KASTroid: A Static Taint Analysis Framework for Kotlin-Based Android Applications

1st Bara' Nazzal
School of Computing
Queen's University
Kingston, Canada
21bn7@queensu.ca

2nd Manar H. Alalfi
Department of Computer Science
Toronto Metropolitan University
Toronto, Canada
manar.alalfi@torontomu.ca

3rd James R. Cordy
School of Computing
Queen's University
Kingston, Canada
cordy@queensu.ca

***Abstract*—We introduce KASTroid[1], a static taint analysis framework designed to detect information leakage vulnerabilities in Kotlin-based Android applications, with a focus on insecure usage of broadcasts. As Kotlin increasingly replaces Java as the preferred language for Android development, ensuring the security of inter-component communication, particularly through broadcasts, is critical. KASTroid employs a novel static analysis approach to identify tainted flows arising from unsafe broadcast practices, such as the use of implicit broadcasts via *sendBroadcast*, which can expose sensitive user data to unauthorized applications. We evaluated KASTroid on a dataset of 1,716 Kotlin Android applications and found that over 70% of broadcasts are implicit, posing significant privacy risks. To demonstrate the framework's effectiveness, we present a case study on a previous version of AndroidAPS, a medical application, where KASTroid identified unsafe broadcast usage that could leak sensitive information to other local applications. Our findings highlight the importance of secure broadcast practices and demonstrate KASTroid's ability to assist developers in detecting and remedying such vulnerabilities.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Smartphones have become integral to everyday life for many people, providing convenience but also introducing potential security risks. As the prevalence of this technology continues to grow, it is crucial to study the security aspects of mobile apps and ensure they comply with established security standards and guidelines.

According to a report by the Open Web Application Security Project (OWASP), inadequate privacy controls rank as the sixth most significant mobile risk [8]. To investigate this issue, we focus on a particular source of vulnerabilities within Android systems: the use of broadcast functions. In Android, broadcasts allow messages to be sent between apps and the system [10]. However, if a broadcast is not properly secured, other potentially malicious applications installed on the device could intercept the message, leading to information leakage.

To illustrate how an attack might occur, consider the example shown in Listing 1, where an app uses a *sendBroadcast* function in *Line 5* to send an intent that may contain sensitive data. A malicious app with a receiver, as demonstrated in Listing 2, could eavesdrop on this broadcast, even if it was not the intended recipient. If the broadcast transmits sensitive information, this could result in a privacy breach for the user.

[1]Materials and a lite version of the tool are available through a web interface:
https://github.com/kastroid/kastroid-materials

Listing 1. sendBroadcast

```
1 fun send(context: Context) {
2     Intent().also { intent ->
3         intent.setAction("com.example.broadcast.
              MY_NOTIFICATION")
4         intent.putExtra("data", "Sensitive Data.")
5         context.sendBroadcast(intent) }}
```

Listing 2. onReceive

```
1 override fun onReceive(context: Context, intent: Intent)
      {
2     intent.takeIf { it.action == "com.example.broadcast.
          MY_NOTIFICATION" }?.let {
3         Log.d("onReceive", "Airdroid  => [${it.
              getStringExtra("data")}]/${it.extras}") }}
```

Broadcast vulnerabilities can lead to various security risks, including intent sniffing, eavesdropping, and hijacking, particularly if a malicious app intercepts or hijacks the broadcast. To mitigate these risks, security guidelines recommend using *LocalBroadcastManager*, which confines the broadcast within the app, or *setPackage*, which designates a specific recipient package. Our research addresses these vulnerabilities in Kotlin apps, where existing Java-based tools fall short.

