SE 801 : Software Project Lab - 3 (Final Report)

IoTWhiz: An IoT Android App Characterization Tool

Submitted By

Shafiq-us Saleheen

Roll: 1125

Internal Supervisor

Dr. Zerina Begum

Professor

Institute of Information Technology,

University of Dhaka

Supervised By :

Dr. Adwait Nadkarni

Assistant Professor

Department of Computer Science,

William & Mary

Mentored By:

Kaushal Kafle

PhD Candidate

William & Mary

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my project supervisor, Dr. Adwait Nadkarni Sir, Assistant Professor, William & Mary, Williamsburg VA, for giving me an opportunity to go on with this project and providing unconditional support throughout the project selection phase. I would also like to thank my internal supervisor, Dr. Zerina Begum Ma'am, Professor, Institute of Information Technology, University of Dhaka. Her invaluable guidance and motivation helped me go further in this project. It is an honor to work under her supervision. Finally, I would like to pay my gratitude to my mentor, Kaushal Kafle, PhD candidate, William & Mary for being available whenever I needed guidance.

ABSTRACT

In the world of mobile applications, the task of characterizing Android apps, specifically in terms of their association with the Internet of Things (IoT) domain, has proven to be quite challenging. In response, we present a comprehensive framework designed to systematically analyze Android applications based on their specific attributes. Our approach starts with collecting a diverse dataset consisting both IoT and non-IoT apps from popular sources for analysis. By examining the collected applications, we extract simple metrics that serve as discriminators between the two categories. These metrics contain diverse aspects, including API usage, permission patterns, dynamic code loading, typical UI layouts, class count and code size measured in LOC, use of reflection, and data storage strategies in the application's codebase. The proposed tool automates the process of analyzing Android applications to extract these metrics related to their functionality and codebase. By applying static code analysis techniques, it collects data from a curated dataset of applications representing both IoT and non-IoT domains. The tool's effectiveness is demonstrated through an empirical analysis of a diverse set of Android applications, wherein it quantifies the differences between IoT and non-IoT apps across various dimensions. These findings shed light on the distinct characteristics that define each category, enabling developers to make informed decisions and researchers to deepen their understanding of IoT application development. The contributions of this work are twofold: 1) the development of a tool that automates the extraction of metrics from Android applications, and 2) the insights gained from the comparative analysis of IoT and non-IoT app characteristics. There will be visualizations to reflect analysis output, ranging from API usage distribution charts to dynamic code loading frequency pie charts, which will showcase the findings. In conclusion, our systematic approach serves as a guiding light in a complex terrain. By breaking down the process into discrete steps, we empower researchers with a potent tool capable of automatically testing between IoT and non-IoT Android applications, leading to more informed and efficient decision-making in the dynamic app development landscape.

Table of Contents

Chapter 1

Introduction to IoTWhiz

IoTWhiz is a specialized software tool designed for in-depth characterization and analysis of IoT (Internet of Things) Android applications. It provides essential insights into the structure, dependencies, security, and user interface elements of IoT apps. IoTWhiz is tailored to address the unique challenges posed by IoT app development.

1.1 Motivation

The advancement of Internet of Things (IoT) technology has ushered in a new era of interconnected devices and applications, revolutionizing the way we interact with our surroundings and making our lives more convenient and efficient. IoT has found applications in various domains, from smart homes and wearable devices to industrial automation and healthcare systems. However, the rapid growth of IoT has also brought forth unique challenges, particularly in the development and security of IoT-centric Android applications.

As IoT continues to integrate into our daily lives, understanding and characterizing IoT Android applications become crucial. These applications play a pivotal role in facilitating communication between the user's mobile device and IoT devices, sensors, and services. Furthermore, ensuring the security, efficiency, and reliability of these applications is paramount, as they often handle sensitive data and control critical functions in IoT ecosystems.

The motivation behind this research project stems from the need to comprehensively analyze and characterize IoT Android applications in terms of their code quality, architecture, and security considerations. By developing an analysis tool and conducting an in-depth examination of a diverse dataset of IoT and non-IoT Android apps, we aim to address the following key objectives:

1. Differentiation: To distinguish IoT Android applications from their non-IoT counterparts based on code characteristics, architecture, and usage patterns.

2. Insight Generation: To generate valuable insights into the common practices, challenges, and anomalies prevalent in IoT app development.

3. Security Assessment: To identify security vulnerabilities and best practices in IoT Android apps, aiding developers in building more secure applications.

4. Contribution to Knowledge: To contribute to the broader academic and research community by offering a comprehensive understanding of the unique features and considerations in IoT Android app development.

This research project not only serves as a valuable resource for developers, researchers, and practitioners in the IoT domain but also contributes to the advancement of knowledge in software engineering, mobile application development, and cybersecurity. By undertaking this, we aim to shed light on the intricate landscape of IoT Android applications, ultimately improving their quality, reliability, and security in an increasingly interconnected world.

1.2 Problem Description

In today's connected world, IoT has become an integral part of our lives. IoT Android applications serve as the bridge between our mobile devices and a myriad of interconnected smart devices and services, offering convenience, efficiency, and functionality. However, with the rapid growth of IoT technology, the development and analysis of these applications pose unique challenges.

1. Distinguishing IoT from Non-IoT Apps: The first challenge is differentiating between IoT Android applications and their non-IoT counterparts. This distinction is crucial for developers, researchers, and stakeholders who need to understand the specific characteristics of IoT apps.

2. Ensuring Code Quality and Efficiency: IoT applications often need to handle large volumes of data and communicate with various devices, requiring efficient code. Ensuring code quality and efficiency is essential to avoid performance issues and inefficiencies.

3. Addressing Security Concerns: IoT apps often deal with sensitive data and control important functions. Security vulnerabilities can have severe consequences. It's vital to analyze and identify potential security issues.

4. Improving Development Practices: Developers require insights into best practices for IoT app development, including architecture choices, library usage, and code patterns.

5. Supporting Research and Analysis: Researchers need a reliable tool to assist in characterizing IoT Android apps to contribute to the knowledge and advancement of the IoT domain.

Our project aims to address these challenges by creating a tool capable of analyzing the code, architecture, and security of IoT Android applications. By providing insights, differentiating features, and security assessments, this tool will empower developers and researchers in making

informed decisions, improving code quality, and contributing to the development of secure and efficient IoT Android applications.

1.3 Scope

IoTWhiz is designed as a specialized desktop tool for the comprehensive analysis and characterization of IoT (Internet of Things) Android applications. Within its scope, IoTWhiz incorporates several key features and focuses on specific objectives tailored to address the unique challenges posed by IoT app development. The primary scope elements include:

- IoT Application Insight
- Code Structure Analysis
- Dependency Mapping
- UI Component Examination
- Data Handling Assessment
- Network Protocol Identification
- Security Evaluation
- Permission Scrutiny

Chapter 2

Project Description of IoTWhiz

2.1 Quality Function Deployment (QFD)

2.1.1 Normal Requirements

1. Users will be able to upload the project folder as it is an essential starting point for the analysis process.

2. Users expect the tool to provide comprehensive metrics and insights to understand the Android app better.

3. Saving analysis reports is crucial for future reference and documentation. The user should be able to save the generated reports.

