compared ApkDiffer with the baselines in 1-to-1 method alignment on $Dataset_{\Delta 1}$ and $Dataset_{\Delta 5}$.

Experiment Design. We use precision and recall to evaluate the effectiveness of ApkDiffer and baselines in 1-to-1 method alignment. The two metrics are calculated as follows, where GT refers to ground truth method mappings and R refers to the set of results returned by the alignment tool.

$$Precision = \frac{|GT \cap R|}{|R|} \quad Recall = \frac{|GT \cap R|}{|GT|}$$

Since ApkDiffer supports n-to-n method alignment, it may return mappings between two method sets, e.g., (C_r, C_t) where $C_r \subsetneq N_r$ and $C_t \subsetneq N_t$. We mark (C_r, C_t) as a correct mapping if there exists a ground-truth method mapping (m_r, m_t) such that $m_r \in C_r$ and $m_t \in C_t$, when evaluating the effectiveness of 1-to-1 method alignment.

Experiment Results. The evaluation results are presented in Table 4. We can find that ApkDiffer significantly outperforms all baselines in both precision and recall on both datasets. We are glad to

Table 4. Effectiveness Results in 1-to-1 Method Alignment.

Tool	$Dataset_{\Delta 1}$		$Dataset_{\Delta 5}$	
	avg. precision	avg. recall	avg. precision	avg. recall
Greedy Search	65.57%	64.64%	57.90%	56.39%
Linear Assignment	65.66%	64.81%	58.73%	57.44%
Neighbor Search	92.37%	70.58%	87.33%	64.07%
PEDroid	90.31%	22.40%	89.24%	33.21%
ApkDiffer	**96.06%**	**86.70%**	**92.76%**	**81.79%**

[1] avg. precision = (precision(app_1) + ... + precision(app_N)) / N
[2] avg. recall = (recall(app_1) + ... + recall(app_N)) / N

find that, even on $Dataset_{\Delta 5}$, ApkDiffer achieves over 90% precision and over 80% recall, which demonstrates ApkDiffer as a useful tool for app diffing.

We manually investigated the alignment errors produced by these baselines. The key reason for the poor performance of *GS* and *LA* lies in that, they ignore the edge relations when aligning methods; thus they cannot determine the correct method mapping when there exist multiple similar candidate methods. *NS* falls short due to the error propagation, i.e., when a method is aligned incorrectly, all its neighbors would be wrongly aligned. The main reasons for the unsatisfactory performance of PEDroid [39] are twofold: (1) PEDroid evaluates the similarity of cross-version methods solely based on normalized instructions, overlooking caller-callee relations, which makes it difficult to distinguish between methods with significant modifications or minimal instructions; (2) PEDroid employs heuristic-based package matching and class matching strategies to decompose codebases. To be more specific, it first identifies identical packages that have identical classes, then identifies similar packages among those sharing the same hierarchical relationships with identical packages, matches similar classes among matched packages based on implementation features, and finally matches methods among matched classes. Obviously, this decomposition strategy is not resilient to code optimization, obfuscation, and refactoring, which might alter package hierarchies or class structures. Compared with the baselines, ApkDiffer not only comprehensively considers the graph structure during method alignment, but also introduces a progressive method alignment strategy to correct previous alignment errors. Also it is important to note that, our design of functionality-driven sub-graph matching also contributes to the high effectiveness of ApkDiffer. More specifically, ApkDiffer has 97.77% precision and 96.81% recall when matching sub-graphs (i.e., handler callbacks) between app versions in $Dataset_{\Delta 1}$, 97.56% precision and 96.76% recall when aligning those in $Dataset_{\Delta 5}$. The detailed matching results for different categories of functionality units are presented in Table 5.

ApkDiffer also produced certain alignment errors (i.e., false positives and false negatives). We took a close look at these alignment errors and find the reasons are three-fold.

- (causing 58.8% of FNs) First, ApkDiffer constructs sub-graphs by traversing the call graph and clustering methods that are reachable from the handler callbacks. However, due to the unsoundness [11] of static call graph construction and incompleteness of callback whitelist (Table 1), 11.26%/11.73% ground-truth method mappings are missed during call graph traversal of apps in $Dataset_{\Delta 1}$/$Dataset_{\Delta 5}$. These methods are thus not involved in the method alignment procedure.
- (causing 30.9% of FPs and 19.4% of FNs) Second, as demonstrated in Table 5, although ApkDiffer shows high precision and recall in matching sub-graphs, there still exist certain

Table 5. Breakdown Analysis of Functionality-driven Sub-graph Matching.

Event Type	$Dataset_{\Delta 1}$		$Dataset_{\Delta 5}$	
	avg. precision	avg. recall	avg. precision	avg. recall
GUI-Event	95.73%	94.64%	92.32%	91.75%
Lifecycle-Event	99.95%	99.77%	99.50%	98.85%
System-Event	96.21%	96.04%	94.19%	93.29%

[1] avg. precision = (precision(app_1) + ... + precision(app_N)) / N
[2] avg. recall = (recall(app_1) + ... + recall(app_N)) / N

mount of mismatched sub-graphs that mislead the method alignment procedure. We also collect false positives and false negatives method alignment results that occur among these mismatched sub-graphs.

- (causing 69.1% of FPs and 21.8% of FNs) Third, the remaining alignment errors all occur among correctly matched sub-graphs. Although the signatures of some methods keep the same during version updates and are collected in our ground-truth dataset, huge code changes are made on the method implementation. This kind of methods are hard to be correlated with totally different implementation.

5.2.2 *Evaluating n-to-n Method Alignment.* In all, ApkDiffer reports 2,989 n-to-n method mappings in $Dataset_{\Delta 1}$ and 6,433 method mappings in $Dataset_{\Delta 5}$. However, since there is no ground truth for the n-to-n method mappings, we cannot directly report the effectiveness of ApkDiffer in n-to-n method alignment. In fact, it still remains an open challenge [10, 17, 33, 50] to construct a benchmark for n-to-n method alignment, due to the huge manual labeling efforts. To evaluate the precision and recall of ApkDiffer in n-to-n method alignment, we designed the following two separated experiments. It is worth noting that we did not compare ApkDiffer with the baselines here because they do not support n-to-n method alignment.

