Uncovering URL-Based Vulnerabilities in Android Apps via Static Slicing Techniques

Omkar Wagle[1], Anand Kumar Singh[2]

[1]ov.wagle@gmail.com, Independent Researcher, CA
[2]anand.krs@gmail.com, Independent Researcher, WA

Abstract
The increasing adoption of the Internet of Things (IoT) has led to a surge in interconnected devices, many of which leverage URLs for data communication between smart devices and servers. However, this reliance on URLs introduces security vulnerabilities, as malicious actors may exploit unprotected URL paths to gain unauthorized access. To mitigate these risks, it is crucial to detect potential vulnerabilities in Android applications during the development stage through static analysis. This paper presents a method for extracting URLs from Android APK files using static analysis, employing the Leakscope tool built on the Soot framework. Leakscope leverages taint analysis to track the flow of data from sources to sinks, identifying potential vulnerabilities that could lead to unauthorized data exposure or system compromise. The analysis process includes backward slicing to trace data flow and forward slicing to reconstruct the URL, thereby enabling developers to examine how user inputs impact URL formation. Experiments were conducted on two Android applications: a single-activity APK developed for this study and the multi-activity Samsung Galaxy Wearable APK. The results demonstrated the efficacy of Leakscope in identifying URL formation in simple applications, while revealing challenges when analyzing complex, multi-activity applications. Additionally, enhancements were made to Leakscope to support various string operations and literal data types. The findings underscore the need to extend Leakscope's capabilities to handle loops and intents, as their absence limits the accuracy of detecting URL vulnerabilities. Future work will address these limitations to improve static analysis accuracy and reliability in complex Android applications.

Keywords: Leakscope, Soot, Internet of Things, Uniform Resource Locator (URL), static analysis, taint analysis, Android APK, source-to-sink

1. Introduction

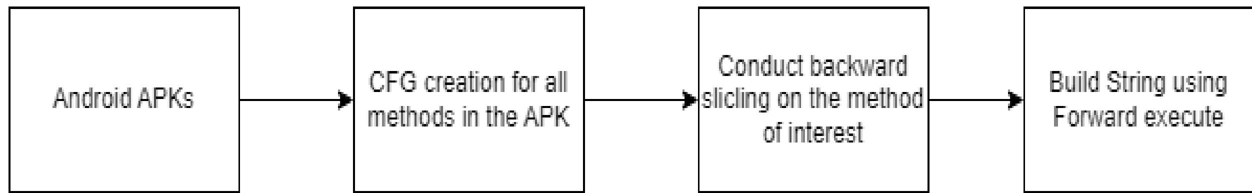
The rise of the Internet of Things (IoT) [1] has led to an explosion of connected devices and applications, all of which rely on various forms of communication protocols to exchange data and interact with other devices. One of the most commonly used communication protocols is the **Uniform Resource Locator (URL)**[2] which enables devices to retrieve information from the internet. Coming to the IoT applications, URLs are often form the backbone of data propagation as its used to propagate data transfers between a smart device and server, or between smart devices, or to control any smart devices from a remote location and many such applications. For example, in a smart home, the user may need to control his HVAC, lights, temperature and many such smart devices. If we take a healthcare facility, we may need to monitor the vitals of the patient remotely. All these data transfers are done through an API that uses URLs to communicate. However, improper validation of URLs or unprotected