The key contributions of our research are as follows:

- The first source-based static taint analysis framework for Kotlin-based Android applications
- A curated dataset of the most popular Kotlin Android projects from GitHub
- A taint analysis tool that identifies tainted flows and unsafe patterns including insecure broadcast usage
- An experimental evaluation of the Kotlin dataset, revealing the prevalence of unsafe broadcast practices

In this paper, we aim to explore the following research questions:

- **RQ1**: How effective is KASTroid in identifying and analyzing tainted flows, particularly insecure broadcast patterns such as unsafe sendBroadcast usage, in Kotlin-based Android applications? This question evaluates KASTroid's core capability to perform static taint analysis on Kotlin code, focusing on its capability in detecting vulnerabilities related to inter-component communication.

- **RQ2**: How prevalent is the unsafe usage of broadcasts in real-world Kotlin Android applications? This question investigates the extent of insecure broadcast practices, particularly implicit broadcasts, in open-source Kotlin applications, leveraging KASTroid's analysis on a curated dataset.

II. Background

There is a well-documented concern for privacy in Android applications. Shrivastava et al. [25] reviewed 110 articles highlighting risks of permission exploitation. Verderame et al. [27] emphasized that many popular Google Play Store apps violate privacy guidelines, allowing access to user private data. These findings underscore the need for robust analysis and vetting of Android applications for privacy concerns.

And while Kotlin is now used by 60% of Android developers, according to Google [11]. However, research and tools for verifying and validating Kotlin apps remain limited compared to Java. Our survey of papers from the last five years using Web of Science, focused on security, privacy, verification, validation, and analysis, shows a notable gap in Kotlin-related research, as shown in Table I. In this paper, we will try to address this gap by introducing a Kotlin analysis tool and applying it to detect broadcast vulnerabilities.

TABLE I
WEB OF SCIENCE 2019-2024 RESULTS

Search Terms	Kotlin	Java	Android	Java Android
Security	17	918	1,935	90
Privacy	4	156	822	25
Verification	7	304	188	15
Validation	4	538	395	15
Static	11	382	586	42
Analysis	55	7,757	2,974	127

III. Methodology

We introduce a Kotlin analysis program, KASTroid, using static analysis techniques. We analyze the code directly allows for quick analysis without the need for bytecode and avoids the need for patching existing Java analysis tools. It also allows for traceability by linking the results to the original code.

We use TXL [6], which is a source transformation language, to perform sink-source-based, backwards slicing of the Kotlin app. In this context, the *sources* are potentially sensitive data, while the *sink* represents the code locations that may leak this data.

If there is a flow from the source to the sink that is not properly guarded, the flow is considered *tainted*. We used this concept to develop a TXL tool for analyzing Kotlin app source code to detect potential tainted flows.

Our TXL program consists of two main components, which are responsible for abstract syntax tree generation and flow detection: **Grammar** and **Transformation Rules**

KASTroid takes a Kotlin source code file as input and outputs the relevant parts of the code that contain the identified data flows. Since no TXL Kotlin grammar was available, we developed our own by referencing the Kotlin syntax documentation [12]. The grammar, similar to a context-free grammar, specifies the language components, ranging from non-terminal annotations, headers, imports, and statements down to terminal components. Correct grammar is crucial for accurate analysis, as it defines the relationships between different components. Our grammar was verified on a large Kotlin dataset and is also flexible and can be modified for other static analyses tasks.

Listing 3. Example Source Code Input

```
1  fun companionFunction () {/*...*/ }// irrelevant function
2  fun fetchPatientData(): String { // Fetches patient data
       from storage
3   val patientId = "PANCREAS_PATIENT"//Sensitive identifier
4   val glucoseLevel = readFromSensor() //e.g., 200 (mg/dL)
5  return "$patientId,$glucoseLevel"
6  }
7  fun processData() {
8   patientData = fetchPatientData()  // formatting without
        encryption
9   formattedData = format(patientData)
10  sendToCloud("$formattedData")
11 }
12 fun mainWorkflow() {
13  processData()
14 }
15 // Sends data to a remote server (sink: unencrypted HTTP)
16 fun sendToCloud(data: String) {
17 //Unsecured endpoint
18  val url = "http://insecure-api.com/upload"
19  HttpClient().post(url, data) // Hypothetical HTTP client
20 }
```

After gathering the targeted Kotlin files, we combine them using a merger and output one combined Kotlin file. The first step occurs when the file is processed through TXL. Using our grammar, the tool parses the input source text according to Kotlin's reference syntax. If parsing fails at this step, the process terminates and an error is returned.