Evaluating False Positives. Our experimental methodology is to randomly sample a subset of n-to-n method mappings returned by ApkDiffer for manual verification. Specifically, we selected 100/2,989 and 100/6,433 n-to-n method mappings reported by our tool on $Dataset_{\Delta 1}$ and $Dataset_{\Delta 5}$, respectively. To avoid the biases of manual verification, each n-to-n method mapping was checked by two authors, each of whom has at least 3-year experience in Android app reversing. After the manual verification, we found that ApkDiffer reports 88 and 80 correct method mappings on $Dataset_{\Delta 1}$ and $Dataset_{\Delta 5}$, respectively. That is, ApkDiffer achieves a precision of 84.0% (=168/200) in n-to-n method alignment.

Evaluating False Negatives. To evaluate the false negatives, our methodology is to construct a small set of true n-to-n method mappings and verify how many of them have been reported by ApkDiffer. To help build the n-to-n method mappings, we resorted to *RefMiner* [65], which can identify extract/inline candidates with AST-based source statement matching. By applying *RefMiner* in analyzing the code repository of the app dataset, we got 80 and 315 method extract/inline pairs from $Dataset_{\Delta 1}$ and $Dataset_{\Delta 5}$, respectively. However, after manual validation, we found only 227 pairs are correct method mappings. The wrong mappings are mainly caused by design limitations [51] of *RefMiner*. In all, we found ApkDiffer successfully identifies 188 of the 227 true method mappings. That is, ApkDiffer achieves a recall of 82.8% (=188/227) in n-to-n alignment.

In the above two experiments, we also found some errors of ApkDiffer in n-to-n method alignment, which are mainly caused by two reasons.

Table 6. Efficiency Results in Method Alignment (RQ2).

Tool	$Dataset_{\Delta 1}$		$Dataset_{\Delta 5}$	
	avg. cost	max. cost	avg. cost	max. cost
Greedy Search	349s	1,650s	448s	3,706s
Linear Assignment	1,183s	8,210s	1,832s	21,075s
Neighbor Search	459s	2,507s	411s	2,356s
PEDroid	**8.0s**	**29.4s**	**9.6s**	**28.2s**
ApkDiffer	521s	2,770s	538s	2,617s

[1] avg. cost = (cost(app_1) + ... + cost(app_N)) / N
[2] max. cost = max(cost(app_1), ... , cost(app_N))

- (causing 78.1% of FPs and 82.1% of FNs) First, ApkDiffer obtains the n-to-n method mappings by expanding 1-to-1 mappings (see §4.2). Thus, errors in identifying 1-to-1 method correspondence inevitably propagate to the expanding procedure.
- (causing 21.9% of FPs and 17.9% of FNs) Second, during version updating, method extract/inline operations are usually accompanied with additional code updates (e.g., non-negligible code changes are applied on the extracted code pieces). These code changes make it hard to determine whether a target method is actually an extracted method or a newly added method during version update.

Despite these limitations, the near 90% precision and recall on $Dataset_{\Delta 1}$ and near 80% precision and recall on $Dataset_{\Delta 5}$ indicate that ApkDiffer is an effective end-to-end automated solution for identifying method extract/inline across app versions.

5.3 Efficiency Evaluation (RQ2)

We evaluated the time cost of ApkDiffer in aligning app versions from $Dataset_{\Delta 1}$ and $Dataset_{\Delta 5}$. As shown in Table 6, ApkDiffer costs about 500s on average to align app versions from $Dataset_{\Delta 1}$ and $Dataset_{\Delta 5}$. To be specific, 19.2% of the runtime is spent on static call graph construction and UI analysis, 23.9% on call graph decomposition and the remaining 56.9% on method alignment. We also compared ApkDiffer with four baselines (i.e., *GS*, *LA*, *NS* and *PEDroid*). The average time cost of ApkDiffer is slightly more than that of *GS* and *NS*. This is mainly because ApkDiffer is enabled with the additional capability of identifying n-to-n method mappings. Among all these tools, *LA* costs the most time, because the computational cost of the Hungarian algorithm is $O(n^3)$. Impressively, *PEDroid* requires only about 10s to align two app versions. The primary reason is that *PEDroid* relies on lightweight heuristics to decompose the codebase (e.g., package hierarchy and class structure). However, as demonstrated in §5.2, this pursuit of high efficiency comes at the expense of unsatisfactory effectiveness of method alignment.

As shown in Table 6, we can also find that ApkDiffer can practically scale to complex apps with large codebase (e.g., less than 0.5h to align the app with over 40,000 methods). Surprisingly, ApkDiffer has better performance on large apps even when compared to *GS* which is an efficiency-oriented approximation algorithm. This mainly owes to the graph decomposition design of ApkDiffer, which limits the matching scope into small sub-graphs.

Table 7. Ablation Study Results on Key Techniques of ApkDiffer (RQ3).

Evaluation Target	Tool	*Dataset*$_{\Delta 1}$ avg. precision	*Dataset*$_{\Delta 1}$ avg. recall	*Dataset*$_{\Delta 1}$ avg. cost	*Dataset*$_{\Delta 1}$ max. cost	*Dataset*$_{\Delta 5}$ avg. precision	*Dataset*$_{\Delta 5}$ avg. recall	*Dataset*$_{\Delta 5}$ avg. cost	*Dataset*$_{\Delta 5}$ max. cost
Functionality-Driven Graph Decomposition	ApkDiffer⊖	90.60%	79.82%	1,244s	7,710s	82.84%	74.02%	1,821s	15,011s
	ApkDiffer$_c$	90.33%	65.28%	**238s**	**817s**	83.77%	64.90%	**224s**	**901s**
	ApkDiffer$_p$	88.10%	71.47%	305s	1076s	81.65%	67.13%	329s	993s
	ApkDiffer	**96.06%**	**86.70%**	521s	2,770s	**92.76%**	**81.79%**	538s	2,617s
Progressive Method Alignment	*ApkDiffer(GS)*	80.85%	72.94%	**485s**	**2,561s**	76.30%	66.76%	480s	**2,470s**
	ApkDiffer(LA)	86.29%	77.80%	519s	2,643s	78.80%	68.95%	508s	2,610s
	ApkDiffer(NS)	93.71%	83.02%	510s	2,587s	88.87%	75.47%	**466s**	2,571s
	ApkDiffer	**96.06%**	**86.70%**	521s	2,770s	**92.76%**	**81.79%**	538s	2,617s

[1] ApkDiffer⊖ denotes ApkDiffer without *Functionality-Driven Graph Decomposition*

[2] ApkDiffer$_{c/p}$ denotes replacing *Functionality-Driven Graph Decomposition* with class-based or package-based decomposition

[3] ApkDiffer(*) denotes replacing *Progressive Method Alignment* with baseline alignment algorithms

5.4 Ablation Study (RQ3)

ApkDiffer features two key techniques to provide scalable and effective cross-version method alignment, i.e., *Functionality-Driven Graph Decomposition* and *Progressive Method Alignment.* We conducted experiments to evaluate how do these two techniques facilitate cross-version app diffing.