endpoints has led to several real-world security incidents. For example, the Kasa Smart Home breach (2021) exposed device control URLs that allowed unauthorized access to camera and lighting systems. Similarly, the Strava fitness app leak (2018) revealed sensitive military base locations due to poorly secured API endpoints accessible via URLs. These incidents illustrate how attackers can exploit open or weakly validated URL patterns to gain unauthorized access or exfiltrate sensitive data. So, the use of URLs in IoT applications introduces new security challenges, which allows attackers to target the path that controls the access control feature that gives complete access to the user of the smart device and can gain unauthorized access to sensitive information through APIs. Therefore, as a software developer, there is a certain amount of responsibility that we should take to understand the possible risks and best practices associated with using URLs in IoT applications to keep the entire IoT ecosystem secure. Furthermore, IoT apps are often distributed via APK files, making them vulnerable to reverse engineering and repackaging attacks. Using tools like JADX or APK Tool, attackers can decompile the APK, extract hardcoded URLs or API keys, and inject malicious payloads or redirect traffic to rogue servers. Once modified, these APKs can be redistributed through unofficial channels. Static analysis during development can help mitigate such risks by flagging insecure URL constructions and potential taint paths early in the

Based on the security risks, it becomes important for the developers to detect such presence of URLs in any applications at the development stages and avoid any security lapses by conducting extensive tests. The most used technique to retrieve such information is by conducting **static analysis** [3] of the application. We used a static analysis tool called **Leakscope**[4] that uses **Soot** [5] as its base to extract strings from the method of interest used inside the application. **Soot** is a static analysis tool used to identify software security vulnerabilities in the system. It works on analyzing the bytecode of the application rather than analyzing it at the runtime. Soot uses a technique known as **taint analysis**[6] which traces the flow of the data through the application and determines whether that data could be manipulated by any user inputs. Soot looks for the sensitive data that could be sent through a user input generally known as **source**[7] and tracks it through the application to check how this data is propagated through the application and used in a way that could compromise system security generally known as **sink**[7]. There are instances that an attacker can manipulate the source input to gain control of the vulnerable sink. This is known as a “source-to-sink” attack.[7] For example, suppose an application accepts user inputs through a login page. All the data entered in the login form is then stored in a SQL database. If the application fails to put a check on the data entered through the login form, the attacker could inject a malicious SQL query as an input to a field of the login form. If the application then uses this input to construct a SQL query, the attacker might be able to execute an arbitrary SQL query on the database. This is an example of a “source-to-sink” attack.

In this paper, we are using Leakscope as our static analysis tool. Leakscope uses taint analysis to detect data flow from source to sink in Android applications. As a URL is the link between a source and sink, it becomes important to track down how the URL was constructed, and the impact of user input on the URL. Speaking in terms of source-to-sink attacks[7], if the Android application does not have enough checks on the user input, the URL might be constructed in such a manner that it can either manipulate data in the sink or can expose the sink and its content to the attacker. Hence, we are using Leakscope to track down the construction of URLs in the Android application and analyze if all checks to the user inputs are added or not.

2. Implementation



To begin with, an Android application is executed with the help of the **Leakscope** tool. The Android application undergoes several stages:

2.1. Control Flow Graph:

Leakscope constructs a **Control Flow Graph (CFG)** of the Android application using the Soot Framework. To generate the CFG, Leakscope uses Soot to analyze the bytecode of the application and produce an Intermediate Representation (IR) of the code. The IR is a low-level abstraction of the application's code that allows for analysis of both control flow and data flow. Using the IR, Leakscope then generates the CFG, which represents the application's control flow. In this graph, each node corresponds to a basic block, labeled with the statements it contains, and each edge represents a control transfer between blocks. The direction of the edges reflects the order of the basic blocks in the program. Similarly, Leakscope constructs a CFG for all the methods within the application.

2.2. Example JSON:

After the CFG creation stage, Leakscope searches for the presence of the method of interest across all the methods
in the application. This method could belong to the application itself or another package. The method and its associated parameter of interest are defined in a JSON file that Leakscope uses for this search. The parameter of interest refers to a specific variable in the method that Leakscope tracks for any modifications. Any changes to this parameter are noted and reported in the results.

