PacDroid: A Pointer-Analysis-Centric Framework for Security Vulnerabilities in Android Apps

Menglong Chen, Tian Tan*, Minxue Pan, Yue Li*

State Key Laboratory for Novel Software Technology, Nanjing University, China

522023320008@smail.nju.edu.cn, {tiantan, mxp, yueli}@nju.edu.cn

Abstract—General frameworks such as FlowDroid, IccTA, P/Taint, Amandroid, and DroidSafe have significantly advanced the development of static analysis tools for Android security by providing fundamental facilities for them. However, while these frameworks have been instrumental in fostering progress, they often operate with inherent inefficiencies, such as redundant computations, reliance on separate tools, and unnecessary complexity, which are rarely scrutinized by the analysis tools that depend on them. This paper introduces PacDroid, a new static analysis framework for detecting security vulnerabilities in Android apps. PacDroid employs a simple yet effective pointer-analysis-centric approach that naturally manages alias information, interprocedural value propagation, and all Android features it supports (including ICC, lifecycles, and miscs), in a unified manner. Our extensive evaluation reveals that PacDroid not only outperforms state-of-the-art frameworks in achieving a superior trade-off between soundness and precision (F-measure) but also surpasses them in both analysis speed and robustness; moreover, PacDroid successfully identifies 77 real security vulnerability flows across 23 real-world Android apps that were missed by all other frameworks. With its ease of extension and provision of essential facilities, PacDroid is expected to serve as a foundational framework for various future analysis applications for Android.

I. INTRODUCTION

Android, launched over a decade ago, has evolved into one of the most widely adopted operating systems globally. As of now, billions of people use smartphones worldwide, with 72.15% of these users running the Android OS [1], [2]. This vast user base highlights the importance of maintaining the security of the apps available on this platform. However, the lack of thorough verification of Android apps has led to numerous security issues, including data leaks and various vulnerabilities [3]. To enhance Android security, static analysis has proven to be a valuable method by detecting potential vulnerabilities before they can be exploited [4]–[13].

To enable effective static security analysis, it is essential to provide a set of fundamental facilities tailored to the Android ecosystem. Representative frameworks in this domain include FlowDroid [4], IccTA [5], P/Taint [6], Amandroid [7], and DroidSafe [8], which address critical Android features such as lifecycle management (entry points, callbacks, layouts, etc.), inter-component communication (ICC), and Android-specific data structures (intents, bundles, shared preferences, etc.).

These few general frameworks have greatly fostered the development of numerous effective Android security tools by providing essential facilities for Android static analysis [11]–[13]; however, these security tools often focus on utilizing the functionalities provided by the frameworks without questioning their design rationale. Despite their success, these popular frameworks possess potential design flaws. For instances, IccTA and DroidSafe rely on separate tools like IC3 [14] and JSA [15] respectively, to resolve intents and other potential features beforehand. This reliance not only introduces extra time but also ties the framework's reliability to these pre-analysis tools. FlowDroid and IccTA reanalyze the program after each iteration of detecting new callbacks to extend the call graph until no new edges are added. This approach results in redundant computations and can slow down the analysis. Frameworks like P/Taint and FlowDroid itself adopt a highly conservative approach when constructing ICC taint flows, leading to imprecise results, while Amandroid employs multiple types of graphs, such as ICFG, DFG and DDG, increasing the overall complexity of the analysis.

In this work, we aim to achieve the following goals through a simple yet effective approach: a) to overcome the issues of existing frameworks mentioned earlier by offering a faster analysis with better trade-off between precision and soundness (a more sound analysis means resolving more real program behaviors); b) to build comprehensive value flows needed for taint analysis in Android. This requires handling alias information, interprocedural value propagation, and all Android features such as ICC, lifecycles and miscs, in a unified manner.

We thus present PacDroid, where we propose to analyze all Android features utilizing a pointer-analysis-centric (PAC) approach, first demonstrating that all such features can be effectively analyzed through pointer analysis, and built upon the plugin system of Tai-e [16], a state-of-the-art static analysis framework which offers an effective pointer analysis system for pure Java without any support for Android analysis. In PAC, each Android feature handling is treated as a plugin that interacts with the PAC engine, centered on pointer analysis, via several straightforward interfaces. All Android feature handlers, along with alias computation and interprocedural value flow, operate on and propagate along the same graph, the Pointer Flow Graph (PFG) [17], which underpins the taint analysis. The PAC approach eliminates the need to reanalyze the program after each iteration since the call graph and all features are resolved on the fly. It also removes the need for separate tools, as all features are handled and values are propagated by the pointer analysis. Such unified approach

offers several benefits: When a new Android feature is added, it can be easily integrated as a new plugin that cooperates with the pointer analysis. Improvements in the resolution of any feature (making it more sound or precise) will be propagated to other feature handlings via pointer analysis, while the enhancements to the pointer analysis will also automatically benefit all feature handlings. The simple design philosophy of PAC will ease both implementation and extensibility.

To thoroughly evaluate PacDroid, we compared it with state-of-the-art general Android analysis frameworks: FlowDroid, IccTA, P/Taint, and Amandroid. Given that the latest version of FlowDroid has integrated IccTA for analyzing ICC, and IccTA relies on FlowDroid as its foundation (with IccTA now maintained by FlowDroid), we refer to their combination as FlowIccTA throughout this paper. DroidSafe is excluded from the evaluation as it does not support Android versions higher than 19 (the latest version is 34), and it fails to analyze all the 24 programs in ICC-Bench [18], and has thus also been omitted from recent works [19]. Our evaluation utilized an extensive set of benchmarks, including DroidBench [20], ICC-Bench [18], and the recent UBCBench [21], with additional features incorporated.

Experimental results show that PacDroid outperforms the other frameworks in terms of F-measure (a higher F-measure value indicates a better trade-off between soundness and precision), analysis speed, and robustness (analysis crash rate). Specifically, PacDroid achieved an F-measure of 90%, compared to 77% for FlowIccTA, 72% for P/Taint, and 65% for Amandroid. It is also 2.6x, 28.8x, and 4.4x faster than FlowIccTA, P/Taint, and Amandroid, respectively, and has the lowest crash rate of 4%, compared to 27% for FlowIccTA, 8% for P/Taint, and 6% for Amandroid. Moreover, out of 250 real taint flows across 45 real-world Android apps, PacDroid successfully detected 187 taint flows, compared to 50 for FlowIccTA, 50 for P/Taint, and 67 for Amandroid; notably, PacDroid detected 77 real taint flows across 23 apps that were missed by all other frameworks, demonstrating its superior capability in detecting vulnerabilities in the real world.

In summary, this work makes the following contributions.

- We introduce PacDroid, a new static analysis framework for detecting security vulnerabilities in Android apps. PacDroid employs a principled pointer-analysis-centric (PAC) approach to handle various Android features in a unified manner, which enables PacDroid to incorporate distinctive designs for Android feature handling, such as integrated ICC and intent analysis.
- We conduct a comprehensive evaluation, showing that PacDroid not only outperforms state-of-the-art frameworks in achieving a superior soundness-precision trade-offs (F-measure), but also surpasses them in both analysis speed and robustness. Moreover, PacDroid successfully detects 77 real vulnerability flows across 23 real-world Android apps, where all other frameworks fail.
- We offer a fully open-source implementation of PacDroid and an artifact to reproduce all the experimental results at https://doi.org/10.5281/zenodo.14749910.

The security team of a Fortune Global 500 software company has developed 40 Android vulnerability analyses based on PacDroid (implemented as 40 handlers of PacDroid). These analyses have been integrated into the company's regular DevSecOps security scans. Over the past two months, 8 to 10 Android apps are analyzed daily, identifying over 100 potential vulnerabilities on average each day. This further demonstrates PacDroid's potential in real-world Android security analysis.

II. A MOTIVATING EXAMPLE

Inter-Component Communication (ICC) is a critical feature in Android, with intents serving as the medium for communication among components. Figure 1 depicts a simplified real code involving explicit and implicit intents in ICC. In this section, we use this example to illustrate the benefits of PacDroid's pointer-analysis-centric (PAC) approach in effectively integrating ICC and intent analysis, and how it addresses complexities that previous frameworks fail short of handling.