Ablation Study on Graph Decomposition. ApkDiffer introduces *Functionality-Driven Graph Decomposition* to boost the efficiency of method alignment by splitting a single call graph into many small sub-graphs. Thus, we first removed the graph decomposition from ApkDiffer to see how the efficiency and effectiveness of ApkDiffer was affected. The experiments were conducted on *Dataset*$_{\Delta 1}$ and *Dataset*$_{\Delta 5}$ and the effectiveness was measured on the ground truth of 1-to-1 method mappings. The evaluation results are shown in Table 7. We can find that graph decomposition largely improves the efficiency of ApkDiffer. Both the average time cost and the maximum time cost of aligning a pair of app versions have been reduced by graph decomposition. More surprisingly, graph decomposition also noticeably improves the precision and recall of method alignment. The main reason is that method alignment is easier to achieve good performance in a smaller matching scope. The results clearly render the *functionality* as an appropriate granularity to decouple Android apps for method alignment, which improves both efficiency and effectiveness.

We conduct a comparative analysis between our functionality-driven codebase decomposition approach and two conventional structural decomposition methods: (1) class-based decomposition (first matching classes across apps, then aligning methods within matched classes) and (2) package-based decomposition (first matching packages across apps, then aligning methods within matched packages). Specifically, we adopt *ApkDiff* [20] and *LibPecker* [71] as representative baselines for class-based and package-based decomposition, respectively. To ensure fair comparisons, we utilize the class/package matching results generated by these baseline tools and apply our *Progressive Method Alignment* technique to matched classes or packages. As demonstrated in Table 7, the existing code-structure-based methods outperform our functionality-driven approach in efficiency, primarily due to the computational overhead of precisely determining functional boundaries in our method (e.g., requiring intensive call graph traversal to identify functionality-relevant code). However, as demonstrated in Table 5, our decomposition methodology is more resilient to code obfuscation and cross-version code updates, thus having more satisfactory precision/recall in method alignment.

Ablation Study on Progressive Method Alignment. ApkDiffer proposes *Progressive Method Alignment* to improve the accuracy of method alignment, which also supports n-to-n method alignment. To evaluate how this technique helps ApkDiffer, we created three new baselines by replacing the progressive alignment framework in ApkDiffer with the three baseline alignment algorithms (i.e., *GS*, *LA*, and *NS*). By comparing ApkDiffer with these new baselines, we can directly demonstrate the advantages/disadvantages of *Progressive Method Alignment*. Table 7 summarizes the comparison results on $Dataset_{\Delta 1}$ and $Dataset_{\Delta 5}$ for 1-to-1 method alignment. Based on the results, we can find that though *Progressive Method Alignment* costs slightly more time than other alignment algorithms, it significantly improves the precision and recall of method alignment.

6 App Evolution Study

In our evaluation, ApkDiffer shows appealing performance in aligning methods across different versions of the same Android app. Particularly, to further demonstrate the utility of our work, we leveraged ApkDiffer to perform *Longitudinal Privacy Testing* [55, 64] on real-world Android apps to uncover veiled privacy issues of app updates.

6.1 Study Design

Novelty of Our Study. We note that this paper is not the first to explore this problem. However, existing studies [55, 64] merely confirmed that wild apps are collecting more privacy-sensitive information from an outside view, e.g., by observing the cross-version changes of permission usage [64] or network traffic [55]. Without the capability of fine-grained app diffing, they cannot further pinpoint the detailed code locations where two app versions deviate in privacy collection, making it hard to diagnose identified privacy issues at the code-level, e.g., to analyze the developer intention, code context or privacy risk of those evolved privacy-collection behaviors. To the best of our knowledge, there are no existing works that utilize fine-grained code alignment techniques to systematically conduct such a study. One possible reason could be, existing alignment tools cannot reliably handle complex real-world apps.

To fill this gap, we choose to leverage ApkDiffer, a reliable code alignment tool, to assist the privacy assessment of app updates. Given an app update, ApkDiffer can enumerate all method-level code changes between pre-update version and post-update version. As such, analysts can thoroughly inspect whether each code change implements previously known privacy-collection behaviors. Through this study, we mainly want to emphasize the usability of ApkDiffer in auditing complex real-world apps. Here, we do not quantitatively compare ApkDiffer with baseline tools, mainly because one can hardly obtain the ground-truth method mappings between real-world app versions.

Research Questions. Generally, we considered three research questions to run this empirical study. In §6.2 (RQ1), §6.3 (RQ2) and §6.4 (RQ3), we introduce the methodologies to investigate these questions and provide answers to them.

- *RQ1. How do the privacy-collection behaviors generally evolve through app updates?*
- *RQ2. What are the causes of privacy-collection deviation across versions?*
- *RQ3. Do the evolving privacy-collection behaviors pose threats to end-users?*

Dataset. We constructed a dataset containing real-world Android apps for this study (i.e., $Dataset_{GP}$). First, we randomly collected 100 popular apps from the Google Play Store, each of which has been downloaded over 50 million times. These apps cover different app categories, e.g., Photography,

Table 8. Evolution Trends of Privacy-Collection (RQ1).