4. Users should be able to easily access and manage their past analysis reports.

2.1.2 Expected Requirements

1. Comparing metrics and insights between projects is a standard feature in similar tools. Combined analysis from all the reports can be seen together for better understanding

2. Accessing and reviewing past insights is important to conclude a research finding.

3. Reports can be downloaded as PDF.

2.1.3 Exciting Requirements

1. Reports should contain clear analysis using charts, graphs and statistical tables if needed.

2. An analysis dashboard can be shown to save insights found from reports.

2.2 User Story

2.2.1 Creating a New Project

To begin the analysis of an Android app, we need to initiate the analysis process. We will be prompted to select the folder containing the Android app project we want to analyze. Once the project folder is selected, the tool will start the analysis automatically.

2.2.2 Analysis Metrics and Insights

During the analysis, the tool will calculate various software metrics to provide us with a comprehensive understanding of the Android app. These metrics include:

Code Analysis:

- 1. Lines of Code: The total number of lines in the codebase.
- 2. Number of Classes/Functions/Methods: The code's structural elements.
- 3. Dynamic Code Loading: Detection of dynamic code loading mechanisms.
- 4. Reflection & Class Loading: Quantification of reflection and class loading usage.

Dependency Analysis:

- 5. Library Usage: Identification and counting of external libraries.

UI Analysis:

- 6. UI Layouts: Analysis of layout files for UI components.

Storage Analysis:

- 7. Data Storage Strategy: Understanding of data storage methods (e.g., SQLite, SharedPreferences).

Data Analysis:

- 8. Data Serialization Format: Determination of data serialization format (e.g., JSON, XML).
- 9. Protocol Usage: Identification of network protocols (e.g., HTTP, MQTT).

Permission Analysis:

- 10. Permission Analysis: Examination of permissions in AndroidManifest.xml.

After the analysis is complete, the tool will generate a comprehensive report containing these metrics and insights.

2.2.3 Saving the Analysis Report

To save the generated analysis report, after the analysis, the user needs to click on the "Save Report" option. They need to choose a location for the report and provide a name. The tool will save the report in an accessible format for future reference.

2.2.4 Accessing Tested Projects

We can access previously analyzed projects by clicking on the "Tested Projects" option. Here, we will find a list of saved analysis reports, including project names, analysis dates, and brief summaries of findings.

2.2.5 Reported Findings and Comparisons

In this section, we can compare metrics and insights between different analyzed projects. We can select multiple projects for comparison, and the tool will display differences and similarities in metrics through graphs, charts, and statistical reports if necessary.

2.2.6 Accessing Saved Insights

This section allows users to access previously generated insights and reports. They can view a list of saved insights and reports, and open them for review. They can easily save and organize insights for reference and future analysis.

