

Detection of Device Triggerable Vulnerabilities in Android Companion Apps through Interactive Triaging

Aditya Vardhan Padala
Purdue University
United States
padalaa@purdue.edu

Saurabh Bagchi
Purdue University
United States
sbagchi@purdue.edu

Aravind Machiry
Purdue University
United States
amachiry@purdue.edu

Abstract

We are increasingly relying on Internet of Things (IoT) devices for most of our daily tasks. However, IoT devices are riddled with security vulnerabilities. Most IoT devices have an associated Mobile Companion App (CApp) that enables users to control and access these devices remotely in a user-friendly manner. CApps are manufactured by the device vendors, and they trust these IoT devices. This blind trust results in DTM vulnerabilities, where attackers can compromise CApps by exploiting the corresponding IoT device. In this paper, we present REARFIND, the first static analysis technique to find DTM vulnerabilities in CApps. We also design an interactive triaging technique to reduce false positive alerts through user feedback. Our evaluation shows that REARFIND was able to find 5 new (*i.e.*, previously unknown) DTM vulnerabilities. Our interactive triaging technique was able to reduce the false positives by 12%.

CCS Concepts

• Security and privacy → Software and application security.

Keywords

taint analysis, vulnerability detection, alert ranking

ACM Reference Format:

Aditya Vardhan Padala, Saurabh Bagchi, and Aravind Machiry. 2025. Detection of Device Triggerable Vulnerabilities in Android Companion Apps through Interactive Triaging. In *The 40th ACM/SIGAPP Symposium on Applied Computing (SAC '25)*, March 31-April 4, 2025, Catania, Italy. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3672608.3707956>

1 Introduction

Our dependence on Internet of Things (IoT) devices has significantly increased, controlling various aspects of our daily lives, including health [15] and homes [3]. The adoption of these devices has seen rapid and extensive growth, with an estimated count of over 50 billion devices [1]. Unfortunately, these devices are riddled with security vulnerabilities [4, 39].

IoT devices typically come with a Companion App (CApp), a mobile app that controls several aspects, such as device configuration, firmware updates, device status, and telemetry collection. CApps are responsible for translating user actions (*i.e.*, button press) into a command encapsulated in a format expected by the target IoT

device. Consequently, many techniques [10, 38, 44] use CApps to test IoT devices.

Problem. CApps are developed by the same organization that manufactured the IoT device. Consequently, most CApps *blindly trust the data received from the devices*. This blind trust on data received from IoT devices results in the *Device to Mobile (DTM) vulnerabilities*,

i.e., attackers on a compromised IoT device can gain control of the CApp and consequently compromise the target Smartphone. Listing 1 shows a real DTM vulnerability, where the SSID of IoT devices (indicated by ⚡) is eventually (flows indicated by ←) used in a JavaScript call (indicated by 📄). An attacker controlling an IoT device can change its SSID to JavaScript code (*e.g.*, Listing 1)

and make the above CApp execute it, resulting in Cross Site Scripting (XSS) attack. These DTM vulnerabilities could have severe security impact because CApps are often overprivileged, *e.g.*, have full network access. Furthermore, previous studies [2, 41] show that CApps are often developed with poor development practices and are prone to contain vulnerabilities. Existing vulnerability detection works [2, 31] on CApps only focus on generic vulnerabilities and privacy leaks. Consequently, as we show in §5, these techniques fail to find such vulnerabilities.

This paper explores the security of CApps from the perspective of the IoT device.

Specifically, we focus on identifying DTM vulnerabilities in CApps. Dynamic analysis [18], specifically random testing [29], is shown to be effective at vulnerability detection. However, effective testing [28] of Android apps is a known hard problem [12].

We design REARFIND, a static analysis technique based on taint tracking to detect DTM vulnerabilities. Given an Android app (*i.e.*, APK[37]), we use call-graph-based analysis [20] to detect taint sources and instrument the app with sink markers. We enhance existing state-of-the-art flow tracking tools, specifically FLOWDROID [16] with our sources and sinks to detect DTM vulnerabilities. It is well known that static analysis techniques are prone to false positives. To handle this, we design an interactive triaging technique wherein we learn false positive flows from user input and present users with alerts that are less likely to be false positives. Our evaluation shows that REARFIND is able to identify 5 new (previously unknown) DTM vulnerabilities across 8 CApps.

We show that our interactive triaging technique is able to reduce false positives on average by 12%.

Furthermore, we also demonstrate the generality of our interactive triaging approach by evaluating it on TAINTBENCH (a well-known taint analysis benchmark) on which we reduced false positives by 12%.

In summary, we make the following contributions:



```

1 public void onReceive(Context context, Intent intent) {
2     WifiManager wm = (WifiManager) context.getSystemService("wifi");
3     if (intent.getAction().equals("android.net.wifi.SCAN_RESULTS")) {
4         BAB.this.apList = wm.getScanResults();
5         ...
6         List<String> result = new ArrayList<>();
7         for (int i = 0; i < BAB.this.apList.size(); i++) {
8             ScanResult scanResult = (ScanResult) BAB.this.apList.get(i);
9             ...
10            result.add(scanResult.SSID);
11        }
12        BAB.this.callWebInterface("setSsid", BAB.this.gson.toJson(result));
13        ...

```

Listing 1: A DTM vulnerability in REDACTED resulting in Cross Site Scripting (XSS)

- We provide the first insight into DTM vulnerabilities and design a static analysis technique to detect them.
- We design an interactive triaging technique of vulnerability alerts that learns from user feedback and reduces false positives, *i.e.*, 12% (on average).
- We implemented REARFIND encompassing our technique and demonstrate its effectiveness by detecting 5 new (previously unknown) DTM.
- We will make our implementation and dataset open-source¹ to enable future research on DTM vulnerabilities.

1

2 Background and Threat Model

IoT devices typically come with a Companion App (CApp), a mobile app that lets users control the IoT device remotely. As shown in Figure 1, CApps communicate with the target IoT device through standard network protocols, specifically Ethernet or Bluetooth. Most of the existing works [38], [10], [44] focus on testing IoT devices through CApp. Specifically, they use or gather information from CApp to test the target device. In this work, we invert that and test the security of the CApp as the IoT devices are often a prime attack vector.

