Attention! Your Copied Data is Under Monitoring: A Systematic Study of Clipboard Usage in Android Apps

Yongliang Chen[†], Ruoqin Tang[†], Chaoshun Zuo[¶], Xiaokuan Zhang[‡], Lei Xue[‖]
Xiapu Luo[§], Qingchuan Zhao[†]
[†]City University of Hong Kong, [¶]The Ohio State University, [‡]George Mason University,
[‖]Sun Yat-Sen University, [§]The Hong Kong Polytechnic University
{cs.ylchen,r.tang}@my.cityu.edu.hk,zuo.118@osu.edu,xiaokuan@gmu.edu,xuelei3@mail.sysu.edu.cn
csxluo@comp.polyu.edu.hk,qizhao@cityu.edu.hk

ABSTRACT

Recently, clipboard usage has become prevalent in mobile apps allowing users to copy and paste text within the same app or across different apps. However, insufficient access control on the clipboard in the mobile operating systems exposes its contained data to high risks where one app can read the data copied in other apps and store it locally or even send it to remote servers. Unfortunately, the literature only has ad-hoc studies in this respect and lacks a comprehensive and systematic study of the entire mobile app ecosystem. To establish the missing links, this paper proposes an automated tool, *ClipboardScope*, that leverages the principled static program analysis to uncover the clipboard data usage in mobile apps at scale by defining a usage as a combination of two aspects, i.e., how the clipboard data is validated and where does it go. It defines four primary categories of clipboard data operation, namely spot-on, grand-slam, selective, and cherry-pick, based on the clipboard usage in an app. *ClipboardScope* is evaluated on 26,201 out of a total of 2.2 million mobile apps available on Google Play as of June 2022 that access and process the clipboard text. It identifies 23,948 , 848 , 1,075 , and 330 apps that are recognized as the four designated categories, respectively. In addition, we uncovered a prevalent programming habit of using the `SharedPreferences` object to store historical data, which can become an unnoticeable privacy leakage channel.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; **Software reverse engineering**.

KEYWORDS

Program analysis, mobile security

ACM Reference Format:
Yongliang Chen[†], Ruoqin Tang[†], Chaoshun Zuo[¶], Xiaokuan Zhang[‡], Lei Xue[‖], Xiapu Luo[§], Qingchuan Zhao[†]. 2024. Attention! Your Copied Data is Under Monitoring: A Systematic Study of Clipboard Usage in Android Apps. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623317

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3623317

1 INTRODUCTION

The clipboard has become a vital feature in modern mobile operating systems, including iOS and Android. Its copy, cut, and paste functions allow users to move text within the same app or across different apps seamlessly. Interestingly, Android supported cross-app clipboard usage from its initial release, while iOS only allowed within-app copy and paste at first and later added cross-app functionality after iOS 3.0. The clipboard's convenience makes it easy for users to perform text-related tasks, such as copying a verification code from a messaging app and pasting it into other apps for multi-factor authentication. As a result, many mobile apps have developed advanced functionality built upon the clipboard, making it an integral part of mobile users' daily routines. For instance, input method apps provide hints that automatically fill in the blanks with the copied text.

Despite the convenience of copying and pasting text across different apps, this clipboard's ability can pose significant security and privacy concerns. That is because the clipboard temporarily stores the copied text, while there is only weak or even no access control mechanism enforced. In other words, once a piece of text is copied, any app running in the foreground could access such data via the clipboard regardless of whether the app is intentionally opened or just happens to be in the foreground when sliding between multiple apps. For instance, input method apps could gain access to the sensitive verification codes when people copy this code from the messaging app and paste it into another app.

Unfortunately, it could be overly optimistic to assume that mobile apps only access clipboard data to display it on the screen; apps may illegally store sensitive text copied from other apps. Recent works have conducted a few case studies that revealed several possible attack vectors targeting these data, including contact records [53] and cryptocurrency wallets [26]. Some apps have even been found to send clipboard data directly to their backend servers [39]. Although these studies were conducted manually and limited to a relatively small set of apps, they highlight the potential privacy risks associated with unrestricted clipboard usage. On the other hand, while mobile operating systems have deployed mechanisms to notify users when their copied text is accessed, these mechanisms may not be sufficient to protect user privacy. For example, Android 10 [12] blocks background apps from accessing the clipboard; however, apps can still secretly conduct suspicious monitoring in the foreground. Android 12 [13] can show a toast message to inform

users that the clipboard has been accessed, but enabling this feature requires a tedious manual switch-on process [36].

According to our preliminary study, the misuse of clipboard functionality in mobile apps may be more alarming than previously reported. Upon manually inspecting several popular apps that access the clipboard, we discovered some concerning practices. For instance, a third-party input method editor app with over *one million* installs automatically logs the clipboard contents to local storage each time the user finishes input. We also found a communication app with over *100 million* installs that continuously monitors clipboard status and automatically saves its content whenever it's updated. Furthermore, a shopping app with over *500 thousand* installs attempts to obtain clipboard data each time users paste it into the search box without any modification and sends the data to its backend server upon resuming from the background.

Therefore, there is an urgent need to understand how apps access the clipboard data, how they deal with such data, and to what extent these actions expose severe security and privacy risks. To this end, in this paper, we propose a new static analysis framework called *ClipboardScope* to systematically and automatically analyze clipboard-related operations in mobile apps on a large scale. At a high level, *ClipboardScope* combines a static taint analysis with particular consideration of efficient and precise control/data flow tracking capabilities in the Android framework to understand how an app accesses and processes data from the clipboard. Specifically, *ClipboardScope* defines a clipboard data operation in a combination of two code aspects, *i.e.*, data validation (*e.g.*, format checking) and data destination (*e.g.*, screen, local storage, and backend servers). Finally, *ClipboardScope* classifies an execution path of a clipboard data operation into various categories based on the combination of their data validation and data destination.

At a high level, we have classified four primary categories of clipboard data operation categories: *(i)* the "spot-on" operation that some apps take the clipboard data and display them on the screen only, which complies with the guideline of clipboard usage. Second, some apps will store the clipboard data locally or send them to the remote regardless of whether displaying them on the screen or not. In particular, we could further divide these operations into several sub-categories based on how apps manipulate the clipboard data. Specifically, these include *(ii)* the "grand-slam" operation that apps have no particular requirement of the data but just store and send them without exception, *(iii)* the "selective" operation that apps will store or send the whole copied text if the data contains certain keywords, and *(iv)* the "cherry-pick" operation that apps only store or send a specific part of the copied text if the whole text containing certain keywords, e.g., the first four numbers after the keyword "OTP". These categories could ultimately assist in recognizing potential security and privacy risks.

We have implemented a prototype of *ClipboardScope*[1] and studied the current state of clipboard data usage by analyzing 2.2 million Android apps from Google Play and a well-maintain repository, AndroZoo [2]. First, we leveraged the system APIs that an app must invoke to get access to the clipboard (e.g., `getPrimaryClip`) and identified 185,423 apps with clipboard-accessing functionality. In particular, we have identified 23,948 apps with the spot-on operation and 2,253 apps that might transfer data, among which 2,228 attempt to store the data locally and 43 transfer the data through the Internet. Interestingly, we uncovered a prevalent programming habit of using the `SharedPreferences` object to store historical data, which can become an unnoticeable privacy leakage channel.

Contribution. In short, we make the following contributions:

- **Novel Research.** We take the first step toward understanding the current state of clipboard usage in mobile apps at scale and classifies the clipboard usage into four primary operations.
- **Automatic Tool.** We devise a systematic tool, *ClipboardScope*, containing a suite of novel static analysis techniques to automatically classify and recognize various clipboard usage operations in mobile apps by analyzing the execution sequence of the program.
- **Comprehensive Evaluation.** We have evaluated *ClipboardScope* on a dataset of 2.2 million up-to-date mobile apps as of June 2022. It discovered 26,201 apps that attempt to access and process the clipboard text and revealed that 2,253 apps can locally store the data or transfer it to externals.

2 BACKGROUND AND MOTIVATION

2.1 Mobile Apps and Clipboard

In Android, the system clipboard is accessible by all apps, and no explicit permission grant is required for manipulations. In particular, the Android system clipboard is represented by the global `ClipboardManager` class, whose reference can be obtained by invoking `getSystemService(CLIPBOARD_SERVICE)`. When a copy action happens, the clipboard is updated by a newly created `ClipData` object with an in-class `Item` object storing the actual contents (*e.g.*, text, Uri, and Intent). In `ClipboardManager`, we can obtain the `ClipData` object through the `getPrimaryClip` method. Although it can contain multiple `Item` objects (*e.g.*, for other data processing purposes), the copying only results in one `Item` object by default. Therefore, we can simply consider the operation of accessing the first `Item` object in the clipboard by calling `getItemAt(int)` on the `ClipData` object with integer 0 as the argument, on which the stored texts can be finally retrieved by invoking `getText`. When a copy action happens, the clipboard is updated by a newly created `ClipData` object with an in-class `Item` object.

2.2 Motivating Example

In Figure 1, we present a motivating example that abstracts from a popular real-world app with over one million installs. This app had the capability to secretly transfer contents that users pasted into the search box to a third-party server. These search records could then be used to build customer profiles for accurate recommendation and advertising, as well as for other profitable purposes. Sending data in the search field within an app may seem legitimate most of the time. However, in this case, the data was sent automatically without the users' consent, implying that the third party could obtain whatever the user pasted, including sensitive content copied by mistake. This constitutes a privacy leak channel.

