EvoTaint: Incremental Static Taint Analysis of Evolving Android Apps

JIAWEI GUO, University at Buffalo, SUNY, USA
HAIPENG CAI*, University at Buffalo, SUNY, USA

In the last decade, Android applications have emerged as a primary interface in consumer technology. With approximately 2.5 billion mobile devices running Android globally, security threats to the Android ecosystem due to vulnerabilities in it become increasingly broadly consequential via user applications (i.e., Android apps). This necessitates efficient methods for defending them against those vulnerabilities. Taint analysis, a popular and fundamental security defense technique, assesses the flow of sensitive information within an app between sources (e.g., reading from user inputs) and sinks (e.g., writing to databases). However, traditional taint analysis is notably resource-intensive. Performing a comprehensive analysis on a single app given a complete list of potential sources and sinks can take hours, a situation exacerbated by the frequent updates typical in mobile app development.

In this paper, we propose EvoTaint, an *incremental taint analysis*, tailored to fit and exploit the evolving nature of Android apps. It aims to substantially reduce the time cost of conventional static taint analysis against an evolved version of a given app by narrowing down the analysis scope from the entire app to only the parts that are changed or impacted by the changes in the evolved version. We have implemented EvoTaint as a practical, open-source tool and evaluated it on 100 Android apps each with 2, 3, or even 5 versions considered. Our results demonstrated a significant (51.8—68.9%) reduction in the time cost of static taint analysis of each of the 1—4 evolved versions on average, without compromising the accuracy of the analysis results (i.e., taint flow paths), compared to using the conventional approach treating each version as a separate/standalone app. Our further analysis aimed to clarify why and when EvoTaint performs favorably. It revealed that the time efficiency gains of incremental taint analysis are strongly correlated with the ratio of changed methods and the proportion of sources/sinks affected by these changes during app evolution.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering → Software evolution**.

Additional Key Words and Phrases: Android apps, incremental analysis, static analysis, taint analysis, software evolution, impact analysis, information flow security, cost reduction, optimization, efficiency improvement

ACM Reference Format:
Jiawei Guo and Haipeng Cai. 2025. EvoTaint: Incremental Static Taint Analysis of Evolving Android Apps. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (June 2025), 44 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Mobile computing, especially that primarily through the Android Operating System, plays a pivotal role in contemporary society. Dominating the smartphone market with over 3.3 billion users worldwide, Android's widespread adoption has also rendered it a prime subject to security threats [18, 30, 58**?**]. In particular, studies focusing on the evolution of Android apps have revealed a concerning trend: with each version update, apps tend to exhibit declining security

*Haipeng Cai is the corresponding author.

Authors' Contact Information: Jiawei Guo, University at Buffalo, SUNY, New York, Buffalo, USA, jiaweigu@buffalo.edu; Haipeng Cai, University at Buffalo, SUNY, New York, Buffalo, USA, haipengc@buffalo.edu.

Manuscript submitted to ACM

with an increase in vulnerabilities in the evolved apps [35, 43, 62]. This degradation in app security during its constant evolution underscores the urgency for continuous security vetting of evolving Android apps.

Problem/Motivation. One widely used method for enhancing application security on Android is (static) taint analysis [30]. This technique evaluates an app for potential exposure or leakage of security-sensitive or private user data to unauthorized or malicious entities. Underlying this technique is often a data-flow analysis computing the reachability between program points where information is retrieved, noted as taint sources (e.g., reading from user inputs), and program points where information leaves the program, noted as taint sinks (e.g., writing to a database). For taint analysis to be efficient in terms of time, it must be precise, generating minimal false positives [52]. Additionally, to minimize the risk of overlooking *real* threats, the analysis needs to be comprehensive, capturing as many actual security threats as possible (i.e., being *safe*) [36]. An *accurate* taint analysis would achieve both high precision and reliability of its results.

However, achieving accurate taint analysis is also notoriously resource-intensive [6, 36], as it requires complex modeling and reasoning about code semantics and app behaviors to identify and eliminate false positives while ensuring safe results. For instance, FlowDroid [6], a well-known taint analysis tool for Android, can take several hours to analyze a single app at its highest precision setting. While adjusting the settings for greater efficiency can speed up the process, it often leads to a significant reduction in precision. Many current solutions face similar efficiency and practicality issues-—they are too resource-intensive to handle large, real-world apps effectively. On the other hand, some approaches prioritize speed over accuracy, resulting in an excessive number of false positives. These persistent false alerts can hinder the adoption of taint analysis techniques, as security analysts may find the time and effort required to sift through the inaccurate results prohibitively expensive.

Existing Solutions. Recent advances in static taint analysis have attended to such challenges through dedicated performance optimizations of the core analysis algorithm. For instance, integrating heap snapshots with FlowDroid aims to optimize the trade-off between precision and performance [11]; optimizing the underlying data flow analysis (IFDS) algorithm helps reduce time and memory requirements for taint analysis [42]. In addition, improving the analysis efficiency has been attempted for through a refinement-based strategy (starting with an over-approximation of taint value propagation, followed by feasibility analysis of taint flows) [67]. Context-sensitive inside-out taint analysis [49] has also been proposed to scale taint analysis to large Java codebases. Adopting just-in-time analysis methodology in static taint analysis helps reduce the time cost of the analysis as well [27]. Despite the progress made by these existing efforts, the challenge persists: conducting highly precise taint analysis remains a taxing endeavor, both in terms of computational resources and the manual effort required to sift through false positives. More importantly, current taint analysis methods [8] [36] [63] handle each version of an app as if it were a standalone program, ignoring the fact that the changes that occur over time are *incremental*. This current approach results in unnecessarily high costs when analyzing multiple versions of an app throughout its development cycle.

Key Insights. Despite the generally costly nature of precise static Android-app taint analysis, we found that the analysis cost is much lower when only a much smaller set of sources and sinks needs to be considered by the analysis. This is intuitive because the analysis is aimed to search any information flow paths between any of the given sources and any of the given sinks. Thus, as expected, the cost is generally positively correlated with the number of sources and sinks considered. When an app evolves to its next version, the changes made may not affect all of those sources and sinks and the reachability between them. The information flow paths between *unaffected* sources and sinks, which are available already if the taint analysis has been performed on the original app, should remain the same for the evolved app. And any additional taint flows induced by the changes are those between the sources and sinks impacted by the

changes. Hence, given the results of a taint analysis on the original app, the taint analysis on the evolved app should be performed incrementally—-only concerning the taint flows between the impacted sources and sinks. If this results in a considerable reduction in the total number of sources and sinks that need to be considered, we may expect that this incremental analysis, though straightforward in terms of the core ideas, may reduce the analysis cost considerably.

Our Approach. Following the above insights, we introduce EvoTaint, a new, *incremental taint analysis* for Android apps. Our approach aims at a significant reduction of the time cost of accurate (yet costly) taint analysis, so as to provide a solution that is both efficient and effective for practical adoption. To that end, our analysis focuses on the changes between different versions of an evolving app and on the parts of the newer version that are affected by the changes, instead of analyzing each version as an independent app. In this way, the analysis cost will be amortized across multiple versions of an app.

Our approach begins with a standard/conventional taint analysis (e.g., FlowDroid) on the initial version of an app, which serves as the basis for subsequent analyses. This traditional taint analysis is performed only once. The incremental taint analysis, used for later versions, involves three main steps.

- The first step, *impact analysis*, identifies which methods are affected by changes in the new app version. It detects modified methods between the two app versions and uses the call graph (of either the base or evolved version) to determine all change-impacted methods.
- In the second step, *impact-guided taint checking*, the technique uses the same call graphs to identify impacted sources and sinks starting from change-impacted methods and thus get impacted flow path results. By tracking backward from impacted methods, potential sources are identified if their predecessors have predefined source invocations. Similarly, tracking forward identifies impacted sinks if successors contain sink invocations. With impacted sources and sinks identified, we proceed to compute the impacted flow paths, naming taint checking.
- Finally, the third step, *taint synthesis*, merges the impacted results—obtained from running the conventional taint analyzer on identified impacted sources and sinks—with the analysis results for the previous version of the app.

Results. To evaluate our approach, we implemented the proposed *incremental taint analysis* algorithm in an open-source tool, EvoTaint, and applied it to 255 real-world benchmarks, representing 100 unique apps with multiple versions each. Specifically, we formed three benchmark groups, with each app in the corresponding group has 2, 3, and 5 versions in total. Our results indicate a substantial improvement achieved by EvoTaint in cost-effectiveness compared to the baseline (i.e., conventional taint analysis with FlowDroid). In the group where each app has two versions, EvoTaint achieves a 51.8% reduction on average in analysis cost, seeing cases in which the reduction reached up to 99%. Notably, the cost reduction is more pronounced with a greater number of evolving app versions. In the groups where each app has three and five versions, EvoTaint can save 66.9% and 68.9% of analysis time, respectively. Moreover, EvoTaint can reduce the space usage of the baseline analysis for most (65%) of the cases (apps) in terms of peak memory consumption. Importantly, EvoTaint achieved all of these efficiency gains without compromising accuracy—it constantly produces the taint flow paths equivalent to those from the conventional taint analyzer.

Through in-depth analysis, we further discovered that the time efficiency gains achieved by EvoTaint are strongly (negatively) correlated with the ratio of an app's methods that are changed between the two versions of the app. Moreover, they are also strongly (positively) correlated with the percentage of sources/sinks that can be saved by only considering those impacted by the changes, with the latter correlation essentially explaining the former. As a potential guideline, EvoTaint tends to be effective when the overall change ratio is below 50%. Among the three types of changes(addition, deletion, and modification), the impact of modifications on EvoTaint's time savings is stronger

than additions and deletions, although none of the individual change types have as strong impact as the overall app change ratio. Between the percentage of sources and that of sinks saved due to the incremental approach, the latter tends to have stronger impact on time savings. However, the source code size of either the base or the evolved version of the app has essentially no correlation with the time efficiency of EvoTaint.

Contributions. Through the development and evaluation of EvoTaint, this paper makes the following contributions:

- We proposed *incremental taint analysis* with EvoTaint, a new incremental approach to static taint analysis, for evolving Android apps, which leverages the incremental nature of changes during app evolution to improve the efficiency hence cost-effectiveness of checking the evolved apps against information flow security vulnerabilities.
- We developed EvoTaint and implemented it as an open-source tool that works practically on real-world evolving Android apps, and demonstrated substantial efficiency merits of the proposed incremental taint analysis via the EvoTaint tool over conventional taint analysis through empirical evaluation on those apps as benchmarks.
- We conducted extensive further studies to reveal and understand the merit conditions of incremental taint analysis, i.e, under what conditions does incremental taint analysis demonstrate its time efficiency advantages, by examining the impact of various relevant factors to the analysis algorithm, including the app change ratio, portions of impacted sources/sinks, and source-code sizes of both base and evolved apps.

The development of EvoTaint started with its preliminary, proof-of-concept version [17]. This paper represents a completion and substantial expansion of that earlier work in terms of both technical contributions and empirical evaluations. Technically, (1) instead of building and holding in memory call graphs for both app versions, which can blow the memory for large-scale apps, we now only build the call graph for the evolved version, largely improving the scalability and efficiency; (2) instead of using method-level dependence abstraction [23] to compute the change impact set, which again hits scalability barriers with large-scale apps as this abstraction itself is expensive also, we now only compute conservative impact set based on the call graph. Another design optimization is to move the incremental taint flow computation for deleted methods entirely to taint synthesis (by simply removing taint flow paths in the original version that pass through any of the deleted methods). Implementation wise, we have updated/upgraded the dependent/supporting libraries/tools of EvoTaint to enable it to work practically and more robustly against apps released in recent years which are built on latest Android platforms. None of these design/implementation optimizations compromise the ultimate/overall accuracy of the analysis, though.

Experimentally, (1) we substantially expanded the scale of evaluation experiments, from 19 unique apps to 100, covering apps of an even greater variety of sizes and application domains, (2) we largely enhanced the diversity of app evolution scenarios of the evaluation benchmarks from considering only two versions of each app to three groups with different numbers (2, 3, and 5) of versions considered for each app, (3) we added the evaluation of space efficiency of incremental taint analysis in terms of peak memory consumption, compared to the conventional analysis, (4) we conducted extensive statistical (i.e., correlation) analyses to examine how the overall ratio of changes and the ratio of different types of changes between the app versions impact the time efficiency of EvoTaint, hence identifying the conditions under which the proposed incremental taint analysis may be worthwhile, (5) we conducted statistical analyses to examine how the percentage of sources/sinks that can be reduced by analyzing taint flows incrementally impacts the time efficiency of EvoTaint, hence further understanding/explaining such impact of the change ratios, and (6) we assessed whether the time efficiency of EvoTaint is affected by the size of analyzed app versions.

In addition, we have provided a new background section on taint analysis, and more detailed accounts about the motivation of EvoTaint (including motivating examples) and its technical design. We also gave more detailed

comparisons to related works and offered new insights into various aspects of using EvoTaint in practical scenarios (via a dedicated discussion section). Finally, in accordance with all of the improved technical design, additional details, and new results, we have also updated the entire paper accordingly in terms of the overall structure and content.

Significance. Given the large number of evolving versions of Android apps on app stores like Google Play, our technique offers a viable solution for rapid and precise security screening of those apps. Furthermore, it enables developers to efficiently assess the security implications of minor modifications, such as altering data flow paths or incorporating third-party libraries, during routine development activities. The incremental analysis approach demonstrated via EvoTaint is particularly relevant given the frequent version updates in Android apps. A study of the Androzoo database [3] indicates that, among 75,963 Android app families, the median number of versions per app is three, with some families featuring apps each with over one hundred versions [3]. Traditional taint analysis overlooks these incremental changes, leading to unnecessary recomputation and excessively high analysis costs.

Although for the ease of presentation, the rest of the paper assumes the most typical scenario of incremental taint analysis (in which the two apps taken by the analysis as inputs are two versions of the same app during its evolution), our approach is not necessarily limited to that scenario. Without loss of generality, EvoTaint can work with any two given apps, regardless of their being the same app—although the technique would only be cost-beneficial when these two given apps are similar enough so that the incremental analysis is worthwhile (i.e., more cost-effective than separately analyzing each app independently). Even more generally, the quite straightforward incremental analysis methodology we demonstrate with EvoTaint should also be applicable to other software application domains beyond Android apps, although a different tool implementation would be necessary.

Artifact. EvoTaint has been made available as an open-source project at

https://bitbucket.org/wsucailab/iterative-taint-analysis/src/v1.0-Evotaint/

2 Background and Motivation

In this section, we first provide basic background on taint analysis as a general technique in software security defense (§2.1). Then, we discuss how conventional taint analysis misses the opportunity of improving efficiency for evolving software (§2.2) through a motivating example (§2.3) in Android taint analysis, so as to motivate our technique presented in this paper.

2.1 Taint Analysis in Software Security

The behaviors of a program can be largely captured by how information flows in the program. Thus, intuitively information-flow security of a program plays a central role in the holistic security of the program. Taint analysis has long since been a fundamental technique for addressing information-flow security of various software systems and applications [33, 34, 51, 60], including Android apps [8, 36, 63, 68]. It is particularly useful for finding security vulnerabilities related to data input validation, such as information/privacy leaks, SQL injection, cross-site scripting (XSS), or other forms of injection attacks.

Taint analysis helps users track and understand the flow of sensitive/tainted data within a program, where "taint" refers to data (e.g., user inputs) that may be influenced by or derived from untrusted or insecure sources. It works by tracing the propagation of tainted data through a program's execution, hence identifying potential security vulnerabilities, especially information-flow vulnerabilities that can be solved as a source-sink problem (i.e., *taint-style vulnerabilities*).

As its core technical step, taint analysis tracks how the tainted data flows through various program entities (e.g., variables, expressions, and functions). How this data-flow tracking is realized differentiates *static* versus *dynamic* taint analysis. In *static taint analysis* [32], the flow is tracked statically (by analyzing the source code without executing it); thus, the analysis analyzes the source code to predict *all* potential flows of tainted data. In contrast, *dynamic taint analysis* [66] performs the flow tracking dynamically (during program execution), observing the actual runtime behavior of the program that is exercised by the run-time inputs considered.

2.2 The Need for Incremental Taint Analysis

Given the rising prevalence of data breaches and the consequences of security lapses, ranging from personal user impacts to substantial financial losses for companies, ensuring the secure handling of sensitive user data in apps via taint analysis is a crucial means against those security threats. In particular, when concerned with a more comprehensive view of such threats rather than only the vulnerabilities exercised in specific program executions, developers would intuitively find *static* taint analysis a more desirable option [28].

However, traditional approaches to static taint analysis can be resource-intensive hence costly. In a study comparing state-of-the-art static taint analyzers for Android apps, it was found that none of the three most representative tools can finish analyzing more than 18 out of 30 chosen relatively-large real-world apps in 30 minutes [55]. In our prior experience with one of the tools, FlowDroid [8], analyzing one app could take up to an hour or longer, while other peer representative tools, DroidSafe [36] and Amandroid [63], can be generally even less efficient for the same or even lower levels of accuracy [68]. When pursuing higher accuracy with this analysis, the cost tends to be even higher [8, 70]. Despite efforts on speeding up the analysis, including through optimized data-flow analysis [42] or a different route such as type analysis [44], static taint analysis remains a generally costly technique. Not only does the cost concern execution time of the analysis, the memory consumption is also quite substantial [41, 44].

This efficiency problem is exacerbated when frequent updates (e.g., during constant software evolution) are necessary—users would need to repeatedly suffer from the significant cost incurred by the taint analysis used for security vetting each updated version of the software. In particular, in the context of Continuous Integration/Continuous Deployment (CI/CD) practices, where small and frequent updates are common, the need for an efficient taint analysis method tailored to these incremental changes becomes increasingly evident. Moreover, during a widely adopted practice of CI/CD, the need to quickly vet each intermediate version to make sure the committed changes do not introduce taint-style security vulnerabilities is further stronger. Compared to software evolution through various official releases, the differences between *committed* versions are even smaller, for which the traditional (standalone/independent) taint analysis tends to be even more wasteful.

2.3 Motivating Example

To illustrate, let us consider the use scenario of static taint analysis for Android apps. The developer routinely performs taint analysis after each app update. The analysis tool chosen is the classical FlowDroid [8], a choice well-justified when considering both the performance and usability of state-of-the-art tools for Android taint analysis—Amandroid is notably less precise while DroidSafe does not provide information flow paths (but only the source and sink methods of identified taint flows) [69]. As normal, the developer runs FlowDroid in a standalone manner—i.e., treating each version of each app as an independent subject of the analysis. Apparently, this approach is unnecessarily expensive and inefficient, particularly for apps undergoing frequent evolution. Yet, at present, this is the common approach.

For example, let us assume one of the Android apps involved is *LMS* [29], an open-source app from F-Droid [1] that is relatively small in size (around 3.5MB) but has over two hundred commits in its version-control repository. Comparing two adjacent versions revealed modest changes: 10 methods modified, 9 added, and 2 deleted, resulting in (method-level) change ratios of 0.95%, 1.06%, and 0.21%, respectively. These minimal differences between versions raise the question: *is it necessary to perform a separate, full taint analysis for each version, especially when changes are so small?* Intuitively, the answer is clearly no—we may perform the analysis of the app in a standard way on its first (i.e., *base*) version and then compute taint flows only for the changed and change-impacted parts of each evolved version.