As we mentioned in §1, there is a prevalence of security vulnerabilities in IoT devices. Furthermore, the lack of exploit mitigation techniques [45] makes it easy to exploit these vulnerabilities. In this work, we focus on the impact of IoT devices on CApp and, consequently, on the Smartphone. Specifically, *we are interested in detecting external device triggerable vulnerabilities (i.e., DTM) in the companion app that runs on a Smartphone.*

Threat model. The Figure 1 shows our threat model. We assume that *attackers have complete control of the software stack of IoT devices (e.g., through code injection vulnerability).* The attacker can make the device send arbitrary data through network channels, *e.g.*, TCP/IP or Bluetooth. The attacker knows that the device is communicating with a Smartphone through a CApp, and the attacker has access to the CApp, *i.e.*, can reverse engineer it. The *goal of the attacker is to gain complete control of the Smartphone* through interactions (*i.e.*, network messages) through CApp.

¹github.com/rearfind/rearfindpublic

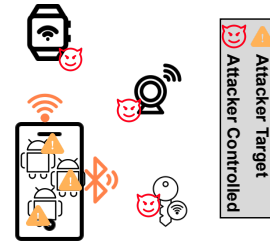


Figure 1: Threatmodel of REARFIND

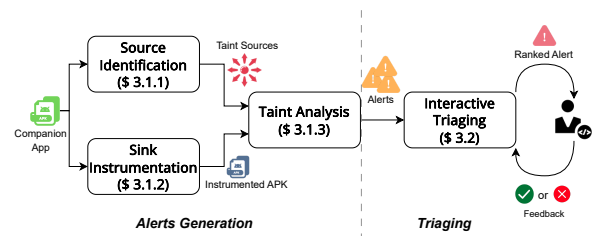


Figure 2: Overview of REARFIND

3 Design

We plan to use static analysis, specifically taint tracking or data flow tracking, to find DTM vulnerabilities. However, as mentioned in §1, static taint tracking is prone to false positives. We handle this by using an interactive triaging technique (§3.2), where we learn false positive flows from developer interaction and provide alerts that are more likely to be true positives. The Figure 2 shows the overview of REARFIND that has the following two phases.

3.1 Alerts Generation

In this phase, we aim to generate DTM vulnerability alerts. We model DTM vulnerability identification as the problem of taint analysis, where the taint source (*i.e.*, that produces untrusted/attacker-controlled data) is an external device and important operations in the companion app constitute sensitive sinks.

There are several existing tools that perform taint (or flow) tracking on Android apps. However, these tools need to be configured with our sources and sinks.

3.1.1 Identifying Device Receive Functions (Taint Sources). It is well-known [38, 44] that most companion apps communicate with the target device through TCP/UDP or Bluetooth. Consequently, we consider all the functions that can receive data through TCP/UDP or Bluetooth as taint sources. An app can receive data through standard SDK methods or native code via JNI (Java Native Interface)[34]. The Listing 2 shows examples of these two cases.

Finding Receiver Functions: We analyzed the entire Android SDK and collected all methods that can receive data through above protocols *e.g.*, `Ljava/net/URLConnection;.getInputStream` and we found 82 methods that deal with receiving data, we call these leaf level

```

1 private InputStream handleServerResponse(HttpURLConnection connection)
2 throws IOException {
3     int $i0 = connection.getResponseCode();
4     if ($i0 == 200) {
5         InputStream $r2 = *jconnection.getInputStream();
6         return $r2;
7     }
8     ...
9 }
10
11 public static int GetMotionDetectConfig
12 (
13     int i,
14     MotionDetectConfig motionDetectConfig
15 ) {
16     HashMap hashMap = new HashMap();
17     hashMap.put("cfgType", 1);
18     String jsonObject = new JSONObject(hashMap).toString();
19     Response response = new Response();
20     int sendCommand = *nREDACTEDJni.sendCommand(i, 24033, jsonObject,
21     ↳ response, Priority.WARN_INT);
22     Log.m21982b("", "GET_MOTION_DETECT_CONFIG:" + response.resp);
23     ...
24     JSONObject jsonObject2 = new JSONObject(response.resp*n);
25     ...
26 }

```

Listing 2: Example illustrating two types of taint sources, i.e., Standard SDK methods (*_j) and JNI methods (*_n)

methods. Previous works [38, 44] show that APKs usually wrap the receive functionality into application-specific wrapper functions, which also take care of sanitization. Listing 2 shows an example of such method `handleServerResponse` that returns an `InputStream`, which we consider as tainted data since the data is from an TCP source. We consider such application-specific wrapper functions (that call these leaf-level functions) as taint sources.

Identifying JNI Sources: Apps can also receive data through JNI. Chenxiong Qian et al. [11] showed that tracking data through JNIs can help find info leaks through the shared objects. To handle this, for each app, we perform a static call-graph analysis [8] of all native libraries to identify all native methods (fn) that can read from a socket. We consider all JNI methods that can reach any function in fn as taint sources.

3.1.2 Instrumenting Companion App with Sinks. For sinks, we consider all operations that can affect the Smartphone (Category 1) or the execution of the companion app (Category 2). We consider all methods that affect the system resources (i.e., file system, process control) as Category 1 sinks, e.g., `java.net.URLConnection`, `java.io.OutputStream.getOutputStream()`.

We identified a list of 235 methods by referring to prior works [27] [16] [33].

For Category 2, we consider all sensitive operations in the app, such as array indexing, division, etc. We also consider all variables controlling loops as sensitive, as they can be used to cause denial of service attacks. We instrument all sensitive operations by adding a call to a dummy function (i.e., `byteSinkClass.bytesinkMet(...)`) and passing the sensitive variable as an argument. Listing 3 shows an example of this where the index used to access the array in line 12 is instrumented before the access, and the variables controlling the loop exit in line 8 are also instrumented with respective calls to their sink methods. This enables us to have a generic mechanism to specify sinks, i.e., argument to a function.

```

1 private void modRequest(String cc, Context context) {
2     String[] $r3 = cc.split(";");
3     int i0 = 0;
4     while (true) {
5         int $i1 = $r3.length;
6         *flashintSinkClass.intsinkMet(i0);
7         *flashintSinkClass.intsinkMet($i1);
8         if (i0 >= $i1) { return; }
9         *sinkClass.sinkMet(i0);
10        String cc2 = $r3[i0];
11        ...
12        sendSMS("79262000900", $r4.append($r1).toString());
13        i0++;
14    }
15 }

```

Listing 3: Example demonstrating taint sinks instrumentation (*_s)

3.1.3 Taint Analysis. We perform Static Taint Analysis [9] to track the flow of tainted data, i.e., data produced by taint sources. We will raise an alert when we detect tainted data flow to a sink. Several tools [35] exist to perform taint analysis of Android apps. Instead of re-designing a new tool, we want to use an existing best-performing tool. However, Pauck et al. [35] showed that the effectiveness of existing taint analysis (or flow tracking) tools varies across different types of apps, and no single tool is superior for all apps. We enhanced the most robust tool, i.e., FLOWDROID [16] with our taint sources and various robustness fixes. This also enables us to demonstrate the generality of our interactive triaging technique (§3.2).