First, let's understand this example. It involves three components: two Activities (line 1 and line 8) and a Receiver (line 20). In line 4, an explicit intent object o1 (pointed to by i1) is created, with its target component explicitly specified as Activity2.class. This means the target component sent by calling startActivity(i1) in line 6 is Activity2, and the intent received by getIntent() in line 11, pointed to by r1, is o1. In line 15, an implicit intent object o2 is created, with its target component implicitly specified by the string value of its constructor's argument action. The value of action is retrieved in line 12 and stored in line 5 through the map-like extra information of intent o1, which is xxx.ACTION. To determine the target component of o2 sent by sendBroadcast(i2) (line 17), Android uses the xxx.ACTION value to match the component in the *AndroidManifest.xml* file, which is Receiver1 in this case (line 26). As a result, intent o2 should be passed to r2 in line 21, forming an ICC flow from i2 in line 17 to r2 in line 21, and making i2 in line 16 an alias of r2 in line 22. This establishes a taint flow from line 14 to line 23 through the ICC flow between components Activity2 and Receiver1.

a) Previous Work: Previous frameworks like FlowIccTA, Amandroid, and DroidSafe handle features like ICC and intents separately, and cannot resolve the ICC flow in Figure 1. Take FlowIccTA as an example. Before applying ICC analysis, it uses a separate intent resolution tool called IC3 to resolve intent-related string values. In Figure 1, IC3 correctly resolves that the target component of intent o1 in line 4 is Activity2 and identifies the extra information of o1 in line 5. However, it fails to resolve the string value of action in line 15 because it does not know what r1 (line 12) points to at that moment, as ICC analysis has not yet been applied. Consequently, it cannot resolve the target component of the implicit intent o2 in line 15. This leads to missing the ICC flow from i2 in line 17 to r2 in line 21, and thus the taint flow from line 14 to 23. On the other hand, P/Taint, like the original FlowDroid, handles ICC conservatively by treating any getIntent() (e.g., the one in line 11) as a source and any startActivity()

```
1    class Activity1 extends Activity {
2      protected void onCreate(Bundle b) {
3        …
4 ①    Intent i1 = new Intent(this, Activity2.class); //o1
5 ②    i1.putExtra("act", "xxx.ACTION");
6 ③    startActivity(i1);
7    }}

8     class Activity2 extends Activity {
9       protected void onCreate(Bundle b) {
10        …
11④     Intent r1 = getIntent();
12⑤     String action = r1.getStringExtra("act");
13      if (action != null) {
14        String s = source(); //source, so
15⑥     Intent i2 = new Intent(action); //o2
16⑦     i2.putExtra("tainted", s);
17⑧     sendBroadcast(i2);
18      }…
19    }}

20    class Receiver1 extends BroadcastReceiver {
21⑨   public void onReceive(Context c, Intent r2) {
22⑩    String tainted = r2.getStringExtra("tainted");
23     sink(tainted) //sink
24    }}
```

```
25    //AndroidManifest.xml
26    <receiver android:name="Receiver1" >
27      <intent-filter>
28        <action android:name="xxx.ACTION" />
29      </intent-filter>
30    </receiver>
```

Update (Write) / Notify (Read)

IntentExtra Analysis

Engine

IntentInfo Analysis — **SendICC Analysis**

① Obtain i1's points-to set (PTS), i.e., o1; resolve o1's intent info as ⟨o1, Activity2.class⟩ (means o1's target component is Activity2)

② Obtain i1's PTS, i.e., o1, resolve intent o1's extra info as ⟨o1, ("act", "xxx.ACTION")⟩ (extra info serves like a map-like field of intent object)

③ Obtain i1's PTS, i.e., o1, determine the target component of Activity1 is Activity2 based on o1's intent info resolved in ①

④ Update r1's PTS with o1 (i.e., let r1 point to o1) based on the ICC relation between Activity1 and Activity2 resolved in ③

⑤ Obtain r1's PTS, i.e., o1 (thanks to the PTS update in ④); resolve action's value based on intent o1's extra info resolved in ②, and update action's PTS with "xxx.ACTION" (as a string constant object)

⑥ Obtain i2's PTS, i.e., o2, and action's PTS, i.e., xxx.ACTION; resolve o2's intent info as ⟨o2, xxx.ACTION⟩

⑦ Obtain i2's PTS, i.e., o2, and s's PTS, i.e., source object so; resolve intent o2's extra info as ⟨o2, ("tainted", so)⟩

⑧ Obtain i2's PTS, i.e., o2, determine the target component of Activity2 is Receiver1 based on o2's intent info "xxx.ACTION" resolved in ⑥ along with the info in lines 26 and 28

⑨ Update r2's PTS with o2 (i.e., let r2 point to o2) based on the ICC relation between Activity2 and Receiver1 resolved in ⑧

⑩ Obtain r2's PTS, i.e., o2; resolve tainted's value based on intent o2's extra info resolved in ⑦, and update tainted's PTS with the source object so
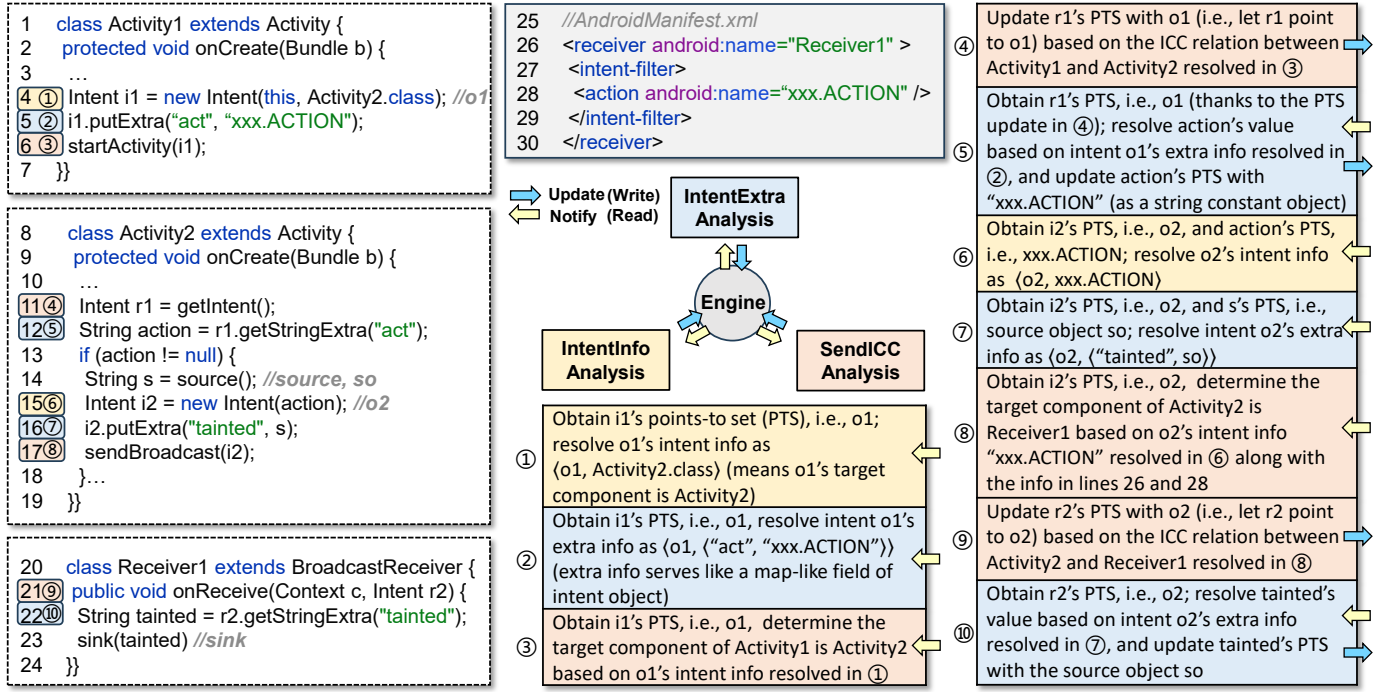
Fig. 1. A simplified real code illustrating integrated ICC and intent analysis in pointer-analysis-centric approach (PAC)

(e.g., the one in line 6) as a sink without resolving the intent information, which can result in many false ICC flows.

b) Our Work: PacDroid adopts a pointer-analysis-centric approach (PAC) to manage various Android features in a unified manner. In PAC, each feature handler obtains points-to set information for any variable via the PAC engine and reflects the side effects of feature handling by *updating* the points-to set of related variables. Consequently, the side effects of one feature handler are propagated by pointer analysis, meaning any side effect of any feature handler can be *notified* (by the engine) to all other feature handlers through the whole-program points-to information.

Now we invite readers to follow the ten self-explanatory steps in Figure 1 to grasp the workings of PAC and understand how PacDroid detects the taint flow from lines 14 to 23 by resolving the ICC flow from lines 17 to 21. For the case of Figure 1, the PAC approach involves three analyses: IntentInfo, IntentExtra, and SendICC, all interacting with the PAC engine, which manages the whole-program points-to information.

To facilitate understanding these ten steps, we illustrate Step ① here, with readers encouraged to follow the subsequent nine steps similarly. Step ①, highlighted in yellow, is part of the IntentInfo analysis and corresponds to handling the statement in line 4, as identified by the step number. From a Java semantics perspective, this is a typical `new` statement, so the pointer analysis will create an object `o1` of type `Intent`, invoke its constructor with two arguments, and let `i1` point to `o1`. Consequently, IntentInfo analysis can directly obtain the points-to set information of `i1` (i.e., which objects `i1` points to) from the engine. The yellow arrow on the

right side of the step rectangle signifies obtaining points-to information from the engine. Note that the engine actually proactively notifies the points-to information whenever related variables are updated, which we will explain in Section III. For now, you can consider it a read operation from the engine for simplicity. According to the semantics of `Intent`'s constructor with a meta-class object `Activity.class` as the second argument (line 4), IntentInfo analysis can resolve that the target component of intent object `o1` is `Activity2` and record it as `<o1,Activity2.class>` for future use. This result is a global intermediate analysis result accessible by any feature handler. Since it does not modify the points-to set of any variable, no update operation to the engine is required (unlike the blue arrow in Step ④).