If parsing is successful, the TXL transformation rules carry out the subsequent tasks. The transformation rules for our analysis are divided into four main components discussed in the following subsections.

1) Sink identification: To illustrate, consider a simplified example where the input is as shown in Listing 3, and we are interested in tracking the function *sendToCloud()*. The tool would first parse the source code and match functions that correspond to the specified sink. In this case, line 10 would be marked. The next step involves backward-tracing variables and constants passed to the sink, along with tracking their assignments within the local scope. For this example, variables *formattedData* and *patientData* at lines 9 and 8.

2) Local backward tracing: The process begins by searching for lines containing variables that influence the sink, such as parameters or assignments. In Listing 3, line 9 would be marked because it contains the variable *formattedData* , which is a parameter for *sendToCloud*. The analysis then recursively repeats for lines affecting the newly marked line; line 8 would be marked next because it contains the variable *patientData*, which is assigned to *formattedData* .

3) Global tracing: The next step involves examining the remainder of the application for instances where the function containing the sink is invoked, as well as other functions present in the flow.

This step is carried out in the global scope. The tool marks and tags these instances, recursively repeating the process until all areas of the code that affect the relevant sink are tracked. In this example, *processData* is invoked within *mainWorkflow* at line 13. Since the function *fetchPatientData* is called to assign a value to *patientData*, it is marked, along with its return statement and the flow leading to it.

4) Custom rules, code cleanup, and vulnerabilities reporting: Additionally, if there are custom rules like ignoring flows that are encrypted or mitigated, they are enforced, otherwise, the flow is retained.

Finally, the tool performs a cleaning step by removing code sections not marked as relevant to the flow towards the sink. For example, *ompanionFunction* on line 1 would be removed, as it is not part of the flow. The output, featuring the relevant slice, is presented in Listing 4, along with the data flow represented by the tags that illustrate the relationships between the different variables and function calls.

Listing 4. Example Source Code Output

```
1
2  <fetchPatientData called from patientData,in processData>
       fun fetchPatientData(): String {
3   <source1>val patientId = "PANCREAS_PATIENT"</>
4   <source2> glucoseLevel = readFromSensor() </>
5   <returns patientId, glucoseLevel when called>
                  return "$patientId,$glucoseLevel"</>
6  } </>
7  fun processData() {
8   <patientData assigned to formattedData>
        patientData = fetchPatientData() </>
9   <formattedData passed to sendToCloud>
        formattedData = format(patientData) </>
10  <sink>sendToCloud("$formattedData")</sink>
11 }
12 fun mainWorkflow() {
13  <invokes processData()> processData() </>
14 }
```

A. Static Analysis Tool Development: Overcoming Kotlin-Specific Challenges

It should be noted that static-analysis can be susceptible to false positives, since for example it could mark parts of the code that are not usually executed. A finer grained analysis and handling of control-statements and callbacks can be obtained by introducing flow and context-sensitivity. This is explored further by Nazzal et al [21] and Arzt et al. [2].

Compared to previous literature by Krishnamurthy et al. [17] that showed that we are able to use Java analysis tools with Kotlin after converting it to bytecode, we do the analysis directly on the Kotlin code. This is because while the previous literature showed the possibility of the analysis, it also showed that the Kotlin bytecode is still different than Java bytecode and would require fixes and extensions to take that into consideration.