This is where our concept of *incremental taint analysis* comes into play. By building upon the analysis of prior versions, it seeks to provide an efficient alternative to the traditional, standalone method. For instance, in the *LMS* app case, the conventional taint analysis for the newer version required 1,104 seconds, whereas the *incremental taint analysis* took only 404 seconds. This resulted in a 63.4% efficiency improvement and a time saving of 700 seconds. Given the app's extensive commit history, the cumulative benefits of using our approach over conventional methods become even more pronounced. We anticipate that this efficiency will be greater in larger apps where conventional taint analysis is typically more time-consuming.

In summary, the introduction of *incremental taint analysis* and EvoTaint, our instantiation of this analysis methodology for static taint analysis of Android apps, is designed to make taint analysis a more practical and cost-effective part of a developer's workflow, particularly for apps that are frequently updated. In the rest of the paper, we present our design and evaluation of EvoTaint.

3 Approach

In this section, we present the design of our incremental taint analysis approach, starting with an overview of the methodology (§3.1) and followed by technical details of each of the three core components of our technique: *impact analysis* (§3.2), *impact-guided taint checking* (§3.3), and *taint synthesis* (§3.4).

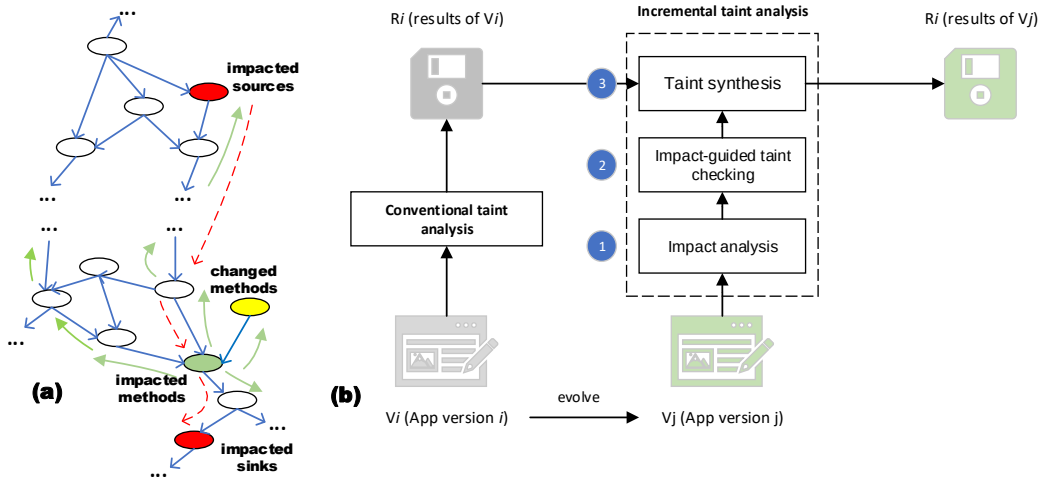

Fig. 1. An overview of our incremental approach to static taint analysis. It examines only the parts impacted by the incremental changes (from V_i to V_j, $i < j$) to compute taint flows to sources/sinks from each impacted entity (a), and synthesizes the solution for V_i with the impact-guided taint checking results to obtain the solution for V_j (b).

3.1 Overview of EvoTaint

Figure 1 depicts an overview of our incremental approach to static taint analysis of Android apps. In particular, the overall idea of our approach is illustrated in Figure 1 (a). Multiple versions of an evolving Android app under vetting are considered and sensitive information flows are examined on the basis of the analysis results of previous versions. Analysis of evolved versions focus on program entities (methods) affected by the incremental code changes between two versions of the same app to save analysis cost, rather than exhaustively examining all possible paths between any source and any sink in the evolved app. More specifically, for each evolved version of an app, the *changed methods* (added, deleted, and modified) between this version and a previous version of the same app are first obtained. Next, methods that are impacted by any of the changed methods (i.e., *impacted methods*) are identified. Then, backward-reachable sources and forward-reachable sinks are considered the *impacted sources* and *impacted sinks*, respectively. And the taint analysis can be performed incrementally (based on the results on the previous version) by searching (statement-level) taint flow paths between these impacted sources and sinks only.

This high-level process flow of our technique is illustrated in Figure 1 (b). As shown in this figure, our approach takes two apps that are typically different—ideally adjacent—versions (named V_i and V_j (i<j)), of the same app, as **input**. Given this input, first, a conventional taint analysis is applied to V_i to obtain the analysis result R_i for V_i, either through a whole-program (i.e., conventional) taint analysis if it is the first/base version of the app or, without loss of generality, through *incremental taint analysis* on its earlier version (iteratively). Then, based on R_i, a pass of *incremental taint analysis* is performed on V_j to obtain the taint analysis result R_j for V_j, which consists of three stages: *impact analysis*, *impact-guided taint checking*, and *taint synthesis*. In particular, the impact analysis aims to find methods impacted by code changes—i.e., impacted methods. It first performs app differencing between V_i and V_j to find changed methods, thereby finding impacted methods by forward traversing the call graph of the app. In Figure 1 (a), the method in yellow is such an example. Then, impacted sources and sinks are identified through backward and forward search on the same cal graph as noted above, and incremental taint flow paths between these impacted sources and impacted sinks are computed—a procedure referred to as impact-guided taint checking. In Figure 1 (a), solid gree arrows starting from impacted methods (in green color) correspond to identifying impacted sources and sinks by forward and backward graph traversal, whereas dashed red arrows starting from methods that have impacted sources and ending in methods that host impacted sinks (both in red color) demonstrate the impact-guided checking process. By doing this, we can avoid analyzing apps with numerous sources and sinks in order to get holistic taint flows in V_j. This holistic result (R_j) is obtained by taint synthesis, the last step of our approach, which combines/merges the impact-guided taint checking analysis result (i.e., incremental taint flow paths) with R_i to derive the full set of taint flows in V_j—the ultimate **output** of our technique for the given input.

Illustrating/working example. As noted above, each pass of *incremental taint analysis* addresses two versions of an app: the *original* version (e.g., V_i) before changes are made, and the *changed/evolved* version (e.g., V_j) that incorporated the changes. To illustrate our approach more vividly, consider a simple app with two versions V_i and V_j corresponding to Listing 1 and Listing 2, respectively. Suppose this app only has a button which will gather sensitive information (device id and SIM serial number) after click. As shown in Listing 1, taint sources are located in method onClick whilst taint sinks are located in methods *m_sink_*∗. Methods *m*1, *m*2, and *m*3 are bridge methods which are the intermediate nodes on the taint flow paths. The same app then evolves into the next version V_j with all the three kinds of code changes: method *m*3 is removed, a new method *m_sink_new* is added, and *m*2 is modified. As shown in Figure 2, every program path of the app that originates from method onClick and ends in any sink method is a taint flow path. After

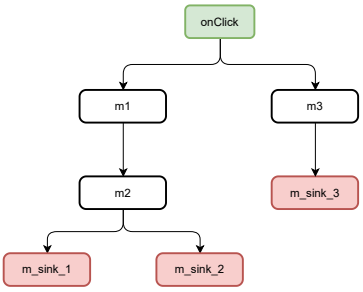

Fig. 2. The call graph of the example app V_i (Listing 1).

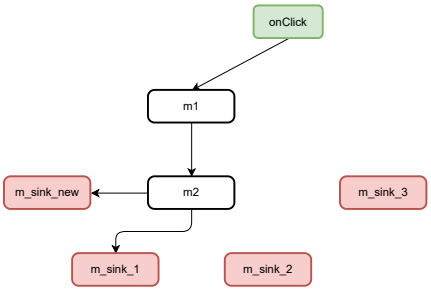

Fig. 3. The call graph of the example app V_j (Listing 2).

having evolved to V_j, this remains true for the app. However, there may be previous flow paths disappearing and new paths emerging between the two app versions. Below we use this example to walk through how our approach works in detail, performing taint analysis *incrementally* for this evolving app.

```
1   @Override
2   public void onClick(View arg0) {
3       Intent i = new Intent(Intent.ACTION_SEND);
4       i.setType("text/plain");
5       TelephonyManager tManager =(TelephonyManager) this.
      act.getSystemService(Context.TELEPHONY_SERVICE);
6       String uid = tManager.getDeviceId();//SOURCE
7       String num =tManager.getSimSerialNumber();//SOURCE
8       i.putExtra("secret", uid);
9       m1(uid,i);
10      m3(num);
11  }
12
13  public void m1(String uid, Intent intent) {
14      m2(uid, intent);
15  }
16
17  public void m2(String uid, Intent intent) {
18      m_sink_1(uid);
19      m_sink_2(intent);
20  }
21
22  public void m3(String num) {
23      m_sink_3(num);
24  }
25
26  public void m_sink_1(String uid) {
27      Log.i("uid", uid);  // SINK
28  }
29
30  public void m_sink_2(Intent intent) {
31      this.act.startActivityForResult(intent, 0); //SINK
32  }
33
34  public void m_sink_3(String num) {
35      Log.i("num", num); //SINK
36  }
```

Listing 1. Code snippet of V_i, lines in red are deleted in V_j.

```
37  @Override
38  public void onClick(View arg0) {
39      Intent i = new Intent(Intent.ACTION_SEND);
40      i.setType("text/plain");
41      TelephonyManager tManager =(TelephonyManager) this.
      act.getSystemService(Context.TELEPHONY_SERVICE);
42      String uid = tManager.getDeviceId();//SOURCE
43      String num =tManager.getSimSerialNumber();//SOURCE
44      i.putExtra("secret", uid);
45      m1(uid,i);
46  }
47
48  public void m1(String uid, Intent intent) {
49      m2(uid, intent);
50  }
51
52  public void m2(String uid, Intent intent) {
53      m_sink_1(uid);
54      m_sink_new(uid);
55  }
56
57  public void m_sink_1(String uid) {
58      Log.i("uid", uid);  // SINK
59  }
60
61  public void m_sink_2(Intent intent) {
62      this.act.startActivityForResult(intent, 0); //SINK
63  }
64
65  public void m_sink_3(String num) {
66      Log.i("num", num); //SINK
67  }
68
69  public void m_sink_new(String uid) {
70        Log.v("uid", uid); //SINK
71  }
72 }
```

Listing 2. Code snippet of V_j, lines in green are newly added.

To start with, the impact analysis identifies three categories of changes in V_j relative to V_i, which are addition (i.e., a new method is added to V_j since V_i; here *m_sink_new*), deletion (i.e., a current method in V_i is deleted during its evolution to V_j; here $m3$), and modification (i.e., a method is carried over from V_i to V_j, whose definition is changed but the name remains the same; here $m2$).

Next, the impact-guided taint checking is performed on V_j, concerning only the added and modified methods—after all, the deleted methods are not present in V_j anymore. Here the impacted methods of added and modified methods are *m_sink_*1 and *m_sink_*2, plus these changed methods themselves—changed methods are trivially considered their impacted methods as well by convention in the area of change impact analysis [20, 23–25]. Our approach traverses both backward and forward from each of these impacted methods on the call graph of V_j to find reachable methods that include calls to any sources and sinks, respectively.[1] All such sources and sinks are regarded as impacted sources (in this example, *tManager.getDeviceId* and *tManager.getSimSerialNumber*) and impacted sinks (in this example, *Log.v* and *Log.i*). In this example app which only has one simple component, it is not a surprise that most of the methods (hence sources and sinks) are impacted. In a real-world app which is generally more complex, it is reasonably expected that usually only a small part of the whole app (and a smaller subset of the sources/sinks considered) would be impacted because typically the changes are usually small (e.g., in a repository commit or between adjacent app versions).

After obtaining these impacted sources and sinks, our approach applies the conventional (statement-level) taint analysis (e.g., FlowDroid [8]) to V_j with the analysis configuration set such that the analysis only considers these impacted sources and sinks, so as to get the taint analysis result R_{ij}—i.e., statement-level taint flow paths between the impacted sources (*tManager.getDeviceId* and *tManager.getSimSerialNumber*) and the impacted sinks (*Log.v* and *Log.i*). Here R_{ij} consists of two flow paths: $42 \hookrightarrow 45 \hookrightarrow 49 \hookrightarrow 53 \hookrightarrow 58$ and $42 \hookrightarrow 45 \hookrightarrow 49 \hookrightarrow 54 \hookrightarrow 70$.

Now, a similar impact-guided taint checking would be performed on V_i, concerning only the modified and deleted methods—after all, the added methods are not presented in V_i. However, to be more efficient, our approach takes a shortcut, directly dealing with the impact of the code changes between V_i and V_j on the taint analysis result of V_i (i.e., R_i) during the taint synthesis step. The rationale is two-fold. First, for the deleted methods, it is intuitive that our approach would just compute taint flow paths between sources and sinks impacted by these methods and their impacted ones, and then remove these paths from R_i in order to obtain the remaining flow paths that should be carried over to the taint analysis result of V_j (i.e., R_j). Second, given that we have only computed the taint analysis results induced by the modified methods for V_j, taint flow paths between sources and sinks impacted by the modified methods and their impacted ones in V_i should also be removed from R_i for the purpose of obtaining R_j. To identify these taint flow paths in R_i to remove, we do not really need to conduct impact-guided taint checking on V_i, though: we can simply identify them directly from R_i—i.e., flow paths in R_i that pass through any of the deleted or modified methods.

In the last step, our approach performs taint synthesis, merging R_i and R_{ij}. Let us assume that R_i has been computed in advance, which includes three taint flow paths according to the sources and sinks considered as marked in Listing 1: $6 \hookrightarrow 9 \hookrightarrow 14 \hookrightarrow 18 \hookrightarrow 27$, $6 \hookrightarrow 9 \hookrightarrow 14 \hookrightarrow 19 \hookrightarrow 31$, and $7 \hookrightarrow 10 \hookrightarrow 23 \hookrightarrow 35$. As justified above, we would remove taint flow paths in R_i that are impacted by deleted or modified methods. Thus, the first two paths are removed since they pass through the modified method $m2$ and the third path is also removed as it passes through the deleted method $m3$. As a result, the synthesis results in R_j as $42 \hookrightarrow 45 \hookrightarrow 49 \hookrightarrow 53 \hookrightarrow 58$ and $42 \hookrightarrow 45 \hookrightarrow 49 \hookrightarrow 54 \hookrightarrow 70$.

[1] Note that for efficiency purposes, our control flow analysis does not descend down to the Android SDK/libraries; thus, the code of sources and sinks (which are usually library/SDK APIs) is not analyzed during the call graph construction. As a result, the call graph does not include nodes corresponding to source/sink methods.

3.2 Impact Analysis

The key to the efficiency enhancement with *incremental taint analysis* is to avoid the redundant computation for program entities of the original app that are not impacted by the changes during the evolution. Given this insight, the first step of EvoTaint is to identify the impacted entities—which is achieved by (change) impact analysis [13, 22].

For impact analysis, given an earlier version V_i and a later (evolved) version V_j of an app, the first step is to find the difference between V_i and V_j, i.e., what changes are introduced in V_j. In our incremental analysis methodology, the main idea is to identify change-impacted sources and sinks, so that the fine-grained (statement-level) standalone taint analysis can be performed with respect to those impacted sources and sinks only. Thus, overall, the changes are defined and computed at method level: the resulting *change set* is a set of methods. Also, we quantify the changes by the percentage of changed methods, named *change ratio*. Specifically, changes can be classified to three different kinds: addition, modification, and deletion. For added and deleted methods, we can find them simply by comparing the method signature for all the methods between the two versions.

For modified methods, the detection is performed at statement level. In other words, if any statement within a method is different, we consider this method modified. This fine-grained detection is necessary, because statement-level changes may lead to changes in (ultimately statement-level) taint flows relative to the results of the previous version V_i. In particular, even if the backward-reachable sources and forward-reachable sinks of a method remain the same between V_i and V_j, the method still needs to be treated as changed if the method body or signature (e.g., the set of parameters) is changed, because the taint flows between those sources and sinks may be impacted by the changes in the method. Including such methods in the change set in our impact analysis ensures that those sources and sinks are identified as impacted hence will still be considered during the impact-guided taint checking step (in particular, during the step of applying the conventional statement-level taint analysis for the impacted sources/sinks), essential for the soundness of our incremental taint analysis.

The (method-level) impact analysis [14, 21] itself is straightforward. Once the changed methods are identified, EvoTaint computes their corresponding impacted methods by traversing on the call graph of the app: all the forward-control-flow-reachable methods on the call graph are considered impacted. Per our discussion of the intuition behind the design of EvoTaint (*S*3.1), impact-guided taint checking only needs to be performed on V_j. So, we only need to identify impacted taint sources and sinks in V_j. Accordingly, the impact set only needs to computed in V_j. Thus, the call graph used in the impact analysis is that of V_j.

Next, we elaborate on the two major steps of the impact analysis module in EvoTaint: *identifying changed methods* and *computing impacted methods*.

3.2.1 Identifying Changed Methods. Algorithm 1 delineates the process for detecting changes between two versions of an app. Initially, the entire set of methods in the original and evolved apps is retrieved. Our version differencing algorithm then iterates through each method in the original app (line 6). For every method in the original version, the algorithm searches for a corresponding method with an identical signature in the evolved app (line 7). In cases where a matching method is not found, it is classified as deleted (D) and subsequently added to the set of deleted methods (lines 8-9). Upon detecting a match between methods of the original and evolved apps, the algorithm conducts a statement-level comparison (line 11). Any disparity in statements of such methods leads to the classification of those method as modified (M) (line 12).

Following the identification of both deleted (D) and modified (M) methods via the traversal of the original app's methods, the algorithm proceeds to analyze the methods in the evolved app (line 13). It mirrors the earlier process:

Algorithm 1: Identify Changed Methods

Input: M_{org}: the set of methods in the original version (V_i)
Input: M_{evo}: the set of methods in the evolved version (V_j)
Output: M_{AMD}: the map from each change type to the set of methods changed in that type

```
1  Function FindChangedMethods(M_org, M_evo)
2      M_AMD ← ∅;
3      M_A ← ∅;
4      M_M ← ∅;
5      M_D ← ∅;
6      foreach method om in M_org do
7          m ← searchMethodInM (om, M_evo)
8          if m = ∅ then
9              M_D.add(om)
10         else
11             if twoMethodsEqual (om,m) then
12                 M_M.add(m)
13     foreach method em in M_evo do
14         if searchMethodInM (em, M_org) = ∅ then
15             M_A.add(m)
16     M_AMD.put(A,M_A);
17     M_AMD.put(M,M_M);
18     M_AMD.put(D,M_D);
19     return M_AMD
```

for each method in the evolved version, it looks for a method with a matching signature in the original app (line 14). Methods in the evolved app without corresponding matches are marked as added (line 15).

3.2.2 Computing Impacted Methods. After categorizing methods into the three distinct change types: added (A), modified (M), and deleted (D), the next step is to compute impacted methods. The main idea is to (1) build the app call graph and then (2) compute impact set based on control flow reachability on the call graph.

To optimize efficiency without compromising accuracy, added and modified methods are grouped together for impact analysis on V_j. This treatment is inspired by two insights. First, modification can be essentially broken down into *deletion + addition*: modified methods can be treated as their original version being deleted from V_i, followed by their evolved version being added to V_j. Thus, the three kind of changes can be reduced to two kinds: deletion in V_i and addition in V_j. Second, deleted methods, which will not be involved in the impact-guided taint checking; instead, they will be directly handled during the taint synthesis step, as discussed earlier (§3.1).

Therefore, only the call graph of the evolved app (V_j) is constructed, while that of V_i is not needed. To reiterate the justification, this decision is driven by two factors: (1) efficiency, as modified methods in the original app are treated as deleted (and then added to the evolved app); and (2) correctness, since some newly added methods, which only exist in the evolved app, are inherently impacted.