After understanding these ten steps, we now provide key summarizations to enhance comprehension. In this case, IntentInfo analysis relies on IntentExtra analysis to resolve the target component (e.g., intent `o2` in line 15 needs to resolve `action`, whose value is extra information from another intent `o1`). IntentExtra analysis further depends on SendICC analysis to identify which intent objects the extra information belongs to (e.g., the intent object pointed to by `r1` in line 12 should be resolved first by SendICC analysis before identifying the `xxx.ACTION` value of the key `act` stored in line 5). This interdependence of feature handling is managed by PAC, which enables on-the-fly resolution of all feature handlers in conjunction with pointer analysis, in a decoupled manner. PAC naturally resolves scenarios involving multiple features, fundamentally differing from existing frameworks where feature handling is separate, as previously described.

III. DESIGN OF PACDROID

First, we present an overview of PacDroid, detailing its basic components, including the PAC engine and various Android feature handlers, and outlining the high-level idea of how the PAC approach enables them to work collaboratively. Subsequently, we delve into the detailed PAC approach, providing an explanation of the PAC engine algorithm and a template for the Android feature handlers' algorithms.

A. Overview of PacDroid

PacDroid contains two key parts: the PAC engine and various Android feature handlers that interact with it (Figure 2).

a) The PAC engine: It is based on the plugin system of Tai-e [16], a state-of-the-art static analysis framework for Java that does not support Android analysis at all. The core of the PAC engine is a whole-program pointer analysis that maintains the points-to information of all variables in a program, by manipulating the traditional pointer flow graph (PFG) [17] (Details on PFG are omitted as they are not essential for understanding PacDroid). As shown in Figure 2, the PAC engine acts as a delegation agent, providing facilities that enable feature handlers to *read* and *write* whole-program points-to information (expressed in PFG) in a decoupled manner. This design allows handlers to remain independent of the complex pointer analysis algorithm. Any changes to a variable's points-to set in a handler are *notified* to all other handlers by the engine, and any side effects from handlers on variables are *updated* through the engine to the whole-program points-to information (The detailed algorithms will be explained in Section III-B). Consequently, the effects of each handler reinforce one another, as illustrated in the case of Figure 1. Moreover, any improvements in handler or pointer analysis (e.g., precision) will benefit all handlers.

PacDroid supports four categories of Android feature handlers, as shown in Figure 2: ICC, Lifecycle, Misc, and other feature analyses. Below is a brief introduction to each category.

b) ICC analysis: A simple usage scenario of SendICC and IntentInfo analyses has been presented in Section II. The ReplyICC analysis complements the SendICC analysis, while the Message analysis handles another communication scheme among various components and the service component. The Message analysis is more complex than intent analysis, as it relies on the results of the latter. Unlike existing frameworks that handle features like intents and ICC separately, our PAC approach integrates them into one. As shown in Section II, PacDroid handles complex implicit intents to further resolve implicit ICC by using several handlers: the IntentExtra Handler (in Misc analysis) to parse extra information from intent objects, the IntentInfo Handler to process the strings used in implicit ICC, and the SendICC Handler to match these strings with configuration items in the AndroidManifest.xml to identify target components and resolve the data flows of implicit intents. In contrast, frameworks like FlowIccTA, DroidSafe and Amandroid resolve these features, including the implicit data flows of intents and ICC, separately, making it difficult to leverage one feature analysis to on-the-fly improve
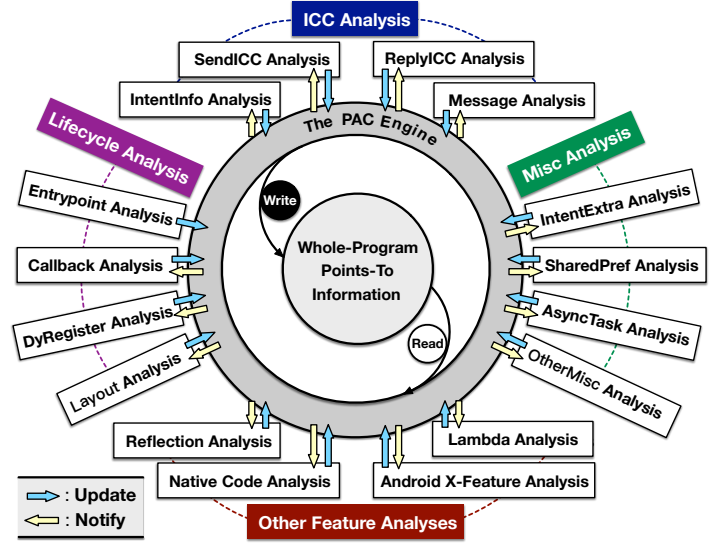

Fig. 2. Overview of PacDroid.

another, as explained in Section II. Specifically, FlowIccTA and DroidSafe first use IC3 and JSA to preprocess the strings involved in ICC, and then use the results to apply ICC analysis in later stages, which cannot handle implicit ICC cases like those in Figure 1. Similarly, Amandroid also fails to handle such cases effectively. It first preprocesses ICC-related strings, then extract the strings to resolve ICC using data flow analysis. When ICC cannot be resolved, it conservatively matches all other components like what P/Taint does, as described in Section II, leading to many false ICC flows.

c) Lifecycle analysis: It adds to the analysis the methods that are registered by calling Android's APIs in the code or configured in files. Specifically, Entrypoint analysis resolves and adds the event methods of Android components specified in `AndroidManifest.xml`, while Layout analysis identifies and incorporates the callback methods for activity components configured in `Layout.xml`. Callback analysis resolves callback methods (e.g., `onClick`) registered in the code (e.g., through a system call `setOnClickListener(c)`) and incorporates callback methods into the analysis. To handle callbacks, frameworks like FlowIccTA and Amandroid create a dummy main method as the analysis entry. When they resolve a callback method `cbm`, they add `cbm` to `main` and reanalyze the program starting from `main` to exploit more reachable callback methods possibly invoked from `cbm`. This process repeats until no new call graph edges are added. To avoid the redundant computations caused by reanalyzing a program constrained by the dummy main strategy, PacDroid adopts an on-the-fly scheme for callback analysis, enabled by PAC: whenever a new callback method is resolved, it is added to the set of methods to analyze. The PAC engine's pointer analysis solver then treats this callback method like any other newly reachable method found after resolving a normal callsite, thereby avoiding a full program reanalysis.

d) Misc analysis: It encompasses a set of Android analyses that are important but not included in the ICC and Lifecycle analyses. For example, we have seen the IntentExtra analysis in Section II. SharedPref analysis resolves another crucial data structure in Android, which involves two nested maps with value dependency inside, making it more complex than IntentExtra. AsyncTask analysis resolves asynchronous methods invoked from Android apps and adds them to the set of reachable methods to analyze. OtherMisc analysis includes the analyses to address or partially address other Android features like WebView, TextView Hint and Handler.

e) Other Feature analyses: As Android apps are encoded in Java, to deliver an effective analysis, we must also handle Java features that may affect the analysis, such as reflection [22], lambda expressions, and native code. The handling of these features in PacDroid is similar to P/Taint, which offers more advanced analyses for certain features than other frameworks. Thanks to the PAC approach, we can also integrate these Java feature analyses as feature handlers, like other analyses in PacDroid, as shown in Figure 2. The Android X-Feature analysis shown in the figure indicates that any new feature introduced by Android in the future, can be addressed as an additional handler of PacDroid. Note that PacDroid currently does not support implicit control flow analysis, which involves correlating path conditions. This decision is based on King's study [23], which found that such analysis can significantly overwhelm the results with many false positives [6].

B. The Pointer-Analysis Centric (PAC) Approach

We detail the two algorithms underpinning the PAC approach. Specifically, Algorithm 1 outlines the operation of the PAC engine, which builds upon the plugin system of Tai-e but includes enhancements by PacDroid for Android analysis. Algorithm 2 shows the core template for PacDroid's Android feature handlers, outlining how each handler functions and interacts with the engine. Currently, all 17 Android feature handlers of PacDroid were developed following this template.

For conciseness, Algorithm 1 shows the core of the PAC engine, excluding the pointer analysis handling, which is relatively standard [24]. It is assumed that pointer analysis runs on-the-fly within the current engine, processing each normal statement such as new, assign, load, and store, updating the related points-to sets for the involved variables, and adding them to the worklist. The key structure of the algorithm comprises two nested while loops, as shown from lines 3 to 8. The outer loop and the ONPHASEFINISH call in line 8, introduced by PacDroid, are designed to optimize specific Android analyses, which will be explained later.

Now, let us examine these two algorithms. Functions ONSTART() and ONFINISH(), called in lines 2 and 9 of Algorithm 1 and defined in lines 2 and 20 of Algorithm 2, manage the steps before and after a feature analysis implemented by its handler registered in PacDroid. Specifically, ONSTART() adds the entry point, specifying the methods of the app where the analysis begins, to the set of methods to analyze by calling ADDNEWMETHOD(). ONFINISH() reports analysis results.