2.3 Project Timeline

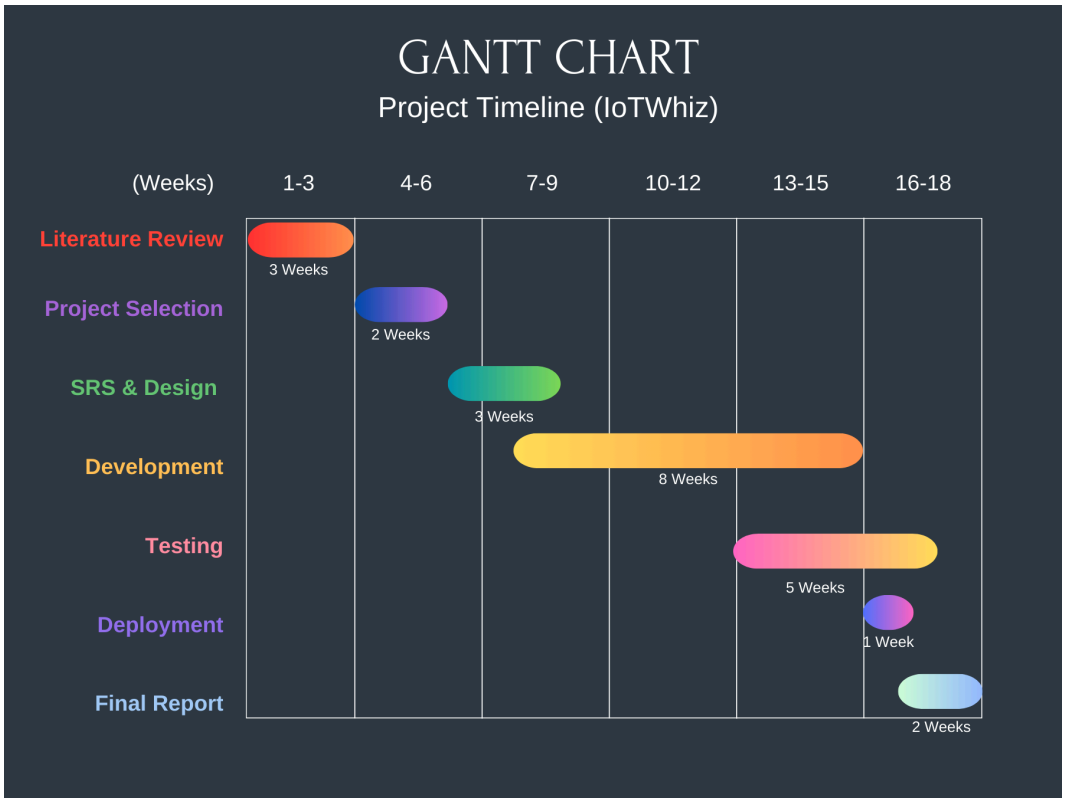


FIGURE: Timeline of IoTWhiz

Chapter 3

Scenario-based Modeling

3.1 Use-case Diagram

3.1.1 Level-0

Primary Actor: User

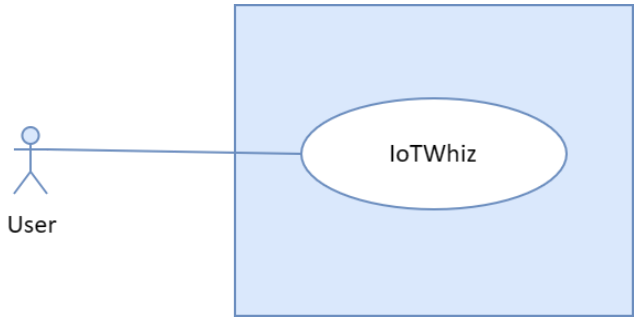


FIGURE: Level-0 of Use-case Diagram

3.1.2 Level-1

Primary Actor: User

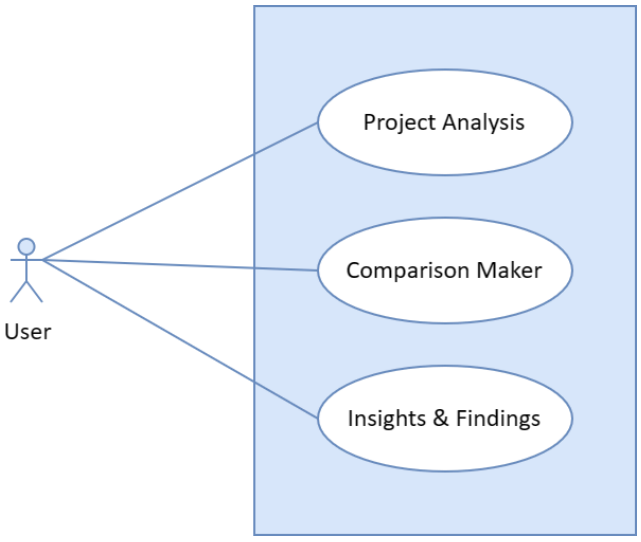


FIGURE: Level-1 of Use-case Diagram

3.1.2.1 Level-1.1

Primary Actor: User

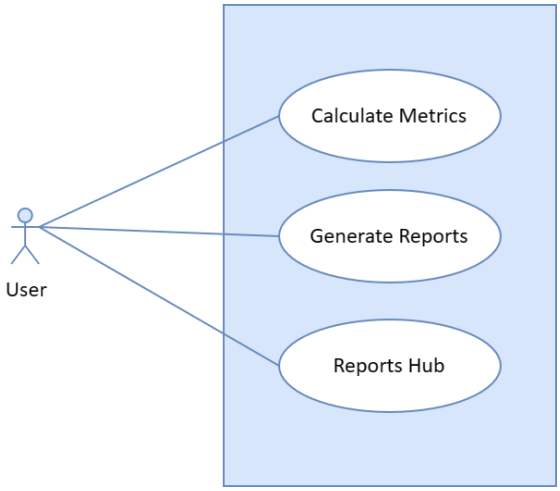


FIGURE: Level-1.1 of Use-case Diagram

3.1.2.1.1 Level-1.1.1

Primary Actor: User

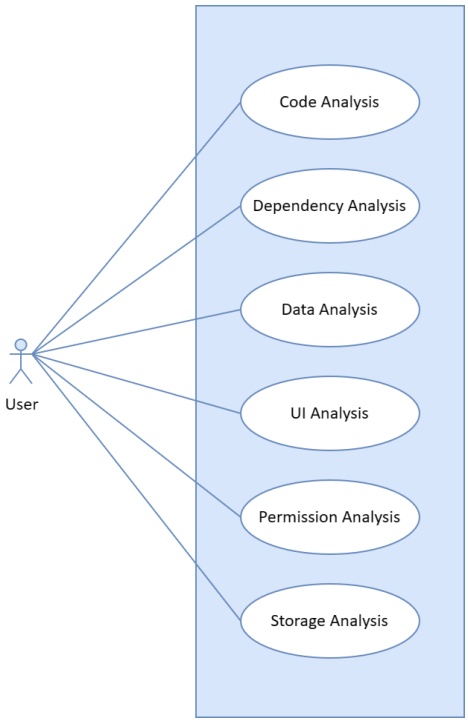


FIGURE: Level-1.1.1 of Use-case Diagram

3.2 Data Flow Diagram

3.2.1 ID-0

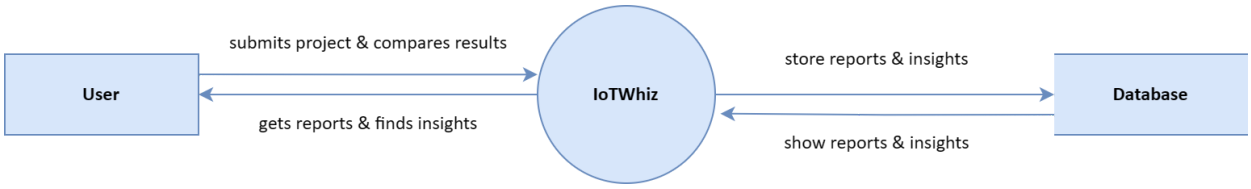


FIGURE: Data Flow Diagram - Level 0

3.2.2 ID-1

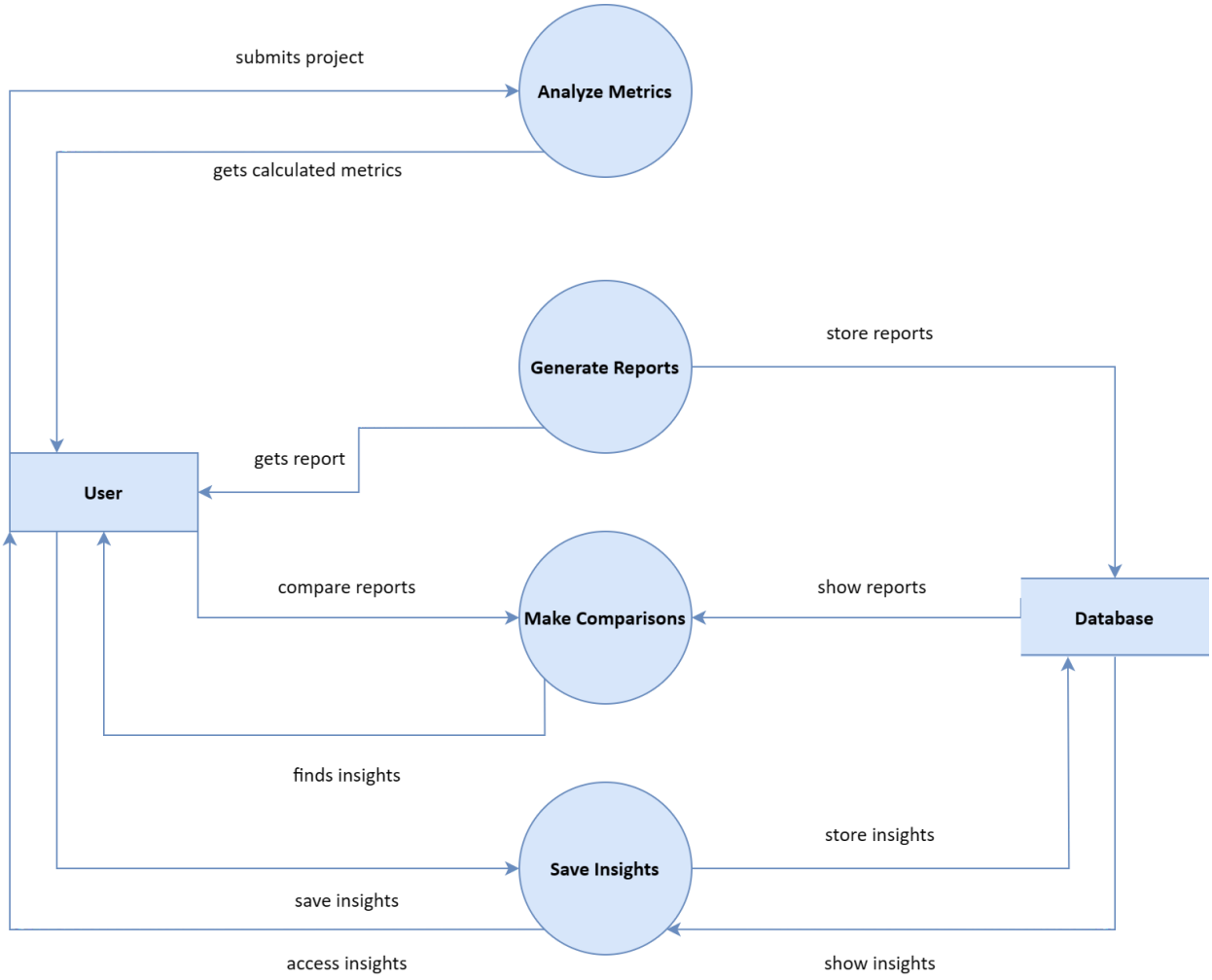


FIGURE: Data Flow Diagram - Level 1

Chapter 4

Class-based Modeling

Class-based modeling identifies classes, attributes and relationships that the system will use. It represents the object. The system manipulates the operations.

Let's take only solution space nouns to further do Class Based Modeling-

4.1 Noun List with General Classifications

1	Analysis	2
2	Android	2
3	Application	2, 7
4	Process	2, 7
5	Folder	2, 7
6	Project	2, 3, 7
7	Tool	2, 7
8	Metrics	2, 7
9	Insights	2, 3, 7
10	Software	2
11	Lines of Code	2
12	Code Analysis	2, 3, 7
13	Class	2
14	Function	2
15	Method	2
16	Codebase	2, 7
17	Element	2
18	Comments	2

19	Dependency Analysis	2, 3, 7
20	Library Usage	2
21	Permission Analysis	2, 3, 7
22	Cyclomatic Complexity	2
23	Dynamic Code	2
24	Reflection	2
25	Data Analysis	2, 3, 7
26	Data Serialization Format	2, 7
27	JSON	2, 7
28	XML	2, 7
29	Protocol Usage	2
30	HTTP	2, 7
31	MQTT	2, 7
32	Security Considerations	2, 3
33	UI Analysis	2, 3, 7
34	Storage Analysis	2, 3, 7
35	SQLite	2, 7
36	Report	2, 3, 7
37	User	4
38	Name	2
39	Description	2
40	Date	3
41	Time	3
42	Link	2
43	Comparison	2, 3, 7

44	Text	2
45	Summary	2
46	Reference	2
47	Graphs	2
48	Charts	2
49	Location	6
50	Future	3
51	Findings	2, 7

Potential Classes -

1. Project
2. Report
3. Insight
4. Comparison
5. Code Analysis
6. UI Analysis
7. Permission Analysis
8. Storage Analysis
9. Data Analysis
10. Dependency Analysis

4.2 Selection Criteria

1	Project	1, 2, 3, 4, 5
2	Report	1, 2, 3, 4, 5
3	Insight	1, 2, 3, 4, 5
4	Comparison	1, 2, 3, 4, 5
5	Metrics Analysis	1, 2, 3, 4, 5
6	UI Analysis	1, 2, 3, 4, 5
7	Tool	1, 2, 3, 4, 5
8	Metrics	1, 2, 3, 4, 5

9	Insights	1, 2, 3, 4, 5
10	Software	1, 2, 3, 4, 5

4.3 Verb List

1	create
2	initiate
3	select
4	start
5	calculate
6	provide
7	include
8	save
9	generate
10	click
11	choose
12	access
13	find
14	compare