It is important to note that the mere identification of impacted methods does not immediately lead to the efficiency merits of *incremental taint analysis*. Oftentimes, these impacted methods do not always yield new taint flow paths. The focus, instead, is on the sources and sinks within data flow paths affected by the change set, forming the basis of the taint analysis. This strategy is central to the cost-effectiveness of our methodology: it eliminates the need to reanalyze unchanged data flow paths between the base and evolved apps. Our algorithm distinctively analyzes only the changed/impacted data flow paths between given sources and sinks, enhancing both efficiency and precision.

3.3 Impact-Guided Taint Checking

This section outlines the procedure of our impact-guided taint checking module, which computes the effects of code changes between the two given apps (app versions) on the static taint analysis of the original app version. The essence of this approach is to refine the scope of sources and sinks between which the analysis needs to search information flow paths, based on the (indirect) influence of those code changes on them. The input of this module is the set of impacted methods and the output is the additional taint flow paths in the evolved app/version compared to the base analysis result (R_i) of the original app/version (V_i). As discussed in §3.2, the impacted methods here are the set of methods impacted by the added and modified methods, and the taint checking is performed on the evolved app/version (V_j) only.

Algorithm 2: Impact-Guided Taint Checking

Input: V_j: the evolved version of the app
Input: M_{AMD}: the map from each change type to the set of methods changed in that type between the prior version of V_j and V_j
Output: R_{ij}: incremental taint flow paths in V_j

1 **Function** *FindImpactedSourcesSinks*(M_{AMD}, CG_{evo})
2 $M_A \leftarrow M_{AMD}.get(A)$;
3 $M_M \leftarrow M_{AMD}.get(M)$;
4 $M_{AM} \leftarrow M_A \bigcup M_M$;
5 $M_{impacted} \leftarrow$ `getSuccessors` (M_{AM}, CG_{evo});
6 $M_{sources} \leftarrow$ `getPredecessors` ($M_{impacted}$, CG_{evo});▷ identify sources *backward*-reachable from any impacted method
7 $M_{sinks} \leftarrow$ `getSuccessors` ($M_{impacted}$, CG_{evo}); ▷ identify sinks *forward*-reachable from any impacted method
8 $SourceSinkDefinition \leftarrow deserialize(SourcesAndSinks.txt)$;
9 $Sources \leftarrow$ `retriveSources` ($M_{sources}$, $SourceSinkDefinition$);
10 $Sinks \leftarrow$ `retriveSinks` (M_{sinks}, $SourceSinkDefinition$);
11 **return** $S_{impacted} \leftarrow Sources, Sinks$
12 **Function** *TaintChecking*(M_{AMD}, V_j)
13 $CG_{evo} \leftarrow$ `constructCallGraph` (V_j);
14 $Src_{impacted}, Sink_{impacted} \leftarrow$ `FindImpactedSourcesSinks` (M_{AMD}, CG_{evo});
15 $R_{ij} \leftarrow$ getTaintFlowPaths($Src_{impacted}$, $Sink_{impacted}$, V_j);
16 **return** R_{ij}

Since precise taint checking addresses the (information-flow) reachability from sources to sinks that are specified beforehand, the key to our impact-guided taint analysis lies in reducing the scope of sources and sinks by identifying those that are (indirectly) affected by the code changes between V_i and V_j. As depicted in Algorithm 2, the process begins with identifying impacted methods (lines 2-5). For each impacted method, the algorithm traverses backward in the evolved app's call graph (line 6), marking all backward reachable methods as ancestors. Following this, it traverses forward (line 7), marking forward reachable methods as descendants. Notably, the descendant methods are essentially impacted methods themselves, allowing for the omission of this step to enhance efficiency.

Furthermore, the algorithm retrieves the signatures of methods within these ancestors and descendants. It then compares these signatures with source/sink definitions from a complete, predefined source/sink set (noted as *SourcesAndSinks.txt*) (lines 8-9). Identified source methods are added to the set of *impacted sources*. Similarly, it then identifies sinks within descendant methods (line 10), adding them to the set of *impacted sinks*. The expectation is that the impacted sources and sinks represent a small fraction of the complete source/sink set, which aligns with the condition of efficiency merits of our approach. These impacted sources and sinks are serialized in a format compatible with the conventional taint analyzer (e.g., FlowDroid [8]).

Once the call graph of V_j is constructed (line 13) and the impacted sources and sinks are identified (line 14), EvoTaint applies the conventional taint analyzer of choice to V_j while considering these impacted sources/sinks only (line 15), rather than the complete set of predefined sources/sinks in a common use scenario. This impact-guided taint checking step results in the set R_{ij} of taint flow paths induced by the added and modified methods between V_i and V_j (line 16). Here, performing the conventional taint analysis is denoted by invoking a subroutine getTaintFlowPaths (line 15) which essentially computes taint flows between the given lists of sources and sinks.

3.4 Taint Synthesis

The synthesis process integrates the taint analysis results for the evolved app version, R_j, by combining the original/base version's results, R_i, with the outcomes of the impact-guided taint checking, denoted as R_{ij} (as per Algorithm 2). Specifically, for the base version V_0, R_0 is derived using the standard conventional static taint analysis—i.e., invoking the subroutine getTaintFlowPaths on the complete sets of all possible sources and sinks considered. For other versions, R_i ($i > 0$) is derived from the prior incremental taint analysis between V_i and the version prior to V_i.

The taint synthesis occurs in two primary stages. In the first stage, any flow paths in R_i that pass through (any statements enclosed in) any of the modified (M) or deleted (D) methods are excluded. In the second stage, the remaining elements of R_i are merged with R_{ij}, the result of the impact-guided taint checking for the evolved app version, to get the final result R_j (for the evolved app). This approach ensures a comprehensive and updated representation of data flow paths in the evolved app.

This synthesis approach also ensures that no security vulnerabilities are overlooked, regardless of whether they exist in changed or unchanged code. When analyzing an evolved version of an app, any security issues present in unchanged code would have already been identified in the base version's taint analysis results (R_i). During the taint synthesis step, we carefully merge these existing results with the new analysis results from changed code (R_{ij}) to produce the final results (R_j). Specifically, only the taint flows that pass through modified or deleted methods are removed from R_i, while all other taint flows—including those representing security issues in unchanged code—are preserved in the final results. Taint flows with respect to the effects of the modifications (in V_j) are later added to R_j. In this way, only the taint flows that become absent due to the evolution are removed from R_i.

To illustrate how EvoTaint performs taint synthesis, consider the example shown in Figure 4, where the top diagram shows a base version of an app and the bottom shows its evolved version, with corresponding taint flow paths listed on the right. The synthesis process starts with the base version containing five taint flow paths (shown in "Base Paths"). When handling the evolved version, EvoTaint first removes paths that are impacted by modified or deleted methods from the base version's results. In this example, four paths are removed: two paths through m2 (which was modified), one path through m3 (which was deleted), and one path through m_sink_4 (which was deleted), leaving only the path "onResume -> m_sink_5" from the base results. Then, EvoTaint adds the taint flow paths computed for the evolved version that are impacted by modified or added methods. These include two recomputed paths through the modified method m2 to existing sinks m_sink_1 and m_sink_2, one new path through m2 to a newly added sink (m_sink_new_1), and one completely new path to another added sink (m_sink_new_2). Finally, EvoTaint synthesizes the complete set of taint flows for the evolved version by combining the remaining path from the base version with these newly computed paths, resulting in five synthesized paths as shown in the bottom-right box of the figure.

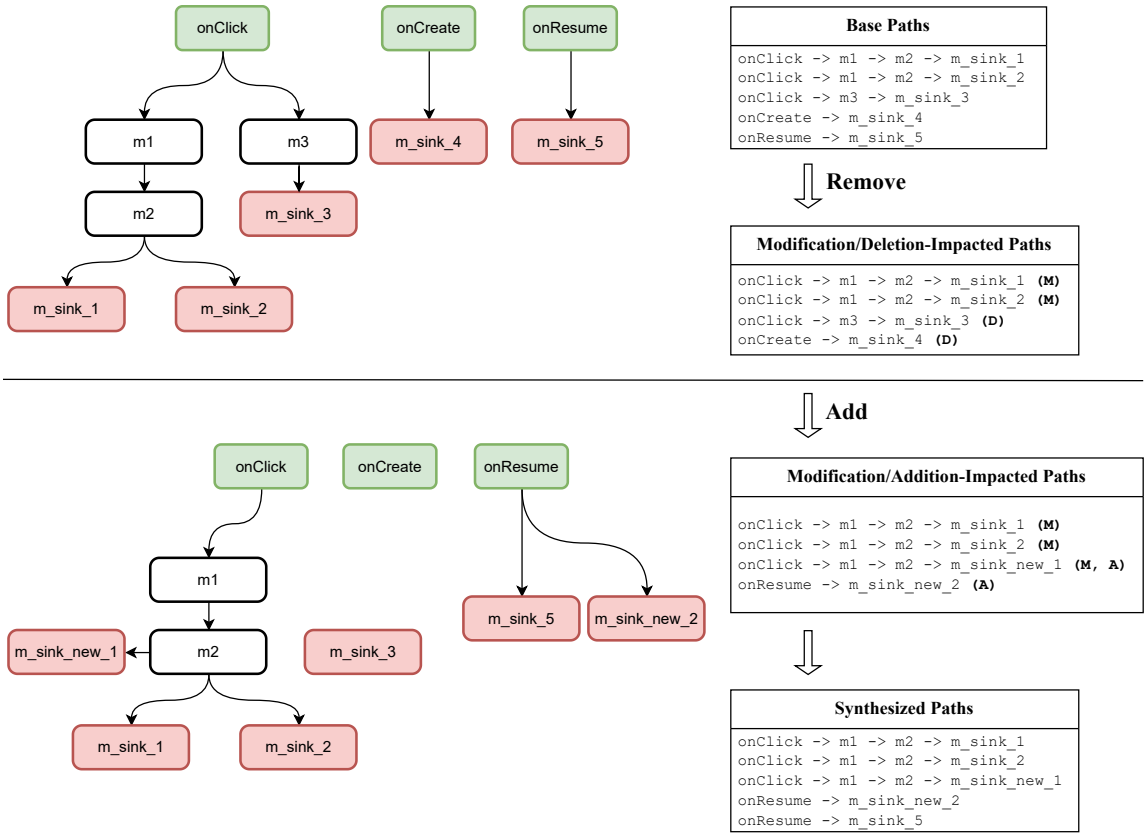

Fig. 4. An example of taint synthesis.

4 Implementation and Limitations

We implemented our technique as a tool, EvoTaint, based on Soot [46] and FlowDroid [6]. These supporting tools expect APK inputs and provide essential functionality that would be complex to reimplement for source code analysis. For instance, for impact analysis, we use Soot to extract methods from APK files to conduct differencing. In this way, we can skip the engineering effort of analyzing version history, parsing source code, and then identifying changed methods. This is why we chose to work with APKs despite targeting developers who have source code access. In general, for the conventional taint analysis (i.e., the subroutine getTaintFlowPaths) employed in both the impact-guided taint checking and taint synthesis modules, the core is an exhaustive, pair-wise reachability analysis that tracks all possible sensitive data flows between all predefined sources and sinks. In particular for Android apps in this work, we use FlowDroid [6] for this conventional whole-program taint analysis. For precise analysis results, we adopt the default, conservative settings for flow and context sensitivity.

In addition, FlowDroid is also used for constructing the call graph of V_j (i.e., the subroutine *constructCallGraph*) underlying the impact-guided taint checking module of EvoTaint. Soot is used by FlowDroid itself, and additionally in EvoTaint for retrieving/traversing all of the methods and statements in a method of a given Android app (APK), so as to enable the app differencing step of the impact analysis in EvoTaint (§3.2.1).

Since many real-world Android apps are obfuscated, which impedes both the app differencing and impact-guided taint checking in EvoTaint, EvoTaint applies deobfuscation to the given apps (V_i and V_j) prior to the impact-analysis phase. While deobfuscation is orthogonal to EvoTaint and dealing with itself is out of scope of this work, it is a necessary engineering step for a practical tool. To deobfuscate an app, we used two state-of-the-art Android deobfuscators, Simplify [31] and DeGuard [12] (as the primary and secondary options, respectively). Thus, our current implementation will not work well on apps that either cannot be analyzed by Soot/FlowDroid or have been obfuscated but cannot be deobfuscated with the two deobfuscation tools (a more advanced deobfuscator can be plugged in to replace them). By the same token, EvoTaint relies on the ability to obtain the taint analysis results of the base version of the given evolved app—if for the given app the base-version result cannot be obtained (e.g., when using FlowDroid against the base version, because it ran out of the user's time budget while not producing the taint flow result), then EvoTaint would not work properly for this app either. While for increasing the applicability of our tool, we deal with obfuscation purposely, we note that in the most typical use scenario of incremental taint analysis, developers would want to check each evolved (committed or released) version against taint flows as part of the security vetting during continuous integration/development, where there is no motive for obfuscating the app at that point. Thus, in those common use scenarios, deobfuscation is not essential or even necessary. Accordingly, possible failures to deobfuscate successfully should not become a practicality barrier to EvoTaint.

The source/sink identification is orthogonal to our incremental taint analysis approach. The complete, predefined source/sink set (i.e., *SourcesAndSinks.txt* in Algorithm 2) is generated using SuSi [5]. Susi is a machine-learning based automated tool that classifies each of the SDK APIs in a given version of the Android platform as a source, a sink, or neither. Considering the evolution of the Android platform, we applied SuSi to all the existing known Android versions (by the time this paper is written), obtaining the complete source/sink set for each Android version in terms of API level. For a given Android app, EvoTaint automatically retrieves the minimum Android platform version the app is aimed to run on (as indicated by the *minSdkVersion* attribute in the app's *AndroidManifest.xml* file located in the app's APK) and chooses the corresponding source/sink set to use for the static taint analysis.

Our technique currently achieves an efficiency improvement in the *incremental taint analysis* mainly by reducing the sources and sinks to be applied to the conventional taint analysis. Directly computing affected (statement-level) taint flows via incremental data flow analysis [53, 56, 59] would avoid invoking the getTaintFlowPaths subroutine, which however may be more expensive than our current straightforward approach hence making the overall *incremental taint analysis* for Android apps less cost-beneficial.

Moreover, it should be noted that the cost-effectiveness of EvoTaint is constrained by the number of impacted flow paths, which intuitively would be affected by the changes between the two (preferably adjacent) versions of an app under analysis. In the case of this number being very large, the cost of incremental analysis may not be paid off.

Regarding the usability of our tool, EvoTaint follows a straightforward installation process as a standard Java application. The tool requires minimal configuration, so developers who are already familiar with FlowDroid can easily transition to using EvoTaint, as it maintains similar input and output formats. We provided simple command-line interface pattern: `java -jar evotaint.jar base.apk evolved.apk` to run EvoTaint. The tool can also be incorporated into continuous integration pipelines with minimal additional effort.

5 Evaluation Design

In this section, we describe the design of empirical evaluation of EvoTaint, including the research questions that guide our evaluation (§5.1), the datasets used as study subjects (§5.2), and the evaluation methodology (§5.3), following the

guidelines from [65]. The goal of our empirical evaluation is to assess whether incremental taint analysis provides statistically significant efficiency improvements over conventional taint analysis while maintaining the same level of accuracy and explore factors that impact EvoTaint's effectiveness.

5.1 Research Questions

Our evaluation aims to address the following six research questions (RQs):

- **RQ1:** What are the time efficiency advantages of *incremental taint analysis* compared to conventional (standard/whole-app) taint analysis?
- **RQ2:** Under what conditions does *incremental taint analysis* demonstrate its time efficiency advantages?
- **RQ3:** How does the number of impacted sources and sinks influence the time efficiency of *incremental taint analysis* (in terms of cost reduction)?
- **RQ4:** How does app size influence the time efficiency of *incremental taint analysis*?
- **RQ5:** Does our *incremental taint analysis* lead to reductions in memory usage?
- **RQ6:** Does our incremental taint analysis achieve the same accuracy in taint checking as the conventional approach?

To be specific, these RQs are set as justified as follows. **RQ1** examines whether our approach can reduce the time cost of taint analysis when analyzing evolved versions of Android apps; **RQ2** investigates the relationship between code changes and analysis efficiency; **RQ3** explores how the reduction in analysis scope affects performance, helping understand the underlying mechanisms of our efficiency gains; **RQ4** examines whether the benefits of our approach vary with app size, helping understand its scalability; **RQ5** investigates whether our approach provides benefits beyond time efficiency by examining its impact on memory consumption. **RQ6** validates that our efficiency improvements do not come at the cost of reduced analysis accuracy.

The subsequent sections detail the experimental setup and methodology employed to answer these research questions.

5.2 Dataset

To evaluate EvoTaint, we applied our incremental taint analysis to a diverse set of Android apps sourced from the F-Droid [1] repository. Our dataset comprises 100 unique apps, encompassing 255 distinct versions. This selection includes 100 apps each with 2 versions, 10 apps each with 3 versions, and 5 apps each with 5 versions, together making up of 255 distinct versions. Specifically, 10 apps from the 2-version group were also used in the 3-version group and 5-version group. This overlap was intentional and serves a specific research purpose: it allows us to evaluate how EvoTaint performs on the same app across different lengths of evolution history. For each app with multiple versions, the lowest version number was designated as the base version, while subsequent versions were treated as evolved versions, resulting in a total of 140 app pairs, i.e., 100 pairs for the second versions in the 2-version group, plus 20 pairs for the second and third versions in the 3-version group, plus 20 pairs for the second through fifth versions in the 5-version group., [2] undergoing *incremental taint analysis*.

Although many of the chosen subject apps each has more versions than we considered, we chose 2, 3, and 5 versions for research evaluation purposes. In particular, we consider these three groups of benchmarks in order to account for different use scenarios of incremental taint analysis in terms of the extent to which developers may benefit from incrementally checking sensitive information flow during app evolution. In particular, by analyzing an app pair, i.e., 2

[2] 2-version group: $100 \times (2-1) = 100$, 3-version group: $10 \times (3-1) = 20$, 5-version group: $5 \times (5-1) = 20$

Table 1. 2-version benchmark group, including for each app the package name, functionality description, and the version no. and size (in terms of #methods) of each of the two considered versions (V_i and V_j).