2.3. Backward Slicing:

Once the method of interest is detected, Leakscope begins a process called **Backward Slicing**. Backward Slicing is a static analysis technique in which the tool identifies a set of statements that influence the target variable. This technique works by analyzing the control flow and data flow backward to trace statements that affect the variable. In security analysis, backward slicing is used to identify statements that may introduce vulnerabilities. This helps developers focus on critical areas of the code to address vulnerabilities, such as preventing sensitive data from leaking to insecure locations. Essentially, when Leakscope identifies the method of interest, it tracks the specified parameter for any changes. Then, it traverses the CFG backward. In cases where multiple paths lead to a node, Leakscope adds to the stack the node or statement that affects the target variable. It continues this process, traversing all nodes to reach the base node of the CFG, adding statements that impact the target variable to the stack.

2.4. Forward Slicing:

Upon completing backward slicing, Leakscope proceeds with a process called **Forward Slicing**. Forward slicing involves analyzing the program starting at a specific point and following the code forward to determine which parts of the program may be influenced by that point. This technique is the opposite of backward slicing, which traces the program backward to understand how a particular point

was reached. In forward slicing, Leakscope pops statements from the stack and executes them. This process continues until the stack is empty. Once the stack is empty, Leakscope has executed all the statements impacting the target variable and generates the final string. This final string serves as the output of the Leakscope analysis.

3. Comparison with Other Static Analysis Tools

Static analysis has become a fundamental approach for identifying security vulnerabilities in Android applications. Numerous tools are available to assist in this task, each with unique capabilities, strengths, and limitations. In this section, we compare Leakscope with three widely recognized tools in the domain: FlowDroid, DroidSafe, and Androguard.

3.1. FlowDroid:

FlowDroid is a widely used static taint analysis tool that performs context-sensitive, flow-sensitive, field-sensitive, and object-sensitive analysis on Android applications. It builds on the Soot framework and leverages detailed modeling of the Android lifecycle to produce accurate taint flows.

- Strengths:
  - Lifecycle-aware analysis improves accuracy for Android-specific behavior.
  - Offers high precision in detecting data leaks from sensitive sources to sinks.
- Limitations:
  - Complex to set up and configure, requiring detailed environment preparation.
  - Can be computationally intensive on large or obfuscated APKs.
- Suitability for URL Analysis:
  - While FlowDroid is excellent at taint tracking, it lacks native support for string reconstruction, making it less effective for tracing complex URL formation logic involving string concatenations and builder patterns.

3.2. DroidSafe

Developed by MIT Lincoln Laboratory, DroidSafe extends beyond basic taint tracking by offering inter-component data flow analysis and support for inter-procedural slicing. It is designed for deep semantic analysis of Android bytecode.

- Strengths:
  - Tracks information flows across activities and services.
  - Supports advanced features like policy enforcement checks and object modeling.
- Limitations:
  - High memory and processing requirements make it impractical for rapid or lightweight analyses.
  - Limited updates and support for newer Android SDKs.
- Suitability for URL Analysis:
  - DroidSafe is capable of cross-component tracking which is advantageous in cases where URLs are assembled over multiple activities. However, it lacks a user-configurable mechanism for method-level string tracking like Leakscope's JSON-based setup.

3.3. Androguard

Androguard is a Python-based tool primarily used for reverse engineering and analyzing Android applications. It includes functionalities for disassembly, control flow graph generation, and signature-based malware detection.

- **Strengths:**
  - Lightweight and easy to integrate into automated pipelines.
  - Useful for inspecting DEX files, certificates, and permissions.
- **Limitations**:
  - Lacks advanced taint analysis capabilities.
  - Not intended for detailed data flow or string tracing.
- **Suitability for URL Analysis**:
  - Androguard is more suited for structural APK inspection and not ideal for tracing URL construction through taint propagation.

3.4. Why Leakscope Was Chosen

Leakscope was selected as the primary tool for this research due to its focused capability in analyzing string construction paths using a combination of backward and forward slicing. It supports**:**

- Custom JSON-based method targeting, allowing selective tracking of sensitive methods like StringRequest or URL.
- Slicing operations that are essential for identifying how a string (e.g., a URL) is assembled over multiple instructions or through various utility methods.
- Extensibility, allowing enhancements such as support for StringBuilder.append(), String.valueOf(), and various literal types, which were crucial for improving its URL analysis accuracy in this study.