15	display
16	allow
17	view
18	open
19	organize

4.4 CRC Cards

Class: Project	
Attributes	**Methods**
- projectName - projectDescription - createdDate - projectFolderLocation	+ getProjectName() + getProjectDescription() + getCreatedDate() + getProjectFolder() + setProjectName(name) + setProjectDescription(description) + setCreatedDate(date) + setProjectFolder(folderPath)
Responsibilities	**Collaborator**
1. Create project 2. Test project	Report

Class: Report	
Attributes	**Methods**
- reportIDanalysis - Date	+ getReportID() + getAnalysisDate() + getInsights()
Responsibilities	**Collaborator**
1. Generate reports 2. Save reports 3. See reports	Project Code Analysis Dependency Analysis UI Analysis Data Analysis Permission Analysis Storage Analysis

Class: Insight	
Attributes	**Methods**

- insightID - Text - dateCreated - reportRef	+ getInsightID() + getText() + getDateCreated() + getReportAssociated()
Responsibilities	**Collaborator**
1. Derive Insights 2. Save Insights	Report

Class: Comparison	
Attributes	**Methods**
- Reports - comparisonResults	+ compareReports() + getComparisonResults() + createVisualizations()
Responsibilities	**Collaborator**
1. Compare multiple reports 2. Create visualizations	Report Insight

Class: Code Analysis	
Attributes	**Methods**
- linesOfCode - numClasses - numFunctions - codeCommentsPercentage - cyclomaticComplexity - dynamicCodeLoadingDetected - reflectionAndClassLoadingCount	+ calculateLinesOfCode() + calculateNumClasses() + calculateNumFunctions() + calculateCodeCommentsPercentage() + calculateCyclomaticComplexity() + detectDynamicCodeLoading() + countReflectionAndClassLoading()
Responsibilities	**Collaborator**
1. Calculate LOC 2. Calculate number of classes/methods/functions 3. Calculate code comments percentage 4. Calculate cyclomatic complexity	Project Report

Class: Dependency Analysis	
Attributes	**Methods**
- libraryUsageCount	+ analyzeLibraryUsage()
Responsibilities	**Collaborator**
1. Analyze library usage	Project Report

Class: Data Analysis	
Attributes	**Methods**
- dataSerializationFormat - protocolUsage - securityConsiderationsFound	+ determineDataSerializationFormat() + identifyProtocolUsage() + performSecurityAnalysis()
Responsibilities	**Collaborator**
1. Determine serialization format 2. Identify protocol usage 3. Perform security analysis	Project Report

Class: UI Analysis	
Attributes	**Methods**
- uiLayoutAnalyzeResults	+ analyzeUILayouts()
Responsibilities	**Collaborator**
1. Analyze UI layout structure	Project Report

Class: Storage Analysis	
Attributes	**Methods**
- dataStorageStrategy	+ determineDataStorageStrategy()