Package name	Functionality description	V_i		V_j	
		version no.	#methods	version no.	#methods
agrigolo.chubbyclick	FOSS Metronome with the gigging musician in mind.	22	399	23	410
app.fedilab.mobilizon	a tool to help manage your events, your profiles and your groups.	1	3,446	2	3,442
app.fedilab.openmaps	Display maps with OpenStreetMap	11	1,943	12	1,967
app.mlauncher	Forked from Olauncher. Minimal and clean	64	13,991	65	13,937
at.bitfire.nophonespam	Block calls from annoying or private numbers on your Android device.	13	349	14	349
at.linuxtage.companion	Browse the "Grazer Linuxtage" schedule	1700005	6,978	170000009	6,531
be.brunoparmentier.wifikeyshare	Share Wi-Fi passwords with QR codes and NFC tags	2	509	3	512
be.ppareit.swiftp_free	Access your phone wirelessly	30000	842	30001	843
bou.amine.apps.readerforselfossv2.android	A new RSS reader for selfoss.	123051471	26,711	123061651	26,711
ca.mimic.apphangar	Access recent and top apps	62	587	74	611
ca.rmen.android.networkmonitor	Check network connectivity	13100	3,195	13101	3,197
click.dummer.have_radiosion	Internet radio player with comic figure as music visualizer	25	318	26	321
click.dummer.imagesms	Send very small photos with long text SMS and without MMS or internet	13	297	14	297
click.dummer.textthing	a simple text file editor	214	184	215	184
click.dummer.UartSmartwatch	Bluetooth App for a UART-Smartwatch	31	155	32	155
cloud.valetudo.companion	Easily find and connect Valetudo robot vacuums on and to your network	11	235	12	237
com.aaronhalbert.nosurfforreddit	Browse the top posts on Reddit without infinite scrolling or addictive features	5	16,007	6	16,089
com.abh80.smartedge	Android alternative for Dynamic Island	20202	6,015	20203	6,032
com.adam.aslfms	Last.fm/Libre.fm Scrobbler	55	904	58	906
com.addi	Math calculation environment	37	2,092	40	2,091
com.akdev.nofbeventscraper	Import Facebook-Events to the calendar	13	4,654	14	4,654
com.alaskalinuxuser.justcraigslist	Search posts on Craigslist	9	113	10	115
com.altillimity.satpredict	An offline satellite tracking app	1	6,852	2	6,856
com.aptasystems.dicewarepasswordgenerator	Generate diceware passwords	8	1,083	9	1,192
com.benny.pxerstudio	Pixel drawing tool for Android	7	1,466	8	1,465
com.bleyl.recurrence	Get reminded about notifications	22	998	23	998
com.cgogolin.library	Browse BibTeX files	60	278	61	278
com.chesire.nekome	Keep track of your anime and manga, through the use of the Kitsu API.	23062323	14,851	23062419	14,910
com.concept1tech.instalate	Online dictionary - Translate content directly in your apps	11	2,419	12	2,419
com.corvettecole.gotosleep	Reminds you to go to sleep… until you do	50	1,536	57	1,552
com.coste.syncorg	Take and organize notes	10	9,309	7	9,293
com.craigd.lmsmaterial.app	Simple webview wrapper for MaterialSkin on an LMS server	301	919	400	954
com.danefinlay.ttsutil	Text-to-Speech Utility Application	6	4,218	7	4,220
com.danhasting.radar	An application featuring customizable, real-time doppler radar images	3	7,155	4	6,485
com.danielkim.soundrecorder	Record audio files	130	1,049	5	1,037
com.decred.decredaddressscanner	Scans Decred addresses for available funds and notifies of changes.	8	993	9	946
com.denytheflowerpot.scrunch	Play sounds whenever you fold or unfold your Galaxy Z Fold.	2	3,438	3	3,438
com.dfa.hubzilla_android	Client for the Hubzilla social network	44	1,710	45	1,710
com.dkanada.chip	Chip8 emulator	1	2,759	2	2,759
com.dozingcatsoftware.bouncy	Pinball game	36	12,851	37	12,856
com.eventyay.organizer	Event management app for Organizers using eventyay platform	12	28,960	13	29,274
com.example.tobiastrumm.freifunkautoconnect	Add multiple Freifunk SSIDs to your device	10	1,022	8	385
com.fabienli.dokuwiki	Access in local to your dokuwiki	71	1,211	72	1,211
com.fastebro.androidrgbtool	Get RGB and HEX values of a color	11	4,714	12	5,172
com.fediphoto	Fedi Photo - quickly post photos to the Fediverse.	24	147	25	144
com.flasskamp.subz	Subscription and contract management with notification function.	2	664	3	1,665
com.forrestguice.suntimescalendars	A calendar provider add-on for Suntimes.	15	587	16	589
com.fredhappyface.fhcode	Code editor for android	20211104	257	20220110	263
com.freshollie.monkeyboard.keystoneradio	Monkeyboard FM & DAB/DAB+ radio interface	115	851	116	851
com.gbeatty.arxiv	Browse, search, and download arXiv articles with arXiv eXplorer!	34	2,529	39	2,649
com.github.cythara	Musical instrument tuner	25	1,590	27	1,601
com.github.dawidd6.andttt	simple tic tac toe game	64	8,125	65	10,009
com.github.gianlucanitti.expreval	Math expression calculator with variables and functions support	1	237	2	258
com.github.igrmk.smsq	Receive your SMS messages in Telegram	13	1,127	14	1,131
com.github.muellerma.coffee	Keep display awake	37	742	38	755
com.github.muellerma.mute_reminder	Remind you to mute media	15	697	16	696
com.github.muellerma.prepaidbalance	Keep track your prepaid balance	30	907	31	907
com.github.muellerma.stopwatch	Stopwatch as quick tile	1	46	2	46
com.github.nutomic.controldlna	DLNA/UPnP control point	12	5,400	13	5,400
com.github.pires.obd.reader	Car diagnostics	12	7,910	13	7,916
com.gitlab.dibdib.dib2calc	The crazy calculator. Once you get used to it, you will love it :-)	2324	1,130	2326	1,130
com.glanznig.beepme	Experience sampling (ESM/DES)	20	916	21	936
com.gmail.afonsotrepa.pocketgopher	A fast, elegant and modern Gopher Client	1	375	2	375
com.haringeymobile.ukweather	View weather forecast	27	1,464	30	1,478
com.hwloc.lstopo	Display the topology of your hardware using the hwloc library and lstopo tool	269	562	270	562
com.james.status	An overlay-based statusbar replacement	41	5,075	42	5,282
com.jarsilio.android.autoautorotate	Automatically remembers your auto-rotate setting for every app	30	6,508	31	6,518
com.jereksel.libresubstratum	Bucket is UNOFFICIAL app for Substratum Theme Engine	1	5,888	3	6,500
com.jim.sharetocomputer	Share anything to your computer	1120	2,150	1130	2,155
com.ketanolab.nusimi	Nusimi is your dictionary of native american languages	6	332	7	332
com.knirirr.beecount	Knitting row counter	119	3,299	120	3,299
com.kolserdav.ana	Foreign Language Dictionary	135	1,105	136	1,119
com.ktprograms.watertracker	reminds you to drink water	3	63	4	63
com.launcher.silverfish	A launcher with focus on a simple UI	7	1,348	8	1,350
com.manimarank.websitemonitor	This app helps to monitor your websites periodically with notification.	5	6,143	6	6,143
com.mehmetakiftutuncu.eshotroid	Bus times in Turkey	3	449	4	451
com.mikifus.padland	Padland is a tool to manage, share, remember and read collaborative documents ba	23	786	24	789
com.mrbimc.selinux	Set SELinux mode on boot	20170724	6,266	20171031	6,268
com.naman14.stools	Collection of system tools	10	1,047	11	1,047
com.nathaniel.motus.cavevin	Wine cellar manager	2	301	3	301
com.nhellfire.kerneladiutor	Manage kernel parameters	245	7,391	246	7,290
com.nilhcem.hostseditor	Edit system hosts file	3	4,562	4	4,712
com.oF2pks.jquarks	LineageOS Jelly browser +adsBlocker +tiles +offline_.mht +...	17	507	18	507
com.philliphsu.clock2	View time, set alarms and timers	112	2,015	113	2,015
com.phpsysinfo	Monitor phpSysInfo	940	549	950	549
com.pindroid	Bookmark manager	68	1,633	69	1,635
com.quaap.bookymcbookface	An extremely unfancy and very basic ebook reader	420	697	430	697
com.rastating.droidbeard	Manage SickBeard installations	1500	854	1502	854
com.sensirion.smartgadget	Connect to your Sensirion Smart Gadget via BLE	90	2,644	96	2,578
com.servoz.appsdisabler	Disable any applications, and easily enable it with a pretty customisable launch	121	1233	122	1233
com.stoptheballs	Flying balloons shooting. Action. Shooter for kids and adults.	108	1,794	114	11,901
com.tomer.draw	Draw and take notes everywhere on your device	51	1,553	100	1,555
com.truchsess.send2car	send geo-locations to your bmw car from any app that allows sharing locations	11	3,220	12	3,220
com.tunerly	A minimalistic pitch tuning app for Guitar, Bass and Ukulele.	6	1,236	7	1,236
com.veniosg.dir	Modern file manager	1523	2,449	1560	2,455
com.vlath.beheexplorer	Browse the web	20051	472	20064	485
com.workingagenda.fissure	Create and view GIFs	2	146	3	146
com.chooloo.www.koler	uniquely stylized phone app with customizable features	145	15,524	146	15,716
com.github.muellerma.mute_reminder	Remind you to mute media	22	728	23	728
de.lukasneugebauer.nextcloudcookbook	View all your recipes stored in your Nextcloud instance.	140	12,866	141	12,866

Table 2. 3-version benchmark group, including for each app the package name, and the version no. and size (in terms of #methods) of each of the three considered versions (from V_0 to V_2).

Package name	V_0		V_1		V_2	
	version no.	**#methods**	**version no.**	**#methods**	**version no.**	**#methods**
com.github.nutomic.controldlna	12	5,400	13	5,400	14	5,406
com.bleyl.recurrence	22	998	23	998	24	986
app.fedilab.mobilizon	1	3,446	2	3,442	3	3,444
com.flasskamp.subz	2	1,664	3	1,665	4	1,828
agrigolo.chubbyclick	22	399	23	410	24	437
com.gianlu.dnshero	38	2,991	39	3,072	40	3,072
com.craigd.lmsmaterial.app	301	919	400	954	401	961
com.forrestguice.suntimescalendars	15	587	16	589	17	625
com.eventyay.organizer	12	28,960	13	29,274	14	30,768
com.launcher.silverfish	6	1,348	7	1,347	8	1,355

Table 3. 5-version benchmark group, including for each app the package name, and the version no. and size (in terms of #methods) of each of the five considered versions (from V_0 to V_4).

Package name	V_0		V_1		V_2		V_3		V_4	
	version no.	**#methods**	**version no.**	**#methods**	**version no.**	**#methods**	**version no.**	**#methods**	**version no.**	**#methods**
com.chooloo.www.koler	8d8d785	15,513	80966ea	15,460	e3c38de	15,478	fc1df6d	15,524	da638ee	15,716
de.lukasneugebauer.nextcloudcookbook	e536f2f	12,866	a396f00	12,866	1472453	12,866	7f2e2e7	12,867	559938d	12,867
com.craigd.lmsmaterial.app	a5ac007	944	064cc0b	944	a5f8b53	951	0bfcfae	951	2c672b2	955
com.github.muellerma.mute_reminder	4ccf8fc	728	c61e0ae	728	acf869c	739	a588102	741	9d69f18	688
com.github.cythara	a74da7d	1,546	a419097	1,590	9a6165e	1,601	b435ee6	1,601	1871ad9	1,628

versions of an app, we are able to assess the efficiency benefits of *incremental taint analysis* in the one-time evolution scenario—i.e., when developers just want to use the incremental analysis on one evolved version only. By analyzing apps with multiple versions, i.e., groups of apps of 3 and 5 versions, we can assess the cumulative efficiency benefits of incremental taint analysis in a continuous evolution scenario—i.e., when developers use the incremental analysis on one evolved version after another.

Moreover, for the 2-version group, we considered 100 benchmarks as 100 is a sizable scale for evaluating a static analysis for Android apps relative to peer works in the relevant literature. For the other two groups, we considered fewer benchmarks because (1) it is not as handy to get as many benchmarks each with many (continuously) evolved app versions (that can all be deobfuscated successfully and analyzable by Soot/FlowDroid) as for the 2-version group and (2) the total experimentation cost is considerably higher for each benchmark than those in the 2-version group. For apps in the 2- and 3-version groups, we were able to find and use the most adjacent version in respective release histories, whilst in the 5-version group, the versions were retrieved from their version-control/commit histories (starting from a randomly selected commit, followed by the next 4 consecutive commits) and then manually built to be the continuously-evolving versions. This approach reflects the incremental nature of app development during continuous integration/deployment practices. Tables 1, 2, and 3 summarize the 2- 3-, and 5-version group of our study benchmarks, respectively. The core of EvoTaint *itself* works at method level (although in its last phase synthesizing base and incremental taint flow results at statement level by resorting to the conventional, statement-level information flow analyzer). Therefore, this size measure will help us understand the efficiency gains EvoTaint brings. This is why for each benchmark app, the size is measured via the total number of methods in that app (rather than a more commonly used measure such as SLOC).

Based on preceding decisions we made, we formulated three app selection criteria. First, we chose F-Droid as the primary data source for as it hosts open-source apps for which we can find evolved versions that are mostly analyzable

with the tooling support (i.e., Soot+FlowDroid) available to us. This choice aligns with our target users—developers who have access to their application codebases and need to check for potential security vulnerabilities during development. F-Droid's open-source nature also allows other researchers to replicate our studies. To automate the app collection process, we selected and downloaded apps using F-Droid's repository metadata file. This file contains package name and available release versions of hosted app. From this data source, we selected benchmarks in each group randomly—for each group, we randomly sampled an app that has at least the corresponding number of versions, checking the analyzability and repeating the process until we obtain the target number of benchmarks for that group. Note that for each benchmark app, we specify the minimal version available in this file as the base version. This is because, conceptually, the minimal version is aligned with the *base version* in our technique/evaluation design for which the conventional taint analysis needs to be applied. Then, the next version is intuitively the immediate later version.

Second, if the conventional taint analysis with FlowDroid took more than 8 hours for the base version of an app and still did not finish, we also skipped that app. This decision is justified by the need to contain the total experimentation cost under a reasonable bound, and we chose 8 hours as the limit because it is a reasonably long enough time for one single app, considering the total number of apps we need to analyze in our evaluation studies.

Third, we included only apps that were either distributed as unobfuscated release APKs or could be built from source without errors. This requirement was necessary to ensure reliable analysis results, as heavily obfuscated code (e.g., with identifier renaming) can interfere with differencing.

Note that our selection methodology did not explicitly filter apps based on the frequency or magnitude of changes between versions. Change ratios were calculated post-selection rather than used as selection criteria. This approach allowed our dataset to naturally include a diverse spectrum of evolution patterns, ranging from apps with minimal changes (e.g., nearly identical) to apps with substantial differences across their versions (e.g., with 50%+ change ratio).

5.3 Methodology

Baseline selection. To establish a baseline for comparison, consider an app a with initial version v_0 (denoted as a_0) and subsequent versions $v_1, v_2, \ldots, v_n$, where v_n is the latest or the maximum version considered. v_{i+1} is the version immediately following v_i in the commit history, where $0 \leqslant i < n$. The total time for the conventional approach (running FlowDroid separately/independently on each version) is expected to be $\sum_{i=0}^{n} t_{a_i}$, with t_{a_i} representing the conventional static taint analysis time for a_i. In contrast, the total time for our approach would be $t_{a_0} + \sum_{i=0}^{n-1} t_{a_{i+1}-a_i}$, where $t_{a_{i+1}-a_i}$ includes the time for each step of our incremental taint analysis (i.e., app differencing and impact analysis, impact-guided taint checking, and taint synthesis). The primary comparison lies between $t_{a_{i+1}}$ and $t_{a_{i+1}-a_i}$.

Thus, we measured the time cost of EvoTaint versus FlowDroid as the baseline on the evolved version of each benchmark, since both approaches share the same analysis cost on the base versions. On the evolved versions, FlowDroid treats each as an independent app and performs a whole-app taint analysis, while EvoTaint performs the incremental taint analysis proposed.

Metrics. As explained above, we only compared the costs for the evolved versions incurred by between EvoTaint and FlowDroid. More specifically, we computed the percentage of cost reduction achieved by EvoTaint relative to the cost of FlowDroid on each benchmark (for answering **RQ1** and **RQ4**). For the same reason, we only compared the resulting taint flow paths in the evolved versions as produced by the two tools for accuracy validation (i.e., confirming that our incremental taint analysis produces taint flow paths that are equivalent to those that are produced by the conventional analyzer, for answering **RQ6**).

This cost reduction percentage is our primary efficiency metric (E), as computed as

$$R = \frac{(c_{baseline} - c_{EvoTaint})}{c_{baseline}} = 1 - \frac{c_{EvoTaint}}{c_{baseline}}. \quad (1)$$

where c stands for cost of time or other resources (e.g., memory).

For apps of multiple evolved versions, we compute the cost reduction for each evolved version separately, just for the apps in the 2-version benchmark group. In addition, to measure the overall cumulative efficiency benefits of iteratively applying incremental taint analysis, we also measured aggregate reduction, with which we assume that each app has $n + 1$ versions, starting with v_0 and ends with v_n. Suppose the time cost of conventional taint analysis on app of version v_i is denoted as t_{cta_i} and the time cost of incremental taint analysis on app of version v_i is denoted as t_{ita_i}. The aggregate efficiency can be computed as:

$$R_{agg} = 1 - \frac{t_{cta_0} + t_{ita_1} + \cdots + t_{ita_n}}{t_{cta_0} + t_{cta_1} + \cdots + t_{cta_n}} \quad (2)$$

Essentially, this metric measures the cumulative time saving when considering the *total* cost of analyzing *all* the $n + 1$ versions, using incremental taint analysis versus using conventional analysis.

In addition, we use the number changed methods to quantify changes between two apps of different versions, referred to as app change ratio. Suppose there are two apps a_i and a_j. If a_i has N_{a_i} methods, and the number of any category of changed (i.e., added, modified, or deleted) methods in a_j is N_{chg}, the change ratio r_{ij} is computed as:

$$r_{ij} = \frac{N_{chg}}{N_{a_i}} \quad (3)$$

Finally, to understand when our incremental taint analysis works, we examine the statistical correlation between the cost reduction it achieves and a few influence factors, including the app change ratio, the numbers of impacted sources/sinks, and the code size of apps (for answering **RQ2**, **RQ3**, and **RQ5**, respectively). In particular, to quantify the correlation statistics, we used Spearman's rank correlation coefficient ρ. Specifically, efficiency (in terms of cost reduction percentage) will be treated as the dependent variable X, whereas each of the influence factors (such as app change ratio) will be the independent variable Y. To quantitatively assess the correlation, we adopted the interpretations of correlation strength according to varied value ranges of the Spearman coefficient r in [64], as Spearman's method does not assume linear relationships between variables and is robust against outliers, making it appropriate for our data which may contain some extreme values. Specifically, the correlation is considered *very weak* if $\|r\|$ is below 0.20, *weak* if $\|r\|$ is between 0.20 and 0.39, *moderate* if $\|r\|$ is between 0.40 and 0.59, *strong* if $\|r\|$ is between 0.60 and 0.79, and *very strong* if $\|r\|$ is 0.8 or above. Moreover, we report the p-value associated with a correlation coefficient, which represents how likely the correlation is due to random chance. Typically, a p-value < 0.05 is considered statistically significant, hence indicating a significant correlation here.

Procedure. We ran our incremental taint analyzer EvoTaint versus the baseline FlowDroid separately on every considered version of each chosen benchmark app in our evaluation dataset. By comparing the performance between these two tools in terms of the above-mentioned metrics, we answer our research questions based on the performance numbers (e.g., time/memory cost reduction) collected in the experiments and statistics (e.g., correlation coefficients) further computed based on those numbers. EvoTaint was configured to collect and report extensive metadata related to the execution of the static taint analysis. We focus on several unique attributes, particularly time and space metrics, to compare our approach with the baseline. These include memory usage and time cost for function retrieval, app differencing, and impact analysis. For detailed timing statistics including those of data flow analysis, taint propagation,

and path reconstruction, we utilized the relevant capabilities offered (through its *InfoflowResults.getPerformanceData()* function) by FlowDroid. For time efficiency, we further conducted two-tailed Wilcoxon signed-rank tests [2] as this non-parametric test is appropriate when comparing paired samples without assuming normal distribution of the differences. We verified this was appropriate by first confirming our time measurement data did not follow a normal distribution using Shapiro-Wilk tests ($p < 0.001$) [61]. We choose two-tailed tests as the directionality of the difference (< 0 or > 0) is not known in advance. This was verified as we later observed that EvoTaint mostly reduces the time cost of taint analysis, but sometimes incurred higher costs, so overall whether the difference is in one direction or the other is not known. We calculated effect sizes in terms of Cliff's delta [26] to measure the statistical significance and magnitude of the differences between EvoTaint and the baseline, respectively.