While tools like FlowDroid and DroidSafe offer broader analysis coverage and greater precision in general taint tracking, they are either not optimized for string reconstruction or are too heavyweight for rapid experimentation. Leakscope strikes a balance by providing targeted insights into data flow involving URLs, making it ideal for static analysis of IoT and mobile apps where URL manipulation is a potential attack vector.

4. Steps to run Leakscope

Step 1: Download Leakscope Source Code: First, download the Leakscope source code from the GitHub repository by visiting the following link: https://github.com/OSUSecLab/LeakScope

Step 2: Download the .jar File: Next, download the ValueSetAnalysis.jar file from the Leakscope releases page here:

https://github.com/OSUSecLab/LeakScope/releases/download/1.0/ValueSetAnalysis.jar

Step 3: Place the Test APK: Take the test .apk file you want to analyze and place it in the example folder inside the Leakscope directory.

Step 4: Edit the .json Configuration File:
In the example folder, you will find a .json configuration file. Open this file for editing. This .json file contains three key fields:

i. apk: Specify the path to the APK file you placed in the example folder. This is the APK that Leakscope will analyze.
ii. method: Enter the name of the method in the APK that you are interested in monitoring for leaks.
iii. parmIndexs: Provide the index of the parameter in the specified method that you want to track.

Step 5: Run the Leakscope tool with the following command:

void <init>(parameters…) specifies as the constructor of the method of interest.

java -jar ValueSetAnalysis.jar ./libs/android.jar ./example/example.json

5. System Requirements (Windows OS)

- Leakscope requires Java 8 to be installed on your local machine.
- Set the system path to include the following directories:
  - C:\Program Files\Java\jdk1.8.0_361\bin
  - C:\Program Files\Java\jre1.8.0_361\bin
- Set the JAVA_HOMEenvironment variable to:
  - C:\Program Files\Java\jdk1.8.0_361

6. Experiment

I conducted experiments with two Android applications. The first is a simple, single-activity Android application where a button press triggers an HTTP GET request. The second is the Samsung Galaxy Wearable application, available on the Google Play Store. I ran Leakscope with both APKs to analyze their behavior.

Experiment 1: Leakscope on a Single-Activity APK

The single-activity application [11] has basic functionality: when the user clicks a button on the main screen, an HTTP GET request is sent to an API. A single button click constructs the URL and initiates the GET request. Since the experiment involves URL extraction, I focused on a method called String Request[12]. This method is relevant because after the URL is fully constructed, it is passed to String Request. Therefore, the method of interest for this experiment is String Request.

Example of my JSON file:

```
{
  "apk": "example/app-debug.apk",
  "methods": [{
     "method": "<com.android.volley.toolbox.StringRequest: void
<init>(int,java.lang.String,com.android.volley.Response$Listener,com.android.volley.Response$ErrorListener)>
",
     "parmIndexs": [1]
   }]
}
```

When analyzing an Android APK using LeakScope, the tool performs the following steps: Control Flow Graph (CFG) Construction:

- LeakScope generates CFGs for all methods within the APK.

Identification of Target Method and Parameter:

- It references a configuration file, such as example.json, to identify the method of interest—StringRequestin this case—and the specific parameter to track.

Method Location in Bytecode:

- LeakScope searches for the identified method within the application's Java bytecode.

Backward Slicing:

- Upon locating the method, LeakScope initiates a backward slicing process, tracing the code backward to determine all statements or nodes that directly influence the tracked variable.

String Reconstruction:

- Once the base node is reached, LeakScope reconstructs the complete string by sequentially processing the collected statements from the stack until it is empty.

This analysis yields the following result:

dev.navids.multicomp1[{"1":["https://api.coingecko.com/api/v3/simple/price?ids=bitcoin&vs_currencies=usd/"]}]

This indicates that the application dev.navids.multicomp1 makes a request to the specified URL.