When developing an analysis tool for Kotlin applications, several technical challenges must be addressed. The grammar is responsible for parsing the code. Kotlin-specific features such as class properties, internal modifiers, local functions, are handled by the grammar and parsed as part of the language syntax. For example, class properties are accessed via getters and setters, even if not explicitly. These are considered

Listing 5. Kotlin Features and Challenges

```
1 class Example {
2     var property1: String = "private"
3     internal var property2: String = "benign"
4     infix fun String.foo(target: String) {/*...*/}
5     fun bar() {
6         property1 foo property2 }}
```

statements in the language; the analysis rules takes them into consideration when identifying the sinks and propagating the flow, as they are considered within the program statements.

Other features in the app's structure can be difficult to accurately represent. For example incorporating interfaces, which contain abstract methods and lack bodies that can be analyzed. The default behavior is to skip these methods. To enhance our approach, we define custom rules to identify abstract methods and include their declarations in the analysis flow. To preserve the structural integrity of the code, we assign unique identifiers to variables and methods, ensuring that the interface is correctly included and not conflated with overridden methods.

Another challenge arises when a non-abstract class is implemented without explicitly declaring a constructor. In such cases, Kotlin automatically generates a primary constructor with no arguments, allowing methods to instantiate objects of that class without an explicit constructor declaration. If this auto-generated constructor is overlooked, it may result in gaps in the data flow, as the tool could fail to identify the corresponding constructor. To mitigate this issue, we preprocess the app's code to insert empty constructors into classes that lack them, enabling comprehensive analysis with the default setup. Alternatively, we can leverage custom rules to instruct the tool to mark the class directly whenever an object of that class is instantiated.

In the following Listing 5 we showcase a code example with unique Kotlin features. These include class properties, an internal modifier for *property1*, an infix function that can be called without dot or parenthesis. For KASTroid the Kotlin grammar parses this as specified by Kotlin's documentation. The specified rules then can track the flow through the different constructions; either the TXL rules handle the features implicitly or or explicitly. For example class properties can be handled implicitly under the general rules relating to variable declarations. On the other hand, an explicit TXL rule can be added to look into infix functions calls due to its unique syntaxt, then the rules tracks the variables to the function implementation.

B. Detecting Broadcast Leakage

This approach can be used to test different properties by accepting different sources, sinks, and add custom rules. In our case, for example, to detect broadcast vulnerabilities, we preform the following steps: Find *sendBroadcast* functions in the code, track the flow and check for *localbroadcastmanager* and *setPackage* functions which mitigates the vulnerability by limiting the broadcast to the app itself or set a specified target package for it. If found, the flow is considered benign. If they are not used, then the flow is further checked to see if the data

TABLE II
RESULTS WHEN TESTING THE DATASET FOR SENDBROADCAST USAGE

	FAMAZOA	Github Repos	Total
Projects	387	1329	1716
Uses sendBroadcast (% out of total)	80 (20%)	120 (9%)	17 (%14)
Uses localbroadcastmanager (% out of sendBroadcast)	22 (%27.5)	200 (11%)	39 (%19.5)
Uses SetPackage (% out of sendBroadcast)	2 (%2.5)	13 (%10.3)	15 (%7.5)
Potential unsafe sendBroadcast (% out of sendBroadcast)	56 (%70)	90 (%75)	146 (%73)

being broadcast is a sensitive source or not. If it is sensitive, then the flow is marked for sensitive data leakage.

This answers **RQ1** and demonstrates the possibility of analyzing Kotlin apps directly using a static analysis approach, which requires a tool that can handle Kotlin's syntax as well as providing rules for marking and tagging relevant data.

C. Dataset and Results

For the dataset we began with FAMAZOA [9] dataset and augmented it with a dataset of the most popular GitHub Kotlin repositories. FAMAZOA is described as the largest publicly available dataset of open-source applications written in Kotlin. It has 387 open source applications from F-Droid AndroidTimeMachine, and AndroZoo. For the GitHub dataset we added all of the Kotlin GitHub projects with over 500 stars, which gave us a much larger dataset containing 1,329 repositories.