Responsibilities	Collaborator
1. Determine data storage strategy	Project Report

Class: Permission Analysis	
Attributes	**Methods**
- permissionAnalysisResults	+ analyzePermissions()
Responsibilities	**Collaborator**
1. Analyze permission patterns	Project Report

4.5 CRC Diagram

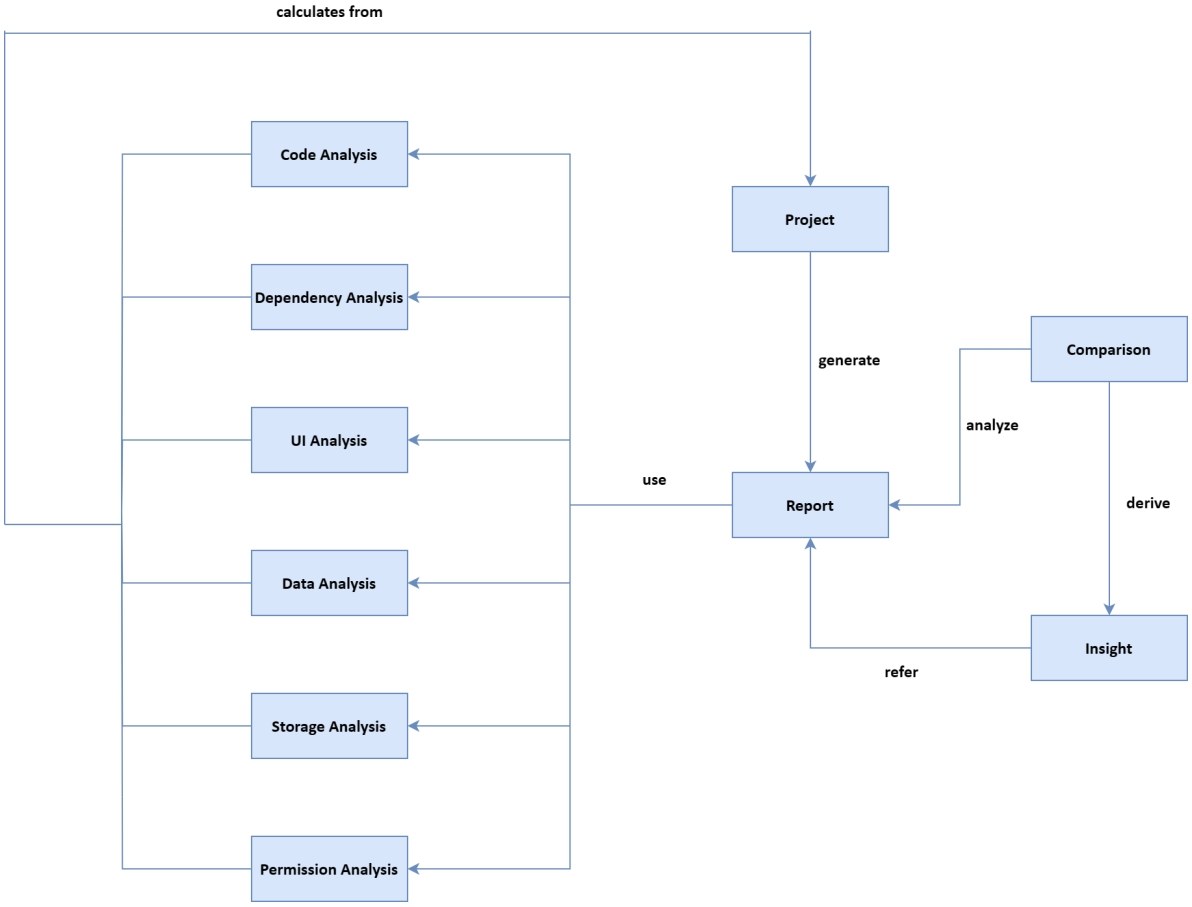


FIGURE: CRC Diagram

Chapter 5

Architectural Design

5.1 Architectural Context Diagram

The architectural context diagram for the tool is given below-

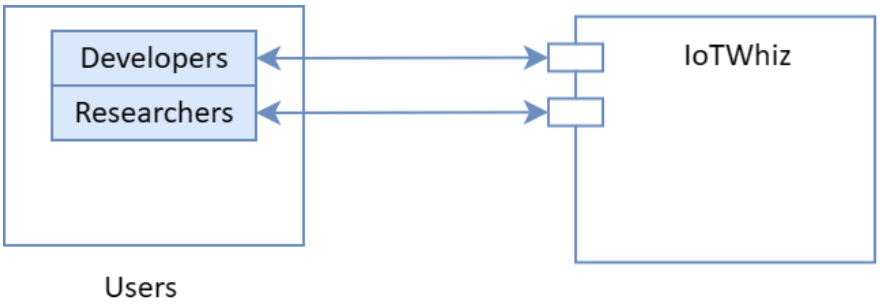


FIGURE: ACD of IoTWhiz

Two categories of users are supposed to be the target for the tool. They are software developers and researchers. Both types of users are the producers and also consumers of the information through the tool. There are no subordinate, superordinate or peer-level systems.

5.2 Archetypes

5.2.1 MVC Pattern

The MVC architecture pattern is used as a primary archetype for the application. The models of the archetypes are:

- Model
- View
- Controller

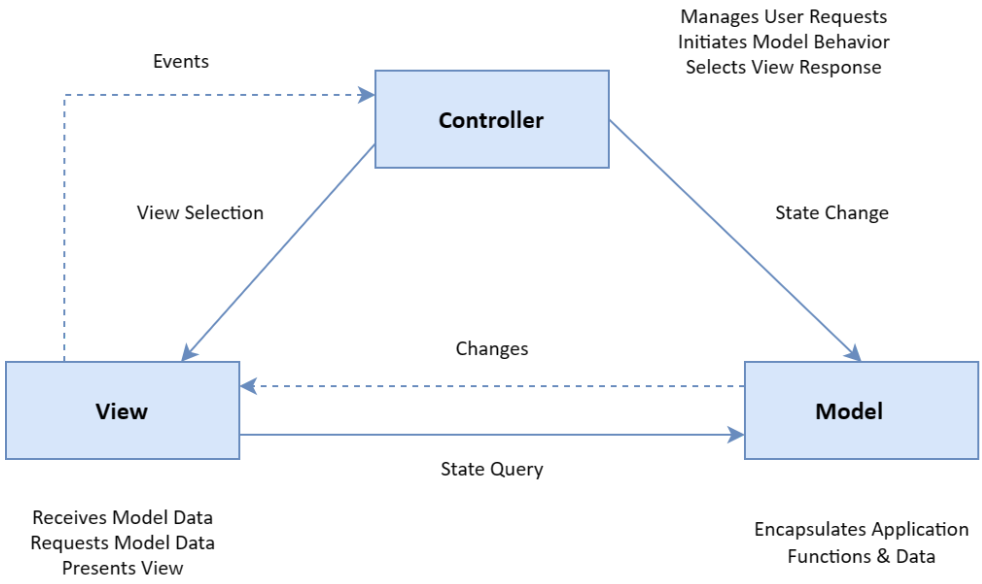


FIGURE: MVC in IoTWhiz

5.2.2 Pipe & Filter Pattern

This pattern is applied when input data is to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern has a set of components, called filters, connected by pipes that transmit data from one component to the next.