The fundamental approach to detecting URLs in this application involves examining how they are constructed, which allows us to assess the influence of user inputs on the final URL. By analyzing the URL construction process, we can determine whether specific user inputs could lead to the formation of URLs that might compromise the security of critical components, such as databases or servers. Implementing input validation mechanisms at the initial stages of data processing is essential to prevent the acceptance of potentially harmful strings. This proactive measure ensures that user inputs are scrutinized before they can impact the application's behavior, thereby mitigating security risks.

Experiment 2: Applying LeakScope to a Multi-Activity APK (Samsung Galaxy Wearable)

In this experiment, I utilized LeakScope to analyze the Samsung Galaxy Wearable APK, obtained from the Google Play Store. Unlike the previous experiment, where I had prior knowledge of the specific methods and parameters to target due to designing the APK myself, this analysis required a different approach. I focused on the commonly used URL class for HTTP requests, which has several overloaded methods. I conducted experiments across these various methods, and the results are as follows:

- URL (String address):


```
com.samsung.android.app.watchmanager[{"0":["https://diagmon-serviceapi.samsungdm.com/v2/common/authtoken"]}]
com.samsung.android.app.watchmanager[{}]
com.samsung.android.app.watchmanager[{"0":["https://android.googleapis.com/attestation/status"]}]
com.samsung.android.app.watchmanager[{"0":["https://diagmon-serviceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]},{"0":[
"https://diagmon-serviceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]},{"0":["https://diagmon-serviceapi.samsungdm.com/v
2/policy?policyVersion=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersio
n=&dmaVersion=ged&tmcc=&smcc="]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]},{"0":["htt
ps://diagmon-serviceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/po
licy?policyVersion=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersion=&d
maVersion=ged&tmcc=&smcc="]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]},{"0":["https:/
/diagmon-serviceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/policy
?policyVersion=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersion=&dmaVe
rsion=ged&tmcc=&smcc="]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]},{"0":["https://dia
gmon-serviceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/policy?pol
icyVersion=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersion=&dmaVersio
n=ged&tmcc=&smcc="]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]},{"0":["https://diagmon
-serviceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/policy?policyV
ersion=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersion=&dmaVersion=ge
d&tmcc=&smcc="]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]},{"0":["https://diagmon-ser
viceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/policy?policyVersi
on=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersion=&dmaVersion=ged&tm
cc=&smcc="]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/policy?policyVersion=&currentPolicyVersion=&dmaVersion=ged&tmcc=&smcc="]}]
com.samsung.android.app.watchmanager[{}]
com.samsung.android.app.watchmanager[{"0":["https://stg-api.samsungcloud.com/device/v1/token"]},{"0":["https://api.samsungcloud.com/device/v1/token"]}]
com.samsung.android.app.watchmanager[{"0":["https://diagmon-serviceapi.samsungdm.com/v2/eventreport/result"]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/common/
serviceregistration"]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/eventreport"]},{"0":["https://diagmon-serviceapi.samsungdm.com/v2/eventreport/result"]}]
com.samsung.android.app.watchmanager[{}]
com.samsung.android.app.watchmanager[{"0":[""]},{"0":[""]},{},{}]
com.samsung.android.app.watchmanager[{}]
com.samsung.android.app.watchmanager[{},{}]
com.samsung.android.app.watchmanager[{"0":["https://qa-odc.samsungapps.com/ods.as"]}]
com.samsung.android.app.watchmanager[{"0":["https://diagmon-policy.samsungdm.com"]}]
com.samsung.android.app.watchmanager[{}]
com.samsung.android.app.watchmanager[{}]
com.samsung.android.app.watchmanager[{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{},{}]
com.samsung.android.app.watchmanager[{}]

C:\Omkar-Docs\LeakScope_Test>
```

- The example.json for above experiment is:

```
{
    "apk":"example/Galaxy-Wearable.apk",
    "methods":  [{
                    "method":"<java.net.URL: void <init>(java.lang.String)>",
                              "parmIndexs":[0]
                 } ]
}
```

- No other overridden method of URL was used in the APK to make any HTTP calls.