To test each app, we merged the Kotlin source files in each of the app's folders and ran our static analysis tool on the merged result. For the FAMAZOA dataset, we found that 283 projects did not use *sendBroadcast* while 80 projects did use it. Out of the 80 projects, 22 use *localbroadcastmanager* and 2 use *setPackage*. 24 projects were initially skipped due to parsing errors.

This means that 56 apps from this dataset could potentially be leaking user data. To explore this further, we manually examined these apps and what type of data is used to in broadcasts. Our criteria for benign data is when the application is sending publicly available data or data that is normally accessible to any other installed app, or if if it uses *sendBroadcast* alongside *localbrodcastmanager* or *setPackage* which mitigates the vulnerability. Otherwise if the app sends data that is usually not publicly available or has user information, we label it as sensitive. Overall, out of 80 apps we found 23 or around 29% to have sensitive data.

For the GitHub dataset, we found that 120 of the apps were using *sendBroadcast*. Of these, 17 of them were using *localbroadcastmanager* and 13 were using *setPackage*. The summary of these results is given in Table II. This answers **RQ2** and shows that while a minority of apps use *sendBroadcast*, many use them with implicit intent, which is unsafe and could potentially lead to sensitive information leakage.

IV. CASE STUDY

One noteworthy app in the Kotlin repository dataset is AndroidAPS [1], the only medical app included. We tested

Listing 6. AndroidAPS SendBroadcast Usage with setPackage

```
private fun sendBroadcast(intent: Intent) {
    val receivers: List<ResolveInfo> = context.
        packageManager.queryBroadcastReceivers(intent, 0)
    for (resolveInfo in receivers)
        resolveInfo.activityInfo.packageName?.let {
            intent.setPackage(it)
            context.sendBroadcast(intent)
            aapsLogger.debug(LTag.CORE, "Sending
                broadcast " + intent.action + " to: " +
                it)
```

Listing 7. AndroidAPS Unsafe SendBroadcast Usage

```
override fun sendCalibration(bg: Double): Boolean {
    val bundle = Bundle()
    bundle.putDouble("glucose_number", bg)
    bundle.putString("units", if (profileFunction.
        getUnits() == GlucoseUnit.MGDL) "mgdl" else "mmol
        ")
    bundle.putLong("timestamp", System.currentTimeMillis
        ())
    val intent = Intent(Intents.ACTION_REMOTE_CALIBRATION
        )
    intent.putExtras(bundle)
    intent.addFlags(Intent.FLAG_INCLUDE_STOPPED_PACKAGES)
    context.sendBroadcast(intent)
```

AndroidAPS *version 3.1.0*. Given that the security and safety of medical device systems can have serious implications for patient health and even life, we selected this app for our case study. Additionally, medical devices often handle sensitive information, making it crucial for these systems to adhere to guidelines and standards that ensure privacy. Moreover, the very use of a medical device can be considered private information, highlighting the need for the application to restrict broadcasts to known targets only. AndroidAPS serves as the controller app for an Artificial Pancreas System (APS), which consists of three main components: a blood glucose sensor, known as a continuous glucose monitor (CGM), a controller, and an insulin pump. The system can operate in either an open-loop or closed-loop configuration. In an open-loop system, after detecting blood glucose levels, the controller algorithm provides recommendations for the user to manually administer insulin. In contrast, a closed-loop system automates this process, allowing the controller to directly send commands to the insulin pump to deliver the appropriate insulin dosage without user intervention.

AndroidAPS is capable of communicating with various sensors and pumps and can integrate with other applications for data sharing and remote control. The app supports plugins for compatible glucose sensors. Based on the algorithm selected, the app reads input from the chosen sensor, calculates the necessary actions for the pump, and sends the corresponding commands. Throughout this process, AndroidAPS can be used in conjunction with optional wearables or cloud-based applications such as Nightscout [22].