Specifically, a customized class `A`, serving as a search box in the app, which is inherited from the `EditText` widget, overrides the callback function `onTextContextMenuItem` to modify the default behavior taken when users click "paste" in the text box's pop-up

[1] Available at https://github.com/CityuSeclab/ClipboardScope_open.

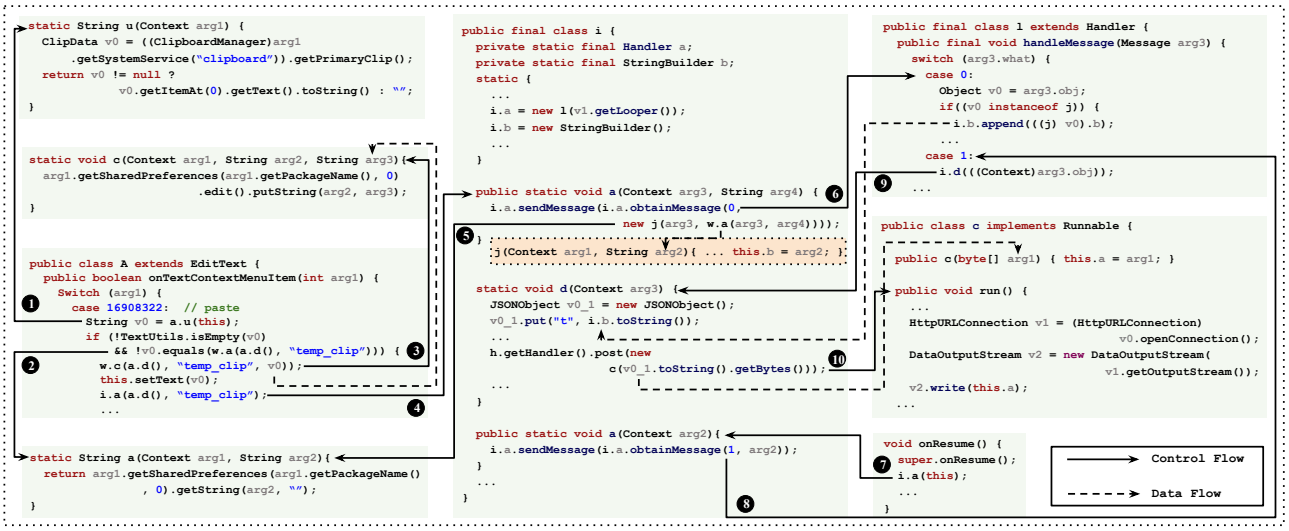


Figure 1: A simplified running example sending pasted contents to the back-end server each time it resumes.

menu. It first retrieves the clipboard text with a helper function (step ❶), and verifies whether the content exists (emptiness check) and is newly copied (duplication verification with the stored up-to-date data obtained in step ❷). Then, if both conditions hold, it updates the `SharedPreferences` object with the new data and the key "temp_clip" (step ❸). In the next step, the app displays the copied content in the search box with the `EditText.setText` method, after which it calls `i.a` (step ❹), obtains (step ❺) and writes the data to a field `j.b` inside the constructor of `j`, and then constructs a `Message` object with the newly created `j` as the `obj` argument. Next, the created `Message` is pushed to the processing queue of the thread held by the `Handler` object `i.a` (step ❻). Recall that the sent `Message` is constructed with 0 as the `what` argument, therefore, it is processed in the `case 0` branch of the `switch` statement inside the `handleMessage` method, where the current field of interest, `arg3.obj.b`, is appended to the static class variable `i.b`.

On the other hand, when the app resumes from the background, the called `onResume` method invokes `i.a` (step ❼), inside which another `Message` is sent with `what` being 1 (step ❽) to further call `i.d` (step ❾). Then, the static variable `i.b`, which holds data from the clipboard, is put into a `JSONObject`, converted to bytes, and stored in the field `a` of a newly created `c` object. Finally, this object, implementing the `Runnable` interface, is posted to the task queue, and its `run` method will be executed (step ❿), where `this.a` is transferred to the backend server.

3 OVERVIEW

3.1 Challenges

We must tackle two key challenges to build *ClipboardScope*.

C1: How to precisely pinpoint the execution sequence related to the clipboard data in an app. While static taint analysis techniques have been well-studied in app vulnerability discovery, *e.g.*, Flowdroid [5], IccTA [27] and DroidSafe [19], their capabilities and overhead in analyzing large apps (*e.g.*, over 50 MB) hinder their practicalities in nowadays apps on the market. Although there are numerous works on large-scale app analysis, they all have specific focuses (*e.g.*, cloud-service API misuse [63], privilege-escalation vulnerabilities in pre-installed apps [15], and input validations [58]). They are difficult to be generalized to our case. More importantly, in the example shown in Figure 1, apps are multi-threaded with asynchronous function calls (*e.g.*, step ❻, ❽, and ❿), and data might flow in global instances (*e.g.*, static field `i.b` and `SharedPreferences` objects) or be used in inter-component communication (*e.g.*, `Intent`), which results in discontinuous control and data flows, making the analysis challenging. Therefore, we must overcome the above issues to uncover the whole execution sequence of interest.

C2: How to recognize different clipboard usages from the execution sequence. Having detected the execution sequence of the retrieved clipboard data, we need to recognize the corresponding clipboard usage. However, this is by no means trivial because the purposes of using these data remain unknown. For instance, in the scenario where an app retrieves the clipboard content and sends it to its backend server through the Internet; if this is an IME app, it might be malicious since arbitrary contents on the clipboard are leaked to third parties. However, suppose this appears in a shopping app that attempts to automatically scan the clipboard for the sharing texts and request the commodity's page; it could be regarded as an approved functionality. In these cases, simply identifying taint sources and sinks (*i.e.*, system APIs for clipboard text retrieval and data transferring) like the previous work [5, 28, 40] is not sufficient. It is significant to accurately and effectively analyze the clipboard usage from the execution sequence with limited information.

3.2 Insights

S1: pinpointing clipboard data related execution sequence via context, flow, path, and object-sensitive static analysis. To conduct hidden operations in the background by invoking asynchronous tasks, apps must implement specific systematic interfaces or derive customized code from abstract callbacks. Therefore, we first identify the method entry for asynchronous calls through a context and flow-sensitive backward search and then construct

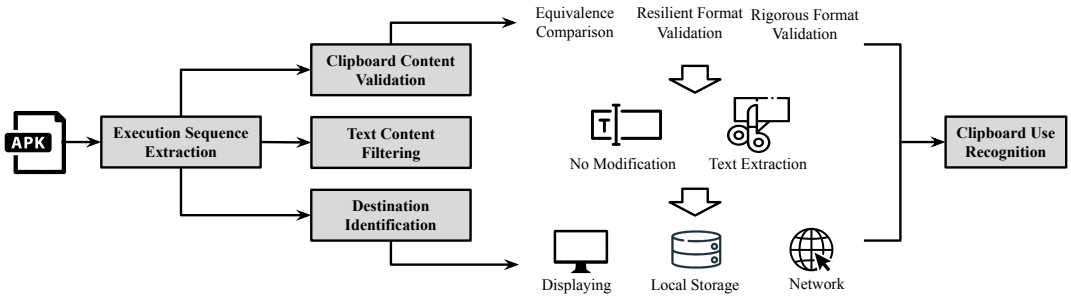


Figure 2: Overview of *ClipboardScope*.

new control flows. In the case of discontinuous data flows, our approach depends on the cause of the issue. If the issue is caused by tainted static variables, we adopt an object-sensitive approach that broadcasts tainted static variables and searches for all variables of interest. If the taint flows into key-value structures, we resolve the key value and search for any retrieved values via this key. This approach effectively addresses discontinuous data flows.

S2: recognizing a clipboard usage through a clipboard operation context. An important insight in this context is that clipboard data undergoes certain operations before being locally stored or transferred to servers. By analyzing these operations, we can infer the purpose of the clipboard data usage. We construct the semantic information of an execution sequence based on two orthogonal aspects: (*i*) the type of format validation methods used on the clipboard text, and (*ii*) the existence of text extraction operations. For instance, consider the code in Figure 1 which sends out clipboard data without any format validations. This indicates that no functionalities in the app require clipboard sniffing, making it a case of clipboard snooping. In particular, some manipulations on the retrieved clipboard `String` data can be categorized as:

- **Equivalence Comparison.** Conducting some equivalence comparisons after getting the clipboard text is a common programming pattern in apps. The purposes of this operation are: (*i*) **emptiness check** by invoking, *e.g.*, `String.isEmpty` and `String.equals("")`, to avoid further processing on empty strings, (*ii*) **duplication verification** that compares the current clipboard text with the previous one to avoid duplicate operation (*e.g.*, auto-filling a text box twice), and (*iii*) **hidden function** that is triggered by special copied contents on the clipboard (*e.g.*, the debug mode is activated if a special pre-defined string is detected), which is aligned with the previous work [58].
- **Compatibility Examination.** Apps may leverage the clipboard to realize functionalities (*e.g.*, directing to webpages of goods and auto-filling the invitation code). To achieve that, they need to confirm that the copied texts are compatible with certain formats by either performing: (*i*) **rigorous format validation** by calling, *e.g.*, `String.startsWith` and `Sring.endsWith`, to check whether the string starts or ends with certain identifiers, or `indexOf` to examine the existence and position of a certain sub-string; (*ii*) **resilient format validation** to examine strings with looser rules, *e.g.*, `String.matches` or `String.contains`, to validate whether a string matches some regular expressions or a specific identifier occurs in the string.
- **Content Filtering.** Regarding the content an app actually uses, we consider two orthogonal cases: (*i*) **no modification** that the raw text retrieved from the clipboard is kept (*e.g.*, logging the copied texts), and (*ii*) **text extraction**, in which portions of the raw text are filtered out for further processing (*e.g.*, a shopping app extracts the URL after the symbol ":" from a piece of sharing text via `System.substring` for the direction to an online shopping page and a VPN app firstly converts the JSON-like string into a JSON object and then parses configuration settings with `JSONObject.optString`).