3.2 Interactive Triageing

This phase aims to prioritize alerts based on their likelihood of being a true positive. We maintain a *rank score* for each alert, indicating the likelihood of the alert being a true positive.

Overview. Initially (without any other information), all the alerts will have the same rank score, i.e., equally likely to be true positives. We use an iterative system (right half of Figure 2), displaying the alert to the user with the greatest rank score in each iteration. In contrast, other non-iterative-based ranking systems (e.g., Z-Ranking [24]) display all alerts at once based on their score. The user provides binary feedback, i.e., whether the alert is a true positive or not. If it is true positive (✓) (or false positive (✗)), then the rank score of all other alerts will be increased, i.e., *rewarded* (or decreased, i.e., *penalized*), based on how *similar* they are to the current alert. In the next iteration, we again select an alert with the highest rank score (which is now adjusted based on the user feedback). This process continues for a fixed number of iterations or until the developer desires.

3.2.1 Similarity Computation. There are various ways to compute the similarity between two alerts. We want the similarity computation to be deterministic and interpretable, which makes it easy to reason about our triaging technique. Consequently, we did not explore any learning-based techniques, such as the one used in ARBITRAR [25], because of their non-determinism and the lack of interpretability. Furthermore, these techniques require an effective vectorization mechanism, i.e., converting alert to a numerical vector.

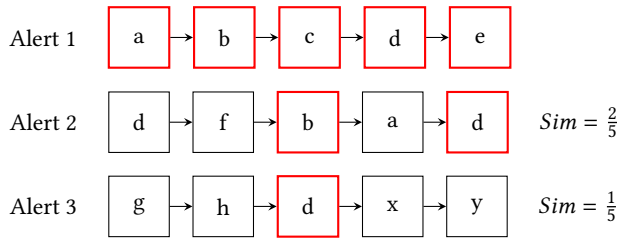


Figure 3: Alert Recalibration

A taint analysis alert is a sequence of instructions (*i.e.*, DEX instructions) indicating the flow of tainted data from a taint source (the first instruction) to a sensitive sink (the last instruction). A true positive alert implies that the data flow is valid, and no sanitization makes the flow safe. On the other hand, a false positive implies either the data flow is invalid (*e.g.*, imprecision in static analyses) and/or there is a sanitization. The similarity between alerts should measure *how much of the flow is shared between alerts*. We model the computation of similarity as the *longest subsequence matching problem* [5]. Specifically, we consider each instruction as an element and model alert as a sequence of elements. We compute the similarity score (*i.e.*, Sim) between two alerts as the fraction of the longest common subsequence (LCS) between them. The Figure 3 shows an example illustrating the similarity score computation of Alert 2 and 3 w.r.t Alert 1. The length of LCS between Alert 2 and 1 is 2, with the length of the Alert 1 being 5. So the Sim score of Alert 2 is $(2/5)$, similarly, for Alert 3.

3.2.2 Interactive Scoring. Given the user feedback for an alert a , we adjust the score of all other alerts based on the above similarity score. Specifically, if a is true positive, we add the similarity score to the rank score multiplied by a *reward factor*. If not (*i.e.*, false positive), we subtract the similarity score multiplied by a *penalty factor*. The reward and penalty factors provide flexibility in handling the confidence level of users providing different kinds of feedback. The Listing 4 shows the pseudocode of our scoring algorithm. In every iteration, we just display the alert with the highest score to the user.

4 Implementation

We implemented REARFIND for Android apps. Specifically, *our system works directly on APKs and does not require the app's source code*. We implemented source identification and sink instrumentation on top of SOOT [42] instrumentation framework. Since Java and Kotlin applications are compiled into Dex bytecode. This allows us to have a single analysis for both Java and Kotlin based applications since Soot lifts the Dex bytecode to its own IR called Jimple IR. Our call-graph analysis of native libraries (*i.e.*, *.so* files) is implemented using BINARYNINJA [7]. We used FLOWDROID as our taint analysis framework as it is well-maintained and has reasonable support for the latest versions of Android SDK. FLOWDROID also supports adding custom sources and sinks through a very configurable format as shown in Listing 5.

```

1 def Recalibration(alerts, user_feedback):
2     reward_factor = 1.0
3     penalty_factor = 1.0
4     ranked_alerts = []
5
6     selected_alert = alerts[0]
7     if user_feedback == True: # True positive
8         selected_alert.rank_score += reward_factor
9     else: # False positive
10        selected_alert.rank_score -= penalty_factor
11
12    for alert in alerts:
13        Sim = compute_similarity(alert, selected_alert)
14        if user_feedback == True:
15            alert.rank_score += Sim * reward_factor
16        else:
17            alert.rank_score -= Sim * penalty_factor
18
19    ranked_alerts.append((alert, alert.rank_score))
20
21    sort(ranked_alerts)
22    return ranked_alerts
23
24 def compute_similarity(alert1, alert2):
25     lcs_length = len(lcs(alert1.trace, alert2.trace))
26     return lcs_length / len(alert1.trace)

```

Listing 4: Rank Score Recalibration based on user feedback.

```

1 <method signature="java.io.PrintWriter: void write(java.lang.String)">
2   <param index="0" description="Output Data">
3     <accessPath isSource="false" isSink="true" />
4   </param>
5   <additionalFlowCondition>
6     <signatureOnPath signature="java.net.URL:
7     ↪ java.net.URLConnection.openConnection()" />
8   </additionalFlowCondition>
9 </method>

```

Listing 5: Model used by FLOWDROID to define a sinks

Table 1: Real World CApps

ID	Name	Classes	Sources	Instrumented Sinks
R1	com.lgref.android.smartref.us.mp2012	1,697	43	3,439
R2	com.ipc360	9,466	83	33,332
R3	com.sjty.imotornew	2,957	26	6,145
R4	android.in.start	2,335	84	7,060
R5	cn.aviador.hybrid	13,013	69	53,953
R6	com.bergson	1,000	69	53,953
R7	com.evo.gimbal	5,289	28	18,435
R8	com.techwin.shc	5,372	39	18,811

Our interactive training is implemented as a standalone Python script. In total, REARFIND is implemented through 1.2K lines of Java and 4K lines of Python.