The app is primarily written in Kotlin, totaling approximately 157,000 lines of code. There are 12 instances where *sendBroadcast* is used in the code. These include employing a custom *DataBroadcastPlugin* where *sendBroadcast* is safely used with *setPackage*, as shown in Listing 6, and a custom *sendBroadcastMessage* function that safely utilizes *LocalBroadcastManager*. The app uses broadcasts for various

functionalities, such as sharing app status, transmitting battery information, and communicating with widgets and wearable devices. We define our sink to be the *sendBroadcast* function, and we can define our sources to be any variables containing sensitive information; in the case of AndroidAPS this includes information relating to blood glucose, basal and bolus information, or user private information. In one instance, the app uses broadcast communication to interact with another application, xDrip [7]. xDrip functions as a data hub, facilitating data exchange between various devices and enabling intercommunication among different apps, including AndroidAPS. The *sendCalibration* function in AndroidAPS, shown in Listing 7, transmits blood glucose values via a broadcast without using *setPackage* or *LocalBroadcastManager*, thereby could expose the data to other apps. This is considered an unsafe use of broadcast communication, and it is recommended to explicitly specify xDrip as the intended receiver to enhance security. We also note that since writing this article, AndroidAPS received multiple updates.

V. Related Work

Several notable tools have been developed to analyze Android applications. Wei et al. [28] introduced Amandroid, which utilizes taint analysis by constructing data flow and dependency graphs for security assessments. Artz et al. [2] presented FlowDroid, emphasizing precision through call-graph analysis using the Soot framework [18]. HybriDroid, proposed by Chen et al. [4], integrates static and dynamic analysis to model application behavior comprehensively. Furthermore, Khedkar and Bodden [15] employed static analysis techniques to evaluate app compliance with privacy regulations, leveraging Java's intermediate representation. However, these tools are predominantly designed for Java-based Android apps and may not effectively address Kotlin-specific vulnerabilities, such as issues related to *sendBroadcast*.

Chin et al. [5] previously investigated inter-application communication vulnerabilities in Java-based Android apps, with a focus on *sendBroadcast*. Their research identified potential intent-based attacks, such as unauthorized reception and intent spoofing, and introduced ComDroid, a tool for detecting these vulnerabilities. However, ComDroid is not publicly accessible and does not target Kotlin applications. Abdul Moiz and Alalfi [20] examined inter-application communication vulnerabilities in automotive apps using a static analysis tool, AAVD, based on TXL [6], but their research was confined to the automotive sector and did not address Kotlin-specific issues.

Krishnamurthy et al. [16] explored the limitations of applying Java taint-analysis tools to Kotlin applications, highlighting specific challenges and suggesting possible solutions. They introduced *SecuChecl-Kotlin* as an extension to support to *Secucheck* and a proof of concept for the possibility of extending Java analysis tools to Kotlin. Out of 18 identified challenges, they proposed conceptual solutions for 8 and implemented six of them. In comparison, our approach targets Kotlin code directly and handles its grammar fully, not as an extension. Using a vulnerable repository of *PetClinic* made by the authors [16], our tool was able to detect the 6 flows that are detected by *SecuCheck-Kotlin*. Table III shows the results of the comparison, showing the source, sink and propagator functions within the *PetClinic* app. Our tool is able to match *SecuCheck-Kotlin* in detecting all the flows.

Existing static analysis tools, including SonarQube [26], PMD [24], Polyspace [19], and CodeSonar [13], are primarily designed to detect general code issues, such as bugs, errors, and code smells. Our research aims to bridge this gap by employing static analysis techniques specifically tailored to identify privacy vulnerabilities in Kotlin applications, with a particular emphasis on *sendBroadcast*-related risks. Other than Krishnamurthy's work, previous literature is not compatible with Kotlin. KASTroid differs in that it can handle Kotlin code directly and does not require executing or adding extensions that arises from handling bytecode.