On the other hand, apps finally direct the obtained text clips to different destinations, which can be categorized as:

- **Displaying to Users.** This is the most common behavior of an app that meets users' expectations, in which the copied contents are observable, and the users are aware that the app perceives the copied contents. This kind of operation is mainly processed by invoking methods that show contents in a text box, such as `EditText.setText` and `EditText.setHint`.
- **Storing Locally.** This behavior is often hidden from users and can be suspicious since the clipboard is typically used for temporary data transfer rather than long-term data storage. It's worth noting that aside from APIs in the `Java.io` category for direct file writing, inserting data into objects like `SharedPreferences` and `SQLiteDatabase` can also lead to local data copies in storage.
- **Transferring to Servers.** Data from the clipboard might be used in some networking functionalities. It can either be a URL through which the app can initiate a connection with the corresponding service provider, or be enclosed and sent out by post request to third-party servers.

Having such a context, we derive several rules to classify and recognize different clipboard usages. The recognition could further assist in justifying the necessity of the clipboard data for the functionality of an app and identifying potential malicious usages or privacy leaks from the execution sequence.

3.3 *ClipboardScope* Overview

The overall design of *ClipboardScope* is depicted in Figure 2. Given an APK file, the proposed static analysis technique is used to extract the execution sequence, revealing how the app processes the clipboard text. We then analyze the execution sequence from three dimensions: (*i*) the content validation type showing how the clipboard text is verified, (*ii*) the text content filtering process extracting information from the text, and (*iii*) the final destination identification indicating how the data is used. Using the information revealed

Type		Class	API
Sources		ClipboardManager	getPrimaryClip() ->[ClipData.Item: getText() | ClipData.Item: coerceToText()]
		ClipboardManager	getText()
Sinks	Re. V.	String	contains(CharSequence), matches(String)
		Pattern	matches(String, CharSequence), matcher(CharSequence)
	Ri. V.	String, StringBuffer	indexOf(String), lastIndexOf(String)
		String	startsWith(String), endsWith(String)
	T. E.	String	subString(int, int), subSequence(int, int)
		JSONObject	optString(String), opt(String, String)
	Data Dest.	java.io.*	write(*), print(*), println(*)
		java.net.*	<init>(URL), getOutputStream() ->java.io.*
		okhttp3.*	add(String, String), url(String)
		SQLiteDatabase	insert*(String, String, ContentValues), execSQL(String)
		SharedPreferences.Editor	putString(String, String), putStringSet(String, Set<String>)
		EditText	setText(CharSequence), setHint(CharSequence)

Table 1: The taint sources and sinks used in *ClipboardScope*: "->" indicates the execution order, Re. V. for resilient format validation, Ri. V. for rigorous format validation, T. E. for text extraction, Data Dest. for data destinations.

from the execution sequence, with formulated policies, we can infer the purpose of the clipboard use at a high level and uncover any privacy-related issues that might arise.

3.4 Scope and Assumptions

This paper focuses on understanding clipboard usage in Android apps while it could be extended to applications on other platforms. Specifically, *ClipboardScope* only analyzes in-app operations on *text-based* contents on the clipboard (*i.e.*, `String`-type data), which can be obtained through `Item.getText` or `ClipboardManager.getText` methods. This is because manually copied texts and most app-posted clipboard data are always stored as `Text`. While apps may attempt to retrieve other MIME types of data from the clipboard, such as `Intent` for application shortcuts or `Uri` for resolving complex data (*e.g.*, images) from the `Content Provider`, analyzing these operations is beyond the scope of this paper, even though they might raise privacy concerns. In addition, the analysis is based on the code implemented at the Java bytecode level in Android apps and is resilient to many code obfuscation techniques (*e.g.*, class and method renaming) but may fail in cases when the apps use native libraries or invoke methods through reflection. Moreover, our focus is solely on operations conducted via standard system APIs, and other third-party APIs are out of scope.

4 DETAILED DESIGN

4.1 Identifying Clipboard Text Manipulations

In *ClipboardScope*, we use static taint analysis techniques to pinpoint manipulations on the clipboard text. Since there are numerous operations of interest, including format validations, text extractions, and data destination identification (*i.e.*, displaying, local storage and remote servers), we need to customize its sources and taint sinks to record this information. Specifically, in this study, we have systematically investigated standard Android APIs and defined the sources and sinks, which are listed in Table 1.

- **Taint Sources.** As described in Section 3.4, in this study, we only focus on text data retrieved from the clipboard. Therefore, the `getText` or `coerceToText` method following the `getPrimaryClip` are both set as our sources. In addition, we also notice that a deprecated method, `getText` of the `ClipboardManager` class, can directly obtain the text data stored on the clipboard; therefore, we also consider it as the source.
- **Taint Sinks.** The sinks contain APIs for text manipulations of interest. As discussed in Section 3.2, we categorize them into four different groups, including resilient and rigorous format validations, text extractions, and data destinations. Note that, for the networking APIs, we consider either leveraging the URL from the clipboard to initialize a network communication or transferring the clipboard data in a built network channel.

4.2 Execution Sequence Extraction

We first describe the construction of the inter-procedural Control-Flow Graph (ICFG) and inter-procedural Data-Flow Graphs (IDFG) and then elaborate on the solutions to the discontinuity problem in both control and data flows discussed in Section 3.1. At last, we summarize the properties of the proposed taint analysis technique.

Building ICFG and IDFG. The first step is to construct an app's ICFG. The ICFG consists of nodes, which are basic blocks containing consecutive statements with no branches in except at the entry and no branches out except at the exit. Each directed edge in the ICFG represents a control-transfer statement, *e.g.*, an `if-else` conditional transfer or a call-in and the corresponding call-out edges between the caller and callee sites either within or across classes.

Atop the built ICFG, we traverse the statements and construct IDFG. The IDFG defines the data dependency among the instructions in the ICFG, with each node representing one data flow source or destination corresponding to the specific operation semantic (*e.g.*, an assignment statement propagates data from the RHS to the LHS, a method call directs data from the caller site to the parameter of the invoked method, and the return statement routes data from the method body back to the corresponding variable at the caller that receives returned values). In particular, since Android apps are developed using Java, an OOP language, we track data flows through member fields and maintain the data hierarchy. For instance, after step ❺ in Figure 1, the returned tainted value flows to `j.b`, which makes both field `j.b` and the object `j` registering this field tainted, but leaves other fields (*e.g.*, `j.a`) unchanged. In addition, if the tainted field is killed (*e.g.*, `j.b = new c()`), we check upward to ensure whether there still exists any other tainted fields. If not, `j` is untainted, and this back-propagation keeps on until it subsequently reaches the first still-tainted field (suppose `j` is a member field of another outer class) or un-taints the whole class instance. Therefore, with this design, in *ClipboardScope*, we can track data flow more precisely in an object- and field-sensitive manner.

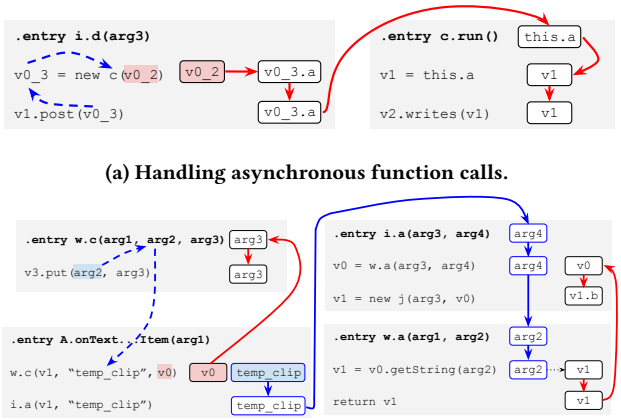
Handling Asynchronous Function Calls. The Android apps are always multi-threaded, with tasks processed in the background. According to our preliminary study, sensitive functionalities, such as I/O and networking operations, are usually conducted in the background to avoid blocking the foreground activity. This introduces challenges in constructing the correct ICFG. This is because, unlike invoking a method by its method signature, in asynchronous cases, the corresponding callbacks are processed by the Android

system, making the control flows cannot be identified and constructed directly from the call-site, *e.g.*, the tainted `Message` object is forwarded by the `sendMessage` method and processed in the `handleMessage` callback at step ❻ in Figre 1; while at step ❿, the `post` method queues the tainted `c` object in the background and eventually the implemented `run` method is executed.