5.4 Experiment Environment

Hardware specification. The experiments using EvoTaint and baseline were conducted on a machine equipped with a 64-Core AMD Ryzen Threadripper PRO 5995WX 2.70GHz processor and 512 GB DDR4 RAM.

FlowDroid configuration. For our experiments, FlowDroid was mostly set to default specifications for achieving high analysis precision—after all, a rough and cheap taint analysis producing highly imprecise results would take very little time on each app version, hence diminishing the need for incremental taint analysis. Particularly, we set *PathReconstructionMode* to $Precise$ and *PathAgnosticResults* to $false$, so that FlowDroid can produce taint flow paths with all the path element details rather than just the involved sources and sinks. Such details are desirable for many use scenarios as they allow better understanding of and diagnoses against sensitive information flows [33, 34, 51]. These details are also essential for our taint synthesis to be able to remove the flow paths in V_i that pass through any of the deleted methods relative to V_j. Additionally, *MergeDexFiles* was enabled to ensure *complete* app analysis.

6 Results and Analysis

In this section, we present and discuss our main evaluation results, answering each of our research questions (§5.1).

6.1 RQ1: What are the time efficiency advantages of incremental taint analysis compared to conventional (standard/whole- app) taint analysis?

This research question investigates whether *incremental taint analysis* offers greater efficiency compared to standard whole-app conventional taint analysis across various Android apps, in terms of the reduction of time cost.

We conducted experiments on the three sets of benchmark apps, grouping them based on the number of (2, 3, and 5) versions considered in our study as detailed earlier on evaluation dataset (§5.2). Each app underwent both *incremental taint analysis* and conventional taint analysis for a comparative evaluation, as described in our study procedure. Below, we will first separately look at the results from each of the three experiments each focusing on one set (group) of benchmarks.

Table 4 lists the (time) cost reduction (Column *Reduction*) EvoTaint achieved on each of the 2-version benchmarks (Column *Package Name*) relative to using the conventional taint analyzer. To assist with understanding the extent to which the taint analysis time was reduced, all the reduction numbers are annotated through background cell colors of varying depth. In particular, to facilitate observing the overall reduction distribution and meritorious cases, positive reductions are marked in green background. For instance, as shown in the second row and first column, for the benchmark `agrigolo.chubbyclick`, EvoTaint reduced by 75.19% (i.e., only incurred 24.81% of) the cost of the conventional

Table 4. Time cost reduction brought by EvoTaint on each of the apps in the 2-version benchmark group

Package Name	Reduction	Package Name	Reduction
agrigolo.chubbyclick	75.19%	com.github.cythara	94.68%
app.fedilab.mobilizon	83.64%	com.github.dawidd6.andttt	-172.73%
app.fedilab.openmaps	54.72%	com.github.gianlucanitti.expreval	-40.32%
app.mlauncher	-73.91%	com.github.igrmk.smsq	56.79%
at.bitfire.nophonespam	93.55%	com.github.muellerma.coffee	53.94%
at.linuxtage.companion	38.68%	com.github.muellerma.mute_reminder	58.59%
be.brunoparmentier.wifikeyshare	89.71%	com.github.muellerma.prepaidbalance	80.02%
be.ppareit.swiftp_free	28.13%	com.github.muellerma.stopwatch	99.93%
bou.amine.apps.readerforselfossv2.android	99.83%	com.github.nutomic.controldlna	96.15%
ca.mimic.apphangar	-17.06%	com.github.pires.obd.reader	45.69%
ca.rmen.android.networkmonitor	54.79%	com.gitlab.dibdib.dib2calc	45.95%
click.dummer.have_radiosion	19.51%	com.glanznig.beepme	96.62%
click.dummer.imagesms	63.64%	com.gmail.afonsotrepa.pocketgopher	95.65%
click.dummer.textthing	-37.14%	com.haringeymobile.ukweather	-14.30%
click.dummer.UartSmartwatch	22.84%	com.hwloc.lstopo	55.32%
cloud.valetudo.companion	53.57%	com.james.status	-14.29%
com.aaronhalbert.nosurfforreddit	95.65%	com.jarsilio.android.autoautorotate	65.28%
com.abh80.smartedge	-1.35%	com.jereksel.libresubstratum	84.20%
com.adam.aslfms	17.49%	com.jim.sharetocomputer	55.07%
com.addi	54.29%	com.ketanolab.nusimi	99.78%
com.akdev.nofbeventscraper	55.34%	com.knirirr.beecount	-58.81%
com.alaskalinuxuser.justcraigslist	-49.05%	com.kolserdav.ana	42.52%
com.altillimity.satpredict	92.22%	com.ktprograms.watertracker	38.57%
com.aptasystems.dicewarepasswordgenerator	-49.24%	com.launcher.silverfish	91.72%
com.benny.pxerstudio	43.57%	com.manimarank.websitemonitor	84.65%
com.bleyl.recurrence	84.38%	com.mehmetakiftutuncu.eshotroid	69.01%
com.cgogolin.library	78.80%	com.mikifus.padland	84.72%
com.chesire.nekome	39.68%	com.mrbimc.selinux	0.94%
com.concept1tech.instalate	90.70%	com.naman14.stools	98.86%
com.corvettecole.gotosleep	90.45%	com.nathaniel.motus.cavevin	52.00%
com.coste.syncorg	14.93%	com.nhellfire.kerneladiutor	69.47%
com.craigd.lmsmaterial.app	57.43%	com.nilhcem.hostseditor	43.92%
com.danefinlay.ttsutil	94.77%	com.oF2pks.jquarks	98.37%
com.danhasting.radar	67.19%	com.philliphsu.clock2	99.80%
com.danielkim.soundrecorder	28.57%	com.phpsysinfo	65.52%
com.decred.decredaddressscanner	39.70%	com.pindroid	79.11%
com.denytheflowerpot.scrunch	60.69%	com.quaap.bookymcbookface	10.35%
com.dfa.hubzilla_android	67.60%	com.rastating.droidbeard	89.69%
com.dkanada.chip	41.56%	com.sensirion.smartgadget	52.54%
com.dozingcatsoftware.bouncy	84.51%	com.servoz.appsdisabler	79.16%
com.eventyay.organizer	90.06%	com.stoptheballs	78.50%
com.example.tobiastrumm.freifunkautoconnect	-106.90%	com.tomer.draw	59.18%
com.fabienli.dokuwiki	99.11%	com.truchsess.send2car	64.31%
com.fastebro.androidrgbtool	1.85%	com.tunerly	50.20%
com.fediphoto	10.13%	com.veniosg.dir	78.19%
com.flasskamp.subz	94.56%	com.vlath.beheexplorer	37.58%
com.forrestguice.suntimescalendars	90.81%	com.workingagenda.fissure	88.89%
com.fredhappyface.fhcode	73.82%	com.chooloo.www.koler	70.83%
com.freshollie.monkeyboard.keystoneradio	95.53%	com.github.muellerma.mute_reminder	99.82%
com.gbeatty.arxiv	39.13%	de.lukasneugebauer.nextcloudcookbook	84.73%

taint analysis on the (only one) evolved version considered. To facilitate identifying cases in which EvoTaint did not help (i.e., incurring an even higher cost on the evolved version with incremental analysis versus running the chosen conventional taint analysis on that version), negative reductions are marked in red background. For instance, as shown in the third row and fourth column, for the benchmark `com.github.dawidd6.andttt`, EvoTaint reduced the conventional taint analysis cost on the evolved version of this app by -172.73%—i.e., the incremental taint analysis took 72.73% longer time than the conventional analyzer. In both the positive and negative reduction cases, the absolute values of the reduction are encoded through the respective depth of the annotating background color—the larger the value, the greater the color depth.

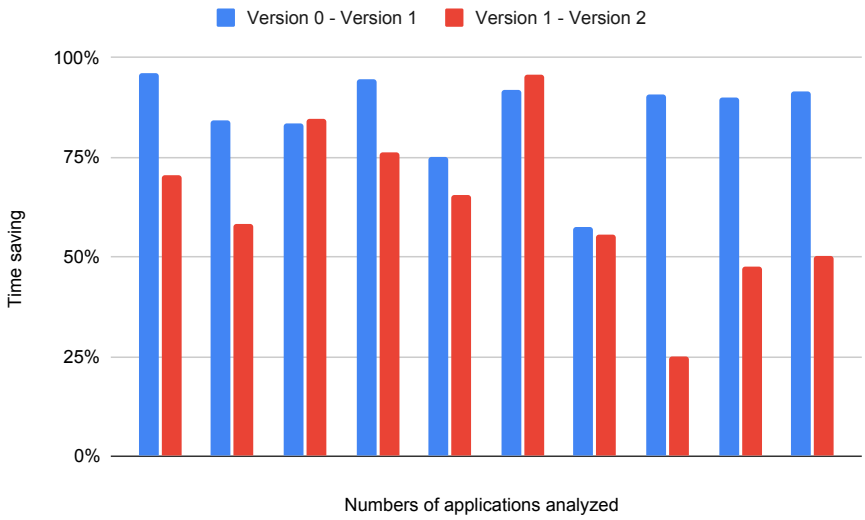

Fig. 5. Cost reduction by performing *incremental taint analysis* over conventional taint analysis on each evolved version of apps in the 3-version benchmark group.

Overall, EvoTaint exhibited its merits through incrementally computing taint flows through the reduction of time cost that would otherwise be incurred using the conventional approach treating each app version independently. Among the 100 cases, EvoTaint was able to save that time cost in 88 cases, with the reduction ranging from merely 1% to as high as 99.8%. In the remaining 12 cases, incremental taint analysis with EvoTaint was not worth it—using the conventional approach would be more efficient. Accounting for all of these 100 cases, the mean time cost reduction EvoTaint achieved was 51.8%, a significant saving of time.

For the 3-version benchmark group, Figure 5 shows the cost reduction EvoTaint achieved against each evolved version treated independently (as opposed to cumulative efficiency merits). In particular, the cost reduction on the first evolved version is indicated by bars labeled as *Version 0 - Version 1*, while the reduction on the second evolved version is indicated by bars labeled as *Version 1 - Version 2*. In most of the cases, cost reduction on the first evolved version appeared to the greater. Our inspection revealed that this was simply due to the respective apps were chosen such that the further evolution introduced greater and/or more complex changes relative to the first evolved version than the changes made in the first evolved version relative to the base version. Overall, the cost reduction is 68.9% on average.

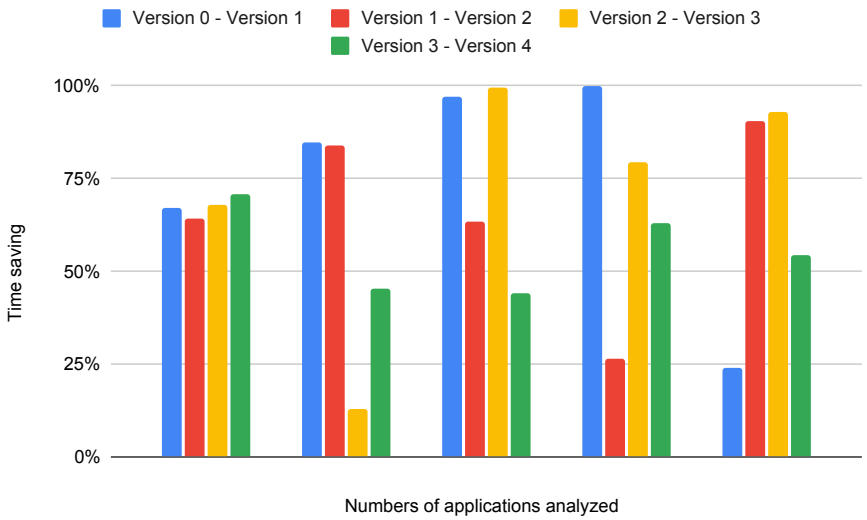

Fig. 6. Cost reduction by performing *incremental taint analysis* over conventional taint analysis on each evolved version of apps in the 5-version benchmark group.

Similarly, for the 5-version benchmark group, Figure 6 shows the cost reduction EvoTaint achieved against each evolved version, in the same format as Figure 5. The results confirmed that the cost reduction on each evolved version varied without a clear/consistent pattern, which is intuitively justified by the fact that the size and complexity of code changes made in each evolved version do not necessarily follow a consistent pattern. The average cost reduction for this benchmark group is 66.9%.

Note that the cost reduction was generally higher in absolute terms in the 3- and 5-version benchmark groups than in the 2-version group. In fact, there were no negative reduction at all in the 3- and 5-version benchmark groups. This greater efficiency for the benchmark groups of more versions considered can be attributed to the closer proximity between adjacent versions in those groups (each version corresponds to a release from the app's version history based on commits) than in the 2-version group (each version corresponds to a major/stable release). Table 5 presents descriptive statistics comparing conventional taint analysis and EvoTaint across our benchmark groups and the statistical significance test results, i.e., the statistical significance and effect size for the differences between the baseline and EvoTaint in terms of time cost for all the benchmarks. The 3,5-version group results are presented together because we combined these data points for statistical analysis due to: (1) their similar characteristics in terms of representing continuous software evolution scenarios with multiple (more than the minimum of two) versions, (2) the relatively limited sample sizes of each group individually (20 data points each), and (3) their consistent performance patterns as observed in our initial analysis—they both have greater time reduction (from our incremental taint analysis) compared to 2-version group and have no negative time reduction as seen by the 2-version group. As shown in Table 5, substantial differences in central tendency measurements highlight EvoTaint's efficiency improvements. For example, for the 2-version group, EvoTaint reduced the mean analysis time from 3,706s to 1,299s (65% reduction) and the median from 1,089s to 229s (79% reduction). The difference is even more pronounced in the 3,5-version group, with mean times decreasing from 1,139s to 315s (72% reduction) and median times from 696s to 135s (81% reduction). EvoTaint also

shows substantially lower standard deviations, indicating more consistent/stable performance across different apps compared to the conventional taint analysis.

For significance tests, we used a significance level of $\alpha = 0.05$. For the 2-version benchmark group (n=100), the two-tailed Wilcoxon signed-rank test yielded a p-value of 4.56×10^{-13}, which is substantially below our significance threshold ($p < 0.05$), allowing us to reject the null hypothesis that there is no difference between the performance of EvoTaint and conventional taint analysis. The effect size (Cliff's delta) of 0.724 indicates a statistically large difference according to the guidelines proposed by [57], where $|d| > 0.474$ is considered large. Similarly, for the combined 3,5-version group (n=40), we obtained a p-value of 1.82×10^{-12} and an effect size of 0.871, which also represents a significant and large difference. These results provide strong statistical evidence that EvoTaint consistently outperforms conventional taint analysis in different app evolution scenarios.

Table 5. Descriptive statistics and statistical significance tests comparing EvoTaint and baseline.

Metric	Mean		Median		Standard Deviation		p-value	effect size
	Conventional	EvoTaint	Conventional	EvoTaint	Conventional	EvoTaint		
2-version group	3706	1,299	1,089	229	6,359	3065	4.56×10^{-13}	0.724
3,5-version group	1,139	315	696	135	1,240	452	1.82×10^{-12}	0.871

Measuring the efficiency benefits of incremental taint analysis in terms of the *aggregated reduction* metric, Figures 7 and 8 depict the results for the 3- and 5-version benchmark group, respectively, in the same format as Figures 5 and 6. With this measure, the merits of incremental taint analysis with EvoTaint are apparently much more pronounced when perceived iteratively over the evolution history than considering each evolved version separately. This is why there are much more cases here (compared to what Figures 5 and 6 show) in which the later evolved versions saw greater (aggregated) cost reduction than the earlier versions. On the other hand, in terms of absolute cost reduction numbers, the efficiency gains (53.2% and 51.3% cost reduction on average for the 3- and 5-version group, respectively) seem to be even smaller than when considering the cost reduction on each evolved version separately. This is due to the way we calculate the aggregated cost reduction—the numerator (i.e., total cost of analyzing all of the app versions considered) includes the base version's cost incurred by the conventional taint analysis (see Eq. 2): oftentimes, the base cost (conventional analysis of v_0) significantly outweighs the incremental analysis cost, resulting in larger numerators hence smaller aggregate cost reduction.

Answer to RQ1: *Overall, incremental taint analysis with EvoTaint can bring substantial time efficiency in terms of cost reduction, either in individual evolved version treated separately (on average 51.8–68.9% cost reduction) or when considering the total cost of analyzing all the app versions during the evolution history considered (on average 51.3–53.2% cost reduction). Generally, cost reduction can vary widely across different apps and different evolved versions of the same app.*

6.2 RQ2: Under what conditions does incremental taint analysis demonstrate its time efficiency advantages?

This research question examines the conditions under which (i.e., use scenarios where) *incremental taint analysis* with EvoTaint is meritorious (i.e., leading to positive cost reduction hence being worthwhile). An underlying premise here is that the efficiency of EvoTaint is relatively greater when changes between two versions of an Android app are relatively smaller. Thus, an intuitive route to investigating the conditions is to examine *app change ratio* (as defined earlier in §5.3) as a main factor that is presumably correlated with the cost reduction that may be achieved by EvoTaint. After all, change ratio is exactly a metric we proposed to quantify the extent of changes between two apps (or app versions). Note

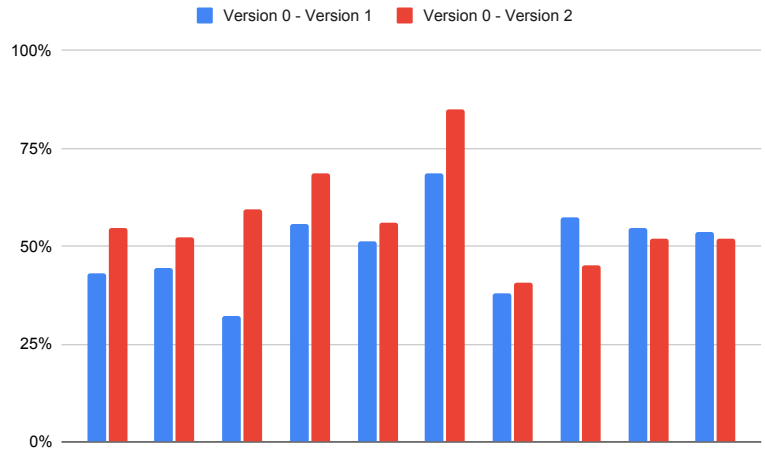

Fig. 7. Aggregate (cost) reduction by performing *incremental taint analysis* over conventional taint analysis on each evolved version of apps in the 3-version benchmark group.