Regarding vulnerabilities, the Android broadcast mechanism enables apps to communicate with the system and other applications. Security guidelines from the CERT Android Coding Standard [3], NowSecure Secure Mobile Development [23], and the Japan Smartphone Security Association (JSSEC) [14] advise against sending sensitive information via public implicit broadcasts. Instead, they recommend using explicit intents or restricting broadcasts to designated receivers.

VI. Threats to Validity

Some factors may affect the validity of our study. To our knowledge, this is the only study focused on detecting broadcast vulnerabilities in Kotlin Android apps. Consequently, there are no prior studies against which we can directly compare the performance and accuracy of KASTroid. To address this limitation, we conducted manual analysis to evaluate the tool's correctness and performance; however, the lack of benchmark studies remains a potential threat to external validity.

Another threat relates to the criteria for defining sensitive data. In our study, we considered data not publicly accessible or exposed by default as potentially sensitive. While developers may have insight into the nature of the data being shared and may intentionally expose some information they deem benign, this judgment may vary. Moreover, seemingly innocuous data can still pose security risks; for instance, information such as the night and day status could be exploited to infer the physical location of a user. Thus, our classification of sensitive data may not capture all possible threat scenarios.

VII. Conclusion

In this research, we studied the issue of potential data leakage arising from the unsafe use of *sendBroadcast* in Kotlin Android apps. To address this, we developed a tool which performs automatic backward slicing based on specified sinks and rules. We compiled a dataset of publicly available Kotlin apps by combining FAMAZOA with the most popular Kotlin repositories from GitHub. Our analysis revealed that *sendBroadcast* was used in 11% of the apps, and nearly three-quarters (73%) of these instances were implemented unsafely,

TABLE III
COMPARISON OF TAINTED FLOW DETECTION IN CUSTOM VULNERABLE PETCLINIC REPOSITORY

	Source	Propagator	Sink	SecuCheck-Kotlin	KASTroid
Flow 1	ShowOwner (OwnerController: 116)	createQuerry (OwnerRepositoryCustomImp: 26)	singleResult (OwnerRepositoryCustomImp: 30)	✓	✓
Flow 2	initUpdateOwnerForm (OwnerController: 92)	createQuerry (OwnerRepositoryCustomImp: 26)	singleResult (OwnerRepositoryCustomImp: 30)	✓	✓
Flow 3	processFindForm (OwnerController: 70)	findByLastName(LastName) (OwnerRepositoryCustomImp: 14)	resultList (OwnerRepositoryCustomImp: 21)	✓	✓
Flow 4	findOwner (PetController: 44)	createQuerry (OwnerRepositoryCustomImp: 26)	singleResult (OwnerRepositoryCustomImp: 30)	✓	✓
Flow 5	processCreationForm (OwnerController: 54)	-	merge (OwnerRepositoryCustomImp: 38)	✓	✓
Flow 6	processCreationForm (OwnerController: 54)	-	persist (OwnerRepositoryCustomImp: 41)	✓	✓

Legend: File and line numbers in parenthesis. ✓ = Flow Detected

without employing *LocalBroadcastManager* or *setPackage*. While this finding suggests potential sensitive data leakage, it is important to note that the broadcasted data could be benign or publicly accessible to other apps. A detailed manual examination indicated that around 23% of the apps in FAMA-ZOA that used *sendBroadcast* were indeed leaking sensitive information.

REFERENCES

[1] AndroidAPS. Androidaps. androidaps.readthedocs.io/. Accessed: 2024-09-20.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 259–269, New York, NY, USA, 2014. Association for Computing Machinery.

[3] CERT Coordination Center. CERT Android coding standard. https://wiki.sei.cmu.edu/confluence/display/android/DRD03-J.+Do+not+broadcast+sensitive+information+using+an+implicit+intent. Accessed: 2024-09-20.