ClipboardScope processes asynchronous function calls separately from the normal ones. Although the callee corresponding to the caller cannot be directly matched, apps must use system APIs to invoke asynchronous processing. These APIs are known; therefore, we first identify them at the call site. Then, given the API information, we locate the object of interest and afterward determine the method in charge. This is because even though asynchronous tasks can be invoked in different ways, the final method responsible for the processing is deterministic for each class and can be correctly located by backward searching for the initialization site (*i.e.*, the `new` statement) of the instance. For instance, an object of a class implementing `Runnable` interface can be invoked by either `Handler.post` or `Thread.start`, once we identify that the first argument in the former case and the argument used to initialize the base object in the latter case is of interest and trace back to its initialization site, we can locate the `run` method in the associated class and complete the ICFG. Notice that if the located class is inherited from another class and the method of interest is not directly implemented in the current one, we search the class hierarchy upward until the method is found.

Handling Discontinuous Data Flows. We perform on-demand IDFG construction that only propagates tainted values atop the ICFG for acceptable computational overhead; however, this might lead to missing some important data flows. For instance, in Figure 1, the static field `i.b` is tainted in the `case 0` block and reaches the final sink `write` method following the data flow. However, the actual control flows (step ❼ to ❾) are not traversed since there are no tainted values involved, leading to missing the resultant leak from `i.b`. As for the tainted value flows in and out from the `SharedPreferences` object by the `putString` and `getString` method, we cannot directly taint it (*i.e.*, values returned by `w.a`) since it contains other untainted values, doing so might result in many false positives. However, if we neglect the tainted values stored, we might omit to traverse control flows of interest (*i.e.*, step ❹).

The discontinuity in data flows mainly stems from using globally-accessible instances and inter-component communication (ICC). In particular, *ClipboardScope* tackles this issue by considering three cases, *i.e.*, static fields, `SharedPreferences`, and `Intent`, which we found prevalent in apps from the preliminary study. Unlike common instance fields in an object, which can be referred to by `This` in class methods and thus traceable to the object's construction site, a static field, however, is shared across all instances of a particular class and requires no instantiation, making the on-demand tracking unable to cover these data flows. To address this issue, in *ClipboardScope*, once a static field is tainted, we flood the taint to all references of this field in the program to rebind the discontinuous data flow. Therefore, in the example from Figure 1, though we miss the control flow from step ❼ to ❾, by directly propagating the tainted static field `i.b`, the data flow is reconnected.



(a) Handling asynchronous function calls.

(b) Reconnecting discontinuous data flows.

Figure 3: A simplified illustrative figure of the IDFG constructed by *ClipboardScope* from portions of the code of Figure 1. The dotted blue edges denote the backward tracing for (a) the entry of the targeted `run` function invoked by the asynchronous function call and (b) the hard-coded key value. The red edges represent tainted paths. The search for the key of interest is pinpointed by the blue edges, from which the tainted value is resumed by `getString` in (b).

As for `SharedPreferences` and `Intent`, although they are used for different purposes, they manage data in key-value pairs (*e.g.*, `putString` and `getString` for `SharedPreferences`, and `putExtra` and `getStringExtra` for `Intent`). Therefore, the first step is identifying the key value of interest such that the site retrieving these data can be located. In Android programs, both structures use string-type keys, and thus we must resolve the string value by backward slicing. In particular, we only target hard-coded string values since it is used in most apps[2] and maintains the balance between performance and overhead. Once the value is resolved, *ClipboardScope* searches it over the program, follows the ICFG to the targeted function (*e.g.*, `getString`), and reconnects the data flow by tainting the returned value with the key.

4.3 Execution Sequence Analysis

Having extracted the execution sequence, *ClipboardScope* recognizes these clipboard operations via their three phases, *i.e.*, Clipboard Content Validation, Text Content Filtering, and Destination Identification, and classifies them into four categories:

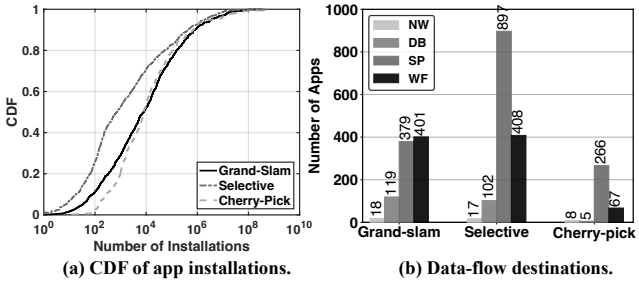
(I) "Spot-on" operation. The clipboard facilitates the flow of copied text within and across different apps. When users copy text to the clipboard and navigate to another app with an input box, the app is prompted to retrieve the copied text from the clipboard. Therefore, an app's primary task is to display the clipboard data on the screen without any additional operations. We refer to this task as "spot-on" as it accurately and efficiently displays the copied text.

(II) "Grand-slam" operation. In this case, the clipboard text might be examined with equivalence comparisons to guarantee that it is

[2]We manually analyze 300 apps with these data structures, and only one app uses the time the data is stored as the key.

Item	Value
# Apps tested	2.2 million
# Apps accessing the clipboard	185,423
# Apps with clipboard text manipulations	26,201
# Apps with Spot-on	23,948
# Apps with Grand-Slam	848
# Apps with Selective	1,075
# Apps with Res. Form. Val. only	743
# Apps with Rig. Form. Val.	332
# Apps with Cherry-Pick	330
# Apps with Res. Form. Val. only & Text Ext.	201
# Apps with Rig. Form. Val. & Text Ext.	129

Table 2: Overall statistics of the evaluated app.



(a) CDF of app installations. (b) Data-flow destinations.

Figure 4: Overall statistics of the inspected apps.

non-empty and a new one. For instance, a customized IME app can first check whether there are contents existing on the clipboard and then send out the content if it is not the same as the lastly-kept one after each time the user finishes inputting. The most important characteristic of these behaviors is that the compatibility examination is not involved in the execution sequence, meaning that no functionalities in the app rely on any information from the clipboard. Therefore, this operation mainly aims at storing the whole copied text in the clipboard locally or sending them out to the remote server, and we call it "grand-slam".

(III) "Selective" operation. To identify whether the text on the clipboard meets the requirement of curtain purposes, the app should conduct the compatibility examination. In general, apps may check the existence of some identifiers (*e.g.*, `contains`) or the compatibility to some regulated formats (*e.g.*, `matches`). For example, an auto-generated sharing text from a shopping app can be "Check out ... from our website! link: ...," where the phrase "link:" can be regarded as an identifier of the link processable in the target app, which might be used to request the corresponding webpage. Unlike the "grand-slam" operation, this operation is "selective" because it seeks certain keywords.

(IV) "Cherry-pick" operation. Similar to the "selective" operation, some apps may only need a specific part of the copied text in the clipboard to implement their advanced functionality. Consequently, besides validating the format of the clipboard text, which is similar to the "selective" operation, these apps have to manipulate the text to extract their interested part of the text and take further actions on these parts only. For example, a download helper app might conduct a continuous check on the clipboard to identify whether there is a magnet URL; if yes, it automatically downloads the content shared via this URL. The app might conduct a rigorous format validation to verify that the clipboard text is started with the identifiable phrase "magnet:?" using the `startsWith` function, and then use the information following the "magnet:?" only to initiate the downloading process. Considering this feature, we call these operations "cherry-pick".

5 APP ANALYSIS

5.1 Experimental Setup

Dataset. In this work, we wish to analyze all apps available in Google Play, the official Android app market, to have a better understanding. Unfortunately, we encountered many challenges in crawling all apps from Google Play because of various restrictions. Interestingly, we find another well-maintain repository, AndroZoo [2], that not only contains the most up-to-date apps from Google Play but also provides the older versions of almost every app. Considering this dataset has been used in many similar studies and is friendly for downloading, we crawled the whole repository, filtered duplicate apps to keep the up-to-date version only, and cross-checked with Google Play to exclude those off-shelf apps. As such, we have created a dataset of 2.2 million apps in total as of June 2022. Next, we leverage the system APIs that an app must invoke to access the clipboard, e.g., `getPrimaryClip`, to find apps we are interested, and finally identified 185,423 apps.

Testing Environment. *ClipboardScope* is implemented atop FlowDroid [5], which is a mature static analysis framework in Android app analysis, with a suite of novel techniques for optimizations. The entire experiment was conducted on a Linux server running Ubuntu 16.04 equipped with two Intel Xeon E5-2695 CPUs, CPU 48 cores in total, alongside 96 GB memory and 20 TB storage.

5.2 Overall Results

As presented in Table 2, there are 185,423 out of 2.2 million apps (8.4%) that were found accessing the system clipboard (*i.e.*, invoking `ClipboardManager.getPrimaryClip`). Among these apps, 29,116 apps (15.7%) failed in the analysis due to exceeding the 10-minute timeout, which complies with the findings in recent works [7, 29, 42] that the time and memory consumption for static analysis atop FlowDroid on new apps significantly increases. Nevertheless, *ClipboardScope* achieves a much lower failure rate, compared with these similar works (40%–70%). Among the 156,307 successfully-analyzed apps, 26,201 were found to attempt to manipulate the clipboard text. The remaining 130,106 apps invoking `ClipboardManager.getPrimaryClip` are mostly dead-code from third-party SDKs and libraries or aiming at other MIME types clipboard data, which are out of the scope of this paper (Section 3.4). Among the rest 26,201 apps, we further filter out 2,253 apps containing data-flows to local storage I/O or networking APIs, including 848 apps with grand-slam (*GS*), 1,075 (743 only contain resilient format validations and 332 contain rigorous validations) apps with selective (*ST*) and 230 (201 only contain resilient format validations and 129 contain rigorous validations) apps with cherry-pick (*CP*).