5 Evaluation

We evaluate REARFIND along with following aspects:

- Q1 (*Effectiveness of REARFIND*): How effective is REARFIND in finding DTM vulnerabilities in CApps?
- Q2 (*Effectiveness of Interactive Triaging*): How effective is our interactive triaging technique?

Table 2: TaintBENCH Dataset

ID	Name	Classes	Sources	Instrumented Sinks
T1	cajino_baidu	3,457	514	246
T2	proxy_samp	739	49	21
T3	smssend_packageinstaller	1,293	53	51
T4	death_ring_materialflow	869	101	80
T5	save_me	2,668	93	74
T6	stels_flashplayer_android-update	1,115	42	38
T7	sms_send_locker_qqmagic	783	20	14
T8	fakedaum	1,280	97	83
T9	roidsec	785	40	22
T10	hummingbad_android_samp	5,717	461	369
T11	backflash	1,402	53	35
T12	chat_hook	1,155	22	21
T13	overlaylocker2_android_samp	1,754	60	48
T14	vibleaker_android_samp	5,410	428	246
T15	tetus	920	45	43
T16	fakeplay	1,550	43	34
T17	fakebank_android_samp	1,207	59	44
T18	threatjapan_uracto	1,213	16	14
T19	overlay_android_samp	1,054	55	35
T20	phospy	853	43	37
T21	jollyserv	1,681	200	159
T22	dsencrypt_samp	587	18	9
T23	exprespam	686	4	6
T24	xbot_android_samp	914	220	100
T25	scipix	1,268	295	125
T26	smsstealer_kysn_assassincreed-android_samp	1,523	33	18

Table 3: New Bugs discovered in CApps.

ID	Bug Class	Num. Bugs	Ablation
R4	SQL Injection	1	1
R5	Plaintext Credentials	2	2
R2	Hardcoded AES Keys	1	1
R6	Unchecked FW for DFU	1	1

Table 4: Comparison and Ablation Study. ✘ indicates that the corresponding bug was not detected by the technique.

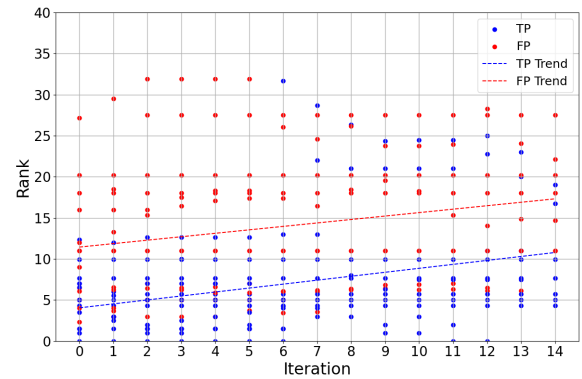
ID	FlowDroid		C1			REARFIND		
	Alerts	TP	Alerts	TP	Default Rank	Alerts	TP	Interactive Rank
R1	290		76			106		
R2	794	✘	394	1	28	567	1	8
R3	133		73			91		
R4	273	✘	497	1	190	940	1	14
R5	140	✘	80	1	8	93	1	1
R6	159	✘	74	1	2, ✘	157	2	1,2
R7	39		35			53		
R8	121		90			250		

Q3 (Comparative Study): What is the effectiveness of REARFIND in comparison with other state-of-the-art techniques?

Q4 (Ablation Study): What is the contribution of individual phases on the overall effectiveness of REARFIND?

5.1 Dataset

Our dataset is composed of real-world CApps and ground truth Android apps with known true positives (to evaluate our interactive triaging). We selected 8 real-world CApps (Table 1). Our selection is guided by how representative is the CApp and whether our underlying framework (*i.e.*, FlowDroid) is able to handle them.


Figure 4: Rank trend of REARFIND.

We selected TaintBENCH[27] (5) as our ground truth dataset as it contains realistic apps, with each app containing multiple issues. Furthermore, TaintBENCH also provides ground truth information in a machine-readable format (*i.e.*, JSON), making it easy for evaluation. We ignored other simplistic ground truth datasets, such as DroidBench[13] and Ghera[32], as they contain either simple (unrealistic) apps or apps with single vulnerability per app. Table 1 and Table 2 show the details for the datasets we are using.

5.2 Experimental Setup

REARFIND was run on an Ubuntu 22.04 LTS machine with AMD EPYC 7543P (64 cores) and 128 GB memory with 64 GB reserved for the analysis. We allow the taint analysis frameworks to run for a maximum of 15 minutes and the JNI analysis to run for 5 mins per shared object. We used Soot 4.4.1, Flowdroid 2.13.0 which were the latest version at the time of running the analysis.

5.2.1 Default Ranking. We use a default severity ranking scheme for all taint analysis alerts. Our ranking is based on the severity of sources, sinks, and the likelihood of flows present in the alert being false positives. Listing 6 shows the pseudocode of our ranking technique. The scores for the sources and sinks are assigned based on their severity. For instance, a network source will be assigned a higher score than a non-network score. Similarly, a command execution sink (*e.g.*, `exec`) will be assigned a higher score than the file open. We assign scores for flows based on the likelihood of it being a false positive. Consider the example in Listing 2. At line 16, we convert `response.resp` to an `java.lang.Integer`. Such conversions in Java are most likely safe, and considering them, they might result in false positives. Hence, we assign such type conversions (*i.e.*, from object types to scalar) a lower score. On the other hand, conversions to other object types (*e.g.*, `byte[]`) will have a higher score. Without our interactive triaging, we will display alerts in the descending order of the score computed above.