We've used this pattern on Report & Comparison components.

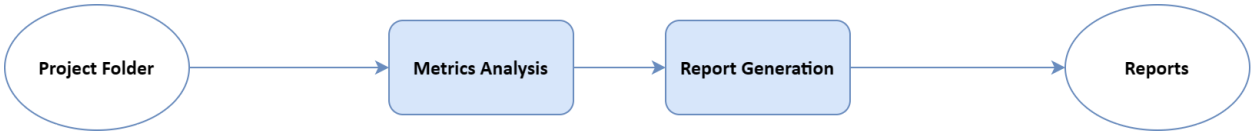


FIGURE: Pipe-filter in Report Component

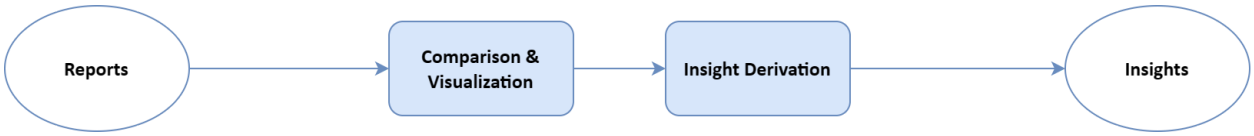


FIGURE: Pipe-filter in Comparison Component

5.3 Refining the Architecture into Components

Defining the set of top-level components that address the following functionality (Architectural structure with top level components):

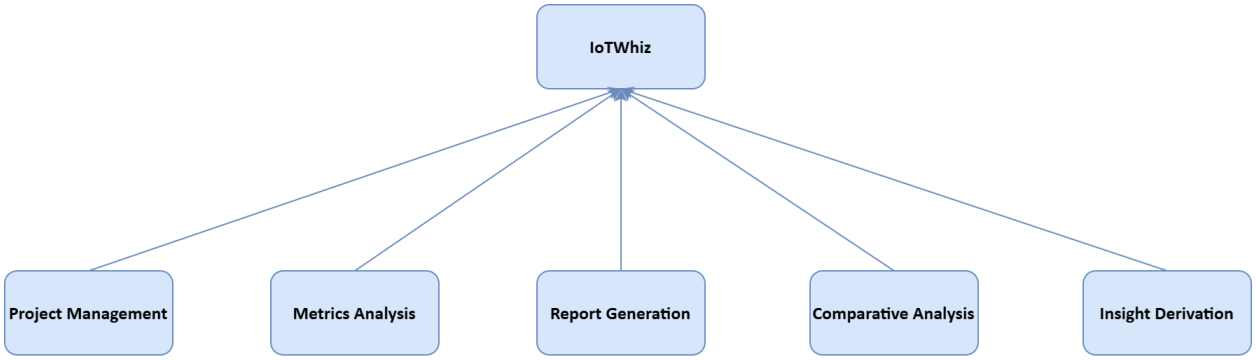


FIGURE: Top-level view of Components

5.4 Describing Instantiations of the Components

To accomplish this, an actual instantiation of the architecture is developed. By this we mean that the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.

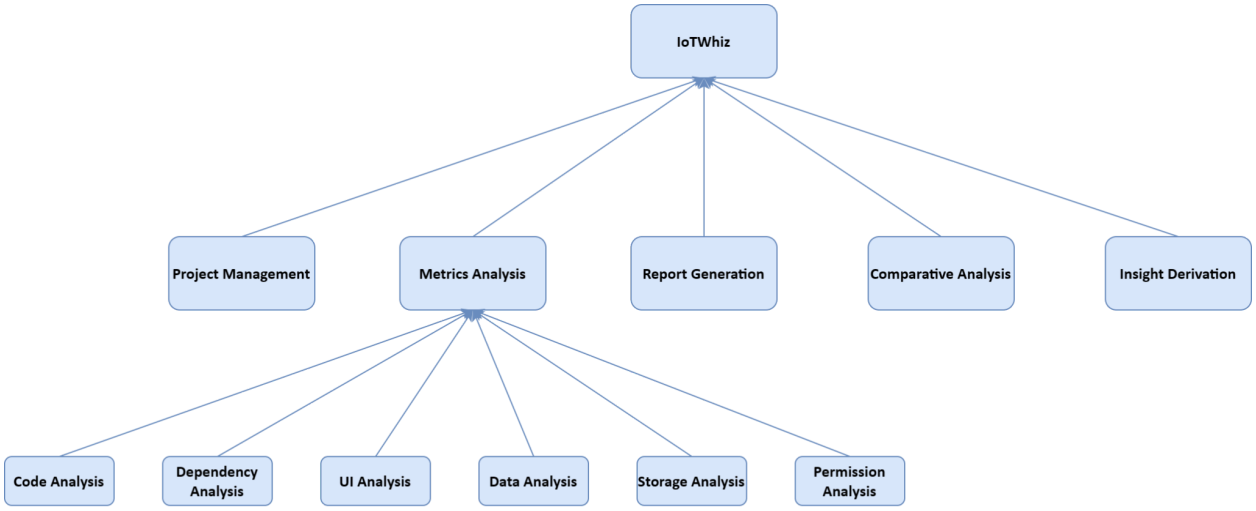


FIGURE: Top-level view of Components (with instantiations)

Chapter 6

Methodology

This chapter outlines the development of our project and the methods employed to address different situations.

6.1 Overview of the Workflow

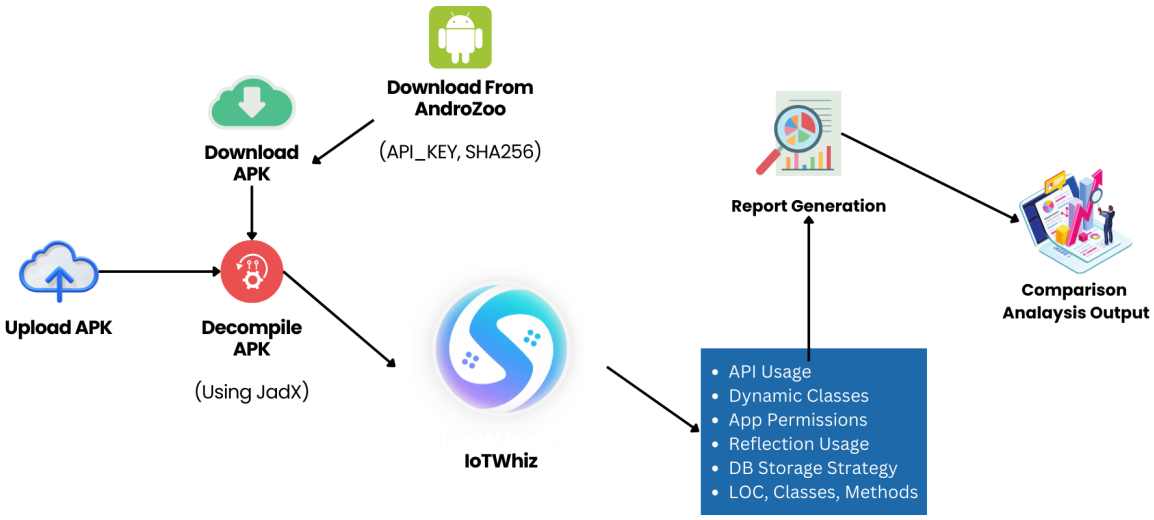


FIGURE: Workflow of IoTWhiz

6.2 Input Methods

The input methods of the IoTWhiz tool allow users to access Android Package (APK) files for analysis through two primary avenues:

6.2.1 APK Input

Users have the option to directly upload APK files into the IoTWhiz tool. This method enables them to select and provide specific APK files stored on their local devices or accessible directories. Once uploaded, the tool keeps the file so that later it can be used for the analysis process.

6.2.2 Androzoo Repository Integration

IoTWhiz incorporates integration with Androzoo, a comprehensive collection of Android applications, by leveraging its repository. This integration is facilitated by providing specific credentials:

1. **API Key:** A unique identifier or authentication key provided by Androzoo to access its repository.
2. **SHA256 Code:** A cryptographic hash function used as an identifier for a specific APK file within Androzoo's vast collection.

Users input the API key and the SHA256 code associated with the desired APK file(s) into IoTWhiz. Subsequently, the tool uses this information to authenticate and access the Androzoo repository, allowing for direct retrieval and download of the respective APK file(s) identified by the provided SHA256 code.

This integration offers users a convenient way to access a wide array of APK files stored within Androzoo's repository without manually sourcing or storing each APK individually.

6.3 Dataset Handling

The dataset utilized by IoTWhiz consists of a significant collection of approximately 37,000 Android applications. These apps have been categorized into two primary groups: IoT and non-IoT applications.

When users interact with IoTWhiz, the tool leverages this categorized dataset as a reference point. It helps in providing comparative insights and identifying trends or patterns specific to IoT or non-IoT applications during the analysis process.

Dataset Link:

shared_sha256_androzoo.csv

6.4 APK Handling

The APK handling functionality within IoTWhiz encompasses various methods to access and analyze APKs, offering users multiple options for retrieval and decompilation:

6.4.1 Download APK via SHA256

Users can request the download of specific APKs stored within a repository by providing their unique SHA256 hash codes. This method serves as an efficient means to obtain APK files directly based on their cryptographic hash identifiers.

6.4.2 Get Source Code via SHA256

This feature allows users not only to download APKs based on their SHA256 codes but also initiates the decompilation process to retrieve the source code of these APKs. By simply inputting the SHA256 code, the tool automatically performs both downloading and decompilation, providing access to the application's source code for analysis.

6.4.3 Direct Decompilation

Alternatively, users have the option to directly upload APK files into the IoTWhiz tool. This feature enables users to manually select and upload specific APKs from their local devices or accessible directories.

6.5 Decompilation Process

IoTWhiz leverages JADX, a popular tool for Android application decompilation, to facilitate the retrieval of source code from the provided APKs. JADX serves as the engine driving the decompilation process, allowing the tool to extract the original source code from the APK files, making it accessible for further analysis and inspection.

These functionalities collectively empower users to access, retrieve, and decompile APK files through various methods within IoTWhiz. The tool provides a process for obtaining the source code from APKs, allowing for in-depth analysis and examination of the application's inner workings and functionalities.

6.6 Source Code Analysis

The methodology employed by IoTWhiz involves a comprehensive analysis of Android application files to derive insights into various critical aspects of the app's structure, behavior, and functionality.

6.6.1 API Usage Examination

The API usage methodology in the provided code involves a systematic search through the project files to detect instances where specific APIs are utilized. The code traverses the project

directory to examine Java files. It utilizes regex patterns to identify API-related code snippets and patterns like HttpURLConnection, OkHttp, etc., are used to spot specific API usages. When a pattern matches, it logs the API call with file path and line number. Unique API calls are collected and counted for a comprehensive overview.

6.6.2 Dynamic Class Detection

The tool identifies the usage of dynamic classes within the app. Dynamic classes are loaded during runtime rather than being statically compiled into the app. It traverses the project directory structure to inspect *.java* files. It uses some specific type of regex or other pattern (dynamic_loading_pattern) to identify variations of ClassLoader or DexClassLoader instantiation indicating dynamic code loading. When a match is found in a line, logs the occurrence with file name, line number, and the triggering line of code. It gathers these instances into a set for uniqueness and counts the total occurrences.

6.6.3 Permissions Analysis

The tool scrutinizes the permissions granted to the Android application. It navigates through the provided directory structure. It targets files named *AndroidManifest.xml* known for containing permission declarations in Android projects. It employs a specific regex pattern to spot lines within the manifest file that declare Android permissions using <uses-permission> tags. It matches patterns representing permissions like android.permission.<name>. It records instances of permission-related lines found, noting file name, line number, and line content. It collects these occurrences into a set to ensure uniqueness and tallies the total permissions detected.

6.6.4 Reflection Usage Analysis

Reflection is a technique used by apps to access normally inaccessible methods and fields. IoTWhiz scrutinizes different types of reflection usage, including - class loading, method retrieval, instance creation, invocation, field retrieval, access control, and annotation retrievals. The function begins by defining the project folder's path. It specifies a set of reflection-related patterns covering various types. It traverses the project directory to explore files, focusing on *.java* files. It applies regex patterns from the defined set to each file's content, searching for occurrences of reflection-related code.

6.6.5 Database Storage Strategy Assessment

IoTWhiz delves into how the app stores its data using various storage strategies such as cursor, content resolver, media store queries, SQLiteOpenHelper, RoomDatabase patterns, RealmDatabase, ObjectBox Database, Firebase Database, and SQLiteDatabase. It traverses the

project folder to explore *.java* files, seeking occurrences of each database-related pattern within the file content. When a pattern matches, records occurrences including file paths, line numbers, and corresponding line content into a structured format. It describes identified strategies using predefined descriptions based on pattern matches.

6.6.6 Code Metrics Analysis

The tool provides an overview of the codebase by highlighting metrics such as total lines of code, classes, and methods. These metrics offer a quantitative assessment of the application's complexity and scale. It reads each line from the files, excluding empty lines, whitespaces, and comments, to count non-comment and non-whitespace lines.