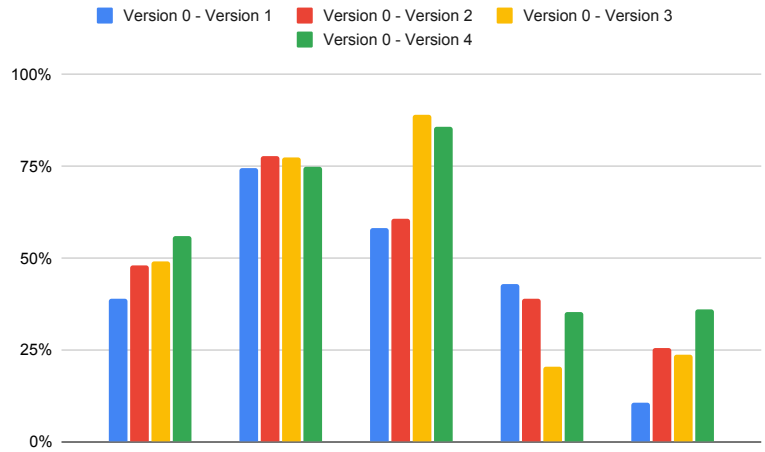

Fig. 8. Aggregate (cost) reduction by performing *incremental taint analysis* over conventional taint analysis on each evolved version of apps in the 5-version benchmark group.

that at its core, EvoTaint's effectiveness is determined by how code changes affect the reachability between sources and sinks, rather than by the semantic nature of the changes themselves. Whether a change involves UI modifications, backend logic updates, or API changes, our approach considers it relevant only if it impacts source/sink reachability. For instance, UI changes that only modify layout parameters or visual elements without affecting information flow paths would not impact our analysis. However, if a UI change involves modifying how user input is collected or displayed (affecting sources or sinks), our impact analysis would identify and analyze these changes. Therefore, we did not consider how semantic categorization of changes affects EvoTaint's effectiveness.

Figure 9 illustrates the (1) change ratio (the chart on the top) of each app in the two-version benchmark group, represented by stacked bars that each contains three different kinds of ratio of change (i.e., percentage of methods

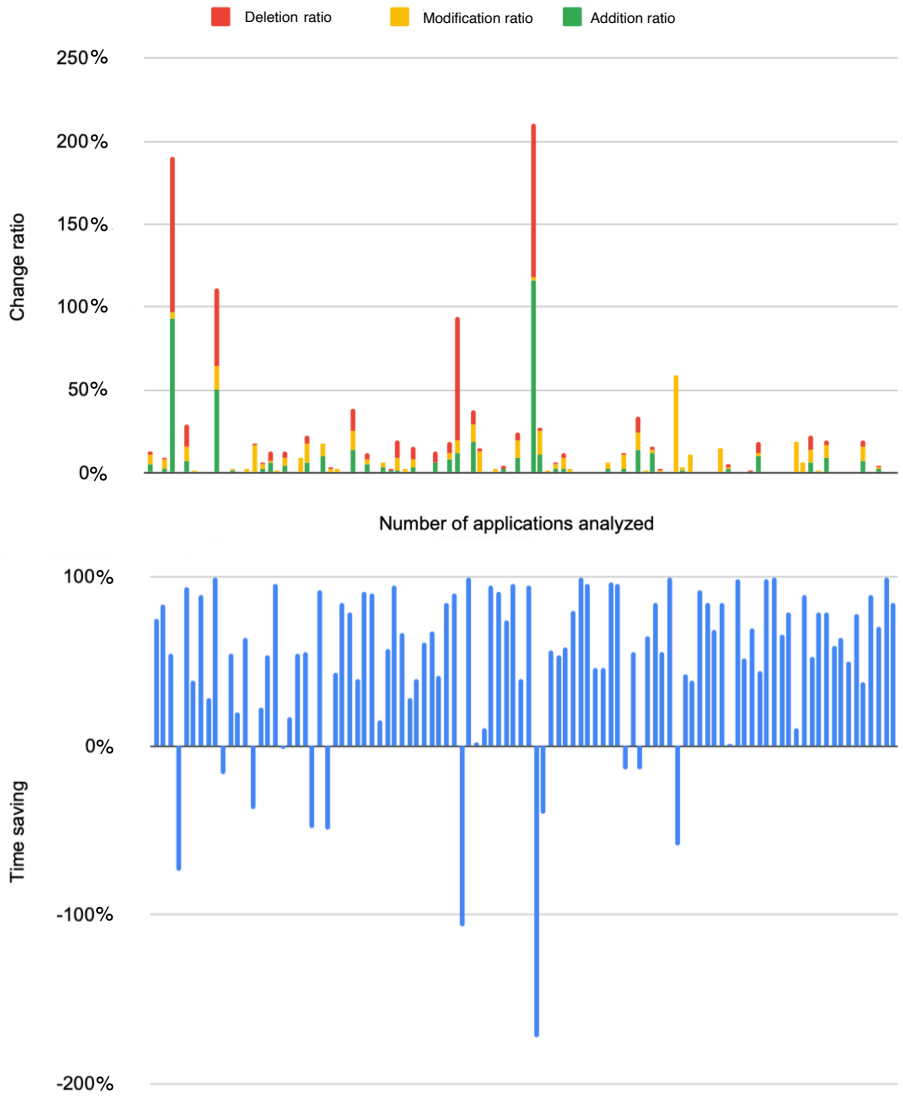

Fig. 9. Change ratio (of the three kinds of changes, in the top chart) between the two app versions versus cost reduction achieved by EvoTaint (in the bottom chart) for each app in the 2-version benchmark group.

deleted, modified, and added, noted as *Deletion ratio*, *Modification ratio*, and *Addition ratio*, respectively), and (2) time savings (i.e., cost reduction, in the chart on the bottom) achieved by EvoTaint on each app in the benchmark group. Note that both charts share the same x-axis, which indicates the individual apps while linking the stacked bar (i.e., change ratios) for each app to the respective blue bar (i.e., time savings) for that same app. For example, the (total) change ratio of the 1st app is approximately 13%, and if we need to check its corresponding time saving, we can check the 1st column in the chart below, which is approximately 75%. In all, the figure visualizes how the efficiency merits achieved by EvoTaint on an evolved version of a benchmark Android app are related to the extent to which the evolved

Table 6. Correlations between app (overall and different kinds of) change ratio and EvoTaint's efficiency gains (time saving).

Independent Variable	Dependent Variable	Coefficient $\|\|r\|\|$	p-value
Overall ratio of (all) changes	Time Saving	**-0.627**	<.001
Ratio of additions	Time Saving	-0.461	<.001
Ratio of modifications	Time Saving	**-0.599**	<.001
Ratio of deletions	Time Saving	-0.453	<.001

version has changed relative to the base version. Some of the bars are hardly visible in this visualization, indicating that the associated change ratio or time saving is unnoticeable.

Putting the two charts in contrast, we can readily observe a clearly negative correlation between change ratio and time saving for these benchmark apps: i.e., EvoTaint tends to save more static taint analysis time when the overall change ratio is lower. In particular, EvoTaint often achieved the highest (nearly 100%) cost reduction when the ratio of change was almost negligible—mostly, the time saving was positive when the change ratio was under 50%. On the flip side, for the apps that were changed substantially (e.g., almost or even over 100%), the cost reduction was negative—e.g., at the greatest change ratio (over 200%), the time saving was the lowest (-180%). This rough observation preliminarily supports the general premise that our incremental taint analysis is worthwhile when relatively modest changes are made between the two app versions considered by the analysis.

There did not seem to be clear/consistent relationship between ratios of individual kinds of changes between the analyzed app versions and the time savings achieved by EvoTaint, though. Even when looking at the different kinds of changes separately, the ratio of the changes still mattered much more than what kinds they are (among deletion, modification, and addition). A side observation is that at least for these 100 studied apps, code changes made during their evolution were rarely concentrated on one kind (e.g., deletion or addition only). In fact, in most of the cases in which the total change ratio was noticeable, all of the three kinds of changes were involved in the evolved version of the apps relative to their previous versions.

To have a closer and quantitative look into the above-mentioned correlation/relationships, we calculated the Spearman's rank correlation coefficient between the change ratio and time saving, which is shown in Table 6. Overall, these numbers corroborate our previous visual observations: the correlations are all clearly negative and the ratio of changes matter more than the kinds of changes. The p values are consistently much lower than the significance level (0.05), indicating that all of the correlations are statistically significant. More specifically, the correlation between the overall change ratio and time saving is strong (coefficient of -0.627), which suggests that as overall change ratio increases, the time saving can be generally expected to decrease as much. For the ratio of individual kind of change, which is listed from the second row to the fourth row of the table, we can see that they are relatively less impactful on the efficiency gain achieved by our incremental taint analysis.

In particular, the gain only has moderate correlations with the ratios of addition and deletion of methods (with a coefficient of -0.461 and -0.453, respectively) between the two apps/versions that the analysis is concerned with. By contrast, modifications of existing methods (in the base app/version) have notably stronger impact on the efficiency gain, as indicated by the nearly strong correlation (with a coefficient of -0.599) between this kind of change and the time saving. In other words, when looking beyond the overall change ratio into specific kinds of changes, how many merits our incremental taint analysis approach has is primarily tied to how many methods are modified, as opposed to how many of the existing methods are removed or how many new methods are added.

On the other hand, under the hood of these changes, what really matters ultimately is how many sources and sinks the changes would impact, since our techniques works by reducing the analysis cost induced by searching taint flow between relevant sources and sinks. Therefore, next we dive even deeper into the merit conditions of our incremental

taint analysis approach with EvoTaint by examining how the number of sources/sinks relates to the time cost of static taint analysis on the evolved apps that EvoTaint can reduce.

> **Answer to RQ2:** *Incremental taint analysis with EvoTaint tends to be meritorious when the changes made in the evolved apps are relatively moderate (e.g., lower than 50%). Meanwhile, the overall change ratio is strongly negatively correlated with the cost reduction EvoTaint achieves; also, among the three kinds of changes, modifications tend to have a greater impact on the cost reduction than additions and deletions.*

6.3 RQ3: How does the number of impacted sources and sinks influence the time efficiency of incremental taint analysis?

This research question examines how the number of impacted sources and sinks influences the efficiency of our *incremental taint analysis*. As noted earlier, our approach achieves savings in the cost of analyzing the evolved apps mainly by reducing the scope of sources/sinks between which taint flows need to be computed using a conventional taint analyzer backed by statement-level data flow analysis (i.e., FlowDroid [8]). Thus, examining the influence of impacted sources/sinks is a natural means of looking more closely into the mechanisms of EvoTaint's cost reduction.

Figure 10 correlates cost reduction (i.e., time savings) accomplished by EvoTaint with the source/sink scope reduction (i.e., sources and sinks savings) it achieved. The bars in red represent the percentages of total sources and sinks are reduced when performing *incremental taint analysis* compared to conventional whole-app taint analysis, whereas the bars in blue represent corresponding time saving. Since the total number of sources and sinks in *incremental taint analysis* can never surpass that in the conventional whole-app taint analysis, the source/sink saving is always non-negative. Therefore, if there is a positive time saving for an app, the two quantities (source/sink saving and time saving) are depicted as a stacked bar for that app.

For the same 12 cases (app) as shown in Table 4 and Figure 9 for which the time saving was negative, the source/sink savings were generally the lowest among the benchmark group. In those cases, the source/sink scope of the expensive data-flow analysis (at the core of the static taint analysis) was not much reduced, making the cost reduction by our incremental taint analysis negative—as the extra cost for identifying the impacted sources/sinks outweighs the slightly reduced cost for computing the taint flows between the sources and sinks (which are only slightly fewer than those fed to the conventional analysis). Yet otherwise, where the source/sink saving was greater (smaller), the time saving was generally greater (smaller) as well. Thus, overall, the source/sink saving and time saving are (positively) correlated, as expected per the inner workings of our technique.

To quantify this visually perceived correlation relationship between time saving and the reduction of sources and sinks, we calculated the Spearman correlation coefficient between these two quantities, as shown in Table 7.

Table 7. Correlations between the reduction of # sources/sinks (*sources/sinks saving*) and reduction of analysis cost (*time savings*)

Independent Variable	Dependent Variable	Coefficient $\|r\|$	p-value
Sources and Sinks Saving	Time Saving	0.642	<.001
Sources Saving	Time Saving	0.502	<.001
Sinks Saving	Time Saving	0.673	<.001
Overall change ratio	Sources and Sinks Saving	-0.801	<.001

From the first row of Table 7, we can observe that the time saving and sources and sinks saving has a moderately strong positive correlation with a Spearman coefficient 0.642, which suggests that generally with fewer impacted sources and sinks, the more analysis time cost can be saved in the *incremental taint analysis*. Noted that in the second

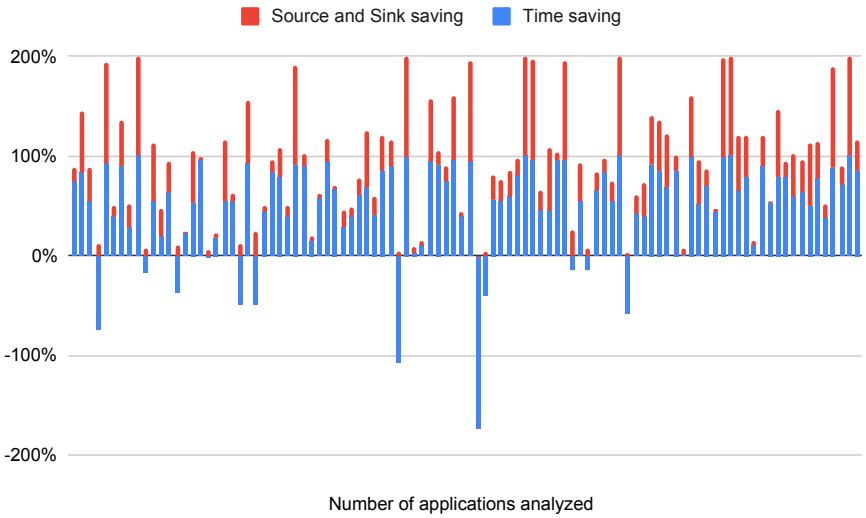

Fig. 10. Source/sink saving (percentage of total sources and sinks reduced) versus time saving (cost reduction) achieved by EvoTaint for each app in the 2-version benchmark group.

row and the third row, we also broke down the sources and sinks saving in total to sources saving and sinks saving separately, and we found that the number of impacted sinks has a more significant correlation with the time saving, with Spearman coefficient being 0.673, versus the lower strength of correlation of the impacted sources saving with the time saving (0.502).

As shown in the fourth row in Table 7, we also computed the correlation between overall change ratio and sources and sinks saving. As a result, they exhibit a strong negative correlation with a Spearman coefficient −0.801, which implies impacted sources and sinks are directly induced by the change ration between the given two app versions. Since we have already revealed the role the number of impacted sources and sinks plays, it is beneficial to explore how change ratio is inextricably intertwined with the number of impacted sources and sinks. This strong statistical connection between sources/sinks saving and overall change ratio explains our earlier observation about the correlation between the (method-level) code change ratio and taint analysis cost reduction brought by EvoTaint. It also consolidates our earlier conclusion that under the hood of the impact of change ratio on the cost reduction is essentially the impact of sources/sinks saving.

Answer to RQ3: *The percentage of sources and sinks (considered in the taint analysis) that is saved by EvoTaint is strongly correlated with the reduction of analysis time it can bring, with the impact of saving in the number of sinks being greater than that of sources. The very-strong correlation between the change ratio and the total sources and sinks saving justifies/explains the earlier observed impact of the change ratio on the time cost reduction.*

6.4 RQ4: How does app size influence the time efficiency of incremental taint analysis?

Considering how it works in terms of the technical nature, EvoTaint's time efficiency is intuitively clearly influenced by how much of the code in the base version of an app is changed in the evolved version (i.e., measured by the overall change ratio). Yet the influence does not separately account for the total size of the code, both of the base and evolved

versions. Intuitively, per its definition in our study (Eq. 3), the app change ratio has the base app version's size in its calculation; yet that size is measured in terms of the number of methods. How the app size in terms of the number of Source Lines Of Code (SLOC), for both the base and evolved versions, affects the time saving that our incremental taint analysis may bring about remains to be examined.

This research question investigates the influence of app size in terms of SLOC on the time efficiency of EvoTaint, from the perspective of statistical correlation analyses. We hypothesized that the efficiency of *incremental taint analysis* might be influenced by the complexity of the app, with complexity in this context being quantified as SLOC. To assess this hypothesis, we again calculated the Spearman correlation coefficient between SLOC and time savings. Again, as the 3- and 5-version benchmark groups do not have enough apps to carry out a meaningful statistical analysis, this correlation analysis here is still performed against the 2-version benchmark group.

Table 8. Correlation between the app size (of both the base and evolved versions) and reduction of analysis cost (time savings)

Independent Variable	Dependent Variable	Coefficient $\|\|r\|\|$	p-value
Base App Source Lines Of Code	Time saving	0.002	<.001
Evolved App Source Lines Of Code	Time saving	0.002	<.001

As shown in Table 8, the correlation between the time saving achieved by EvoTaint and the app size, with respect to either (base or evolved) version of the app, is so weak (the Spearman coefficient value being 0.002) that we should consider it essentially none.

Contrary to a possible hypothesis, which might suggest that larger apps (with greater SLOCs) would result in more methods in total, at least in the base versions, hence potentially smaller overall app change ratio (Eq. 3)—then further being connected to greater time cost reductions (through the strong correlation between time saving and overall app change ratio as we observed earlier). Or, a greater total number of methods, as a possible result of greater SLOCs, may imply more sources and sinks in total invoked in the app, hence potentially smaller percentages of impacted sources and sinks—which then also is connected to greater time cost reductions (through the strong correlation between time saving and sources/sinks saving). However, our results here do not support any of these hypotheses. In fact, even apps with relatively smaller SLOCs exhibited a significant number of impacted sources and sinks between the base and evolved versions in our benchmarks.

Answer to RQ4: *There is essentially no correlation between app size (in terms of SLOC) and the time efficiency of our proposed incremental taint analysis, making the sheer code size of apps a largely irrelevant factor in the merit conditions of EvoTaint.*

6.5 RQ5: Does our incremental taint analysis lead to reductions in memory usage?

In RQ1, we examined the *time efficiency* of EvoTaint in terms of how much time cost reduction it can bring by computing taint flows incrementally. Taking a step further, this research question explores whether our *incremental taint analysis* leads to reductions and, if any, how much, in space usage in terms of memory consumption. Specifically, we tracked the peak memory usage during each run of EvoTaint and the conventional taint analysis baseline. The results on these space usages along with how they are related to the app change ratio for the 2-version benchmark group are depicted in Figure 11. Specifically, in Figure 11, the top chart shows the total and dissected change ratio between the two versions of each app in the 2-version benchmark, identical to its counterpart in Figure 9. Juxtaposed to

this chart is the bottom chart, which shows the memory saving (i.e., percentage of peak memory consumption reduced) brought by EvoTaint for each of the same suite of apps.