5.2.2 Interaction Budget for Interactive Triaging. We chose an interaction budget of 15 rounds for the TaintBENCH dataset and 20 rounds for the real-world CApps. This is in line with previous studies [25]. We triaged a total of 246 alerts across 26 APKs in TaintBENCH and 160 alerts across 8 CApps.

```

1 def severity_score(issues):
2     for each issue in issues:
3         score = 0
4         score += get_source_score(source)
5         score += get_sink_score(sink)
6         score += calculate_trace_score(feature)
7     issue_score = normalize(score)
8     normalize_scores(issues)
9     return sort_issues_by_score(issues)

```

Listing 6: Severity Scoring Algorithm

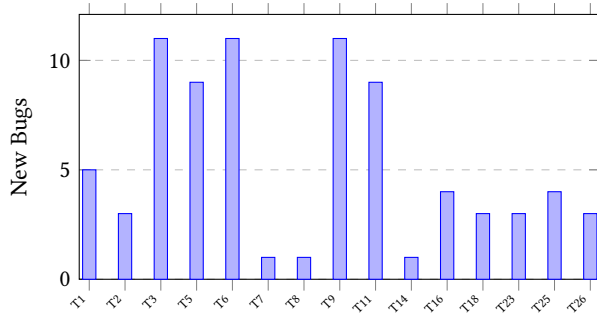


Figure 5: New Bugs found in the TaintBench

5.3 Q1: Effectiveness of REARFIND




Alerts Generation. The columns *sinks* and *sources* of Table 2 and 1 show the number of taint sources (java/native) and sinks found in each app. We randomly sampled 10 apps and verified that all these are valid. This demonstrates the importance of our sources and sinks identification techniques (§3.1.1 and 3.1.2).

Bug Finding Ability. We used REARFIND with its interactive triaging on both datasets.

On TaintBench: Given our interaction budget of 15 interactions, the maximum number of bugs that can be detected in each APK is 15, and consequently, the total number of detectable bugs is 135.

Table 5 shows the summary of results across all APKs. REARFIND through its interactive triaging, found a total of 128 bugs in TaintBench. Out of which, 97 are known bugs, *i.e.*, REARFIND was able to find 71% of detectable bugs in TaintBench demonstrating the effectiveness of REARFIND in identifying previously known bugs.

Interestingly, we found 31 new bugs in TaintBench, *i.e.*, bugs that are not listed in the ground truth. Figure 5 shows the distribution of these bugs across various APKs. These results demonstrate the effectiveness of REARFIND as a general bug-finding tool for Android APKs.

On Real World Dataset: We found a total of 5 security vulnerabilities in Real-world CApps. Table 3 shows the split of vulnerabilities and their categories across different APKs. Listing 7 shows a real SQL injection found in android.in.start (R4) CApp. As we can see, the user input (indicated by ) is used (path highlighted through ) in an SQL query (indicated by ) without any sanitization resulting in a SQL injection vulnerability.

Responsible Disclosure. We are in the process of disclosing these vulnerabilities to the respective vendors.

```

1 public void syncUserData(CommCloud commCloud) {
2     String str;
3     DataBaseOperator dataBaseOperator;
4     boolean z;
5     ...
6     dataBaseOperator2.updateData(DataBaseOperator.TABLE_TB_USERINFO,
7     ↪ "iHealthCloud = '" +  this.txt_username.getText().toString() +
8     ↪ "'", "isRememberPassword = '" + (this.cb_rememberPwd.isChecked() ?
9     ↪ "'T' : 'F') + '"', isAutoLogin='F', isDisplayWelcome='" + str2 +
10    ↪ "'");
11    ...
12 }
13 }
14 }
15 }
16 }
17 }
18 }
19 }
20 }
21 }

```

Listing 7: SQL Injection in R4

Table 5: TaintBench Results.

APK	Ground Truth	Alerts	Observed	Ablation
T1	12	168	9	0
T2	17	41	10	5
T3	5	79	13	4
T4	1	29	1	0
T5	25	31	12	5
T6	3	63	12	0
T7	6	8	1	0
T8	2	13	1	0
T9	6	30	11	0
T10	2	13	0	0
T11	13	57	14	2
T12	12	17	9	2
T13	7	14	1	1
T14	4	46	3	1
T15	2	1	0	0
T16	2	46	4	2
T17	5	7	4	0
T18	2	3	3	0
T19	4	2	1	0
T20	2	2	2	0
T21	1	1	1	0
T22	1	3	0	0
T23	2	5	5	0
T24	3	1	0	0
T25	3	7	4	0
T26	5	5	5	2
Total	147	692	128	25

5.4 Q2: Effectiveness of Interactive Triaging

The Figure 7 shows the performance of our interactive triaging on each APK of TaintBench dataset. As we can see, our technique learns from user feedback and picks true positives with higher probability. This can be observed graphically (in Figure 7) as there are more green squares (*i.e.*, true positives) when we move from left to right. The Figure 4 also illustrates this, where more feedback (*i.e.*, as we go right) results in higher rank score for true positives (*i.e.*, blue dots). This demonstrates our interactive triaging technique's effectiveness and ability to learn from user feedback.

We further evaluate the effectiveness of our interactive triaging along the following three aspects.

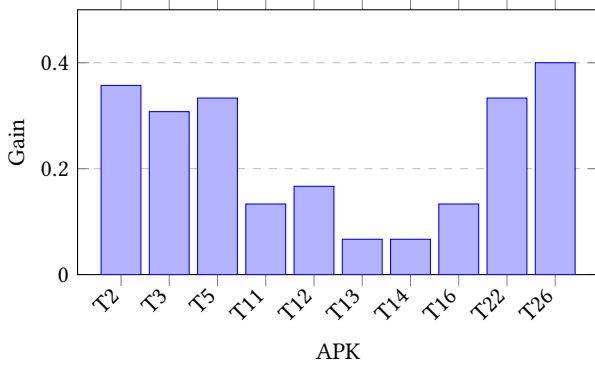


Figure 6: Gain (§5.4.3) for each APK of TAINTBENCH

5.4.1 *Ablation.* Given a triaging budget t , we compute the number of additional true bugs detected using our interactive triaging compared to the default method (*i.e.*, default scoring). Specifically, we interact with REARFIND for t alerts and use the top-ranked t alerts for the default method. As mentioned in §5.2.2, we use a budget of $t = 15$ for TAINTBENCH and $t = 20$ for real-world CApps. The column *Ablation* of Table 5 and 3 shows the results for TAINTBENCH and real-world CApps, respectively. In total, our interactive triaging enabled us to find an additional 24% bugs in TAINTBENCH. The results are much more impressive in real-world CApps (Table 3), where the default method has found none of the vulnerabilities found by our interactive triaging.