Algorithm 1: PAC Engine

1   $workList \leftarrow []$, $handlers \leftarrow$ registered handlers
2   $allHandlers$.ONSTART()
3   **while** $workList$ *is not empty* **do**
4   **while** $workList$ *is not empty* **do**
5       $\langle p, newPTS \rangle \leftarrow worklist$.POP()
6   PROCESSCALL($p, newPTS$)
7       $handlers$.ONNEWPOINTSTOSET($p, newPTS$)
8     $handlers$.ONPHASEFINISH()
9   $handlers$.ONFINISH()

---

10   PROCESSCALL *(p, newPTS)*
11   **foreach** *callsite c: p.m(...)* **do**
12     $callee \leftarrow$ RESOLVECALL($c, newPTS$)
13   ADDNEWMETHOD($callee$)
14     $handlers$.ONNEWMETHOD($callee$)

---

15   ADDPOINTSTO *(p, newPTS)*
16    $workList \leftarrow workList \cup \langle p, newPTS \rangle$

Algorithm 2: PAC Handler for Android Feature X

1   Initialize *resolvedInfos*, *androidVars*

---

2   ONSTART *()*
3    $entrypoints \leftarrow$ components in AndroidManifest.xml
4   **foreach** $entrypoint \in entrypoints$ **do**
5   ADDNEWMETHOD($entrypoint$)

---

6   ONNEWMETHOD *(callee)*
7   **foreach** *callsite c: r = p.m(a_1...) in callee* **do**
8   **if** *c is related to Android Feature X* **then**
9      $androidVars \leftarrow androidVars \cup \{r, p, a_1...\}$

---

10   ONNEWPOINTSTOSET *(p, newPTS)*
11   **if** $p \in androidVars$ **then**
12     $resolvedInfo, \{\langle v_1, newPTS'_1 \rangle...\} \leftarrow$
      RESOLVEFEATUREX($p, newPTS$)
13   **foreach** $\langle v, newPTS' \rangle \in \{\langle v_1, newPTS'_1 \rangle...\}$ **do**
14   ADDPOINTSTO($v, newPTS'$)
15     $resolvedInfos \leftarrow resolvedInfos \cup resolvedInfo$

---

16   ONPHASEFINISH *()*
17    $\{\langle v_1, PTS'_1 \rangle...\} \leftarrow$ DELAYEDRESOLVEFEATUREX()
18   **foreach** $\langle v, PTS' \rangle \in \{\langle v_1, PTS'_1 \rangle...\}$ **do**
19   ADDPOINTSTO(v, PTS')

---

20   ONFINISH *()*
21   Report analysis result

If the popped variable p (and its points-to set *newPTS*) (line 5) serves as the receiver variable of a callsite c (line 11), the target *callee* of c will be resolved according to the objects pointed to by p, by calling RESOLVECALL() (line 12), and then adds the *callee* to the set of methods to analyze (line 13). Then, the newly added method will be notified to all handlers by calling ONNEWMETHOD() (line 14). Now, let us look into ONNEWMETHOD, defined in line 6 of Algorithm 2.

From the perspective of a handler for Android feature X (e.g., ICC, IntentInfo and others as in Figure 2), whenever a method *callee* is added, it checks whether a callsite *c* within *callee* is related to X and adds the involved variables of *c* into the set of X-related variables *androidVars* (line 9), maintained as a kind of global variable among all handlers (line 1).

Now we can move to line 7 of Algorithm 1, which calls ONNEWPOINTSTOSET*(p, newPTS)* to notify all handlers that the points-to set of variable *p* has possibly changed. All feature handlers should be aware of such changes to perform any necessary actions affected by the new points-to set, as defined in line 10 of Algorithm 2. Let us take an example to illustrate this function. Assuming that feature X is IntentInfo and *p* in line 11 refers to `i1` in line 4 of Figure 1, then *p* is an Android-related variable. So the method RESOLVEFEATUREINTENTINFO() is called in line 12 of the algorithm to handle the `new Intent()` statement in line 4 of Figure 1 and resolve the intent information. In this case, $\langle o1, Activity2.class \rangle$ is returned as the *resolvedinfo* in line 12 of the algorithm, indicating that intent `o1`'s target component is `Activity2`, which can then be used by other handlers. For another example, let X be SendICC, corresponding to the case in line 11 of Figure 1. Here, *p* in line 11 of the algorithm represents the hidden `this` variable invoking `getIntent()`. The analysis then searches through the previously resolved information *resolvedinfos* to identify which intent object's target component matches `this`'s type, i.e., `Activity2`. As a result, intent `o1` is identified, and the points-to information for `r1` in line 11 of Figure 1 is added accordingly. This operation is reflected in line 12 of the algorithm, where $\langle v_1, newPTS'_1 \rangle$ corresponds to $\langle r1, o1 \rangle$. Then, $\langle r1, o1 \rangle$ is updated in the PAC engine by calling ADDPOINTSTO() (see line 14 of Algorithm 2). As shown in line 15 of Algorithm 1, ADDPOINTSTO adds $\langle r1, o1 \rangle$ to the worklist, which further triggers the while loop in line 4 to continue processing.

The PAC engine incorporates an additional outer loop (lines 3 to 8) beyond the typical worklist algorithm loop (lines 4 to 7). This extra layer serves as an optimization for Android analysis. When analyzing certain Android features like Intent extras and SharedPreferences, there are often numerous store and load operations associated with key structures that influence app behaviors. For instance, consider the methods `putExtra(key,val)` and `getStringExtra(key)` in Figure 1, which we abbreviate as `put(k,v)` and `get(k)`.

As the worklist's pop order is often unpredictable, we cannot guarantee that all `get(k)` callsites are handled after their corresponding `put(k,v)` callsites. Thus, for soundness, the single-layer worklist algorithm needs to check all `get(k)` instances whenever a `put(k,v)` callsite is processed, including previously handled `get(k)` callsites that may not have found a matching `put(k,v)` at the time. The complexity increases with the presence of multiple points-to objects associated with k and the receiver variables of `put(k,v)` and `get(k)`.

The two-layer loop approach addresses this by delaying the propagation of handling `put(k,v)` and the resolution of `r = get(k)`. Specifically, it defers the propagation of objects from `v` to `r` until most `put(k,v)` callsites have been resolved (but not yet propagated), before handling the majority of `get(k)` callsites. This strategy offers two main advantages: it removes the need to search for `get(k)` during each `put(k,v)` callsite and enables consolidated propagation: for any `get(k)`, all corresponding `put(k,v1)`, `put(k,v2)`, etc., can be identified, and the objects pointed to by `v1`, `v2`, and others can be propagated to `r` in a single, efficient operation, reducing the original separate propagation cost.

Now we can understand the two-layer loops of Algorithm 1 with the above example. After the inner loop from lines 4 to 7, a fixpoint is reached, meaning all statements have been resolved and the points-to sets of variables have been propagated and updated, except for any unresolved `get(k)` callsites and resolved `put(k,v)` callsites that have not yet been propagated. This means that by the end of the inner loop, there is a high likelihood that all `put(k,v)` are resolved, and all corresponding `put(k,v)` values are available when handling a `get(k)`. The ONPHASEFINISH() function, called in line 8 of Algorithm 1 and defined in line 16 of Algorithm 2, performs the delayed resolution of all `get(k)` callsites and the propagation of all corresponding `put(k,v)` values by invoking DELAYEDRESOLVEFEATUREX() (line 17). In our example, this results in a set of pairs like $\langle r, o \rangle$ (where o is pointed to by `v`) being added to the worklist by calling ADDPOINTSTO() in line 19. Consequently, the worklist is not empty, and the outer loop in line 3 of Algorithm 1 continues to propagate the effects of updating $\langle r, o \rangle$ in the PAC engine, potentially influencing other handlers.

We conducted experiments on a set of real-world Android apps (collected in RQ2 of Section IV-C). This two-layer worklist optimization shows negligible advantage for simple apps. However, it notably accelerates the original one-layer worklist algorithm for 13 complex apps (e.g., those taking over 300 seconds), achieving an average speedup of 3.2X, with the largest average speedup for three apps being 8.3X.

IV. EVALUATION

In this section, we investigate the following research questions to evaluate PacDroid's effectiveness.

- **RQ1**: How does PacDroid perform compared to other state-of-the-art frameworks in terms of the trade-off between soundness and precision?
- **RQ2**: Is PacDroid robust (analysis crash rate) and fast (analysis speed) in analyzing real-world Android apps?
- **RQ3**: Can PacDroid successfully detect real vulnerabilities in real-world Android apps?