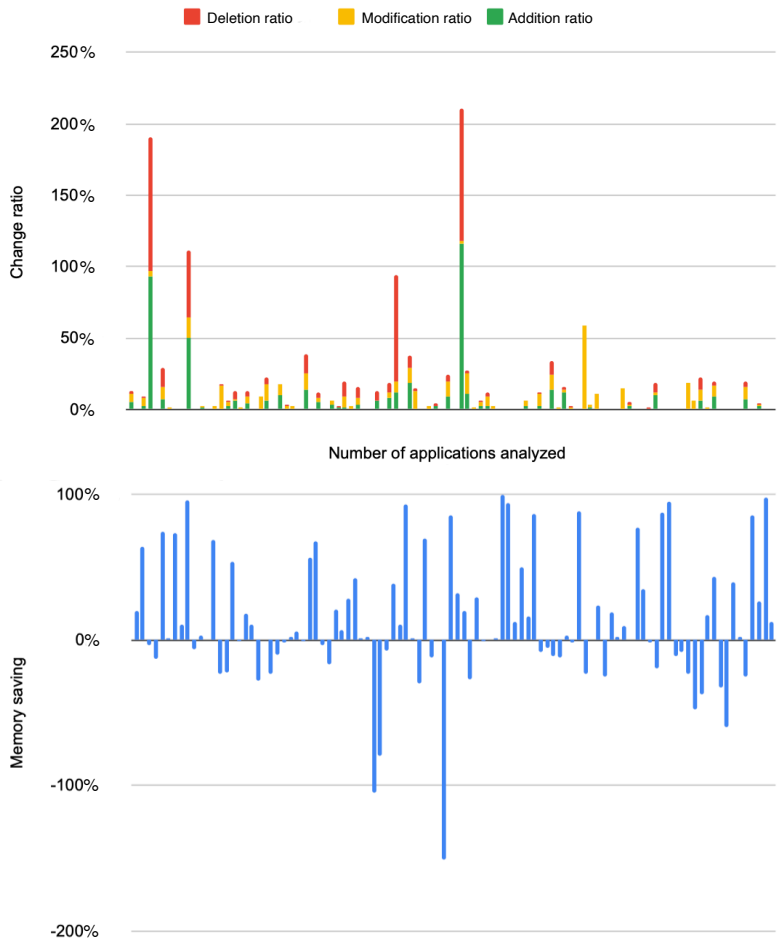

Fig. 11. Space efficiency in terms of the percentage of peak memory consumption reduced (i.e., *memory saving*, shown in the bottom chart) by EvoTaint versus the app *change ratio* between the two versions for each app in the 2-version benchmark group.

Table 9. Correlation between app (overall and different kinds of) change ratio and EvoTaint's space efficiency gains (memory saving)

Independent Variable	Dependent Variable	Coefficient $\|\|r\|\|$	p-value
Overall change ratio	Memory Saving	-0.311	0.002
Ratio of additions	Memory Saving	-0.202	0.438
Ratio of modifications	Memory Saving	-0.254	0.107
Ratio of deletions	Memory Saving	-0.242	0.153

First of all, the memory saving results reveal that our incremental taint analysis brought memory efficiency gains, in addition to time efficiency merits: for the majority (65%) of the 100 benchmarks in this 2-version benchmark group, the memory time savings were positive. On the other (35%) of the benchmarks, EvoTaint consumed higher peak memory

than the conventional analysis baseline. In contrast, EvoTaint achieved positive time savings for 88% of these same benchmarks (Table 4).

Second, putting the two charts in contrast, we observe that, unlike time savings, memory usage savings exhibited more variations and did not show as clear and consistent correlation with the change ratio. To quantify this observation, we calculated the Spearman correlation coefficient as well, detailed in Table 9. The results indicated a far weaker correlation between memory savings and change ratio, as evidenced by smaller absolute values of coefficients, which ranges from −0.311 to −0.202. In addition, the much higher p-values (up to 0.438) indicate that memory savings are not significantly correlated with the ratio of any individual type of changes, although the correlation with the overall change ratio is statistically significant.

These findings suggest that memory savings are not as directly correlated with *incremental taint analysis* efficiency as time savings are. Therefore, while *incremental taint analysis* can offer significant time savings, it does not as consistently lead to reduced memory usage. To explain this, we can think of the ingredients of memory cost of *incremental taint analysis*. Apart from executing FlowDroid, *incremental taint analysis* also involves other steps beforehand such as collecting methods from the app and change impact analysis. All these parts will also consume memory, which might exhaust more than the memory cost reduction benefited from having to analyze fewer sources and sinks.

Answer to RQ5: *While EvoTaint reduced the peak memory consumption of static taint analysis by doing it incrementally for most (65%) of the benchmarks, the memory saving benefits are not as commonly seen as for the time savings (in 88% of the benchmarks). Moreover, there is only a weak correlation between the overall app change ratio and these memory savings. Such correlations are insignificant with the ratio of individual types of changes.*

6.6 RQ6: Does our incremental taint analysis achieve the same accuracy in taint checking as the conventional approach?

Per the motivation and design goal of EvoTaint, it should not compromise the accuracy of static taint analysis while aiming to improve efficiency for analyzing evolved versions of Android apps. Now that the efficiency gains are evaluated with respect to the conventional taint analyzer FlowDroid, a natural way to evaluate the accuracy of EvoTaint is to validate the consistency between these two analyzers in terms of their final sets of flow paths produced (given the same source/sink setting and on the same apps).

Intuitively, per our technical design, EvoTaint should produce the same flow paths (out of the final step of taint synthesis) as FlowDroid does against each evolved version of each benchmark we considered—for the very first/base version of the app, EvoTaint simply runs FlowDroid on it. To empirically validate this assumption, on each of the (1, 2, and 4) evolved versions of each app in our (2-, 3-, and 5-version, respectively) benchmark groups, we compare the set of final taint-flow paths produced from EvoTaint with that from running FlowDroid against the same evolved version treated as a standalone app. We found no difference between the two analyzers on any of our benchmarks, suggesting that EvoTaint does not compromise the accuracy of FlowDroid when improving its efficiency.

Answer to RQ6: *EvoTaint produced the same taint analysis results (flow paths) as the conventional, standalone taint analyzer for all of our benchmarks, hence achieving the same level of accuracy of analysis results. That is, EvoTaint does not compromise taint analysis accuracy while achieving efficiency gains.*

7 Discussion

7.1 Generalizability

While EvoTaint's current implementation focuses on Android apps, the core methodology of incremental taint analysis is platform-independent and can be readily adapted to other software systems and platforms.

The fundamental components of our approach—impact analysis, impact-guided taint checking, and taint synthesis—are designed to work with any software system where static taint analysis is applicable. The key insight of reducing analysis scope based on code changes and their impact on sources/sinks remains valid regardless of the underlying platform. To adapt EvoTaint to a different platform, one would primarily need to take two steps. First, they replace FlowDroid with an appropriate static analyzer for the target platform and programming language. The core algorithms of EvoTaint would remain unchanged—only the interface with the underlying analyzer would need adaptation. Second, they adjust the call graph construction mechanism according to the target platform's characteristics. Our current implementation uses Android-specific call graph construction through FlowDroid, but this component can be substituted with appropriate alternatives like LLVM's [47] call graph analysis for C/C++ programs.

7.2 Applicability

From the perspective of practical usefulness, users of EvoTaint would benefit from knowing in advance whether applying an incremental taint analysis like what we propose on a given app is worthwhile. As we examined earlier, an obvious factor is the amount of code change that has occurred between the base and evolved version of the app, as measured as app change ratio in our study. We have studied the relationship between this merit condition factor and EvoTaint's time efficiency.

In our specific 2-version dataset, only 12 out of 100 showed increased time costs with EvoTaint compared to conventional analysis. Through careful examination of these cases, we identified several meaningful thresholds. For evolved apps with changes (addition, modification, and deletion aggregated) below 1.63%, the incremental taint analysis consistently showed positive efficiency gains, making EvoTaint a safe choice in these scenarios. The risk of increased costs begins to appear with changes between 1.63% and 16%, though most cases in this range still benefited from incremental taint analysis. The risk increases substantially when changes exceed 16%, with the median change ratio for cost-increasing cases being 30.52%. Cases with changes above 58% almost consistently showed increased time costs. Based on these findings, we can provide developers with practical guidelines: EvoTaint is most reliably beneficial when changes are below 16% of methods, should be used with careful monitoring for changes between 16-30%, and may not be advantageous for changes exceeding 50%. However, we maintain our earlier observation that these thresholds should be considered guidelines rather than strict rules, as the specific context of changes (particularly their impact on source/sink reachability) remains important. The more reliable indicator we found was the percentage of impacted sources and sinks, which showed a stronger correlation with time savings (as detailed in Section 6.3). However, this metric is not readily available to developers before running the analysis.

To better understand this relationship in a more tangible manner, we have picked a few concrete cases that appear to be outliers in our evaluation benchmarks for case studies. They represents two opposite ends in the spectrum of changes, categorized as two scenarios, which may help us reveal what is under the mask.

- **Scenario 1**: There are essentially no code changes between the two app versions; hence the overall app change ratio and the change ratio of any of the three types of changes are all zero.

- **Scenario 2**: There are massive code differences between the two app versions; hence a very high overall app change ratio.

Scenario 1 may be exemplified by the development scenario in which developers only modified the assets of an app but did not alter the code (e.g., likely between two adjacent commits or releases). In our 2-version benchmark group, we picked two apps `bou.amine.apps.readerforselfossv2.android` and `com.fabienli.dokuwiki`, which have no code changes between the two versions of their apps. Without surprises, their time savings are extremely high (99.11% and 99.93%, respectively)—i.e., analyzing the evolved version takes an almost negligible amount of time.

Scenario 2 might seem too unusual when considering what *adjacent* versions of an app during its evolution should look like. In our 2-version benchmark group, we found four apps falling in this scenario, at least with respect to the base and evolved versions we considered. These cases can also be noticed from Figure 9, where there are four stacked bars that are apparently higher than others. After looking into these four cases, we found that they are heavily obfuscated; as a result, despite our deobfuscation steps, the remaining unresolved obfuscation still substantially impeded our app differencing, making it detect excessive code changes. The ratio of changes is so high that the incremental taint analysis ended with taking more time than a conventional approach. There are two points that we would like to make here. First, as discussed earlier, obfuscation is widely present in real-world apps after they have been distributed to the app market, yet developers are not expected to obfuscate their apps during the development while using EvoTaint to check if the incremental changes they make introduce more sensitive taint flows in the apps—the primary use scenario targeted by our technique. Second, such heavy obfuscation cases can be viewed equivalently to those in which the two versions of an app are indeed enormously different, similar to the cases in which one may apply incremental taint analysis to two arbitrary, unrelated apps.

By studying these two extreme scenarios discussed above, we reach a further consolidated understanding about the *merit conditions* of incremental taint analysis in terms of the relationship between the portion of code that differs the two app versions during the incremental changes and the merit of the incremental analysis in terms of its time efficiency gains. Looking *at* these extremities, when the changes are minimum, the time saving can be gratifyingly huge, whereas when the changes are drastic, it can incur additional time expenditure which makes the choice of analyzing taint flows incrementally a penalizing rather than rewarding one. Looking *between* these extremities, the efficiency of *incremental taint analysis* on two given app versions is closely tied to the specific extent of code changes between the two versions. As the version gap widens, an increase in method change ratio can be expected, potentially lengthening the analysis time of EvoTaint itself.

On the other hand, the selection of version intervals is critical in deciding whether incremental taint analysis should be a good choice and, if chosen, deciding which two specific app versions to apply the analysis to. Moreover, as observed from our evaluation results on the 2-version benchmark group versus the 3- and 5-version benchmark groups, the length of the evolution history during which the incremental taint analysis is applied consistently also matters: generally, the longer the history (meaning more evolved versions the analysis is applied to in a continuous fashion), the greater the overall merits of the incremental approach to the static taint analysis.

These general interplay between the app change ratio and EvoTaint's efficiency have been quantitatively characterized through the statistical (correlation) analysis we have performed. However, due to the complexity of different Android apps and that of the code changes themselves, it is almost impossible or even misleading to assume or discover a rigorous quantitative relation between the code change ratio and efficiency of our approach (e.g., through a mathematical function). After all, the change ratio only measures the portion of methods changed, rather than that of the

statements changed, let alone the complexity of the changes (at either the method or statement level). In fact, results of our correlation analysis show the Spearman correlation coefficient is slightly higher than 0.6, i.e., to the extent of moderately strong but not yet to the extent of very strong. In other words, there is *no guarantee* that when the app change ratio gets lower, *incremental taint analysis* will definitely save more in the time cost of static taint analysis. For instance, in some cases, the change ratio was only modest (below 50%), yet the incremental taint analysis achieved no or even negative efficiency gains. In fact, there were a couple of cases in which the cost reduction was negative even though the change ratio was less than 25%. Moreover, the time saving was not always reversely proportional to the change ratio: when the ratio was lower (higher), the cost reduction was not necessarily *proportionally* higher (lower).

This is why, to more thoroughly understand the relationship between the change ratios and time savings, we went further to look into the underlying factor that actually causes the impact of change ratio on the efficiency of our incremental taint analysis. Recall that the key insight that enables EvoTaint's cost reduction in analyzing the evolved version of an app lies in the reduced analysis cost due to the reduced number of sources and sinks to be considered by the analysis—we do not directly improve/optimize the statement-level data flow analysis algorithm itself at the core of the static taint analysis. Thus, intuitively, underneath the ratio of code changes, what really matters for EvoTaint is how many sources and sinks can be reduced. Furthermore, considering that the source/sink reduction results from the change impact analysis and impact-guided taint checking steps of EvoTaint, what ultimately matters is the numbers of sources/sinks impacted by the code changes. For instance, in the scenarios where the change ratio is high, EvoTaint may still achieve great cost reductions if the change only impacts very few sources/sinks; while the cost reduction may be little even when the change ratio is not that high but the change impacts many sources/sinks.

7.3 Robustness to Android Platform Evolution

The core challenge raised by Android evolution primarily affects source/sink identification, which is orthogonal to our incremental analysis approach. EvoTaint's methodology focuses on solving the fundamental problem of incremental taint analysis—how to efficiently analyze information flows between any given set of sources and sinks. The actual identification of these sources and sinks is handled separately through a tool like SuSi [5], as mentioned in Section 4.

When Android introduces new APIs or security features that generally affect information flow analysis, EvoTaint can be adapted to them without specific modification in itself. Here we discuss four aspects related to Android evolution: (1) changes in sources, sinks, and sanitization, (2) permission updates, (3) new inter-component communication (ICC) mechanisms, and (4) changes in reflection and native code handling.

For new/outdated sources and sinks (e.g., from new APIs, sensors, or storage mechanisms), only the source/sink lists need updating. We automatically retrieve the appropriate list based on the app's target Android version, as detailed in Section 4. This mechanism allows EvoTaint to remain effective as new platform versions introduce additional APIs.

When Android evolves its permission model (e.g., introducing scoped storage or background location restrictions), these changes primarily affect which APIs are considered sensitive sources or protected sinks, rather than changing how information flows are tracked between them. Our approach separates the definition of what constitutes sensitive information from the core analysis of how it flows. Thus, this evolution will be again adopted via source/sink updating.

For changes in inter-component communication (ICC) mechanisms, EvoTaint inherits the ICC modeling capabilities of its underlying analyzer (e.g., FlowDroid). When that analyzer is updated to handle new ICC mechanisms, EvoTaint automatically adapts through these updates without requiring changes to its incremental analysis algorithm.

Changes in reflection handling, native code execution paths, or security enforcement mechanisms may require updates to the underlying taint analyzer. However, EvoTaint's incremental methodology remains valid—it would

continue to efficiently identify which portions of an app need reanalysis based on code changes, regardless of how the underlying analyzer models these advanced/evolved language features.

The key insight is that EvoTaint operates at a higher abstraction level than the specific Android security model details. It focuses on determining *what parts of this app need to be reanalyzed due to changes*, rather than *how to model specific Android security features*. This design allows EvoTaint to adapt to Android evolution with minimal maintenance, simply by updating the underlying analyzer (e.g., by replacing underlying tools like the conventional analyzer such as FlowDroid to consider the new platform features, an ICC analyzer to resolve ICC induced data/control flow to augment the data flow analysis in FlowDroid, a reflection analysis tool such as DroidRA [50] to resolve reflective control/data flow., a source/sink mapping tool such as PScout [9] to map source/sink APIs per their associated permissions, etc.) and source/sink definitions while keeping its core incremental analysis algorithm unchanged.

8 Threats To Validity

This section addresses potential issues that could impact the validity of our experimental results.

Internal validity threats. The main threat to the internal validity of our results lies in the potential errors in our tool implementation of EvoTaint. To mitigate this threat, we have conducted careful code review of the source code of EvoTaint, followed by unit and integration tests of the tool against relatively small and simpler cases of apps and between-version code changes. Another such threat comes from the inaccuracy of our deobfuscation steps, due to the limited accuracy and capabilities of the deobfuscators used. We chose to use popular and strong ones of their kind, but it is well-known that accurate deobfuscation is an NP problem [4]. Thus, when EvoTaint is used in the presence of obfuscation, its performance and empirical results are both subject to the inaccuracy of deobfuscation.

Another internal validity threat has to do with the nondeterministic nature of FlowDroid, a key component of EvoTaint's tool implementation. Previous studies ([11]) have noted that FlowDroid can produce nondeterministic results. Specifically, running the same analysis on the same app with identical settings can yield different outcomes. This issue with FlowDroid, unresolved at the time of our study, poses a challenge for ensuring absolute consistency between the taint flow paths synthesized by our approach and those obtained through conventional analysis. The potential variability in FlowDroid's output could introduce inconsistencies in our results, affecting the reliability of the comparisons made between *incremental taint analysis* and conventional methods. To deal with this issue, we had to run FlowDroid multiple times when necessary (i.e., in the cases in which the inconsistencies were found) during our accuracy validation (for answering RQ6).

External validity threats. The main threat in this class concerns our benchmark selection. Our experiment involved a set of 100 Android apps, each with 2, 3, or 5 versions. This selection was based on the average number of versions per app, as reported in [3], to ensure reproducibility and relevance. However, this choice may also introduce limitations.

Firstly, there is an implicit assumption that the app with three evolved versions are representative of the broader range of Android apps. While this average is statistically reasonable, it may not account for the diversity and complexity found in families with more than three versions. Apps with a higher number of versions could inherently differ in their structural and functional complexity, potentially affecting the efficiency of *incremental taint analysis*. For example, if such apps are inherently less costly to analyze using our approach compared to those with fewer versions, the observed average cost savings might not accurately reflect the potential savings across a more diverse application set. This limitation suggests that our results might be skewed towards a specific subset of Android apps, potentially impacting the generalizability of our findings.

In short, while our dataset selection was informed by statistical averages, the potential variability in app complexity across different families with varying numbers of versions remains a factor that could influence the outcomes of our study. This discussion highlights the need for caution in interpreting our results and suggests avenues for further research to explore the applicability of *incremental taint analysis* across a more diverse range of Android apps.

Second, in our study, apps with two or three versions were selected based on their adjacency in release version numbers available on F-Droid. However, this selection does not guarantee that these versions are the closest in terms of actual commit history or even in release history. This limitation is particularly relevant for our methodology, which ideally expects to analyze consecutive versions for best efficiency gains. The exception was the subset of apps with five versions, where we manually constructed the dataset by cloning source code repositories and building consecutive app versions (APKs) based on the actual commit history. While this approach is more accurate, it is also more time-consuming and labor-intensive. Therefore, the dataset's composition, especially for apps with two and three versions, represents a compromise between experimental quality and feasibility.

Also, the fact that our benchmarks are garnered from F-Droid limits the diversity of the study dataset. We chose to do so because F-Droid provides apps of evolved versions and source code of Android apps which allows us to build continuous versions from the respective version histories (as we explained earlier in §5.2). While this choice is justifiable, we cannot rule out the possibility that our empirical results and conclusions based on the results may shift to some extent should we use benchmarks obtained from other data sources.

Construct validity threats. The main threat in this class concerns how well our measurements represent the concepts we aim to study. Our metrics for cost reduction may be affected by FlowDroid's nondeterministic nature, as mentioned in our current internal validity section. This nondeterminism means running the same analysis multiple times could produce slightly different execution times and results, potentially affecting our measurement of efficiency gains. Additionally, our measurement of change impact through statement-level analysis, while practical, may yield false positive changes due to refactoring. Even though the final taint paths result will not be affected thanks to taint synthesis, the change ratio we obtained might not be always accurate when apps have refactoring during evolution.