V_{pre}	V_{post}	pcb_{pre}		pcb_{post}		**Evolution Trends**		
		upcb	*dpcb*	*upcb*	*apcb*	**CR**	**AR**	**DR**
V_{Q1}	V_{Q2}	34,803	3,040	34,803	3,921	18.4%	10.4%	8.0%
V_{Q2}	V_{Q3}	35,145	3,547	35,145	4,489	20.8%	11.6%	9.2%
V_{Q3}	V_{Q4}	35,861	3,100	35,861	4,317	19.0%	11.1%	8.0%

[1] Changing Rate (CR) = (# of *apcb* + # of *dpcb*) / # of pcb_{pre}
[2] Addition Rate (AR) = # of *apcb* / # of pcb_{pre}
[3] Deletion Rate (DR) = # of *dpcb* / # of pcb_{pre}

Shopping, Video, etc. Second, for each app, we downloaded their quarterly snapshots released in a year from Androzoo [3], which are denoted as V_{Q1}, V_{Q2}, V_{Q3} and V_{Q4}, respectively. In this study, we practically track the 1-year evolution of collected apps in $Dataset_{GP}$, by diffing the quarterly versions of each app (i.e., $V_{Q1} \leftrightarrow V_{Q2}$, $V_{Q2} \leftrightarrow V_{Q3}$ and $V_{Q3} \leftrightarrow V_{Q4}$).

6.2 RQ1: Evolution Trends of Privacy-Collection

Methodology. We explored the evolution trends of privacy-collection behaviors mainly by inspecting what percentage of source API invocations (i.e., APIs that access privacy information) are unvaried/added/deleted during the version update. Specifically, given a pre-update version V_{pre} and a post-update version V_{post} (e.g., $V_{Q1} \leftrightarrow V_{Q2}$), we first identified all privacy-collection behaviors (i.e., invocations of source APIs) among these two versions, according to an API whitelist curated by SuSi [54]. After that, with the help of ApkDiffer, we correlated these privacy-collection behaviors across versions, and further classified them into 3 categories. For ease of expression, the privacy-collection behavior is denoted as *pcb* in the following of this section.

- *Unvaried Privacy-Collection Behavior (upcb)*: $pcb_x \in V_{pre}$ and $pcb_y \in V_{post}$ form a pair of *upcb*, only if 1) the caller methods of pcb_x and pcb_y can be aligned by ApkDiffer, and also 2) pcb_x and pcb_y invoke same source API.
- *Added Privacy-Collection Behavior (apcb)*: After identifying all *upcb* pairs, we view each remaining $pcb \in V_{post}$ as *apcb*.
- *Deleted Privacy-Collection Behavior (dpcb)*: Similarly, each remaining $pcb \in V_{pre}$ is viewed as *dpcb*.

Results. In total, ApkDiffer can successfully perform method alignment on 97/100 apps in $Dataset_{GP}$, the released APKs of which contain 90,905 methods in average. ApkDiffer failed on 3/100 apps due to implementation issues of Backstage [37] and Soot [66], and we plan to fix them in future. Table 8 presents the 1-year evolution trends of these 97 apps. In general, the quarterly version updates might introduce 11.0% (=(10.4%+11.6%+11.1%)/3) previously unknown privacy-collection behaviors and also delete 8.4% (=(8.0%+9.2%+8.0%)/3) at the same time. Notably, existing studies [55, 64] also reported a similar finding, that is wild apps are accessing more privacy information during version updating. But, as clarified in §6.3 (RQ2) and §6.4 (RQ3), by additionally knowing the detailed code locations of those evolved privacy-collection behaviors, we could carry out further in-depth diagnosis of identified privacy issues.

6.3 RQ2: Causes of Privacy-Collection Deviation

Methodology. Answers to RQ1 reveal that, the privacy-collection behaviors of studied apps are continuously evolving through app updates. In addition, we try to probe the causes of those

Table 9. Potential Causes of Privacy-Collection Deviation across Versions (RQ2).

V_{pre}	V_{post}	Factor 1: Functionality Updates				Factor 2: Library Integration			
		apcb		*dpcb*		*apcb*		*dpcb*	
		F-ADD	**F-EQ**	**F-DEL**	**F-EQ**	**LIB**	**NON-LIB**	**LIB**	**NON-LIB**
V_{Q1}	V_{Q2}	2,743 (**70.0%**)	1,178 (30.0%)	1,822 (**60.0%**)	1,218 (40.0%)	3,458 (**88.2%**)	463 (11.8%)	2,550 (**83.9%**)	490 (16.1%)
V_{Q2}	V_{Q3}	3,038 (**67.7%**)	1,451 (32.3%)	2,179 (**61.4%**)	1,368 (38.6%)	3,739 (**83.3%**)	750 (16.7%)	2,970 (**83.7%**)	577 (16.3%)
V_{Q3}	V_{Q4}	2,978 (**69.0%**)	1,339 (31.0%)	1,776 (**57.3%**)	1,324 (42.7%)	3,806 (**88.2%**)	511 (11.8%)	2,453 (**79.1%**)	647 (20.8%)

[1] **F-Add** refers to added functionalities, **F-EQ** refers to aligned functionalities, **F-DEL** refers to deleted functionalities
[2] **LIB** refers to library codebase, **NON-LIB** refers to app-developer codebase

deviated privacy-collection behaviors across versions, according to their code locations reported by ApkDiffer:

- *Whether the deviation is caused by app functionality update?* Given $apcb \in V_{post}$, we collected handler callbacks CB_{apcb} to represent its served app functionalities, each of which can reach *apcb* through control-flow edges. We think *apcb* should correlate with functionality addition, if there exist $cb \in CB_{apcb}$ that *cb* is a newly added method during update. Similarly, we can determine whether $dpcb \in V_{pre}$ is correlated with functionality deletion.
- *Whether the deviation stems from library codebase or app-developer codebase?* Since third-party libraries are frequently integrated by developers to ease development, we also want to check whether the deviation occurs in library codebase or app-developer codebase. More specifically, given $apcb \in V_{post}$ or $dpcb \in V_{pre}$, we try to verify whether its caller method is located in a library package. To achieve this goal, we followed existing methodology proposed in [14], which enhances LibRadar [45] (a detection tool of library packages) with package name heuristics.