5.4.2 *False positive Rate.* This is a side effect of ablation, where we compute the number of false positives in our interactive triaging v/s default method. On TAINTBENCH dataset, the default ranking has a false positive rate of 62%, whereas our interactive scoring has 50% with a 12% reduction. On real-world CApps, over 20 triages per app for the 4 apps we found bugs in, the default ranking has a false positive rate of 97.5%, whereas our interactive scoring has 93.75% with a 4% reduction. This shows that our interactive triaging is effective at false positive reduction.

5.4.3 *Triaging Gain.* We define this as the amount of information gained by our technique for detecting true positives. We define gain as:

$$\text{Gain} = \frac{\text{No. of New Issues Found using Interactive Triage}}{\text{Last True Positive Iteration}}$$

The gain is defined only for cases where our technique found at least one true positive. A higher value indicates better gain (*desirable*). The ideal value for this metric is 1, where we found a true positive in every iteration, *e.g.*, we found three true positives and the last true positive was found in the 3rd iteration. This indicates that our technique gained a lot of information in every iteration. The worst case is 1/15 (*i.e.*, 0.06), where we detected one true positive, and it was in the last iteration. Specifically, our technique did not learn fast enough, and it required 14 iterations. The Figure 6 shows the gain for various APKs of TAINTBENCH dataset. The gain for most of the APKs is higher than 0.3, indicating that our technique can effectively gain from user interactions.

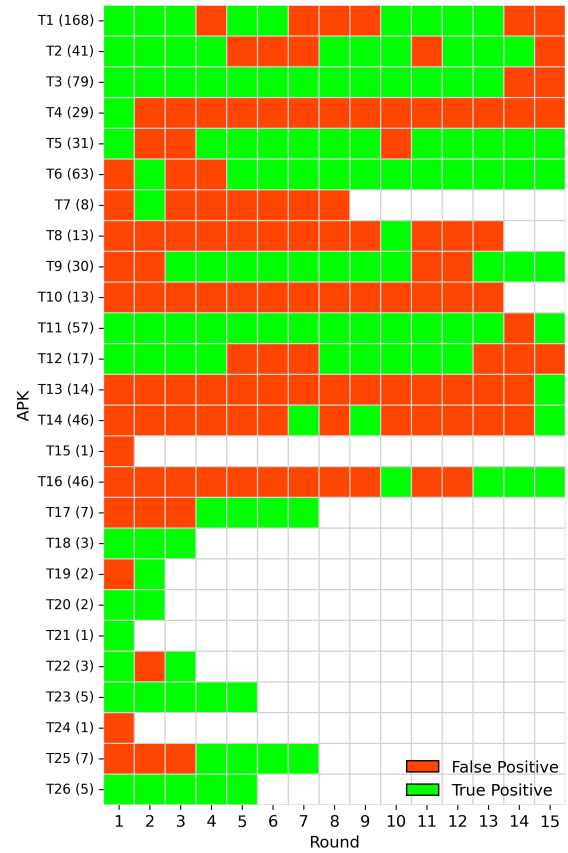


Figure 7: User Feedback and Performance of Interactive Triaging for TAINTBENCH

5.5 Q3: Comparative Study

We compared with FLOWDROID, the state-of-the-art taint tracking framework for Android APKs. We used FLOWDROID with its default configuration (*i.e.*, default sources and no sink instrumentation).

On TAINTBENCH, FLOWDROID found 71% bugs (among the top 15 alerts) compared to 94% bugs by REARFIND.

On real-world CApps, FLOWDROID found none of the bugs found by REARFIND. Table 4 shows the results for each APK. Despite a large number of warnings, FLOWDROID found none of the bugs (100% false positive rate).

5.6 Q4: Ablation Study

In this section, we perform an ablation study to evaluate the contribution of our techniques to the overall effectiveness of REARFIND. Specifically, our taint source identification’s contribution to the overall effectiveness of REARFIND. We created a new configuration C1 of REARFIND with only taint source information (§3.1.1), no taint sink instrumentation, and no interactive triaging. The Table 4 shows the results. First, we can see that C1 failed to find one of the bugs in R6. Second, the default ranking (indicated under *Default Rank* column) of the alerts corresponding to these bugs is very large, indicating that the developer has to go through various alerts

to find the true bug. Finally, our interactive triaging found bugs in a considerably smaller number of iterations, as indicated by the *Interactive Rank* column.

6 Limitations and Future Work

We acknowledge the following limitations of the current state of REARFIND.

- **Lack of a comprehensive DTM Ground Truth:** Even though TAINTBENCH has enough APKs and general taint-related vulnerabilities, there is no dataset specifically for DTM vulnerabilities. Furthermore, TAINTBENCH ground truth is on decompiled Java code and not on the dex instruction level — on which our taint alerts are generated. It required significant effort to map the Java-level information to dex instructions. We believe our findings will serve as the initial dataset of DTM vulnerabilities.
- **Tool Limitations:** We adopt and build on top of existing frameworks, which implies that we also adopt the limitations of the frameworks, so wherever the framework fails, REARFIND also fails. This was pretty evident for the instrumentation engine where Soot failed 50% of the time.
- **Smaller Real World Dataset:** The real world dataset is not very exhaustive. This has to do with the failure rate of the tools, which limits how many APKs move from Taint Source Identification 3.1.1 to Sink Instrumentation 3.1.2.
- **Fine-grained Instrumentation:** Finer instrumentation rules would be able to reduce the false positives by a huge margin and catch new bug classes too. Although this requires manual effort to define the instrumentation constraints, we should see a relatively lower number of alerts and possibly more quality alerts.

7 Related Work

There are many tools [35] that try to perform static taint analysis on Android apps, such as FLOWDROID [16], AMANDROID [43], MARIANATRENCH [30], and DROIDSAFE [19]. There are techniques built on top of these tools for specific purposes, *e.g.*, to find loop bounds [17], malware detection [40], etc. In our work, we customized FLOWDROID to detect DTM vulnerabilities.

Security Analysis of Companion Apps. Several works try to study companion apps. Scoccia et al. [41] analyzed user perception of companion apps. Allen et al. [2] uses existing Static application security testing (SAST) tools to find vulnerabilities in companion apps. However, it focuses on regular app-level vulnerabilities and does not consider DTM vulnerabilities.