Conclusion validity threats. For our primary comparison between EvoTaint and conventional analysis, we employed Wilcoxon signed-rank tests due to the non-normal distribution of time measurements, obtaining strongly significant results ($p = 4.56 \times 10^{-13}$ for 2-version, $p = 1.82 \times 10^{-12}$ for 3- and 5-version groups) with large effect sizes ($r > 0.72$ across all groups). In our correlation analyses (Section 6.2—6.4), we used Spearman's rank correlation coefficient to evaluate relationships between various factors (like change ratio and time savings) because it doesn't assume normal distribution and is robust to outliers.

However, while our 2-version benchmark group with 100 apps provides strong statistical power, the relatively small sample sizes in our 3-version (10 apps) and 5-version (5 apps) benchmark groups may limit the generalizability of our statistical conclusions about cumulative efficiency benefits. We chose $p < 0.05$ as our significance threshold, which is standard in software engineering research. The strong statistical significance we observed (p-values orders of magnitude smaller than this threshold) and large effect sizes provide confidence in our conclusions despite these limitations, though future work with larger samples of multi-version apps would further strengthen these findings.

Our app version selection methodology prioritized adjacency in the app's release/commit history, which may not represent all evolution patterns. Different sampling strategies (e.g., selecting versions spanning longer development periods or focusing on major releases) might yield different patterns of efficiency improvement by our technique, particularly regarding the correlation between app change ratios and efficiency benefits.

9 Related Work

There are mainly two lines of works in the literature that are related to EvoTaint: approaches for improving the performance of static taint analysis and more general-purpose incremental static analysis.

9.1 Improving Taint Analysis Performance

He et al. [42] presented an optimization of the IFDS algorithm, commonly used in tools like FlowDroid. It introduces a sparse propagation method in the IFDS algorithm, significantly reducing time and memory requirements for taint analysis. Their tool, SparseDroid, achieved an average speedup of 22.0x compared to FlowDroid, efficiently completing analysis on all tested Android apps within set resource limits. In contrast, our approach targets the evolving nature of Android apps, selectively reanalyzing altered or affected code segments, thereby offering substantial savings in analysis time and resources for apps with frequent updates. While both methods aim to enhance taint analysis efficiency, they address different aspects: SparseDroid optimizes the IFDS algorithm's internal workings, whereas our method streamlines the analysis process for evolved app versions. On the other hand, integrating insights from SparseDroid's sparsification technique with our incremental approach (especially in the part corresponding to the use of FlowDroid) could potentially yield a more comprehensive and efficient taint analysis framework, particularly beneficial in scenarios necessitating quick reanalysis due to frequent app version changes.

FastDroid [67] deviates from traditional data flow analysis and focuses on the propagation of taint values, thereby improving analysis efficiency. It uses a preliminary flow-insensitive taint analysis to construct Taint Value Graphs (TVGs) and then checks the feasibility of Potential Taint Flows (PTFs) on the control flow graph (CFG) of the given app. Their experiments demonstrated FastDroid's superior performance in terms of precision, recall, and efficiency compared to earlier versions of FlowDroid. FastDroid achieved up to 93.3% precision and 85.8% recall, with significant reductions in analysis time. In contrast, our *incremental taint analysis* specifically exploits the evolving nature of evolving Android apps to reduce the analysis time on evolved versions of apps. While FastDroid improves the overall efficiency and accuracy of taint analysis, our approach focuses on reducing redundancy in successive app version analyses by selectively reanalyzing changed or impacted code segments. This selective reanalysis is particularly suited for apps undergoing frequent updates, allowing for significant savings in time and computational resources. On the other hand, the integration of FastDroid's methodology could potentially augment our incremental approach, especially for initial analyses of base versions, creating a comprehensive framework that combines the benefits of both methods in analyzing Android apps.

FlowTwist [49] introduces an innovative context-sensitive inside-out taint analysis approach, primarily focusing on large Java codebases. This method, unlike traditional taint analyses that track flows from sources to sinks, begins at inner layers of an API and operates outwards, effectively reversing the usual analysis direction. FlowTwist implements two separate sub-analyses: one for integrity and one for confidentiality, both of which operate inside-out. The approach is notable for its extension of the IFDS algorithm, which aids in maintaining context sensitivity and efficiently constructing paths for identified data flows. This method scales better and performs faster compared to pure forward analysis, as confirmed by experiments comparing it against conventional approaches. In comparison, while FlowTwist's inside-out methodology is adept at handling large Java codebases and efficiently tracking integrity and confidentiality issues, our approach focuses on optimizing taint analysis by concentrating on changes between successive Android app versions. This results in significant savings in analysis time and resources, especially for apps with frequent updates. On the other hand, the integration of FlowTwist's inside-out approach and our incremental analysis could potentially yield a

robust system for handling both large-scale Java apps and rapidly evolving Android apps, leveraging the strengths of both methods to provide comprehensive and efficient taint analysis.

Cheetah [27] shares our goal of reducing the time needed for Android app taint analysis. It employs a just-in-time static analysis approach, prioritizing different complexity results based on the development context. While Cheetah is designed for interactive use during coding, our approach is more suited for analyzing completed apps during their production evolution.

9.2 Incremental Static Analysis

Mudduluru et al. [54] presents a novel approach to incremental static analysis, aiming to leverage path abstraction to optimize static analysis across versions. This approach involves analyzing changes in source code versions and the semantically affected code regions, using a path abstraction method. The path abstraction encodes program paths as a set of constraints in boolean formulas, which are then input to a SAT solver. The novelty of this approach lies in its focus on reducing redundant computations that arise from poor abstraction in existing tools. Notably, the method achieved up to a 32% performance gain in analysis time for large codebases (up to 87 KLoC).

Krishnan et al. [45] proposes an innovative technique for analyzing changes in large codebases at the time of commit. This method utilizes forward summaries, in addition to standard backward summaries, to focus the analysis solely on the changed code. Forward summaries encapsulate the effects of callers on a method, thereby reducing the need to reanalyze unchanged callers. This technique demonstrates significant efficiency, especially when only a fraction of the codebase is altered.

Reviser [7] focuses on updating interprocedural data-flow facts for incremental code changes. However, its application in accelerating taint analysis for evolving Android apps remains unclear. Incorporating Reviser's incremental data-flow analysis into our approach could potentially enhance its efficiency while retaining accuracy.

Incremental information-flow analysis has been applied in various software engineering tasks, such as data flow computation [56], program testing [10], and change-impact analysis [40, 48]. However, its application in security defense, particularly in code-based security analysis like taint checking, has been less explored.

In comparison, our method focuses on Android app versions and streamlines the taint analysis process by selectively reanalyzing change-impacted code behaviors. Our approach does not directly improve the efficiency of the underlying fine-grained static (data-flow) analysis itself with incremental code changes. We still delegate the process of that underlying analysis to FlowDroid without tampering it. However, the core idea of incremental static analysis, which is to narrow the analysis scope, is explicitly reflected in our design of EvoTaint.

9.3 App Evolution Analysis for Android

Studies on Android app evolution have characterized significant changes in app structure, behavior [16, 19], compatibility over time [37], and code-level regional variations [38], all highlighting the challenges posed to traditional app analysis. To mitigate the high cost of re-analyzing entire apps after incremental updates, the field of app evolution analysis has explored various strategies as noted earlier [42, 49, 67]. Other research within the broader scope of Android evolution analysis has investigated the sustainability of machine learning-based malware detectors against evolving app features [15] and the use of code change semantics to improve software maintenance tasks like release note generation [39]. These diverse efforts underscore the need for techniques that adapt to the dynamic Android landscape, with incremental methods like EvoTaint offering targeted efficiency gains for complex analyses.

10 Conclusion

In this paper, we introduced *incremental taint analysis* for Android apps, a novel approach designed to enhance the cost-effectiveness of static taint analysis for evolving Android apps. Our method capitalizes on the version history of apps, utilizing previous taint analysis results (for earlier app versions) to minimize re-computation for unchanged or unaffected code segments in evolved app versions. By applying our technique to a dataset of 100 real-world Android apps, we demonstrated a significant reduction in analysis costs brought by the incremental analysis—averaging a 51.8—68.9% decrease compared to conventional taint analysis and exceeding 99% reduction in many cases−−while maintaining accuracy. This underscores the potential of *incremental taint analysis* as a valuable tool for developers in the continuous vetting process of app evolution. We have made our open-source tool, EvoTaint, publicly available, providing a practical resource for conducting *incremental taint analysis* for Android apps.

Acknowledgment

We thank our associate editor and reviewers for insightful and constructive comments. This work was supported in part by the U.S. Office of Naval Research (ONR) under Grant N000142512252 and National Science Foundation (NSF) under Grant CCF-2505223.

References

[1] [n. d.]. F-Droid - Free and Open Source Android App Repository. https://f-droid.org/.

[2] 2024. Wilcoxon Signed-Rank Test. *Wikipedia* (Nov. 2024).

[3] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 468–471.

[4] Andrew Appel. 2002. Deobfuscation is in NP. *Princeton University, Aug* 21, 2 (2002).

[5] Steven Arzt. 2013. SuSi. https://github.com/secure-software-engineering/SuSi.

[6] Steven Arzt and Eric Bodden. 2014. Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering*. 288–298.

[7] Steven Arzt and Eric Bodden. 2014. Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering*. 288–298.

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Acm Sigplan Notices 49, no. 6 (2014)*. 259–269.

[9] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 217–228.

[10] Samuel Bates and Susan Horwitz. 1993. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 384–396.

[11] Manuel Benz, Erik Krogh Kristensen, Linghui Luo, Nataniel P. Borges Jr, Eric Bodden, and Andreas Zeller. 2020. Heaps'n leaks: how heap snapshots improve Android taint analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1061–1072.

[12] Benjamin Bichsel, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2017. Statistical Deobfuscation for Android. http://apk-deguard.com/.

[13] Haipeng Cai. 2015. *Cost-effective dependency analysis for reliable software evolution.* University of Notre Dame.

[14] Haipeng Cai. 2017. Hybrid program dependence approximation for effective dynamic impact prediction. *IEEE Transactions on Software Engineering* 44, 4 (2017), 334–364.

[15] Haipeng Cai. 2020. Assessing and Improving Malware Detection Sustainability through App Evolution Studies. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 2 (2020), 8:1–8:28. https://doi.org/10.1145/3371924

[16] Haipeng Cai, Xiaoqin Fu, and Abdelwahab Hamou-Lhadj. 2020. A study of run-time behavioral evolution of benign versus malicious apps in android. *Information and Software Technology* 122 (2020), 106291.

[17] Haipeng Cai and John Jenkins. 2018. Leveraging Historical Versions of Android Apps for Efficient and Precise Taint Analysis. In *IEEE/ACM Working Conference on Mining Software Repository (MSR)*. 265–269. https://doi.org/10.1145/3196398.3196433

[18] Haipeng Cai, Na Meng, Barbara G Ryder, and Danfeng Daphne Yao. 2016. *Droidcat: Unified dynamic detection of Android malware.* Technical Report. Department of Computer Science, Virginia Polytechnic Institute & State University.

[19] Haipeng Cai and Barbara Ryder. 2020. A Longitudinal Study of Application Structure and Behaviors in Android. *IEEE Transactions on Software Engineering (TSE)* 47, 12 (2020), 2934–2955. https://doi.org/10.1109/TSE.2020.2975176

[20] Haipeng Cai and Raul Santelices. 2014. DIVER: Precise Dynamic Impact Analysis Using Dependence-based Trace Pruning. In *Proceedings of International Conference on Automated Software Engineering*. 343–348.

[21] Haipeng Cai and Raul Santelices. 2015. Abstracting program dependencies using the method dependence graph. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 49–58.

[22] Haipeng Cai and Raul Santelices. 2015. A Framework for Cost-effective Dependence-based Dynamic Impact Analysis. In *International Conference on Software Analysis, Evolution, and Reengineering*. 231–240.

[23] Haipeng Cai and Raul Santelices. 2016. Method-Level Program Dependence Abstraction and Its Application to Impact Analysis. *Journal of Systems and Software (JSS)* 122 (2016), 311–326.

[24] Haipeng Cai, Raul Santelices, and Siyuan Jiang. 2016. Prioritizing Change Impacts via Semantic Dependence Quantification. *IEEE Transactions on Reliability (TRE)* 65, 3 (2016), 1114–1132.

[25] Haipeng Cai, Raul Santelices, and Tianyu Xu. 2014. Estimating the Accuracy of Dynamic Change-Impact Analysis using Sensitivity Analysis. In *Proceedings of International Conference on Software Security and Reliability*. 48–57.

[26] Norman Cliff. 1996. *Ordinal methods for behavioral data analysis*. Psychology Press.

[27] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. 2017. Cheetah: just-in-time taint analysis for android apps. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference*. 39–42.

[28] Lisa Nguyen Quang Do, James R Wright, and Karim Ali. 2020. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering* 48, 3 (2020), 835–847.

[29] Craig Drummond. [n. d.]. LMS Material: Simple webview wrapper for MaterialSkin on an LMS server. https://f-droid.org/en/packages/com.craigd.lmsmaterial.app/. F-droid.org.

[30] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. 2015. Android security: a survey of issues, malware penetration, and defenses. *Communications Surveys & Tutorials, IEEE* 17, 2 (2015), 998–1022.

[31] Caleb Fenton. 2018. Generic Android Deobfuscator. https://github.com/CalebFenton/simplify.

[32] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. 2013. Highly Precise Taint Analysis for Android Application. (2013).

[33] Xiaoqin Fu and Haipeng Cai. 2019. A Dynamic Taint Analyzer for Distributed Systems. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Tool Demos*. 1115–1119. https://doi.org/10.1145/3338906.3341179

[34] Xiaoqin Fu and Haipeng Cai. 2021. FlowDist: Multi-Staged Refinement-Based Dynamic Information Flow Analysis for Distributed Software Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2093–2110. https://www.usenix.org/conference/usenixsecurity21/presentation/fu-xiaoqin

[35] Jun Gao, Li Li, Pingfan Kong, Tegawendé F. Bissyandé, and Jacques Klein. 2018. Poster: On Vulnerability Evolution in Android Apps.. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. 276–277.

[36] Michael Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilhamy, Nguyen Nguyen, and Martin Rinard. 2015. Information-Flow Analysis of Android Applications in DroidSafe. In *NDSS*.

[37] Jiawei Guo, Xiaoqin Fu, Li Li, Tao Zhang, Mattia Fazzini, and Haipeng Cai. 2025. Characterizing Installation- and Run-Time Compatibility Issues in Android Benign Apps and Malware. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 0, 0, Article 0 (2025), 43 pages. https://doi.org/10.1145/3725810

[38] Jiawei Guo, Yu Nong, Zhiqiang Lin, and Haipeng Cai. 2025. Code Speaks Louder: Exploring Security and Privacy Relevant Regional Variations in Mobile Applications. In *IEEE Symposium on Security and Privacy (S&P)*. 3952–3970. https://doi.org/10.1109/SP61157.2025.00225

[39] Jiawei Guo, Haoran Yang, and Haipeng Cai. 2025. VerLog: Enhancing Release Note Generation for Android Apps using Large Language Models. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1–23. https://doi.org/10.1145/3728961 (artifact evaluated; badges: Available).

[40] Shengjian Guo, Markus Kusano, and Chao Wang. 2016. Conc-iSE: Incremental symbolic execution of concurrent software. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 531–542.

[41] Dongjie He, Yujiang Gui, Yaoqing Gao, and Jingling Xue. 2023. Reducing the Memory Footprint of IFDS-Based Data-Flow Analyses using Fine-Grained Garbage Collection. In *Symposium on Software Testing and Analysis (ISSTA'23)*.

[42] Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. 2019. Performance-boosting sparsification of the ifds algorithm with applications to taint analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 267–279.

[43] Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. 2015. Tracking the software quality of android applications along their evolution. In *In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 236–247.

[44] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 106–117.

[45] Padmanabhan Krishnan, Rebecca O'Donoghue, Nicholas Allen, and Yi Lu. 2019. Commit-time incremental analysis. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. 26–31.

[46] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. Soot - a Java Bytecode Optimization Framework. In *Cetus Users and Compiler Infrastructure Workshop*.

[47] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

[48] James Law and Gregg Rothermel. 2003. Incremental dynamic impact analysis for evolving software systems. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on.* IEEE, 430–441.

[49] Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. 2014. FlowTwist: efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 98–108.

[50] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 318–329.

[51] Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. 2022. PolyCruise: A Cross-Language Dynamic Information Flow Analysis. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2513–2530. https://www.usenix.org/conference/usenixsecurity22/presentation/li-wen (artifact evaluated; badges: Available, Functional).

[52] Kangjie Lu, Zhichun Li, Vasileios P. Kemerlis, Zhenyu Wu, Long Lu, Cong Zheng, Zhiyun Qian, Wenke Lee, and Guofei Jiang. 2015. Checking More and Alerting Less: Detecting Privacy Leakages via Enhanced Data-flow Analysis and Peer Voting.. In *NDSS*.

[53] Thomas J Marlowe and Barbara G Ryder. 1989. An efficient hybrid algorithm for incremental data flow analysis. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 184–196.

[54] Rashmi Mudduluru and Murali Krishna Ramanathan. 2014. Efficient incremental static analysis using path abstraction. In *Fundamental Approaches to Software Engineering: 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings 17*. Springer, 125–139.

[55] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do android taint analysis tools keep their promises?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 331–341.

[56] Lori L Pollock and Mary Lou Soffa. 1989. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering* 15, 12 (1989), 1537–1549.

[57] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys. In *annual meeting of the Florida Association of Institutional Research*, Vol. 177.

[58] MEthan M. Rudd, Andras Rozsa, Manuel Günther, and Terrance E. Boult. 2017. A Survey of Stealth Malware Attacks, Mitigation Measures, and Steps Toward Autonomous Open World Solutions. In *IEEE Communications Surveys & Tutorials*. 1145–1172.

[59] Barbara G Ryder and Marvin C Paull. 1988. Incremental data-flow analysis algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10, 1 (1988), 1–50.

[60] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *S&P*. 317–331.

[61] Samuel Sanford Shapiro and Martin B Wilk. 1965. An analysis of variance test for normality (complete samples). *Biometrika* 52, 3-4 (1965), 591–611.

[62] Vincent F. Taylor and Ivan Martinovic. 2017. To update or not to update: Insights from a two-year study of android app evolution.. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*.

[63] Fengguo Wei, Sankardas Roy, and Xinming Ou. 2014. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *CCS*. 1329–1341.

[64] I. Wier. 2016. Spearman Correlation. http://www.statstutor.ac.uk/resources/uploaded/spearmans.pdf.

[65] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. 2012. *Experimentation in software engineering*. Vol. 236. Springer.

[66] Wei You, Bin Liang, Wenchang Shi, Peng Wang, and Xiangyu Zhang. 2017. Taintman: An art-compatible dynamic taint analysis framework on unmodified and non-rooted android devices. *IEEE Transactions on Dependable and Secure Computing* 17, 1 (2017), 209–222.

[67] Jie Zhang, Cong Tian, and Zhenhua Duan. 2019. FastDroid: efficient taint analysis for Android applications. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 236–237.

[68] Jie Zhang, Cong Tian, and Zhenhua Duan. 2021. An efficient approach for taint analysis of android applications. *Computers & Security* 104 (2021), 102161.

[69] Junbin Zhang, Yingying Wang, Lina Qiu, and Julia Rubin. 2021. Analyzing android taint analysis tools: FlowDroid, Amandroid, and DroidSafe. *IEEE Transactions on Software Engineering* 48, 10 (2021), 4014–4040.

[70] Xueling Zhang, Xiaoyin Wang, Rocky Slavin, and Jianwei Niu. 2021. Condysta: Context-aware dynamic supplement to static taint analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 796–812.