A. Experimental Settings

All experiments were conducted on an Intel(R) Xeon(R) E2488 3.2GHz machine with 20GB of memory.

a) State-of-the-art Frameworks: As PacDroid is a general static analysis framework that offers fundamental facilities for building taint analysis for Android apps, we consider well-recognized comparative frameworks, including FlowDroid,

TABLE I
SUMMARY OF BENCHMARK RESULTS. THESE BENCHMARKS CONSIST OF DIFFERENT SUITES, EACH CONTAINING MULTIPLE TEST APPS, AND EACH TEST APP MAY INCLUDE ZERO OR MORE MALICIOUS FLOWS. WE CATEGORIZE THE RESULTS AS FOLLOWS: A CORRECTLY IDENTIFIED MALICIOUS FLOW IS A TRUE POSITIVE (TP), AN INCORRECTLY IDENTIFIED FLOW IS A FALSE POSITIVE (FP), AND A MISSED MALICIOUS FLOW IS A FALSE NEGATIVE (FN).

Suite	Malicious Flows	FlowIccTA			P/Taint			Amandroid			PacDroid		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
Alias	1	1	1	0	1	3	0	1	2	0	1	3	0
AndroidSpecific	11	9	0	2	10	1	1	6	0	5	9	0	2
ArraysAndLists	4	4	4	0	4	4	0	1	4	3	4	4	0
Callbacks	18	18	2	0	9	2	9	8	9	10	18	4	0
Emulator	18	16	0	2	16	0	2	16	0	2	16	0	2
FieldAndObject	3	2	0	1	3	2	0	2	0	1	3	2	0
GeneralJava	22	15	4	7	20	6	2	11	3	11	21	4	1
IAC	19	6	0	13	3	0	16	5	2	14	11	0	8
ICC	41	22	2	19	28	14	13	35	11	6	40	1	1
Lifecycle	24	16	0	8	21	0	3	14	0	10	24	0	0
Reflection	9	8	0	1	3	0	6	1	0	8	9	0	0
Threading	6	3	0	3	4	0	2	4	0	2	6	0	0
UnreachableCode	0	0	3	0	0	4	0	0	4	0	0	4	0
UBCBench	24	17	1	7	13	5	11	11	6	13	23	4	1
Sum, Recall and Precision													
Sum	200	137	17	63	135	41	65	115	41	85	185	26	15
Recall r = TP/(TP + FN)		69%			68%			58%			93%		
Precision p = TP/(TP + FP)		89%			77%			74%			88%		
F-measure: 2rp/(r + p)		77%			72%			65%			90%		

IccTA, P/Taint, and Amandroid. DroidSafe is excluded as it does not support Android versions higher than 19 (the latest version is 34), and it fails to analyze all 24 programs in ICC-Bench, and has thus also been omitted from works [19]. For all the compared frameworks, we use their latest stable versions.

b) Benchmarks: To thoroughly evaluate the capabilities of various frameworks, we consider DroidBench [20] and ICC-Bench [18], commonly used in existing literature [4]–[8]. We use their latest versions to benefit from the additional Android features they support. Besides, we include the recently developed UBCBench [21] (2021 version), which covers a broader range of usage scenarios for Android features. We also incorporate eight cases involving ICC features or their practically adopted combinations that may lead to leaks, which are missed by the above benchmarks. Note that thirty out of 230 malicious flows, involving hard features such as dynamic class loading, which no frameworks can handle, or aspects like implicit flows that most frameworks do not address due to their focus on explicit flow analysis, have been excluded.

B. RQ1: Soundness and Precision Trade-off

The F-measure, calculated as the harmonic mean of recall and precision, effectively balances soundness and precision into a single value. It is commonly used as the primary metric to evaluate the soundness-precision trade-off in the literature on static analysis for Android [4], [5], [7]. Table I details the results of different frameworks on the benchmarks given in Section IV-A. Overall, PacDroid achieves an F-measure of 90%, surpassing all other frameworks, with the next best, FlowIccTA, at 77%. Below, we delve into soundness and precision, with soundness validated through a recall experiment.

a) Soundness (Recall): A recall experiment measures the proportion of real malicious flows detected by a static analysis among all real flows: higher recall values signify better soundness. As shown in Table I, the recall of PacDroid reaches 93%, significantly outperforming all other frameworks, with FlowIccTA at 69%, P/Taint at 68%, and Amandroid at 58%.

Apart from the enhanced handling of certain Android features, the recall advantage of PacDroid primarily stems from its PAC design. As explained in Section III, PAC allows the analysis results of one feature to influence the resolution of another. For example, in an ICC case from our benchmark, communication among activity and service components involves more complex features, including messages, messengers, and intents, than those depicted in Figure 1. As PacDroid supports message analysis, messenger analysis, and intent analysis, and incorporates their results into the ICC analysis via PAC, it is able to detect taint flows through such complex ICC scenarios, where other frameworks fail. A similar explanation applies to the SharedPreferences example from the Lifecycle suite in our benchmark. SharedPreferences is an important Android structure that involves two nested maps, with the outer map's value correlated with the inner map. Building the flow for SharedPreferences requires modeling both maps and tracking the value dependencies between them. Among all evaluated frameworks, only PacDroid can build such flow in analysis as PacDroid's PAC design allows the effects of all feature analyses (including map modeling, which involves string analysis) to be automatically propagated via pointer analysis, enabling it to identify taint flows through SharedPreferences effectively.

b) Precision: Soundness and precision often come at the cost of each other, with more conservative handling improving soundness but potentially reducing precision. We believe that PacDroid has made a better sweet spot between these two aspects for Android analysis, as evidenced by its highest F-measure of 90%, compared to the next best FlowIccTA at 77%. In terms of precision alone, PacDroid achieves a precision of 88%, which is slightly lower than FlowIccTA's 89%, but

TABLE II
RESULTS OF HANDLING CERTAIN CHALLENGING FEATURES

Category	Flows	FlowIccTA			P/Taint			Amandroid			PacDroid		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
Implicit	14	10	0	4	13	3	1	13	1	1	13	0	1
Hybrid	5	1	0	4	2	1	3	5	5	0	5	0	0
Sharepref	6	1	0	5	1	0	5	1	0	5	6	0	0
RPC	7	0	0	7	0	0	7	5	2	2	7	0	0

TABLE III
AVERAGE ANALYSIS TIME AND RATE OF CRASH OF EACH FRAMEWORK.

Tool		Average analysis time(s)		Rate of Crash
		Non-Intersect	Intersect	
FlowIccTA	IC3	116.47	N/A	90%
	FlowIccTA	263.46	252.37	27%
P/Taint		4243.72	4127.89	8%
Amandroid		651.09	458.18	6%
PacDroid		147.37	102.39	4%

significantly higher than P/Taint's 77% and Amandroid's 74%. The main reason why FlowIccTA is slightly more precise than PacDroid is that FlowIccTA is based on IFDS [25], [26], making it inherently context- and flow-sensitive, and unable to provide an insensitivity option. In contrast, PacDroid is context-sensitive and partially flow-sensitive (utilizing flow-insensitivity with SSA form to enhance its flow sensitivity). Since PacDroid is based on pointer analysis, it can choose context insensitivity as its default option, balancing efficiency with its ability to achieve precision even in a context-insensitive mode. For a fair evaluation, we ran PacDroid with its default context-insensitive option throughout the evaluation. However, it is important to note that PacDroid's precision can be directly improved to surpass FlowIccTA simply by selecting any context-sensitivity option available in Tai-e, without requiring any changes to the approach. Despite this, we must acknowledge that PacDroid cannot achieve the necessary precision for few cases where full flow-sensitivity is needed.

To validate PacDroid's ability to handle challenging features compared to other frameworks, we conduct an empirical study, and the results are shown in Table II. The study includes four categories: Implicit, Hybrid, Sharepref, and RPC, all of which involve implicit data flows, in the sense that Implicit and Hybrid include various complex cases involving implicit intents/ICCs, while the other categories address cases that cannot be resolved directly without first handling other features. For example, Hybrid includes cases that require resolving explicit, implicit, and replied intents together to identify the final ICC target. RPC contains cases dealing with remote calls to Service components, which require resolving intents first, followed by the IBinder features. In summary, PacDroid outperforms other frameworks in soundness, with a recall of 97%, compared to FlowIccTA's 38%, P/Taint's 50% and Amandroid's 75%. In addition, PacDroid exhibits excellent precision as FlowIccTA in this experiment, consistent to the precision results of Table I.