7. Enhancements Implemented in LeakScope

During the testing of single-activity applications, it was observed that LeakScope did not recognize modifications to target variables when certain methods were employed. To address this, support for the following methods was incorporated into BackwardContext.java and SimulateEngine.java within the forward execution process. This integration ensures that LeakScope can detect these methods during both backward and forward slicing phases. In backward slicing, the tool identifies methods that alter the target variable and adds them to the stack. Conversely, in forward slicing, the methods are executed on the variable.

- **String.concat()**: Support was added to BackwardContext.java to detect this method during backward slicing and to SimulateEngine.java for consideration during the forward slicing process of string construction.
- **StringBuilder.append()**: Support for StringBuilder.append() was integrated into BackwardContext.java for detection during backward slicing and into SimulateEngine.java for the forward slicing process. Additionally, support was extended to all data types that can be appended to a string, including int, float, double, long, boolean, and char.
- **String.valueOf()**: This method returns the string representation of the passed parameter(s). Support for all its overloaded versions was added to ensure comprehensive detection of string representations involved in URL construction.
- **Arrays**: Support for arrays of all data types was incorporated. Given that arrays are fundamental data structures in software development, this addition was essential.
- **Other Literal Values**: While the primary focus of our experiment is on string construction and its origins, it was necessary to add support for other Java literals. Although not common practice, Java allows concatenation of strings using the + operator with various literals. Therefore, incorporating support for all literal types was crucial to ensure no string modifications within the application are overlooked.

8. Limitations

During the experiments, several limitations of LeakScope were identified:

- Lack of Loop Support: LeakScope does not currently handle modifications to target variables within loops like *for*, *while*, or *do-while*. Consequently, any changes made to a variable inside a loop are not reflected in the final string analysis. For example,

```
String url = https://example.com.api?
For (String param: params) {
url+=pram + "&".
}
```

In the above code, url is constructed dynamically inside a loop. Leakscope is unable to track such iterative concatenations and thus fails to reconstruct the full URL or understand the influence of each loop iteration.

- No Support for Intents and Inter-Activity Communications: If a target variable is modified across different activities within an Android application, LeakScope fails to track these changes. This limitation means that alterations made to the variable in separate activities are not captured in the final string analysis. For instance:

```
Intent i = new Intent(MainActivity.this, SecondActivity.class);
i.putExtra("url_base", "https://iot.example.com/data");
startActivity(i);
```

In Second Activity, this URL might be retrieved and appended with user input to form a complete request. However, Leakscope's static slicing is confined to a single method or activity. It cannot propagate data flow across such inter- component communication, leading to blind spots in URL formation tracking and potential underreporting of security risks.

Additionally, LeakScope has other limitations, such as its reliance on a predefined list of cloud APIs for detecting checked-in secrets, which may restrict its ability to identify all potential data leaks.

Addressing these issues would require augmenting Leakscope with inter-procedural analysis across activities and basic loop unrolling logic, which remains an open area for future work.

9. Future work

While Leakscope offers a powerful method for analyzing URL construction using static slicing, there are several avenues for enhancing its capabilities. One key area involves integrating machine learning-based heuristics to identify obfuscated or dynamically constructed URLs that are not explicitly visible in the bytecode. By training models on labeled datasets of URL-building code patterns, the tool could begin to infer implicit or hidden strings formed through encoded representations, reflection, or runtime method resolution—scenarios where traditional rule-based slicing may fail.

Another promising direction is enabling support for analyzing obfuscated APKs. Android developers frequently use tools like ProGuard or R8 that rename classes and strip metadata, making static analysis significantly harder. Incorporating de-obfuscation methods and advanced control/data flow resolution techniques would help adapt Leakscope for analyzing such code structures and maintain high accuracy even when symbols and method names are masked.