We have summarized the popularity of these 2,253 apps according to their categories, which are shown in Figure 4(a) as CDF. It can be observed that the number of installations occupies a wide range. Specifically, 405 (47.8%), 314 (29.2%), and 156 (47.3%) apps

# Installs	Category	Package Name	Data Dest. SP	DB	NW	WF	Tr.
100,000,000 – 500,000,000	Comm.	com.tencent.mm	●	❍	❍	❍	●
50,000,000 – 100,000,000	Product.	ridmik.keyboard	❍	●	❍	❍	●
50,000,000 – 100,000,000	Personal.	com.cutestudio.neonledkeyboard	❍	❍	❍	●	●
50,000,000 – 100,000,000	Personal.	com.flashkeyboard.leds	●	❍	❍	❍	●
50,000,000 – 100,000,000	Tools	fast.phone.clean	❍	●	❍	❍	●
10,000,000 – 50,000,000	Entmt.	com.mistplay.mistplay	❍	❍	●	❍	❍
10,000,000 – 50,000,000	Tools	com.facemoji.lite	❍	●	❍	❍	●
10,000,000 – 50,000,000	Lifestyle	com.aboutjsp.thedaybefore	❍	●	❍	❍	●
10,000,000 – 50,000,000	Lifestyle	net.milkdrops.beentogether	❍	●	❍	❍	●
10,000,000 – 50,000,000	Personal.	com.jb.gokeyboardpro	●	❍	❍	❍	●

Table 3: Detailed results of top inspected apps with grand-slam operations: Data Dest. for data destination, Tr. for trigger type; SP for `SharedPreferences`, DB for `SQLiteDatabase`, WF for local file, and NW for network; ● for presence, ❍ for absence in Data Dest. column; ● for automatic, and ❍ for user-triggered in Tri. column.

of the *GS*, *ST* and *CP* gain over 10,000 installations. When considering the cases with over one million installations, the numbers become 75 (8.8%), 78 (7.3%), and 25 (10.9%) in the three categories, respectively, indicating the extensive influence.

As for the final data-flow destinations in an app, as shown in Figure 4(b), we have summarized them into four categories, *i.e.*, leaking via inputting to `SharedPreferences` instances (SP), storing to `SQLiteDatabase` objects (DB), writing to files on the local storage (WF), and transferring through the network (NW). Most apps store the clipboard text locally, mainly via file writing and operations on `SharedPreferences` objects. A non-negligible amount of apps directly store the clipboard text in the database, which is suspicious. Although only a few apps transfer data through the network, some are very popular, with over *ten million* installations. The statistics shown above indicate a concerning situation in clipboard privacy.

5.3 Detailed Results

Since most apps are classified as spot-on apps that only display clipboard data on the screen, in this section, we primarily focus on the other three categories because they potentially contain privacy leakages. As such, we manually examined the top apps in each category in order to better understand the use of clipboard data.

Apps with grand-slam operation. Apps in this category directly store or transfer clipboard text without performing any format validations, which can lead to possible privacy leaks. We discovered a list of popular apps with such behaviors, indicating a concerning situation. To understand how this happens, we manually inspected ten apps with over ten million installations and presented the results in Table 3. Our findings are summarized as follows:

- **Continuous clipboard inspecting is prevalent.** We have uncovered the trigger scenario for these behaviors with our best effort. Specifically, one app leverages the clipboard to implement the launch redirection functionality by sending the account login data through the Internet. Besides this, all nine other apps inspect the clipboard data in the `onPrimaryClipChanged` callback method. It is each time when an `onPrimaryClipboardListerner` is registered and the primary clip data is changed (*e.g.*, copying a new text). Therefore, in these cases, apps can continuously monitor the clipboard and keep the latest data whenever it is updated *automatically* without any user-involved interactions (*e.g.*, performing the "paste" action).
- **Clipboard data is transferred without encryption.** Through our investigation, none of the top apps listed in Table 3 consider adopting any encryption algorithms to encrypt the clipboard data when being stored locally or sent to the outside. Only one IME app was found to use the `SHA-256` algorithm on the clipboard data before storing it.
- **Apps might maintain clipboard history.** We have observed that four apps intentionally develop a secret list for historical clipboard data. This can be achieved by maintaining an `Array` instance in the app, and there is always a size limit (*e.g.*, 10). Despite the fact that some functionalities related to these data might facilitate the design purpose of the app, *e.g.*, a customized keyboard app records the clipboard history for fast input suggestions; it inevitably increases the vulnerabilities in users' privacy.

Apps with selective operations. Given format validation conditions discovered in the execution sequence, to understand what types of content the app tries to access and the format validations used, we have manually investigated the top 5 apps with selective operations. Table 4 shows the results, and we have summarized three types of contents that these apps attempt to verify:

- **URL.** Apps might retrieve URLs from the clipboard. To verify whether the copied text is of interest, these apps mainly use the `contains` function to check the existence of specific prefixes, *e.g.*, "http://" for initializing a network connection to other servers, and "taobao.com" for directing to a shopping webpage. In addition, some apps leverage stricter criteria to verify that the text is started with certain sub-strings, such as verifying some special prefixes that can be used as identifiers ("EASEMOBIMG" in a Productivity app). Moreover, besides using fixed identifiers, we have found that two shopping apps also leverage periodically-updated regex patterns or strings to verify the text, indicating that stricter criteria are adopted in these apps.
- **Code.** Identifying the existence of any code (*e.g.*, validation code, invitation code, and security code) on the clipboard is a common functionality found in many apps. To validate that the copied text is a code string, apps mostly resort to APIs related to regular expression matching. According to our investigation, a photography app checks the existence of a code sequence by verifying the existence of a series of numbers, which can be done by matching the "[0-9]+" regex pattern (finding whether there are one or more consecutive digits). Also, this app is found to check on the existence of the string "][", which might be a special indicator of the expected text containing the code.
- **Character.** Another set of apps verifies the existence of certain characters. Dislike the above two categories where apps attempt to verify whether certain contents *exist* in the text, in this case, apps act differently to the validation results. For instance, we have found that a dictionary app tries to ensure that the clipboard text contains no special characters, such as "@", "$", and digits, which could be used as criteria for detecting English words.

V.M.	# Installs	Category	Package Name	Format Validation	Data Dest. SP	DB	NW	WF	Cont.
Res. Val.	100,000,000 − 500,000,000	Product.	cn.wps.moffice_eng	ℂ{"ftp://", "http://", "https://"}	●	❍	❍	❍	URL
	10,000,000 − 50,000,000	Books	com.hdictionary.bn	ℂ{"@", "$"}, 𝕄{"[0-9]"}	●	●	❍	❍	Char.
	5,000,000 − 10,000,000	Art	com.behance.behance	ℂ{"<iframe", ">"}	❍	❍	●	❍	URL
	5,000,000 − 10,000,000	Photo.	com.boo.facecam	ℂ{"]["}, 𝕄{"[0-9]+"}	●	❍	❍	❍	Code
	1,000,000 − 5,000,000	Shopping	com.daigou.sg	ℂ{"taobao.com", "http://", "tmall.com"…}, 𝕄{Dynamic Pattern}	●	❍	❍	❍	URL
Rig. Val.	100,000,000 −- 500,000,000	Product.	com.transsion.carlcare	𝕊{"EASEMOBIMG"}	●	❍	❍	❍	URL
	10,000,000 −- 50,000,000	Tools	com.ume.browser.international	𝕊{"http://", "rtsp://", "https://"}	●	❍	❍	❍	URL
	10,000,000 −- 50,000,000	Tools	tweeter.gif.twittervideodownloader	𝕀{"https://twitter.com", "https://mobile.twitter.com", ...}	●	❍	❍	❍	URL
	5,000,000 −- 10,000,000	Shopping	com.asda.android	𝕊{Dynamic_String}	❍	❍	❍	●	URL
	1,000,000 −- 5,000,000	Arcade	com.ace.shell.production	𝕊{"http://", "https://"}	❍	❍	●	❍	URL

Table 4: Detailed results of top inspected apps with selective operations: V.M. for validation method, Res. Val. for resilient validation, and Rig. Val. for rigorous validation; ℂ for `contains` condition, 𝕄 for regular expression matching, 𝕊 for `startsWith` condition, 𝔼 for `endsWith` condition, and 𝕀 for `indexOf` checking; Data Dest. for data destination, SP for `SharedPreferences`, DB for `SQLiteDatabase`, WF for local file, NW for network, ● for presence, and ❍ for absence; Cont. stands for content.