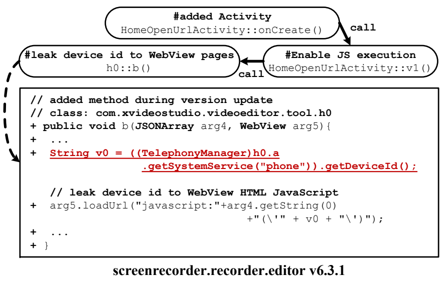
Results. Table 9 illustrates the number of identified *apcb* and *dpcb* during quarterly updates, and their relations with functionality updates and library integration. From the table, we can see that the introduction of *apcb* and *dpcb* is closely associated with the addition and deletion of app functionalities. Besides, it is surprising that, there exist near 30% *apcb* and *dpcb*, whose served functionalities can be aligned across versions. Looking into these cases, we find the reasons are two-fold: 1) Two app versions might probably implement different handling procedure for same runtime event (i.e., aligned callbacks), which causes deviation in accessed privacy data. 2) Some pairs of *apcb* and *dpcb* serve same functionalities on two app versions and meanwhile invoke same source API, which are probably caused by intra-functionality code refactoring. Besides, Table 9 also shows that a large percent (about 80%) of *apcb* and *dpcb* stem from third-party libraries.

Apart from the general statistics mentioned above, more importantly, we can obtain the functionality entry (i.e., callback method) and the offending party (i.e., app developer or libraries) of each suspicious privacy collection in a newer app version. This kind of information can largely ease the assessment of privacy risks.

6.4 RQ3: Privacy Threats of App Updates

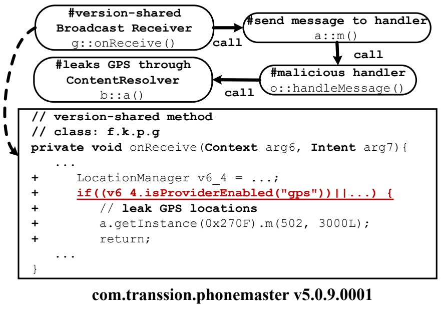
Methodology. Finally, we try to uncover the veiled privacy issues of app updates, by performing manual code-level case studies on identified *apcb*. Note that, a *apcb* is considered to bring privacy threats, if ❶ the accessed privacy information can be leaked through sink APIs (e.g., be sent through the network); ❷ the *apcb* can potentially be triggered by end-users during runtime (i.e., not dead code); ❸ the intended app functionality has no necessary dependency on collected privacy information (e.g., identity theft in a camera app). In the current study, we mainly want to emphasize the utility of ApkDiffer in assisting cross-version privacy testing, while not claiming contribution on how to confirm the privacy threats. In future, we can incorporate some advanced analysis tools to automate the assessment of privacy risks.

Results. During manual case studies, we examined four types of *apcb*, based on a two-factor (see RQ2, §6.3) classification, considering the correlation with functionality addition (i.e., the *apcb* serves added functionalities or version-shared ones) and library integration (i.e., the *apcb* locates in library codebase or app-developer codebase). Unfortunately, we confirmed that all these types of *apcb* may carry privacy threats. Due to space limitations, here we summarize a representative case for each type.


#added Activity
HomeOpenUrlActivity::onCreate()
call
#Enable JS execution
HomeOpenUrlActivity::v1()
call
#leak device id to WebView pages
h0::b()
screenrecorder.recorder.editor v6.3.1

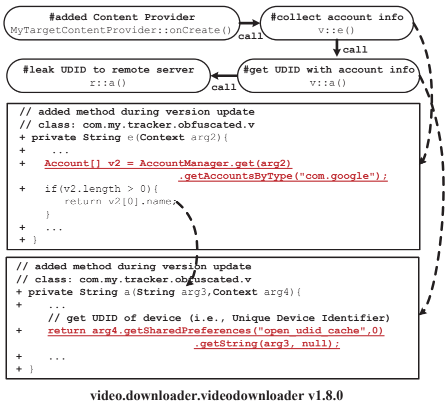
```
// added method during version update
// class: com.xvideostudio.videoeditor.tool.h0
+ public void b(JSONArray arg4, WebView arg5){
+   ...
+   String v0 = ((TelephonyManager)h0.a
                    .getSystemService("phone")).getDeviceId();

    // leak device id to WebView HTML JavaScript
+   arg5.loadUrl("javascript:"+arg4.getString(0)
                                  +"(\'" + v0 + "\')");
+   ...
+ }
```

(a) Case-I: V Recorder


#version-shared Broadcast Receiver
g::onReceive()
call
#send message to handler
a::m()
call
#malicious handler
o::handleMessage()
call
#leaks GPS through ContentResolver
b::a()
com.transsion.phonemaster v5.0.9.0001

```
// version-shared method
// class: f.k.p.g
private void onReceive(Context arg6, Intent arg7){
    ...
+      LocationManager v6_4 = ...;
+      if((v6_4.isProviderEnabled("gps"))||...) {
+         // leak GPS locations
+         a.getInstance(0x270F).m(502, 3000L);
+         return;
    ...
}
```

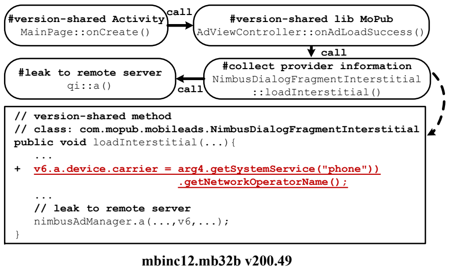
(b) Case-II: Phonemaster


#added Content Provider
MyTargetContentProvider::onCreate()
call
#collect account info
v::e()
call
#get UDID with account info
v::a()
call
#leak UDID to remote server
r::a()
video.downloader.videodownloader v1.8.0

```
// added method during version update
// class: com.my.tracker.obfuscated.v
+ private String e(Context arg2){
+    ...
+   Account[] v2 = AccountManager.get(arg2)
                      .getAccountsByType("com.google");
+   if(v2.length > 0){
       return v2[0].name;
    }
+   ...
+ }
```

```
// added method during version update
// class: com.my.tracker.obfuscated.v
+ private String a(String arg3,Context arg4){
+    ...
     // get UDID of device (i.e., Unique Device Identifier)
+    return arg4.getSharedPreferences("open_udid_cache",0)
                       .getString(arg3, null);
+    ...
+ }
```

(c) Case-III: Video Downloader


#version-shared Activity
MainPage::onCreate()
call
#version-shared lib MoPub
AdViewController::onAdLoadSuccess()
call
#collect provider information
NimbusDialogFragmentInterstitial ::loadInterstitial()
call
#leak to remote server
qi::a()
mbinc12.mb32b v200.49

```
// version-shared method
// class: com.mopub.mobileads.NimbusDialogFragmentInterstitial
public void loadInterstitial(...){
   ...
+  v6.a.device.carrier = arg4.getSystemService("phone"))
                         .getNetworkOperatorName();
   ...
   // leak to remote server
   nimbusAdManager.a(...,v6,...);
}
```

(d) Case-IV: MixerBox

Fig. 6. Cases for different types of privacy threats.