C. RQ2: Analysis Robustness and Speed

The benchmarks in RQ1 cover a wide range of diverse usage cases for Android apps, examining each framework's basic capabilities in handling Android features. However, these benchmarks are typically simple, allowing all frameworks to analyze them without crashes, and most complete the analysis within seconds. To better assess the robustness and analysis speed, we randomly selected 100 real-world apps from Androzoo [27], [28], dated between 2021 and 2023 (with an average about 1 million bytecode instructions). Each framework was run under its default settings, with three runs conducted to calculate average analysis time. The analysis time was capped

at 90 minutes for all frameworks. Only P/Taint exceeded this limit for half of the apps, with these instances recorded as 90 minutes. Below, we discuss the analysis robustness and speed in detail, as summarized in Table III.

a) Analysis Robustness: As a practical tool, the ability to analyze a real-world app without crashes is a basic requirement, indicating the tool's robustness. Since FlowIccTA requires the use of the separate tool IC3 for ICC analysis, its evaluation consists of two parts (i.e., IC3 and FlowIccTA). In our evaluation, PacDroid demonstrated the lowest crash rate at 4%, followed by Amandroid and P/Taint with crash rates of 6% and 8%, respectively. FlowIccTA and IC3 exhibited significantly higher crash rates of 27% and 90%. The crashes in PacDroid were primarily due to issues in parsing the APKs, specifically during the dex-to-IR transfer process, as well as a memory overflow in one program. Amandroid's crashes also stemmed from APK parsing issues, along with exceptions such as null pointer or array out-of-bounds errors and memory overflow. P/Taint's crashes occurred when generating Datalog fact files. Due to lack of maintenance, the newest IC3 still relies on an outdated version of FlowDroid to analyze strings inside Android apps, and replacing it with the new version of FlowDroid causes IC3 to fail to compile. Because FlowIccTA relies on IC3 to resolve intents in advance, its reliability heavily depends on the reliability of this separate tool, which is not ideal in design. FlowIccTA 's crashes were mainly caused by memory overflow and many null pointer exceptions that occurred while building the call graph in its callback analysis.

b) Analysis Speed: Different frameworks may crash on different real-world apps. To fairly examine the efficiency of each framework, we had them analyze the same set of apps that *all* frameworks could analyze without crashes and within the time limit, referred to as the "Intersect" group. The "Non-Intersect" group includes apps that *each* framework could analyze successfully within the time limit. Since IC3 crashed in 90% of the programs, we excluded it when collecting the Intersect group, resulting in 63 out of the 100 apps being included. Table III shows the average analysis times for these two groups. PacDroid demonstrated superior performance in both the Non-Intersect and Intersect groups. In the Non-Intersect group, PacDroid is about 2.6x faster than FlowIccTA, 4.4x faster than Amandroid, and 28.8x faster than P/Taint. In the Intersect group, PacDroid is about 2.5x faster than FlowIccTA, 4.5x faster than Amandroid, and 40.3x faster than P/Taint. It is challenging to pinpoint all causes of inefficiency,

but we can highlight some factors likely to slow down an analysis. For example, FlowIccTA reanalyzes the program after each iteration of detecting new callbacks to extend the call graph until no new edges are added, resulting in redundant computations, which are time-consuming. Amandroid, on the other hand, constructs multiple graph structures (e.g., ICFG, DFG, DDG, Summary Table) and employs algorithms based on them, adding complexity that can increase analysis time. P/Taint, built on Datalog, has to generate a large number of Datalog facts to execute, with its efficiency heavily dependent on the opaque Datalog engine, which may make it much slower than other frameworks.

To further evaluate the scalability stress of PacDroid concerning app size, we analyzed an additional 30 large and complex apps from Androzoo. Each app exceeds 1 million bytecode instructions, averaging 3.36 million, making them significantly larger than the previous dataset. Amandroid failed on 15 apps, with 12 cases attributed to timeouts, resulting in a scalability stress of 3.9 million bytecode instructions. Scalability stress is calculated as the average size of the apps a framework cannot analyze scalably. FlowIccTA failed on 12 apps, all due to memory shortages, with a scalability stress of 1.7 million. However, these results should be interpreted with caution, as FlowIccTA often yields unreliable outcomes in this experiment by resolving very few call graph edges in many instances. For example, for three apps in the 5-6 million range, it completed analyses in just 20 seconds but resolved only around 690 call graph edges on average. Notably, PacDroid does not exhibit these robustness issues, and only failed to analyze 9 apps, mainly due to memory limitations, with a scalability stress of 6.1 million. This larger app set exceeds P/Taint's scalability capabilities, leading to failures in 27 applications, mostly due to timeouts, so we do not compute its scalability stress. While PacDroid shows better efficiency and robustness than other frameworks, there is still room for scalability improvement. Considerable work has been done to optimize analysis efficiency [29]–[37], such as using heap snapshots to enhance scalability [30], [32]. We plan to explore further improvements in this area going forward.

D. RQ3: Real-World Vulnerability Detection

Let's evaluate the ability of each framework to detect real vulnerabilities in real-world Android apps. Fortunately, the recent works TaintBench [19] and UBCBench [21] provide a collection of various real vulnerabilities found in real-world apps. We consider all 39 apps from TaintBench and 6 out of 25 apps from UBCBench that involve privacy information leakage. The remaining 19 apps in UBCBench are excluded because they focus on detecting login encryption scenarios and do not provide precise ground truth through taint flows. In total, we have 250 labeled real taint flows across 45 real-world apps. Note that this does not necessarily mean there are no other real taint flows in those apps; the labeled ones are simply those verified by the studies [19], [21]. Consequently, these labeled taint flows cannot serve as a definitive metric for assessing the precision of different frameworks in real-

world scenarios. Therefore, in this section, we focus solely on soundness, specifically whether the frameworks can detect these real taint flows, which is the most critical aspect of taint analysis. As for frameworks, we observed that the number of taint flows reported by Amandroid varied across different runs on the same machine, with the same app, and under the same configuration settings. This inconsistency has also been noted by other works [21]. Thus, to maintain fairness despite this bug in Amandroid, we consider the highest number of taint flows reported by Amandroid from three runs for each app.

The results of PacDroid are highly promising, as detailed in Table IV, where it successfully detects 187 out of 250 real taint flows. This performance is significantly better than that of Amandroid, which detects 67 flows, FlowIccTA with 50 flows, and P/Taint with 50 flows, showcasing PacDroid's high potential in identifying real-world vulnerabilities. Notably, PacDroid detects 77 taint flows that all other frameworks miss. Of the 63 taint flows that PacDroid fails to detect, only 14 are detected by other frameworks, while the remaining 49 go undetected by any framework. Given the complexity and time-consuming nature of enumerating reasons for a framework's success or failure in detecting real vulnerabilities in real-world apps, we discuss below some representative scenarios where PacDroid succeeds while others fail, and where PacDroid fails but other frameworks partially detect.

For the former, two factors may contribute to PacDroid's better performance. First, PacDroid's PAC approach uniformly handles various Android features and their complex combinations, enhanced by the propagation of pointer analysis. For instance, the program *chulia* involves four taint flows with complex feature combinations, including lifecycle, ICC, intent extras, collections, appendToString, and others. PacDroid effectively manages these combinations and identifies all taint flows in this scenario, whereas other frameworks fail to detect them all. Second, PacDroid supports some more Android features than others. For example, the taint flow in program *xbot* involves the WebView feature of Android, which PacDroid partially supports while other frameworks do not.

Before discussing why PacDroid fails to detect some taint flows that other frameworks can, we first provide some background. Each framework, including PacDroid, models important methods in the Java JDK and Android standard library (android.jar) to enhance analysis effectiveness. However, in addition to the standard library, frameworks like Amandroid handle other library methods conservatively. For instance, they assume the values of a specified library method's parameters will flow to all its fields, such as f, and further to the fields of f. Similarly, FlowIccTA adopts conservative handling in the opposite direction. For example, for a store operation $o.f = g$, if g is tainted, not only will $o.f$ be tainted, but o will also be tainted, with the nesting level of flow expansion limited to a predefined number, such as five. In contrast, PacDroid does not apply such expansions to maintain precision, relying instead on real flow based on statements' or APIs' semantics to propagate values. This approach, while precise, may cause PacDroid to miss some taint flows. For example, in the

App Name	Exp.Flows	FlowIccTA	P/Taint	Amandroid	PacDroid
flashlight	7	Crash	Timeout	1	7
phonemonitor	12	4	8	12	12
win7imulator	3	Crash	3	1	3
win7launcher	11	0	8	3	11
flappybird	7	Crash	6	1	7
hzpermispro	7	Crash	6	1	7
backflash	13	13	0	0	13
beita	3	Crash	0	0	1
cajino	12	8	2	9	8
chat	12	9	3	1	10
chulia	4	0	0	1	4
death	1	Crash	1	0	1
dsencrypt	1	0	0	0	1
exprespam	1	0	0	1	1
fakeappstore	3	0	0	0	3
fakebank	5	0	0	1	3
fakedaum	2	0	0	1	1
fakemart	2	0	0	1	0
fakeplay	2	0	0	0	0
faketaobao	4	0	0	4	4
godwon	6	0	0	0	0
hummingbad	2	0	0	0	0
jollyserv	1	1	0	0	0
overlay	4	Crash	0	0	1
overlay2	7	0	0	0	5
phospy	2	2	1	0	2
proxy	17	6	2	1	9
remote	17	Crash	0	0	15
repane	1	0	0	0	0
roidsec	6	0	0	6	4
samsapo	4	0	0	0	2
save	25	Crash	5	11	25
scipiex	3	0	0	0	1
slocker	5	0	0	1	5
sms_google	4	0	0	1	2
sms_send	6	3	2	2	6
smssend	5	1	2	2	2
smssilience	2	Crash	0	0	0
smsstealer	5	0	0	0	3
stels	3	1	1	0	2
tetus	2	2	0	0	2
the	1	0	0	0	0
threatjapan	2	0	0	2	1
vibleaker	4	Crash	0	2	0
xbot	3	0	0	1	3
Sum	250	50/250	50/250	67/250	187/250

program *cajino*, FlowIccTA and Amandroid detect 2 out of 4 taint flows each, sinking to calls of `putObject` in the com.baidu package, while PacDroid misses these flows due to its lack of conservative modeling.