V.M.	# Installs	Category	Package Name	Format Validation	Text Extraction	Data Dest. SP	DB	NW	WF	Cont.
Res. Val.	50,000,000 − 100,000,000	Finance	com.santander.app	ℂ{"0014BR.GOV.BCB.PIX"}	𝕊𝕊{(s,e), (0, 11), (11, 22), (22, 33), ...}	●	❍	❍	❍	Setting
	500,000 − 1,000,000	Parent.	com.moms.momsdiary	𝕄{URL_Format}	𝕊𝕊{(*s*, *e*)}	●	❍	❍	❍	URL
	100,000 − 500,000	Auto.	com.gsd.carmeets	ℂ{"instagram", ""https://www.instagram.com/p/"}	𝕊𝕊{(0, e)}	❍	❍	●	❍	URL
	100,000 − 500,000	Business	ru.petrolplus.mobileapp	𝕄{"[0-9]+"}	𝕊𝕊{(*s*, *e*)}	❍	❍	❍	●	Code
	100,000 − 500,000	Education	com.sapienmind.bigmd	ℂ{"\\\'", "\\\\", ...}	𝕊𝕊{(0,100)}	❍	●	❍	❍	Text
Rig. Val.	100,000,000 − 500,000,000	Tools	com.mi.globalbrowser	𝕊{"about:", "data:", ...}, 𝔼{"."}, 𝕄{URL_Format}	𝕊𝕊{(*s*, *e*)}	●	❍	❍	❍	URL
	100,000,000 − 500,000,000	Card	com.matteljv.uno	𝕊{"omnisdk"}, 𝕄{"omnisdk://"}	𝕆𝕊{"extInfo", "linkID"}	●	❍	❍	❍	Setting
	10,000,000 − 50,000,000	Tools	me.skyvpn.app	𝕀{"/skyvpn"}, ℂ{"skyvpn://install?", "inviteKey", ...}	𝕊𝕊{(0, e)}	❍	❍	●	❍	URL
	10,000,000 − 50,000,000	Product.	com.tf.thinkdroid.viewer	𝕊{"**writeClipboard*** + pid}	𝕊𝕊{(*s*, *e*)}	❍	❍	❍	●	URL
	5,000,000 − 10,000,000	Business	com.fedex.ida.android	𝕊{"023"}, 𝕄{"^[a-zA-Z0-9]*$"}	𝕊𝕊{(2, 14)}	●	❍	❍	❍	Code

Table 5: Detailed results of top inspected apps with cherry-pick operations: V.M. for validation method, Res. Val. for resilient validation, and Rig. Val. for rigorous validation; ℂ for `contains` condition, 𝕄 for regular expression matching, 𝕊 for `startsWith` condition, 𝔼 for `endsWith` condition, 𝕀 for `indexOf` checking, 𝕊𝕊 for `subString` extraction, and 𝕆𝕊 for `optString` extraction; *s* and *e* mean the starting and ending index of the extracted string; Data Dest. for data destination, SP for `SharedPreferences`, DB for `SQLiteDatabase`, WF for local file, NW for network, ● for presence, and ❍ for absence; Cont. stands for content.

Apps with cherry-pick operations. We have manually analyzed the top 5 popular apps with cherry-pick operations in each validation method, *i.e.*, performing resilient format validation only and adopting rigorous format validation before storing or transferring the clipboard data. The results are summarized in Table 5. Note that, we found that apps mostly adopt self-defined ways to identify the starting or ending indexes of the substring of interest, *i.e.*, a VPN app is found first to replace the interested prefix with an empty string and then extract the component between 0 and the index of the ending character "#"; we denote the range to be (*s*, *e*) since the two indexes are not fixed. Compared with the findings in selective operations, we have the following insights:

- **URL extraction is the most overlapped functionality.** Apps with this purpose always start by finding special identifiers similar to (II). Also, we found apps using well-defined regex patterns to match the interested part of the URL, which we denote as "URL_Format" [3]. After that, the text is truncated by extracting portions of the content, and only the captured part is kept. For instance, an automobile app extracts the post ID from the social media URL and performs quick sharing.
- **Extracting detailed setting configurations.** With the text containing setting information, we have observed that some apps might leverage text extraction to filter out detailed configurations denoted by different sub-fields. For instance, several apps directly convert the extracted strings to JSON objects and invoke `optString` to obtain the corresponding settings. One finance app attempts to extract QR code settings by capturing information from multiple fixed text positions. Another example is a Business app that first verifies the area code "023" on the text and then automatically extracts the 13-digit phone number.

6 DISCUSSION AND FUTURE WORK

6.1 Effectiveness of *ClipboardScope*

We have investigated apps with top installs in each category (40 in total) to evaluate the accuracy of *ClipboardScope* because it relies on static analysis. Specifically, we decompile and inspect the Java code, and then check the execution sequences inside each app to see whether they are consistent with the extracted ones and then further verify whether they are correctly categorized. As

[3] An example of such regex patterns is "(?:[\\W])((ht|f)tp(s?):\\/\\/|www\\. |m\\.)(([\\w\\-]+\\.){1,}?([\\w\\-.]+\\/?)* [\\p{Alnum}.,%_=?&#\\-+()\\[\\]*$ @!:/{};\'\]*)", which is found in a parenting app.

a result, we found three false positives, which are caused by the partial object-sensitivity, where the control flows are redirected to methods inside the unreachable-in-practice sub-classes pointed by the abstract references. In the remaining 37 apps with execution sequences correctly extracted, we have further identified two cases flagged as wrong categories. The first case is an app that first invokes `subString` with `(0,7)` and then compares its equivalence with "http://" to verify a URL, which is the same as calling `startsWith`; however, the `subString` is mistakenly regarded as a text extraction process, and the app is categorized as cherry-pick instead of selective. Another app first replaces some unwanted characters and then uses the `match` function to verify that they are all replaced. Then it truncates the text to get the first 100 characters with `subString` and inserts it into the database. This should be a grand-slam case while *ClipboardScope* classifies it as cherry-pick. Therefore, *ClipboardScope* achieved an accuracy of 87.5%.

To evaluate the effectiveness of the two proposed components to handle asynchronous function calls and discontinuous data flows, we disabled them separately to evaluate their performance. We appliedy them on the 2,253 apps identified to be either grand-slam, selective, or cherry-pick and checked whether the execution sequences in these apps could be uncovered. In particular, when the asynchronous function redirection was disabled, we could discover the target execution sequences in 69.9% (593/848) of the grand-slam, 82.0% (881/1,075) of the selective, and 95.8% (316/330) of the cherry-pick apps. The proportions became 83.7% (710/848), 83.0% (892/1,075), and 47.3% (191/330) when disabling the data flow reconnection component. Having these results, we conclude that the proposed two solutions in our static taint analysis technique contribute significantly by raising the success rate in uncovering execution sequences by 20.6% and 20.4%, respectively.

6.2 Limitations and Future Work

ClipboardScope is built atop static analysis that may inevitably result in inaccuracy. First, false positives inevitably exist in the reported results. The main reasons are two-fold: *1)* our tool routes data flows to all sub-classes pointed by virtual references, and *2)* we flood taints stored in global variables without fully tracing their execution contexts. These two policies might falsely include unreachable paths in the results. Second, our tool only checks the appearance of target APIs irrespective of their combinations, which might lead to false categorizations. Third, the tool now only focuses on the text data stored in the clipboard and analyzes clipboard usage operations at the Java code level in Android apps; therefore, its current version might miss some execution sequences of interest. As such, the improvement of the static taint analysis and enlarging the scope of ClipboardScope to other platforms are our future works.

6.3 Prevalence of `SharedPreferences`

During this work, we find a prevalent usage of `SharedPreferences` in the process of clipboard-related operations. To better understand the rationale behind this practice, we conducted a case study on 623 apps with over 5,000 installations with clipboard data inserted to `SharedPreferences` objects.

Specifically, we investigated whether data stored in these objects is further retrieved somewhere in apps and recorded their reachable

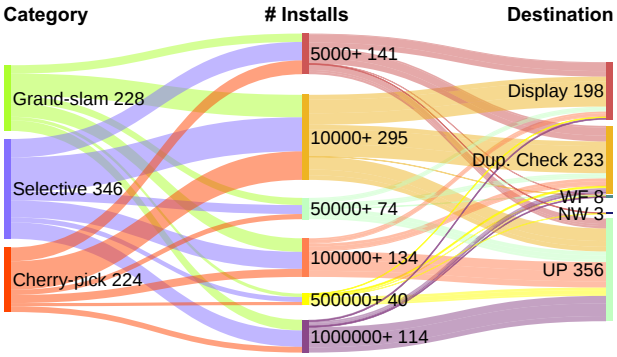


Figure 5: Data flows in different categories from `SharedPreferences` objects in apps with more than 5,000 installations. Each data flow is encoded in a tuple (Type, Installations, Destination): Dup. Check for duplication check, WF for local file, NW for network, UP for cases unprocessed in this study.

data destinations. Note that there can be more than one data flow of interest existing in an app, so the number of reported data flows is more than that of the apps. As shown in Figure 5, there are 267 apps containing 442 data flows directing to the destinations where the data is displayed on the screen, stored locally, and sent to the outside. In addition, we have discovered that a total of 198 apps display the clipboard text to app users, and 233 apps (87.3%) use the stored text to perform equivalence comparisons with the newly inspected clipboard data for duplicate checking. Interestingly, we found one clipboard helper app with grand-slam operations equipped with the functionality of maintaining historical clipboard data in a `SharedPreferences` object that has a backup function to write the stored data to a file in the local storage. Moreover, a VPN app with selective operations is found to encode the stored clipboard text in its `SharedPreferences` object into a URL and post a network request.