Almost all existing works that analyze companion apps focus on vulnerability detection on the corresponding IoT device. Mauro Junior et al. [31] uses dynamic analysis of companion apps to find insecure crypto on IoT devices. Diane [38] uses a combination of static and dynamic analysis to identify Fuzzing Triggers, *i.e.*, functions that send data to the target IoT device. These triggers are then used to fuzz-test the device. Similarly, IoTFuzzer [10] also tries to test the target device through the companion app. IoTProfiler [33] analyzes companion apps through learning techniques to identify

leakage of sensitive data. In this work, we focus on DTM vulnerabilities in companion apps, *i.e.*, those that can be triggered through IoT devices.

Ranking Static Analysis Alerts. Techniques to rank static analysis alerts can be categorized into two categories as (i) *Ad-Hoc Filtering*, and (ii) *Statistical Filtering*.

Ad-Hoc filtering techniques use custom rules for ranking. AWARE[21] uses alert type and code locality to rank alerts. This technique assumes that alerts closer to each other are expected to have a similar outcome and are similarly actionable.

Statistical Filtering techniques uses various statistical and probabilistic methods for ranking. Z-Ranking [24] is one of the first works that tries to rank alerts from most to least probable based frequency counts of successful and failed alerts reported by the static analysis tool. This is a drawback of Z-Ranking, where the system expects the static analysis tool to give successful and failed checks for ranking.

Similarly, FeedBackRank [23], Bing [36] and Bayesmith [22] proposed a Bayesian network [6] based ranking, which also adapts based on user feedback. However, these techniques work well for query-based static analysis techniques, *e.g.*, Datalog [14], and cannot be easily applied to flow-based analysis techniques.

FITS [26] performs filtering at the intermediate taint source finding phase. Consequently, the technique modifies the underlying analysis and cannot be applied to arbitrary taint results. However, REARFIND performs filtering at the end, making it independent of the underlying analysis and thus can be applied to arbitrary taint results. We also demonstrate the generalizability of REARFIND in §5.

8 Conclusion

In this paper we introduce a systematic way to automatically find and generate taint sources that help in better Taint Analysis of IoT Companion Apps. We also introduced an Interactive Triaging system to aid in faster triage so less user effort is spent on analysing False Positives. Our approach performs good in the ground truth by finding 21% new bugs that are not present in the ground truth.

9 Acknowledgements

This research was supported by the National Science Foundation (NSF) under Grant CNS-2340548. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

References

- [1] Mohammed Ali Al-Garadi, Amr Mohamed, Abdulla Khalid Al-Ali, Xiaojiang Du, Ihsan Ali, and Mohsen Guizani. 2020. A Survey of Machine and Deep Learning Methods for Internet of Things (IoT) Security. *IEEE Communications Surveys & Tutorials* 22, 3 (2020), 1646–1685. <https://doi.org/10.1109/COMST.2020.2988293> Conference Name: IEEE Communications Surveys & Tutorials.
- [2] Ashley Allen, Alexios Mylonas, Stilianos Vidalis, and Dimitris Gritzalis. 2024. Security Evaluation of Companion Android Applications in IoT: The Case of Smart Security Devices. *Sensors* 24, 17 (2024). <https://doi.org/10.3390/s24175465>
- [3] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. SoK: Security Evaluation of Home-Based IoT Deployments. *Proceedings - IEEE Symposium on Security and Privacy* 2019-May (2019), 1362–1380. <https://doi.org/10.1109/SP.2019.00013>
- [4] Amnesia 2020. AMNESIA:33 – Forescout Research Labs Finds 33 New Vulnerabilities in Open Source TCP/IP Stacks. <https://www.forescout.com/blog/amnesia33->