- *Case-I Added Functionality & App-developer Codebase:* **V Recorder** is a screen recoder app with over 100 million downloads. When updated from *v5.0.1* to *v6.3.1*, the app implements a new suspicious activity *HomeOpenUrlActivity*. The activity contains an obfuscated method $v1()$, which invokes $setJavaScriptEnabled(true)$ to enable the execution of JavaScript. After that, an obfuscated method $com.xvideostudio.videoeditor.tool.h0 :: b()$ invokes source API *getDeviceId()* and silently leaks the device id via JavaScript to WebView pages.
- *Case-II Version-shared Functionality & App-developer Codebase:* **Phonemaster** is a phone manager with over 500 million downloads, which helps clean junk files and manage phone states. When

updated from *v5.0.5.0002* to *v5.0.9.0001*, the app additionally invokes the source API *Location-Manager.isProviderEnabled()* in a version-shared *onReceive()* method to check the status of the GPS provider. If enabled, it would send a message (message id is 502) to a malicious handler method *b.b.a.b.o::handleMessage()*. Finally, the handler stealthily collects the GPS information and finally leaks it through *ContentResolver* API (i.e., $ContentResolver.insert()$) in an obfuscated method *b.b.a.b.b::a()*.
- *Case-III Added Functionality & Library Codebase:* **Video Downloader** is a popular app for downloading videos and music from the Internet, with over 100 million downloads. When updated from *v1.7.3* to *v1.8.0*, the app integrates a new advertisement library *MyTarget*, which registers a new content provider in *AndroidManifest.xml*. The *onCreate()* method of the provider invokes an obfuscated library method *com.my.tracker.obfuscated.v::e()* which accesses the account information. The account information is further utilized to fetch the *UDID* (i.e., unique device identifier), which is leaked to remote server.
- *Case-IV Version-shared Functionality & Library Codebase:* **MixerBox** is a music player with over 100 million downloads. Different versions of this app integrates the same advertising library *com.mopub.mobileads*. When updated from *v200.46* to *v200.49*, a version-shared activity *mbinc12.mb32.MainPage* still has dependency on the advertising library *MoPub*. However, certain code updates are applied in the library codebase. When successfully loading advertisements, the library method *NimbusDialogFragmentInterstitial::loadInterstitial()* additionally invokes the source API *getNetworkOperatorName()* and leak the provider information through network traffic in the method *qi::a()*.

6.5 Threats to Validity

Firstly, the collected apps in $Dataset_{GP}$ are all popular apps in Google Play Store. Hence, the longitudinal privacy changes of these apps might not be representative of the entire Android ecosystem. In future, we plan to conduct such study based on apps from different app stores. Secondly, ApkDiffer inevitably produce alignment errors when diffing app versions. However, we believe these alignment errors would not largely affect the reliability and rationality of study conclusions, since our extensive experiments have demonstrated the high precision and recall of ApkDiffer.

7 Discussion

Limitations. Though the experiment results show that ApkDiffer noticeably outperforms existing method alignment algorithms, it still has several limitations: ❶ ApkDiffer relies on a manually curated whitelist of callback interfaces (see Table 1) to detect app functionality units. Although our whitelist is more complete than that of FlowDroid [11], a possible optimization solution is to leverage static framework modeling [12] to obtain a more precise whitelist. ❷ ApkDiffer mainly aims to correlate methods in Java/Kotlin codebase, and methods in native libraries are considered out of scope. A recent work proposes *JuCify* [57] to merge the call graphs of app bytecode and native code into a whole-app call graph, which can be integrated to tackle this issue.

Resilience to App Obfuscation. When constructing the benchmark dataset for evaluation, we use the default ProGuard [9] obfuscation configuration to build the apps (see §5.2). Thus, we believe that ApkDiffer can handle a certain degree of obfuscation. For call graph-level obfuscation (e.g., inserting dummy methods), the n-n alignment mechanism of ApkDiffer may help align such methods. For control flow graph-level obfuscation (e.g., CFG flattening), semantic-aware code similarity measures may improve the effectiveness of ApkDiffer in aligning such methods.

8 Related Work

Code Similarity Calculation. The method alignment, formulated as call graph matching problem, should comprehensively consider both node (i.e., method) similarities and edge (i.e., caller-callee relation) similarities to solve precise alignment results. In practice, such node similarities are usually calculated via code similarity calculation techniques [23, 29, 31, 42, 59, 67]. Notably, our contribution is the method alignment methodology for whole-app diffing, independent of existing similarity measures. We believe ApkDiffer can further improve the cross-version code diffing capability from this understudied perspective. It is also an appealing idea to integrate advanced similarity measures into ApkDiffer, and we will leave it for future work.

Method Extract/Inline Identification. Existing techniques in method extract/inline identification either work on source code [35] or commit histories [65], or need heavy manual workloads [30]. Other works [15, 23] in method-level code similarity calculation rely on hard-coded heuristics to reduce the impact of method extract/inline for more precise similarity detection (e.g., Asm2vec [23] assumes callees with less than 10 instructions as extracted methods). Besides, although basic-block-level code alignment [52, 53] can potentially detect method extract/inline, they face even severe scalability issues when aligning ICFGs of Android apps.

App Diffing. Existing app diffing methodologies [20, 39] exhibit two key limitations: 1) excessive reliance on overly coarse comparison granularity (e.g., class-level analysis [20]), and 2) adoption of simplistic code-structure-based matching strategies (e.g., package-method association [39]). These approaches demonstrate insufficient robustness when confronted with cross-version obfuscation variations, structural refactoring operations, and implementation updates. As comparison, ApkDiffer achieves reliable method alignment based on novel functionality-driven codebase decomposition.