It can be observed from the above results that, the `SharedPreferences` object often serves as a "global variable" to perform duplicate checking in most apps since contents stored in these objects are accessible in the whole app; however, unlike the static field with the same functionality, data in the `SharedPreferences` object is kept as a hard-copy of file in the storage. Considering the sensitivity of clipboard text, this behavior makes the user privacy vulnerable.

In addition, we have discovered 356 apps that do not further process the clipboard data after storing it. We randomly selected ten apps and investigated them since they account for a large proportion. We found four apps only storing the contents without further usage, which is redundant dead code. The other three apps do not use hard-coded key values: one app uses the timestamp as the key, one app constructs the key using the generated device ID, and one app generates the key with values stored in other key-value structures. We failed to get the correct keys since *ClipboardScope* only backtraces for hard-coded keys. Two apps use tokens stored in the clipboard to generate device identifiers, which are then used to create names for logs or cached images. They are labeled as unprocessed since we do not record file-constructing APIs. One app retrieves the data and stores it in a `Map` object, a function parameter. This data flow is lost since *ClipboardScope* only tracks the returned value at the return site for efficiency considerations.

6.4 Legitimate Use of the Clipboard

By analyzing the inspected 30 apps with grand-slam, selective, and cherry-pick operations with top installs, we found that most apps leverage the clipboard for legitimate functionalities, while their implementations might leave vulnerabilities in user privacy.

For apps in the grand-slam category, seven out of ten apps cache historical clipboard data locally for input assistance in customized keyboards. These are necessary functionalities in apps; however, only one of these apps chooses to encrypt the data before storing it, leaving potential on-device privacy issues. There is one popular communication app that maintains the latest clipboard data in a `SharedPreferences` object without using it, which is redundant and puts the user's privacy at risk.

As for apps with selective and cherry-pick operations, they use clipboard data with highly-overlapped purposes (e.g., identifying URLs and extracting characters). Most of the inspected apps cache the data for duplicate checking. Some education apps (e.g., dictionary apps) might store historical data for learning purposes. These clipboard monitoring behaviors are designed for necessary functionalities. However, it is possible for these validation methods to fail in classifying information of interest with few simple conditions (e.g., only validating the "http" prefix), leading to recording irrelevant content and exposing users' privacy.

6.5 Clipboard Privacy Protections in Android

Starting with Android 10, only the default IME or apps in the foreground can access the clipboard [12]. Blocking background access behaviors is far from secure since apps can still retrieve clipboard data in the foreground without users' consent. Furthermore, according to Google, on January 2023, 32% of devices were still running Android 9 or lower [11]. In addition, Android 12 and higher can show a toast message to notify users when the app calls `getPrimaryClip` [13]. However, on the one hand, this requires a tedious switch-on process, which is easy to be neglected by most users. On the other hand, Android will not show the toast message if the app repeatedly accesses the clipboard data. Considering that this prompt does not tell the user what clipboard data is being accessed by the application, the user will usually be confused and ignore it.

6.6 Ethics and Responsible Disclosure

We have taken ethical considerations seriously. Although this work does not primarily focuses on security issues, we have contacted Google Play and developers of apps that may contain security issues to verify our findings. We have not disclosed any findings in this paper to any party yet except the relevant parties to who we are working closely to address issues with our best efforts.

7 RELATED WORK

Static Taint Analysis for Vulnerability Discovery. Uncovering malicious behaviors in mobile apps has been a long-lasting and significant research topic and there is a large body of works proposing promising solutions, such as novel unpacking strategies [47–50, 60], traffic analysis [31, 34, 52], and side-channel exploitation [32, 55, 56] Moreover, there is another branch of work focusing on leveraging static taint analysis, a highly-scalable technique, to uncover privacy-intrusive behaviors in apps [17, 23, 28, 35]. In past years, several generic tools [5, 18, 27, 40] have been developed to track sensitive data flows for vulnerability discovery. More specifically, some works focus on revealing privacy leaks from various aspects, including network communications [10, 30, 62], cloud services [4, 63], location services [33, 41, 59, 61], mini-programs [38], and resource operations [9, 20, 44, 51]. Different from these works, in this study, we focus on uncovering privacy leaks from the clipboard on smartphones on a large scale, which has not been well studied.

In addition, previous works have tried to discover malicious app behaviors by analyzing the code semantics, such as analyzing code embedding [3, 8, 14], code similarity distance [46, 64] and function callstacks [22, 45, 54]. Moreover, Zhao *et al.* [58] proposed InputScope to uncover hidden app secrets by analyzing the data types and code dispatch behavior during input validations. Inspired by these works, *ClipboardScope* uncovers clipboard-related usages by investigating execution sequences in apps.

Clipboard Data Privacy. In previous research, the clipboard data has been shown to contain highly sensitive information [1, 16, 24, 26, 37, 53, 57]. In addition, while numerous attacks have shown to be feasible on the data [21, 25, 53, 57] in a small set of apps, which indicates the user privacy on the clipboard can be compromised for malicious purposes. Although several possible countermeasures were proposed, such as enforcing permission controls [43] and data encryption [6], their deployment has not been accomplished yet. Recent work [39] has exposed the existence of clipboard privacy leaks on a small scale by incorporating code analysis and human-effort examination, which only shows the tip of the iceberg. Therefore, we propose *ClipboardScope*, which can automatically analyze the clipboard use in apps and expose possible privacy leaks, to arouse more community attention to this severe problem.

8 CONCLUSIONS

This study takes the first step toward understanding the current state of clipboard usage in mobile apps at scale. It proposes an automatic tool, *ClipboardScope*, that leverages the principled static program analysis to classify and uncover four primary clipboard data usage operations (*i.e.*, spot-on, grand-slam, selective, and cherry-pick) in mobile apps at scale by defining a usage as a combination of two aspects of the corresponding code, *i.e.*, how the clipboard data is validated and where does it go (the end of data-flow). *ClipboardScope* is evaluated on 26,201 out of a total of 2.2 million mobile apps available on Google Play as of June 2022 and identifies 23,948 , 848 , 1,075 , and 330 apps containing the four primary operations, respectively. Moreover, this study also uncovered that the `SharedPreferences` object is prevalently used to store historical data, which might establish an unnoticeable privacy leakage channel.

ACKNOWLEDGMENTS

We sincerely thank all anonymous reviewers for their constructive feedback. This work was partially supported by CityU APRC grant 9610563, the Research Grants Council of Hong Kong (CityU 21219223, C1029-22G, PolyU15224121), National Natural Science Foundation (62372490), and CCF-NSFOCUS KunPeng Research Fund. Any opinions, findings, and conclusions in this paper are those of the authors and are not necessarily of supported organizations.

REFERENCES

[1] Efthimios Alepis and Constantinos Patsakis. 2019. Unravelling security issues of runtime permissions in android. *Journal of Hardware and Systems Security* 3 (2019), 45–63.

[2] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. ACM, New York, NY, USA, 468–471. https://doi.org/10.1145/2901739.2903508

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[4] Omar Alrawi, Chaoshun Zuo, Ruian Duan, Ranjita Pai Kasturi, Zhiqiang Lin, and Brendan Saltaformaggio. 2019. The Betrayal At Cloud City: An Empirical Analysis Of Cloud-Based Mobile Backends.. In *USENIX Security Symposium*, Vol. 19.

[5] 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. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[6] Ahmad Atamli-Reineh, Ravishankar Borgaonkar, Ranjbar A Balisane, Giuseppe Petracca, and Andrew Martin. 2016. Analysis of trusted execution environment usage in samsung KNOX. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*. 1–6.

[7] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining apps for abnormal usage of sensitive data. In *2015 IEEE/ACM 37th IEEE international conference on software engineering*, Vol. 1. IEEE, 426–436.

[8] David Azcona, Piyush Arora, I-Han Hsiao, and Alan Smeaton. 2019. user2code2vec: Embeddings for profiling students based on distributional representations of source code. In *Proceedings of the 9th International Conference on Learning Analytics & Knowledge*. 86–95.

[9] Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury. 2017. Energypatch: Repairing resource leaks to improve energy-efficiency of android apps. *IEEE Transactions on Software Engineering* 44, 5 (2017), 470–490.

[10] Hyunwoo Choi, Jeongmin Kim, Hyunwook Hong, Yongdae Kim, Jonghyup Lee, and Dongsu Han. 2015. Extractocol: Automatic extraction of application-level protocol behaviors for android applications. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 593–594.

[11] Composables. 2023. Android Distribution Chart. https://www.composables.com/tools/distribution-chart.

[12] Android developers. 2019. Privacy changes in Android 10. https://developer.android.com/about/versions/10/privacy/changes.

[13] Android developers. 2023. Copy and paste. https://developer.android.com/develop/ui/views/touch-and-input/copy-paste#PastingSystemNotifications.

[14] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and distributed system security symposium*.

[15] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2020. FIRMSCOPE: Automatic uncovering of privilege-escalation vulnerabilities in pre-installed apps in android firmware. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 2379–2396.

[16] Sascha Fahl, Marian Harbach, Marten Oltrogge, Thomas Muders, and Matthew Smith. 2013. Hey, You, Get Off of My Clipboard: On How Usability Trumps Security in Android Password Managers. In *Financial Cryptography and Data Security: 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers 17*. Springer, 144–161.