- forescout-research-labs-finds-33-new-vulnerabilities-in-open-source-tcp-ip-stacks/. (Dec. 2020).
- [5] Alberto Apostolico and Concettina Guerra. 1987. The longest common subsequence problem revisited. *Algorithmica* 2 (1987), 315–336.
 - [6] Bayesian network 2024. Bayesian network - Wikipedia. https://en.wikipedia.org/wiki/Bayesian_network. (2024). (Accessed on 10/02/2024).
 - [7] Binary Ninja 2024. Binary Ninja. <https://binary.ninja/>. (2024). (Accessed on 10/04/2024).
 - [8] bncallgraph. 2024. psifertex/callgraph: Binary Ninja Call Graph plugin. <https://github.com/psifertex/callgraph>. (2024). (Accessed on 10/02/2024).
 - [9] Dan Boxler and Kristen R Walcott. 2018. Static taint analysis tools to detect information flows. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*. The Steering Committee of The World Congress in Computer Science, Computer ..., 46–52.
 - [10] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Proceedings 2018 Network and Distributed System Security Symposium* (2018). Internet Society. <https://doi.org/10.14722/ndss.2018.23159>
 - [11] Chenxiong Qian, Chenxiong Qian, Liu Wang, Xiapu Luo, Yuru Shao, Yuru Shao, Alvin T. S. Chan, and Alvin T. S. Chan. 2014. On Tracking Information Flows through JNI in Android Applications. (June 2014), 180–191. <https://doi.org/10.1109/dsn.2014.30>
 - [12] Shaunik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?(e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 429–440.
 - [13] Steven Arzt Christian Fritz and Siegfried Rasthofer. 2024. secure-software-engineering/DroidBench: A micro-benchmark suite to assess the stability of taint-analysis tools for Android. <https://github.com/secure-software-engineering/DroidBench>. (2024). (Accessed on 09/17/2024).
 - [14] Datalog. 2024. Datalog - Wikipedia. <https://en.wikipedia.org/wiki/Datalog>. (2024). (Accessed on 10/02/2024).
 - [15] Tamara Denning, Alan Borning, Batya Friedman, Brian T. Gill, Tadayoshi Kohno, and William H. Maisel. 2010. Patients, pacemakers, and implantable defibrillators: human values and security for wireless implantable medical devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. Association for Computing Machinery, New York, NY, USA, 917–926. <https://doi.org/10.1145/1753326.1753462>
 - [16] Flowdroid. 2024. secure-software-engineering/FlowDroid: FlowDroid Static Data Flow Tracker. <https://github.com/secure-software-engineering/FlowDroid>. (2024). (Accessed on 09/16/2024).
 - [17] Yanick Fratantonio, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2015. CLAPP: characterizing loops in Android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 687–697. <https://doi.org/10.1145/2786805.2786873>
 - [18] Frida. 2024. Frida • A world-class dynamic instrumentation toolkit | Observe and reprogram running programs on Windows, macOS, GNU/Linux, iOS, watchOS, tvOS, Android, FreeBSD, and QNX. <https://frida.re/>. (2024). (Accessed on 09/16/2024).
 - [19] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. 2015. Information-Flow Analysis of Android Applications in DroidSafe. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. <https://doi.org/10.14722/ndss.2015.23089>
 - [20] Mary W Hall and Ken Kennedy. 1992. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 3 (1992), 227–242.
 - [21] Sarah Heckman and Laurie Williams. 2008. On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, Kaiserslautern Germany, 41–50. <https://doi.org/10.1145/1414004.1414013>
 - [22] Hyunsu Kim, Mukund Raghothaman, and Kihong Heo. 2022. Learning probabilistic models for static analysis alarms. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1282–1293. <https://doi.org/10.1145/3510003.3510098>
 - [23] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. 2004. Correlation exploitation in error ranking. *SIGSOFT Softw. Eng. Notes* 29, 6 (Oct. 2004), 83–93. <https://doi.org/10.1145/1041685.1029909>
 - [24] Ted Kremenek and Dawson Engler. 2003. Z-ranking: using statistical analysis to counter the impact of static analysis approximations. In *Proceedings of the 10th International Conference on Static Analysis (SAS'03)*. Springer-Verlag, Berlin, Heidelberg, 295–315.
 - [25] Ziyang Li, Aravind Machiry, Binghong Chen, Mayur Naik, Ke Wang, and Le Song. 2021. Arbitrar: User-guided api misuse detection. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1400–1415.
 - [26] Puzhuo Liu, Yaowen Zheng, Chengnian Sun, Chuan Qin, Dongliang Fang, Mingdong Liu, and Limin Sun. 2024. FITS: Inferring Intermediate Taint Sources for Effective Vulnerability Analysis of IoT Device Firmware. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 138–152. <https://doi.org/10.1145/3623278.3624759>
 - [27] Linghui Luo, Felix Pauck, Goran Piskachev, Manuel Benz, Ivan Pashchenko, Martin Mory, Eric Bodden, Ben Hermann, and Fabio Massacci. TaintBench: Automatic Real-World Malware Benchmarking of Android Taint Analyses. *Empirical Software Engineering* 27, 1 (????), 16. <https://doi.org/10.1007/s10664-021-10013-5>
 - [28] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 224–234. <https://doi.org/10.1145/2491411.2491450>
 - [29] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.
 - [30] Mariana Trench 2024. facebook/mariana-trench: A security focused static analysis tool for Android and Java applications. <https://github.com/facebook/mariana-trench>. (2024). (Accessed on 09/16/2024).
 - [31] Davino Mauro Junior, Luis Melo, Hao Lu, Marcelo d'Amorim, and Atul Prakash. 2019. A Study of Vulnerability Analysis of Popular Smart Devices Through Their Companion Apps. In *2019 IEEE Security and Privacy Workshops (SPW)*. 181–186. <https://doi.org/10.1109/SPW.2019.00042>
 - [32] Joydeep Mitra and Venkatesh-Prasad Ranganath. 2017. Ghera: A Repository of Android App Vulnerability Benchmarks. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. Association for Computing Machinery, New York, NY, USA, 43–52. <https://doi.org/10.1145/3127005.3127010>
 - [33] Yuhong Nan, Xueqiang Wang, Luyi Xing, Xiaojing Liao, Ruoyu Wu, Jianliang Wu, Yifan Zhang, and XiaoFeng Wang. 2023. Are You Spying on Me? Large-Scale Analysis on IoT Data Exposure through Companion Apps. In *32nd USENIX Security Symposium*. USENIX Association, Anaheim, CA, 6665–6682. <https://www.usenix.org/conference/usenixsecurity23/presentation/nan>
 - [34] Oracle. 2024. Introduction. <https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/intro.html>. (2024). (Accessed on 09/16/2024).
 - [35] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android Taint Analysis Tools Keep Their Promises?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Lake Buena Vista FL USA, 331–341. <https://doi.org/10.1145/3236024.3236029>
 - [36] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided program reasoning using Bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 722–735. <https://doi.org/10.1145/3192366.3192417>
 - [37] Heena Rawal and Chandresh Parekh. 2017. Android Internal Analysis of APK by Droid_Safe & APK Tool. *International Journal of Advanced Research in Computer Science* 8, 5 (2017).
 - [38] Nilo Redini, Andrea Continella, Dipanjan Das, Giulio De Pasquale, Noah Spahn, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2021. Diane: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices. In *2021 IEEE Symposium on Security and Privacy (SP)*. 484–500. <https://doi.org/10.1109/SP40001.2021.00066>
 - [39] Ripple20 2024. Ripple20. <https://www.jsf-tech.com/disclosures/ripple20/>. (2024).
 - [40] Suzanna Schmeelk, Junfeng Yang, and Alfred Aho. 2015. Android Malware Static Analysis Techniques. In *Proceedings of the 10th Annual Cyber and Information Security Research Conference (CISR '15)*. Association for Computing Machinery, New York, NY, USA, Article 5, 8 pages. <https://doi.org/10.1145/2746266.2746271>
 - [41] Gian Luca Soccia, Romina Eramo, and Marco Autili. 2023. Studying users' perception of IoT mobile companion apps. *Pervasive and Mobile Computing* 92 (2023), 101786. <https://doi.org/10.1016/j.pmcj.2023.101786>
 - [42] Soot. 2024. soot-oss/soot: Soot - A Java optimization framework. <https://github.com/soot-oss/soot>. (2024). (Accessed on 10/02/2024).
 - [43] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2018. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. *ACM Trans. Priv. Secur.* 21, 3 (April 2018), 14:1–14:32. <https://doi.org/10.1145/3183575>
 - [44] Yuhao Wu, Jinwen Wang, Yujie Wang, Shixuan Zhai, Zihan Li, Yi He, Kun Sun, Qi Li, and Ning Zhang. 2024. Your Firmware Has Arrived: A Study of Firmware Update Vulnerabilities. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 5627–5644. <https://www.usenix.org/conference/usenixsecurity24/presentation/wu-yuhao>
 - [45] Miao Yu, Jianwei Zhuge, Ming Cao, Zhiwei Shi, and Lin Jiang. 2020. A survey of security vulnerability analysis, discovery, detection, and mitigation on IoT devices. *Future Internet* 12, 2 (2020), 27.