Additionally, to align with modern development workflows, Leakscope can be packaged into a lightweight plugin or command-line tool that integrates with CI/CD pipelines. This would allow automated analysis of APKs during the build process, enabling developers to catch insecure URL patterns and improper string construction early in the lifecycle— before release to production. Bringing Leakscope into secure DevOps environments would bridge the gap between static analysis research and

real-world application security.

References

1. Oracle, "What is IoT? - Internet of Things Definition," Oracle.com. [Online]. Available: https://www.oracle.com/internet-of- things/what-is-iot/#:~:text=The%20Internet%20of%20Things%20(IoT)%20describes%20the%20network%20of%20physical,and%20syste ms%20over%20the%20internet. [Accessed: 03-Apr-2023].
2. “About this document,” *Uniform Resource Locators*. [Online]. Available: https://www.w3.org/Addressing/URL/url- spec.html. [Accessed: 14-Apr-2023].
3. TechTarget, "Static Analysis (Static Code Analysis),"TechTarget.com, 2021.[Online].Available: https://www.techtarget.com/whatis/definition/static-analysis-static-code-analysis#:~:text=Static%20analysis%2C%20also%20called%20static,code%20adheres%20to%20industry%20standards. [Accessed: 14-Apr-2023]
4. Zuo C, Lin Z, Zhang Y. Why Does Your Data Leak? Uncovering the Data Leakage in Cloud from Mobile Apps. In: 2019 IEEE Symposium on Security and Privacy (SP) [Internet]. San Francisco, CA, USA: IEEE; 2019 [cited 2023 Apr 10]. p. 1296–310. Available from: https://ieeexplore.ieee.org/document/8835301/
5. Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V. Soot: a Java bytecode optimization framework. In: CASCON First Decade High Impact Papers on - CASCON ’10 [Internet]. Toronto, Ontario, Canada: ACM Press; 2010 [cited 2023 Apr 10]. p. 214–24. Available from: http://portal.acm.org/citation.cfm?doid=1925805.1925818
6. C. Kaestner, "Taint Analysis," Carnegie Mellon University, Pittsburgh, PA, USA, Oct. 2018. [Online]. Available: https://www.cs.cmu.edu/~ckaestne/15313/2018/20181023-taint-analysis.pdf. [Accessed: 14-Apr-2023].
7. Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, et al. 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 [Internet]. Edinburgh United Kingdom: ACM; 2014 [cited 2023 Apr 10]. p. 259–69. Available from: https://dl.acm.org/doi/10.1145/2594291.2594299
8. The GCC Development Team, "Control Flow," GCC.gnu.org. [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/Control-Flow.html. [Accessed: 15-Apr-2023].
9. Wikipedia contributors, "Intermediate representation," Wikipedia.org. [Online]. Available: https://en.wikipedia.org/wiki/Intermediate_representation#:~:text=An%20intermediate%20representation%20(IR)%20is,suc h%20as%20optimization%20and%20translation. [Accessed: 15-Apr-2023].
10. Piazza, "Program Slicing," Piazza.com, [Online]. Available: https://piazza.com/class_profile/get_resource/hy7enxf648g7me/i37ncz24ztf1h2. [Accessed: 16-Apr-2023].
11. "Activity," Android Developers, [Online]. Available: https://developer.android.com/reference/android/app/Activity. [Accessed: Apr. 16, 2023].
12. "StringRequest," Javadoc.io, [Online]. Available: https://javadoc.io/static/com.android.volley/volley/1.1.1/com/android/volley/toolbox/StringRequest.html. [Accessed: Apr. 16, 2023].
13. "URL," Oracle Documentation, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/net/URL.html. [Accessed: Apr. 16, 2023].

14. "Intent," Android Developers, [Online]. Available: https://developer.android.com/reference/android/content/Intent. [Accessed: Apr. 16, 2023].