App Clone Detection. Android app repackaging has been raised as a serious problem by various researchers, calling for reliable app clone detection techniques to catch repackaged apps. This line of techniques generally utilizes different kinds of app features (e.g., UI features [61, 69], code features [16, 25, 41] and so on) to determine how similar are two apps. Different from these works, ApkDiffer can further give fine-grained method mappings, indicating the detailed code changes.

9 Conclusion

In this work, we propose ApkDiffer, a cross-version method alignment tool for Android apps. Compared to existing method alignment techniques, ApkDiffer achieves a good balance between scalability and effectiveness via a decomposition-based approach and additionally supports the method extract/inline identification across versions. In evaluation, we performed extensive experiments to demonstrate the effectiveness and efficiency of ApkDiffer. We further conducted a real-world study to showcase the utility of ApkDiffer in assisting privacy assessment of real-world app updates, convincingly revealing the privacy threats of app updates.

Data-Availability Statement

Our artifact is available at https://doi.org/10.5281/zenodo.16736418 [1], which includes the prototype of ApkDiffer and the benchmark dataset.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (U2436207, 62172105, 62402116). Yuan Zhang and Min Yang are the corresponding authors. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - FuDan

University 21TQ1400100 (21TQ012). Min Yang is a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

References

[1] [n. d.]. *ApkDiffer-OOPSLA25-artifact.* doi:10.5281/zenodo.16736418
[2] 2023. AndroGuard. https://github.com/androguard/androguard.
[3] 2023. AndroZoo. https://androzoo.uni.lu/.
[4] 2023. Backdoor Insertion during Update. https://arstechnica.com/information-technology/2020/04/sophisticated-android-backdoors-have-been-populating-google-play-for-years/.
[5] 2023. CamScanner. https://www.tomsguide.com/news/infected-android-app-with-100m-downloads-found-in-google-play.
[6] 2023. Diaphora. https://github.com/joxeankoret/diaphora.
[7] 2023. F-droid: Free and open source android app repository. https://f-droid.org/.
[8] 2023. Modular Programming. https://en.wikipedia.org/wiki/Modular_programming.
[9] 2023. ProGuard. https://www.guardsquare.com/proguard.
[10] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, et al. 2019. On the Impact of Refactoring on the Relationship between Quality Attributes and Design Metrics. In *ESEM*.
[11] Steven Arzt, Siegfried Rasthofer, et al. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI*.
[12] Michael Backes, Sven Bugiel, et al. 2016. On Demystifying the Android Application Framework:{Re-Visiting} Android Permission Specification Analysis. In *USENIX Security*.
[13] Martial Bourquin, Andy King, et al. 2013. BinSlayer: Accurate Comparison of Binary Executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*.
[14] Paolo Calciati, Konstantin Kuznetsov, et al. 2018. What did Really Change with the new Release of the App?. In *MSR*.
[15] Mahinthan Chandramohan, Yinxing Xue, et al. 2016. BinGo: Cross-Architecture Cross-OS Binary Search. In *FSE*.
[16] Kai Chen, Peng Liu, et al. 2014. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *ICSE*.
[17] Zhiyuan Chen, Young-Woo Kwon, et al. 2018. Clone Refactoring Inspection by Summarizing Clone Refactorings and Detecting Inconsistent Changes during Software Evolution. *Journal of Software: Evolution and Process (JSEP)* (2018).
[18] Minsu Cho and Kyoung Mu Lee. 2012. Progressive Graph Matching: Making a Move of Graphs via Probabilistic Voting. In *CVPR*.
[19] Rudi Cilibrasi. 2006. *Statistical Inference through Data Compression.* University of Amsterdam.
[20] Robbe De Ghein, Bert Abrath, Bjorn De Sutter, and Bart Coppens. 2022. Apkdiff: Matching android app versions based on class structure. In *Proceedings of the 2022 ACM Workshop on Research on offensive and defensive techniques in the context of Man At The End (MATE) attacks.*
[21] Jean-Charles Delvenne, Michael T Schaub, et al. 2013. The Stability of a Graph Partition: a Dynamics-based Framework for Community Detection. In *Dynamics On and Of Complex Networks, Volume 2.* Springer.
[22] Anthony Desnos. 2012. Android: Static analysis using similarity distance. In *Proceedings of the 45th Hawaii International Conference on System Sciences.*
[23] Steven HH Ding, Benjamin CM Fung, et al. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *S&P*.
[24] Thomas Dullien and Rolf Rolles. 2005. Graph-based comparison of Executable Objects. *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)* (2005).
[25] Ming Fan, Jun Liu, et al. 2017. Dapasa: Detecting Android Piggybacked Apps through Sensitive Subgraph Analysis. *TIFS* (2017).
[26] Raymond Fok, Mingyuan Zhong, et al. 2022. A Large-Scale Longitudinal Analysis of Missing Label Accessibility Failures in Android Apps. In *Proceedings of the 40th Annual ACM Conference on Human Factors in Computing Systems (CHI).*
[27] Debin Gao, Michael K Reiter, et al. 2008. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *ICICS*.
[28] Jun Gao, Li Li, et al. 2019. Understanding the Evolution of Android App Vulnerabilities. *IEEE Transactions on Reliability* (2019).
[29] Jian Gao, Xin Yang, et al. 2018. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary. In *ASE*.

[30] Michael W Godfrey and Lijie Zou. 2005. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *TSE* (2005).

[31] Yikun Hu, Yuanyuan Zhang, et al. 2016. Cross-Architecture Binary Semantics Understanding via Similar Code Comparison. In *SANER*.

[32] Ang Jia, Ming Fan, et al. 2021. One-to-One or One-to-Many? What Function Inlining Brings to Binary2source Similarity Analysis. *arXiv* (2021).

[33] István Kádár, Péter Hegedűs, et al. 2016. A Manually Validated Code Refactoring Dataset and Its Assessment Regarding Software Maintainability. In *PROMISE*.

[34] Chariton Karamitas and Athanasios Kehagias. 2018. Efficient Features for Function Matching between Binary Executables. In *SANER*.