Note that FlowIccTA can resolve intents on its own, though it can be further enhanced by IC3. Due to IC3's exceptionally high crash rate (shown in Table III), we ran FlowIccTA independently to analyze real-world apps. However, as shown in Table IV, FlowIccTA still exhibited a relatively high crash rate (24%) compared to other frameworks, consistent with the results in Table III. Additionally, as previously mentioned, Amandroid reports different results for the same real-world app under identical settings in different runs. These phenomena underscore the need for reliable research tools in practice.

E. Discussion

Based on the experimental results, PacDroid exhibits notable advantages over established and widely used frameworks, including FlowIccTA, P/Taint and Amandroid, both in extensive benchmarks and real-world applications. This highlights PacDroid's potential as a promising research tool with capabilities for identifying real taint flows in practical scenarios. Nevertheless, as a framework aimed at more comprehensive Android application analyses, PacDroid still has room for further improvement.

Firstly, there are challenging or emerging Android features, such as dynamic class loading, Jetpack Compose [38], and WebView [39], which are either not yet supported or inadequately addressed by PacDroid. Fortunately, PacDroid's PAC method facilitates the flexible addition or enhancement of these features through plugin handlers, allowing for extensibility.

Secondly, recent research suggests that integrating large language models (LLMs) could enhance security issue detection [40]. Given that PacDroid's PAC approach organizes its handling of Android features in a modular fashion, certain features can be further decomposed or recombined into new handlers. This approach may facilitate fine-tuning interactions with LLMs to possibly generate more effective prompts.

Lastly, PacDroid is inherently a versatile static analysis framework. Its PAC approach enables taint analysis, implemented as a handler, to collaborate with Android feature handlers and pointer analysis. This design not only supports the current security analysis but also paves the way for future extensions. By developing new PAC handlers, PacDroid can be adapted to perform other client analyses of Android apps, such as bug detection [41] and program comprehension [42].

V. Related Work

We review related works in the following three categories.

a) Static Analysis Frameworks for Android: We have discussed various static analysis frameworks for Android throughout the paper, including FlowDroid, IccTA, P/Taint, Amandroid, and DroidSafe. Below, we clarify the key differences between frameworks that use pointer analysis, i.e., P/Taint and DroidSafe, to underscore PacDroid's advantages.

Unlike PacDroid, which uniformly resolves all Android features through pointer analysis and successfully demonstrates its feasibility, P/Taint either omits handling certain Android features (e.g., SharedPreferences) or uses non-pointer-analysis approaches to resolve them (e.g., conservative ICC resolution), resulting in worse soundness and precision than PacDroid. Moreover, as P/Taint is declarative and built using Datalog, optimizing feature analysis in P/Taint is hard because it utilizes an opaque Datalog engine to interpret rules, lacking the capability to manipulate specific data structures and execution strategies [43]. This makes it significant slower than PacDroid.

As for DroidSafe, it only partially employs pointer analysis for feature handling. Like FlowIccTA and Amandroid, DroidSafe relies on a separate tool for extracting strings before resolving critical Android features such as intents and ICC. As illustrated in Sections II and IV, this design not only incurs extra time but also ties the framework's reliability to these pre-analysis tools. Moreover, DroidSafe lacks PacDroid's capability to leverage results from one feature analysis to

enhance another for resolving more program behaviors, as DroidSafe does not handle all features under the same on-the-fly pointer analysis algorithm as PacDroid. Additionally, unlike PacDroid, which uses pointer analysis for modeling critical library methods, DroidSafe simplifies these methods by manually rewriting their internal code. Finally, none of these frameworks, like PacDroid, presents the PAC approach to showcase how to imperatively and cohesively utilize pointer analysis to manage various Android features uniformly, achieving both extensibility and performance.

CHEX [9] employs an app-splitting approach to model the execution of multiple entry points, facilitating global data-flow analysis. But it focuses solely on the analysis of Android entry points and does not support more complex Android features.

HornDroid [44] utilizes Horn clause-based static analysis for Android applications, but it complicates the process by needing security attributes to be converted into Horn clauses. The framework only addresses apps with activities, and handles ICC solely for explicit intents.

Some frameworks, despite having their own methods for analyzing Android, still rely on more fundamental frameworks for tasks like identifying entry points, building control flow graphs and call graphs, and conducting data flow analysis [45]–[49]. For example, DidFail [45] depends on Epicc [50] for resolving intent information and on FlowDroid for data flow analysis. R-droid [48] employs a slicing-based analysis to generate data-dependent statements for arbitrary points of interest within an Android app. It follows FlowDroid and Amandroid to analyze lifecycle and utilizes DroidSafe's lightweight framework model to analyze apps.

b) Static Analysis for Specific Android Features: For intent analysis, ICCBot [51] models UI relationships from fragments to activities during the resolution process. ARIA [52] uses a two-pass method: first, it applies data flow analysis to resolve intents and labels those that cannot be resolved initially. It then performs a new analysis to resolve ICC, leveraging its results to resolve previously unresolved intents. In contrast, PacDroid integrates features such as intents, ICC, fragments, and more, resolving them on the fly with pointer analysis in a single analysis, thus deriving enhanced intent information from the aggregated results of all analyzed features. RAICC [53] analyzes atypical intents (e.g., PendingIntent) by parsing them into standard ICC methods (e.g., startActivity) through instrumentation, resulting in a new APK that serves as a preprocessor for other static analyses. PacDroid can also utilize the generated APK to handle atypical intents.

Several studies focus on inter-app communication (IAC), extending ICC analysis across multiple apps [10], [11], [13], [54]. For instance, IAFDroid [13] identifies IAC-related entry and exit points within an app and then performs taint analysis between apps using FlowDroid. No general frameworks, including FlowIccTA, P/Taint, Amandroid, DroidSafe and PacDroid, offer comprehensive IAC analysis. To address this, PacDroid currently employs ApkCombiner [55] to merge multiple apps into a single app for IAC analysis.

c) Static Analysis for Android Security: Numerous studies employ static analysis to identify and address various security issues, including intent-based attack surfaces [12], [54], permission problems [56]–[58], performance concerns [59], [60], and others [61]–[67]. These studies typically use or extend existing general Android analysis frameworks (e.g., FlowDroid) to better tackle security challenges. For example, SEALANT [12] addresses issues like intent spoofing and unauthorized intent reception, helping users block potential attacks by automatically identifying vulnerable ICC paths between applications. Permission Tracer/Tainter [57] focuses on the opacity of custom permission management, which can pose privacy and security risks. IMGDroid [60] analyzes improper practices in image loading and processing, considering both performance and quality aspects. In the future, PacDroid can be extended by conveniently adding handlers to develop and enhance research on these security issues.

VI. CONCLUSIONS

We present PacDroid, a new static analysis framework for vulnerability detection in Android apps. Its core novelty lies in adopting a pointer-analysis-centric approach, a simple yet effective method for cohesively handling various Android features in a unified way. Extensive experiments demonstrate that PacDroid not only achieves a superior trade-off between soundness and precision (F-measure) compared to state-of-the-art frameworks, but also surpasses them in analysis speed and robustness—achieving all of these simultaneously is a big challenge for a fundamental static analysis framework. Moreover, PacDroid successfully detects dozens of real vulnerability flows in real-world Android apps, where all other frameworks fall, indicating its potential for practical use.

Although our primary focus in this paper is on security, PacDroid actually offers a set of fundamental facilities such as alias and points-to information for every variable and the call graph of an Android app, which serve as the basis for various analysis clients. We hope these capabilities will help establish PacDroid as a foundational framework for developing a range of applications, such as bug detection, program understanding, and optimization tools for Android in the future.

VII. ACKNOWLEDGMENTS

REFERENCES

[1] StatCounter, "Mobile operating system market share worldwide," https://gs.statcounter.com/os-market-share/mobile/worldwide.

[2] Statista, "Number of smartphone users worldwide," https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/.

[3] M. Talal, A. Zaidan, B. Zaidan, M. Alsalem, F. Jumaah, and M. Alaa, "Comprehensive review and analysis of anti-malware apps for smartphones," *Telecommunication Systems*, vol. 72, pp. 285–337, 2019.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM sigplan notices*, vol. 49, no. 6, pp. 259–269, 2014.

[5] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 280–291.

[6] N. Grech and Y. Smaragdakis, "P/taint: Unified points-to and taint analysis," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[7] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 3, pp. 1–32, 2018.

[8] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe." in *NDSS*, vol. 15, no. 201, 2015, p. 110.

[9] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 229–240.

[10] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "Covert: Compositional analysis of android inter-app permission leakage," *IEEE transactions on Software Engineering*, vol. 41, no. 9, pp. 866–886, 2015.

[11] A. Bosu, F. Liu, D. Yao, and G. Wang, "Collusive data leak and more: Large-scale threat analysis of inter-app communications," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 71–85.