[17] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. 2014. Android security: a survey of issues, malware penetration, and defenses. *IEEE communications surveys & tutorials* 17, 2 (2014), 998–1022.

[18] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Trust and Trustworthy Computing: 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings 5*. Springer, 291–307.

[19] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information flow analysis of android applications in droidsafe.. In *NDSS*, Vol. 15. 110.

[20] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. 2013. Characterizing and detecting resource leaks in Android applications. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 389–398.

[21] Vincent Haupert and Stephan Gabert. 2019. Short paper: How to attack PSD2 internet banking. In *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23*. Springer, 234–242.

[22] Shifu Hou, Aaron Saas, Lifei Chen, and Yanfang Ye. 2016. Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs. In *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*. IEEE, 104–111.

[23] Rosmalissa Jusoh, Ahmad Firdaus, Shahid Anwar, Mohd Zamri Osman, Mohd Faaizie Darmawan, and Mohd Faizal Ab Razak. 2021. Malware detection using static analysis in Android: a review of FeCO (features, classification, and obfuscation). *PeerJ Computer Science* 7 (2021), e522.

[24] Ilan Kirsh and Mike Joy. 2020. A different web analytics perspective through copy to clipboard heatmaps. In *Web Engineering: 20th International Conference, ICWE 2020, Helsinki, Finland, June 9–12, 2020, Proceedings 20*. Springer, 543–546.

[25] Zeyu Lei, Yuhong Nan, Yanick Fratantonio, and Antonio Bianchi. 2021. On the insecurity of SMS one-time password messages against local attackers in modern mobile devices. In *Network and Distributed Systems Security (NDSS) Symposium 2021*.

[26] Cong Li, Daojing He, Shihao Li, Sencun Zhu, Sammy Chan, and Yao Cheng. 2020. Android-based cryptocurrency wallets: Attacks and countermeasures. In *2020 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 9–16.

[27] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 280–291.

[28] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information and Software Technology* 88 (2017), 67–95.

[29] Haoran Lu, Qingchuan Zhao, Yongliang Chen, Xiaojing Liao, and Zhiqiang Lin. 2023. Detecting and Measuring Aggressive Location Harvesting in Mobile Apps via Data-flow Path Embedding. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 7, 1 (2023), 1–27.

[30] Abner Mendoza and Guofei Gu. 2018. Mobile application web api reconnaissance: Web-to-mobile inconsistencies & vulnerabilities. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 756–769.

[31] Tao Ni, Guohao Lan, Jia Wang, Qingchuan Zhao, and Weitao Xu. 2023. Eavesdropping Mobile App Activity via {Radio-Frequency} Energy Harvesting. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3511–3528.

[32] Tao Ni, Jianfeng Li, Xiaokuan Zhang, Chaoshun Zuo, Wubing Wang, Weitao Xu, Xiapu Luo, and Qingchuan Zhao. 2023. Exploiting Contactless Side Channels in Wireless Charging Power Banks for User Privacy Inference via Few-shot Learning. In *The 29th Annual International Conference On Mobile Computing And Networking*.

[33] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 2019. 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system. In *28th USENIX security symposium (USENIX security 19)*. 603–620.

[34] Suibin Sun, Le Yu, Xiaokuan Zhang, Minhui Xue, Ren Zhou, Haojin Zhu, Shuang Hao, and Xiaodong Lin. 2021. Understanding and detecting mobile ad fraud through the lens of invalid traffic. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 287–303.

[35] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. 2017. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)* 49, 4 (2017), 1–41.

[36] Techtrickz. 2022. How to Get Notification When Apps Read Your Clipboard Content on Android. https://techtrickz.com/how-to/get-notification-when-apps-read-clipboard-content-android/.

[37] Radu Vanciu, Ebrahim Khalaj, and Marwan Abi-Antoun. 2014. Comparative evaluation of architectural and code-level approaches for finding security vulnerabilities. In *Proceedings of the 2014 ACM Workshop on Security Information Workers*. 27–34.

[38] Chao Wang, Ronny Ko, Yue Zhang, Yuqing Yang, and Zhiqiang Lin. 2023. Taintmini: Detecting flow of sensitive data in mini-programs with static taint analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 932–944.

[39] Wei Wang, Ruoxi Sun, Minhui Xue, and Damith C Ranasinghe. 2020. An automated assessment of Android clipboards. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1249–1251.

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

[41] Haohuang Wen, Qingchuan Zhao, Zhiqiang Lin, Dong Xuan, and Ness Shroff. 2020. A study of the privacy of covid-19 contact tracing apps. In *Security and Privacy in Communication Networks: 16th EAI International Conference, SecureComm 2020, Washington, DC, USA, October 21-23, 2020, Proceedings, Part I 16*. Springer, 297–317.

[42] Daoyuan Wu, Debin Gao, Robert H Deng, and Chang Rocky KC. 2021. When program analysis meets bytecode search: Targeted and efficient inter-procedural analysis of modern Android apps in BackDroid. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 543–554.

[43] Hao Wu, Zheng Qin, Xuejin Tian, Edward Sun, Fengyuan Xu, and Sheng Zhong. 2019. Broken Relationship of Mobile User Intentions and Permission Control of Shared System Resources. In *2019 IEEE Conference on Dependable and Secure Computing (DSC)*. IEEE, 1–8.

[44] Tianyong Wu, Jierui Liu, Xi Deng, Jun Yan, and Jian Zhang. 2016. Relda2: an effective static analysis tool for resource leak detection in Android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 762–767.

[45] Cong Xie, Wei Xu, and Klaus Mueller. 2018. A visual analytics framework for the detection of anomalous call stack trees in high performance computing applications. *IEEE transactions on visualization and computer graphics* 25, 1 (2018), 215–224.

[46] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 363–376.

[47] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. 2017. Adaptive unpacking of Android apps. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 358–369.

[48] Lei Xue, Hao Zhou, Xiapu Luo, Le Yu, Dinghao Wu, Yajin Zhou, and Xiaobo Ma. 2020. Packergrind: An adaptive unpacking system for android apps. *IEEE Transactions on Software Engineering* 48, 2 (2020), 551–570.

[49] Lei Xue, Hao Zhou, Xiapu Luo, Yajin Zhou, Yang Shi, Guofei Gu, Fengwei Zhang, and Man Ho Au. 2021. Happer: Unpacking android apps via a hardware-assisted approach. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1641–1658.

[50] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. 2017. Malton: Towards {On-Device}{Non-Invasive} Mobile Malware Analysis for {ART}. In *26th USENIX Security Symposium (USENIX Security 17)*. 289–306.

[51] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for Android. *Automated Software Engineering* 25 (2018), 833–873.

[52] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. 2018. Homonit: Monitoring smart home apps from encrypted traffic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1074–1088.

[53] Xiao Zhang and Wenliang Du. 2014. Attacks on android clipboard. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 11th International Conference, DIMVA 2014, Egham, UK, July 10-11, 2014. Proceedings 11*. Springer, 72–91.

[54] Xueling Zhang, John Heaps, Rocky Slavin, Jianwei Niu, Travis D Breaux, and Xiaoyin Wang. 2022. DAISY: Dynamic-Analysis-Induced Source Discovery for Sensitive Data. *ACM Transactions on Software Engineering and Methodology* (2022), 1–32.

[55] Xiaokuan Zhang, Xueqiang Wang, Xiaolong Bai, Yinqian Zhang, and XiaoFeng Wang. 2018. Os-level side channels without procfs: Exploring cross-app information leakage on ios. In *Proceedings of the Symposium on Network and Distributed System Security*.

[56] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. 2016. Return-oriented flush-reload side channels on arm and their implications for android devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 858–870.

[57] Xiao Zhang, Kailiang Ying, Yousra Aafer, Zhenshen Qiu, and Wenliang Du. 2016. Life after App Uninstallation: Are the Data Still Alive? Data Residue Attacks on Android.. In *NDSS*.

[58] Qingchuan Zhao, Chaoshun Zuo, Brendan Dolan-Gavitt, Giancarlo Pellegrino, and Zhiqiang Lin. 2020. Automatic uncovering of hidden behaviors from input validation in mobile apps. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1106–1120.

[59] Qingchuan Zhao, Chaoshun Zuo, Giancarlo Pellegrino, and Lin Zhiqiang. 2019. Geo-locating Drivers: A Study of Sensitive Data Leakage in Ride-Hailing Services.. In *Annual Network and Distributed System Security Symposium, February 2019 (NDSS 2019)*.

[60] Hao Zhou, Haoyu Wang, Yajin Zhou, Xiapu Luo, Yutian Tang, Lei Xue, and Ting Wang. 2020. Demystifying diehard android apps. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 187–198.

[61] Sebastian Zimmeck, Peter Story, Daniel Smullen, Abhilasha Ravichander, Ziqi Wang, Joel R Reidenberg, N Cameron Russell, and Norman Sadeh. 2019. MAPS: Scaling Privacy Compliance Analysis to a Million Apps. *Proc. Priv. Enhancing Tech.* 2019 (2019), 66–86.

[62] Chaoshun Zuo and Zhiqiang Lin. 2017. Smartgen: Exposing server urls of mobile apps with selective symbolic execution. In *Proceedings of the 26th International Conference on World Wide Web*. 867–876.

[63] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. 2019. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1296–1310.

[64] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2018. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint arXiv:1808.04706* (2018).