[4] Hongyi Chen, Ho-fung Leung, Biao Han, and Jinshu Su. Automatic privacy leakage detection for massive Android apps via a novel hybrid approach. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–7, 2017.

[5] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, page 239–252, New York, NY, USA, 2011. Association for Computing Machinery.

[6] J.R. Cordy, C.D. Halpern, and E. Promislow. Txl: a rapid prototyping system for programming language dialects. In *Proceedings. 1988 International Conference on Computer Languages*, pages 280–285, 1988.

[7] Nightscout Foundation. xDrip. https://github.com/NightscoutFoundation/xDrip. Accessed: 2024-09-20.

[8] The OWASP Foundation. Owasp mobile top 10. https://owasp.org/www-project-mobile-top-10/. Accessed: 2024-09-20.

[9] Bruno Góis Mateus and Matias Martinez. An empirical study on quality of Android applications written in Kotlin language. *Empirical Software Engineering*, Jun 2019.

[10] Google. Android developers - broadcasts overview. https://developer.android.com/develop/background-work/background-tasks/broadcasts. Accessed: 2024-09-20.

[11] Google. Android developers - Kotlin. https://developer.android.com/kotlin. Accessed: 2024-09-20.

[12] Google. Kotlin docs - grammar. https://kotlinlang.org/docs/reference/grammar.html. Accessed: 2024-09-20.

[13] GrammaTech. Codesonar. https://www.grammatech.com/codesonar-cc. Accessed: 2024-09-20.

[14] Japan Smartphone Security Association (JSSEC) Secure Coding Working Group. Android application secure design/secure coding guidebook. https://www.jssec.org/dl/android_securecoding_en/4_using_technology_in_a_safe_way.html#when-sending-sensitive-information-with-a-broadcast-limit-the-receivable-receiver-required. Accessed: 2024-09-20.

[15] Mugdha Khedkar and Eric Bodden. Toward an Android static analysis approach for data protection. In *Proceedings of the IEEE/ACM 11th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '24, page 65–68, New York, NY, USA, 2024. Association for Computing Machinery.

[16] Ranjith Krishnamurthy, Goran Piskachev, and Eric Bodden. To what extent can we analyze kotlin programs using existing java taint analysis tools? In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 230–235. IEEE, 2022.

[17] Ranjith Krishnamurthy, Goran Piskachev, and Eric Bodden. To what extent can we analyze kotlin programs using existing Java taint analysis tools? (extended version), 2022.

[18] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, 2011.

[19] MathWorks. Polyspace. https://www.mathworks.com/products/polyspace.html. Accessed: 2024-09-20.

[20] Abdul Moiz and Manar H. Alalfi. An approach for the identification of information leakage in automotive infotainment systems. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 110–114, 2020.

[21] Bara' Nazzal and Manar H. Alalfi. An automated approach for privacy leakage identification in iot apps. *IEEE Access*, 10:80727–80747, 2022.

[22] NightScout. Nightscout. https://nightscout.github.io/nightscout/new_user. Accessed: 2024-09-20.

[23] NowSecure. Nowsecure secure mobile development. https://github.com/nowsecure/secure-mobile-development/blob/master/en/android/avoid-intent-sniffing.md. Accessed: 2024-09-20.

[24] PMD. PMD. https://pmd.github.io/. Accessed: 2024-09-20.

[25] Gulshan Shrivastava, Prabhat Kumar, Deepak Gupta, and Joel JPC Rodrigues. Privacy issues of Android application permissions: A literature review. *Transactions on Emerging Telecommunications Technologies*, 31(12):e3773, 2020.

[26] SonarSource. Sonarqube. https://www.sonarsource.com/products/sonarqube/. Accessed: 2024-09-20.

[27] Luca Verderame, Davide Caputo, Andrea Romdhana, and Alessio Merlo. On the (un)reliability of privacy policies in Android apps. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–9, 2020.

[28] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 1329–1341, New York, NY, USA, 2014. Association for Computing Machinery.