[12] Y. K. Lee, J. Y. Bang, G. Safi, A. Shahbazian, Y. Zhao, and N. Medvidovic, "A sealant for inter-app security holes in android. in 2017 ieee/acm 39th international conference on software engineering (icse)," *IEEE, 312ś323*, 2017.

[13] B. Wang, C. Yang, and J. Ma, "Iafdroid: Demystifying collusion attacks in android ecosystem via precise inter-app analysis," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 2883–2898, 2023.

[14] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 77–88.

[15] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *International Static Analysis Symposium*. Springer, 2003, pp. 1–18.

[16] T. Tan and Y. Li, "Tai-e: A developer-friendly static analysis framework for java by harnessing the good designs of classics," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1093–1105.

[17] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "Precision-guided context sensitivity for pointer analysis," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[18] https://github.com/fgwei/ICC-Bench.

[19] L. Luo, F. Pauck, G. Piskachev, M. Benz, I. Pashchenko, M. Mory, E. Bodden, B. Hermann, and F. Massacci, "Taintbench: Automatic real-world malware benchmarking of android taint analyses," *Empirical Software Engineering*, vol. 27, pp. 1–41, 2022.

[20] https://github.com/secure-software-engineering/DroidBench/tree/develop.

[21] J. Zhang, Y. Wang, L. Qiu, and J. Rubin, "Analyzing android taint analysis tools: Flowdroid, amandroid, and droidsafe," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4014–4040, 2021.

[22] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 2, pp. 1–50, 2019.

[23] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'em, can't live without 'em," in *Information Systems Security: 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings 4*. Springer, 2008, pp. 56–70.

[24] Y. Smaragdakis, G. Balatsouras *et al.*, "Pointer analysis," *Foundations and Trends® in Programming Languages*, vol. 2, no. 1, pp. 1–69, 2015.

[25] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1995, pp. 49–61.

[26] M. Sagiv, T. Reps, and S. Horwitz, "Precise interprocedural dataflow analysis with applications to constant propagation," *Theoretical Computer Science*, vol. 167, no. 1-2, pp. 131–170, 1996.

[27] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 468–471. [Online]. Available: https://doi.org/10.1145/2901739.2903508

[28] M. Alecci, P. J. R. Jiménez, K. Allix, T. F. Bissyandé, and J. Klein, "Androzoo: A retrospective with a glimpse into the future," in *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 2024, pp. 389–393.

[29] S. Banerjee, S. Cui, M. Emmi, A. Filieri, L. Hadarean, P. Li, L. Luo, G. Piskachev, N. Rosner, A. Sengupta, O. Tripp, and J. Wang, "Compositional taint analysis for enforcing security policies at scale," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1985–1996. [Online]. Available: https://doi.org/10.1145/3611643.3613889

[30] M. Benz, E. K. Kristensen, L. Luo, N. P. Borges, E. Bodden, and A. Zeller, "Heaps'n leaks: how heap snapshots improve android taint analysis," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1061–1072. [Online]. Available: https://doi.org/10.1145/3377811.3380438

[31] J. Wang, Y. Wu, G. Zhou, Y. Yu, Z. Guo, and Y. Xiong, "Scaling static taint analysis to industrial soa applications: a case study at alibaba," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1477–1486. [Online]. Available: https://doi.org/10.1145/3368089.3417059

[32] N. Grech, G. Fourtounis, A. Francalanza, and Y. Smaragdakis, "Shooting from the heap: ultra-scalable static analysis with heap snapshots," ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 198–208. [Online]. Available: https://doi.org/10.1145/3213846.3213860

[33] Y. Gui, D. He, and J. Xue, "Merge-replay: Efficient ifds-based taint analysis by consolidating equivalent value flows," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 319–331.

[34] D. He, H. Li, L. Wang, H. Meng, H. Zheng, J. Liu, S. Hu, L. Li, and J. Xue, "Performance-boosting sparsification of the ifds algorithm with applications to taint analysis," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 267–279.

[35] W. Ma, S. Yang, T. Tan, X. Ma, C. Xu, and Y. Li, "Context sensitivity without contexts: A cut-shortcut approach to fast and precise pointer analysis," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 539–564, 2023.

[36] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "Scalability-first pointer analysis with self-tuning context-sensitivity," in *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 129–140.

[37] T. Tan, Y. Li, and J. Xue, "Efficient and precise points-to analysis: modeling the heap by merging equivalent automata," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 278–291.

[38] https://developer.android.com/courses/pathways/compose?hl=zh-cn.

[39] J. Hu, L. Wei, Y. Liu, and S.-C. Cheung, "ωtest: Webview-oriented testing for android applications," in *Proceedings of the 32nd ACM*

SIGSOFT International Symposium on Software Testing and Analysis, 2023, pp. 992–1004.

[40] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 474–499, 2024.

[41] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 408–419.

[42] E. Aghajani, G. Bavota, M. Linares-Vásquez, and M. Lanza, "Automated documentation of android apps," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 204–220, 2019.

[43] D. Helm, F. Kübler, M. Reif, M. Eichberg, and M. Mezini, "Modular collaborative program analysis in opal," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 184–196.

[44] S. Calzavara, I. Grishchenko, and M. Maffei, "Horndroid: Practical and sound static analysis of android applications by smt solving," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 47–62.

[45] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, 2014, pp. 1–6.

[46] J. Zhang, C. Tian, and Z. Duan, "An efficient approach for taint analysis of android applications," *Computers & Security*, vol. 104, p. 102161, 2021.

[47] A. Tiwari, S. Groß, and C. Hammer, "Iifa: modular inter-app intent information flow analysis of android applications," in *Security and Privacy in Communication Networks: 15th EAI International Conference, SecureComm 2019, Orlando, FL, USA, October 23–25, 2019, Proceedings, Part II 15*. Springer, 2019, pp. 335–349.

[48] M. Backes, S. Bugiel, E. Derr, S. Gerling, and C. Hammer, "R-droid: Leveraging android app analysis with static slice optimization," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 129–140.

[49] L. Luo, G. Piskachev, R. Krishnamurthy, J. Dolby, E. Bodden, and M. Schäf, "Model generation for java frameworks," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2023, pp. 165–175.

[50] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective {Inter-Component} communication mapping in android: An essential step towards holistic security analysis," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 543–558.

[51] J. Yan, S. Zhang, Y. Liu, J. Yan, and J. Zhang, "Iccbot: fragment-aware and context-sensitive icc resolution for android applications," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 105–109.

[52] N. Abolhassani and W. G. Halfond, "A component-sensitive static analysis based approach for modeling intents in android apps," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2023, pp. 97–109.

[53] J. Samhi, A. Bartel, T. F. Bissyandé, and J. Klein, "Raicc: Revealing atypical inter-component communication in android apps," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1398–1409.

[54] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, 2011, pp. 239–252.

[55] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Apkcombiner: Combining multiple android apps to support inter-app analysis," in *ICT Systems Security and Privacy Protection: 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings 30*. Springer, 2015, pp. 513–527.

[56] B. F. Demissie, M. Ceccato, and L. K. Shar, "Security analysis of permission re-delegation vulnerabilities in android apps," *Empirical Software Engineering*, vol. 25, no. 6, pp. 5084–5136, 2020.

[57] J. Gamba, Á. Feal, E. Blazquez, V. Bandara, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "Mules and permission laundering in android: Dissecting custom permissions in the wild," *IEEE Transactions on Dependable and Secure Computing*, 2023.

[58] S. Wang, Y. Wang, X. Zhan, Y. Wang, Y. Liu, X. Luo, and S.-C. Cheung, "Aper: evolution-aware runtime permission misuse detection for android apps," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 125–137.

[59] M. Jalili and F. Faghih, "Static/dynamic analysis of android applications to improve energy-efficiency," in *2022 CPSSI 4th International Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*. IEEE, 2022, pp. 1–8.

[60] W. Song, M. Han, and J. Huang, "Imgdroid: Detecting image loading defects in android applications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 823–834.

[61] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 73–84.

[62] Z. Meng, Y. Xiong, W. Huang, L. Qin, X. Jin, and H. Yan, "Appscalpel: Combining static analysis and outlier detection to identify and prune undesirable usage of sensitive data in android applications," *Neurocomputing*, vol. 341, pp. 10–25, 2019.

[63] M. Tileria and J. Blasco, "Watch over your tv: a security and privacy analysis of the android tv ecosystem," *Proceedings on Privacy Enhancing Technologies*, 2022.

[64] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why eve and mallory love android: An analysis of android ssl (in) security," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 50–61.

[65] J. Samhi, L. Li, T. F. Bissyandé, and J. Klein, "Difuzer: Uncovering suspicious hidden sensitive operations in android apps," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 723–735.

[66] D. Lai and J. Rubin, "Goal-driven exploration for android applications," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 115–127.

[67] Z. Chen, J. Liu, Y. Hu, L. Wu, Y. Zhou, Y. He, X. Liao, K. Wang, J. Li, and Z. Qin, "Deuedroid: Detecting underground economy apps based on utg similarity," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 223–235.