[35] Miryung Kim, Matthew Gee, et al. 2010. Ref-Finder: A Refactoring Reconstruction Tool based on Logic Query Templates. In *FSE*.

[36] Harold W Kuhn. 1955. The Hungarian Method for The Assignment Problem. *Naval research logistics quarterly (NRL)* (1955).

[37] Konstantin Kuznetsov, Vitalii Avdiienko, et al. 2018. Analyzing the User Interface of Android Apps. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MOBILESof)*.

[38] Sehyung Lee, Jongwoo Lim, et al. 2020. Progressive Feature Matching: Incremental Graph Construction and Optimization. *IEEE Transactions on Image Processing (TIP)* (2020).

[39] Hehao Li, Yizhuo Wang, et al. 2022. PEDroid: Automatically Extracting Patches from Android App Updates. In *ECOOP*.

[40] Yu Lin, Semih Okur, and Danny Dig. 2015. Study and Refactoring of Android Asynchronous Programming (T). In *ASE*.

[41] Martina Lindorfer, Stamatis Volanis, et al. 2014. AndRadar: Fast Discovery of Android Applications in Alternative Markets. In *Proceedings of the 11th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.

[42] Bingchang Liu, Wei Huo, et al. 2018. αDiff: Cross-Version Binary Code Similarity Detection with DNN. In *ASE*.

[43] Lorenzo Livi and Antonello Rizzi. 2013. The Graph Matching Problem. *Pattern Analysis and Applications* (2013).

[44] Zhenhao Luo, Pengfei Wang, et al. 2023. VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search. In *NDSS*.

[45] Ziang Ma, Haoyu Wang, et al. 2016. Libradar: Fast and Accurate Detection of Third-party Libraries in Android Apps. In *ICSE*.

[46] Aravind Machiry, Rohan Tahiliani, et al. 2013. Dynodroid: An Input Generation System for Android Apps . In *FSE*.

[47] Noah Mauthe, Ulf Kargén, et al. 2021. A Large-Scale Empirical Study of Android App Decompilation. In *SANER*.

[48] Stuart McIlroy, Nasir Ali, et al. 2016. Fresh Apps: An Empirical Study of Frequently-Updated Mobile Apps in The Google Play Store. *ESE* (2016).

[49] Emerson R. Murphy-Hill, Chris Parnin, et al. 2012. How We Refactor, and How We Know It. *TSE* (2012).

[50] Edmilson Campos Neto, Daniel Alencar Da Costa, et al. 2018. The Impact of Refactoring Changes on the SZZ Algorithm: An Empirical Study. In *SANER*.

[51] Matheus Paixão, Anderson Uchôa, et al. 2020. Behind the Intents: An In-depth Empirical Study on Software Refactoring in Modern Code Review. In *MSR*.

[52] Jannik Pewny, Behrad Garmany, et al. 2015. Cross-architecture Bug Search in Binary Executables. In *S&P*.

[53] Jannik Pewny, Felix Schuster, et al. 2014. Leveraging Semantic Signatures for Bug Search in Binary Programs. In *ACSAC*.

[54] Siegfried Rasthofer, Steven Arzt, et al. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *NDSS*.

[55] Jingjing Ren, Martina Lindorfer, et al. 2018. Bug Fixes, Improvements, ... and Privacy Leaks - A Longitudinal Study of PII Leaks Across Android App Versions. In *NDSS*.

[56] Kaspar Riesen, Xiaoyi Jiang, et al. 2010. Exact and Inexact Graph Matching: Methodology and Applications. *Managing and Mining Graph Data* (2010).

[57] Jordan Samhi, Jun Gao, et al. 2022. JuCify: A Step Towards Android Code Unification for Enhanced Static Analysis. In *ICSE*.

[58] Yuru Shao, Xiapu Luo, et al. 2014. Towards a Scalable Resource-driven Approach for Detecting Repackaged Android Applications. In *ACSAC*.

[59] Paria Shirani, Leo Collard, et al. 2018. BinArm: Scalable and E cient Detection of Vulnerabilities in Firmware Images of Intelligent Electronic Devices. In *Proceedings of the 15th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.

[60] Paria Shirani, Lingyu Wang, et al. 2017. BinShape: Scalable and Robust Binary Library Function Identification Using Function Shape. In *Proceedings of the 15th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.

[61] Charlie Soh, Hee Beng Kuan Tan, et al. 2015. Detecting Clones in Android Applications through Analyzing User Interfaces. In *ICPC*.

[62] Yukun Su, Guosheng Lin, et al. 2022. Self-Supervised Object Localization with Joint Graph Partition. In *AAAI*.

[63] Pang-Ning Tan et al. 2006. *Introduction to Data Mining*.

[64] Vincent F Taylor, Ivan Martinovic, et al. 2017. To Update or Not to Update: Insights from a Two-Year Study of Android App Evolution. In *AsiaCCS*.

[65] Nikolaos Tsantalis, Matin Mansouri, et al. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *ICSE*.

[66] Raja Vallée-Rai, Phong Co, et al. 2010. Soot - a Java Bytecode Optimization Framework. In *CASCON First Decade High Impact Papers (CASCON)*.

[67] Xiaojun Xu, Chang Liu, et al. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *CCS*.

[68] Junchi Yan, Xu-Cheng Yin, et al. 2016. A Short Survey of Recent Advances in Graph Matching. In *ICMR*.

[69] Fangfang Zhang, Heqing Huang, et al. 2014. ViewDroid: Towards Obfuscation-Resilient Mobile Application Repackaging Detection. In *Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*.

[70] Xiaoyi Zhang, Anne Spencer Ross, et al. 2018. Robust Annotation of Mobile Application Interfaces in Methods for Accessibility Repair and Enhancement. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST)*.

[71] Yuan Zhang, Jiarun Dai, et al. 2018. Detecting Third-Party Libraries in Android Applications with High Precision and Recall. In *SANER*.

[72] Zhengyou Zhang, Rachid Deriche, et al. 1995. A Robust Technique for Matching Two Uncalibrated Images Through The Recovery of The Unknown Epipolar Geometry. *Artificial Intelligence* (1995).

Received 2025-03-24; accepted 2025-08-12