6.6.7 UI Layout and Widget Analysis

IoTWhiz also scrutinizes the app's user interface by analyzing the occurrence and placement of UI layouts (e.g., linear layout, relative layout, nested layout) and widgets (e.g., text view, button). This analysis offers insights into the app's design structure and layout preferences. It utilizes os.walk to traverse through the specified folder (folder_path). It identifies and collects files with .xml extension, typically representing layout files. It categorizes components into three groups:

- Widgets and Views,
- Layout Types,
- and Nested Layouts.

It iterates through each layout file in the provided list. It checks each line of the XML content for specific elements (e.g., TextView, Button) indicating Widgets and Views. It identifies various layout types (LinearLayout, RelativeLayout, ConstraintLayout) by parsing XML tags. It detects nested layouts by recognizing closing tags for layout elements. It records identified components along with their file path, line number, and specific component type for Widgets and Views. It gathers layout types and nested layout occurrences with their respective file paths.

6.6.8 Analysis Page Presentation

The collected data across these analyses is compiled into an analysis page within IoTWhiz. This page serves as a consolidated dashboard, presenting all the derived insights and data points, aiding in easy comprehension and evaluation of the application's characteristics.

6.7 Output Report Generation

The output report generation process aims to compare and analyze key aspects between IoT and non-IoT applications. By conducting comprehensive code analysis, this method aims to provide statistical insights and visual representations highlighting differences in code characteristics and usage patterns.

6.7.1 API Usage Report

Objective: The API Usage Report aims to compare and contrast the utilization of Application Programming Interfaces (APIs).

Method:
Statistical Comparisons:

1) **Descriptive Statistics:** Calculation of total counts of observations, mean, maximum, minimum, standard deviation, and percentiles for API usages in IoT and non-IoT apps.
2) **Hypothesis Testing:** T-tests for assessing significant differences in mean API usages between the two categories.

Visualization Techniques:

1) **Histograms:** Representing the distribution of API usages in IoT and non-IoT apps to visualize frequency and patterns across different usage levels.

6.7.2 Dynamic Class Usage Report

Objective: This report delves into the comparison of dynamic class usage between IoT and non-IoT applications.

Method:
Statistical Comparisons:

1) **Descriptive Statistics:** Computation of descriptive metrics (mean, maximum, minimum, standard deviation, etc.) for dynamic class usage in IoT versus non-IoT apps.
2) **Hypothesis Testing:** Similar T-test analysis to assess significant differences in dynamic class usage between IoT and non-IoT app categories.

Visualization Techniques:

1) **Box Plots:** Visualization of the distribution of dynamic class usage in both app categories to depict frequency variations.

6.7.3 App Permissions Comparison Report

Objective: Comparing permission co-occurrences and evaluating required permissions between IoT and non-IoT apps.

Method:
Statistical Comparisons:

1) **Permission Co-Occurrences:** Identifying the top 10 permission co-occurrences between IoT and non-IoT apps, showcasing which combinations are more prevalent in each category.

2) **T-Statistical Analysis:** Assessing if there are significant differences in the permissions required between IoT and non-IoT apps.

Visualization Techniques:

1) **Distribution Path Charts:** Displaying the frequency distribution of permissions for IoT versus non-IoT apps, categorized by permission types.

6.7.4 App Permissions Comparison Report

Objective: Offers insights into the comparison of code length, structure, and complexity.

Method:

Statistical Comparisons:

1) **Descriptive Statistics:** Calculating descriptive metrics for code length, classes, and methods between IoT and non-IoT apps.
2) **Correlation Matrix:** Establishing correlations between lines of code, classes, and methods for both app categories.

Visualization Techniques:

1) **Box Plots:** Visualizing variations in code metrics like lines of code, classes, and methods between IoT and non-IoT apps using box plots.
2) **Scatter Plots:** Visualizing variations in code metrics like lines of code, classes, and methods between IoT and non-IoT apps using scatter plots.

6.7.5 DB Storage Report

Objective: Evaluating diverse database storage methodologies adopted by IoT and non-IoT apps.

Method:

Statistical Comparisons:

1) **Database Strategy Percentages:** Analyzing the prevalence of different database storage strategies between IoT and non-IoT apps.
2) **T-tests and Chi-square Tests:** Statistical tests conducted to compare and evaluate the significance of database strategies.

6.7.6 Reflection Report

Objective: Presents a comparison of reflection usage across IoT and non-IoT apps.

Method:

Statistical Comparisons:

1) **Reflection Types Analysis:** Conducting statistical analyses to compare the usage of different types of reflections between IoT and non-IoT apps.
2) **T-test Results:** Providing insights into significant differences in reflection types utilized.

Visualization Techniques:

1) **Comparative Graphs:** Graphical representation illustrating variations in different reflection types between IoT and non-IoT apps.

These methodologies aim to highlight and compare specific aspects of IoT and non-IoT apps, providing both numerical insights and graphical representations for a comprehensive understanding of their differences in code behavior and characteristics.

Chapter 7

User Interface & Task Analysis

Effective user interface design, guided by established principles, not only shapes how users interact with digital systems but also lays the foundation for an intuitive and user-friendly environment. A thoughtfully designed interface enhances user satisfaction by meeting usability and accessibility standards. User interface design serves as a crucial communication bridge between humans and computers.

7.1 User Analysis

Researchers: The tool caters to researchers involved in comprehensive analysis and comparison of IoT and non-IoT applications. Researchers can leverage the detailed reports and statistical analyses generated by IoTWhiz to draw conclusions and contribute to the body of knowledge in the field of IoT app characterization.

Developers: Software developers are a primary audience, utilizing IoTWhiz for code analysis. They can assess the structure, permissions, and API usage of their apps, gaining insights into optimizing their applications for security, efficiency, and functionality.

7.2 Task Analysis

7.2.1 Upload APK

- Upload an APK file for the decompilation process

7.2.2 Download APK from AndroZoo

- Enter API key & SHA256 code
- Tick on checkbox if an IoT app, do nothing if a non-IoT app and start downloading

7.2.3 Get Source Code from AndroZoo

- Enter API key & SHA256 code
- Tick on checkbox if an IoT app, do nothing if a non-IoT app and start downloading & decompiling

7.2.4 Upload Project Folder for Analysis

- Tick on checkbox if an IoT app, do nothing if a non-IoT app
- Upload a decompiled/existing project source code folder for the analysis

7.2.5 Generate Report PDF

- Generate report of IoT and non-IoT app analysis

7.2.6 Download Generated Report

- Download report of IoT and non-IoT app analysis to see comparison insights

7.2.7 View Top 10 Permission Co-occurrences for IoT & non-IoT

- Show outputs of permission occurrences from app permissions analysis based on app type

7.3 User Interface and User Manual

The user interface is structured to present comprehensive analysis in a visually accessible manner. The layout is designed for ease of navigation and interpretation of various statistical and comparative analyses. The user manual provides a step-by-step guide on using IoTWhiz, explaining input methods, functionalities, and interpretation of analysis outputs.

7.3.1 Home Page

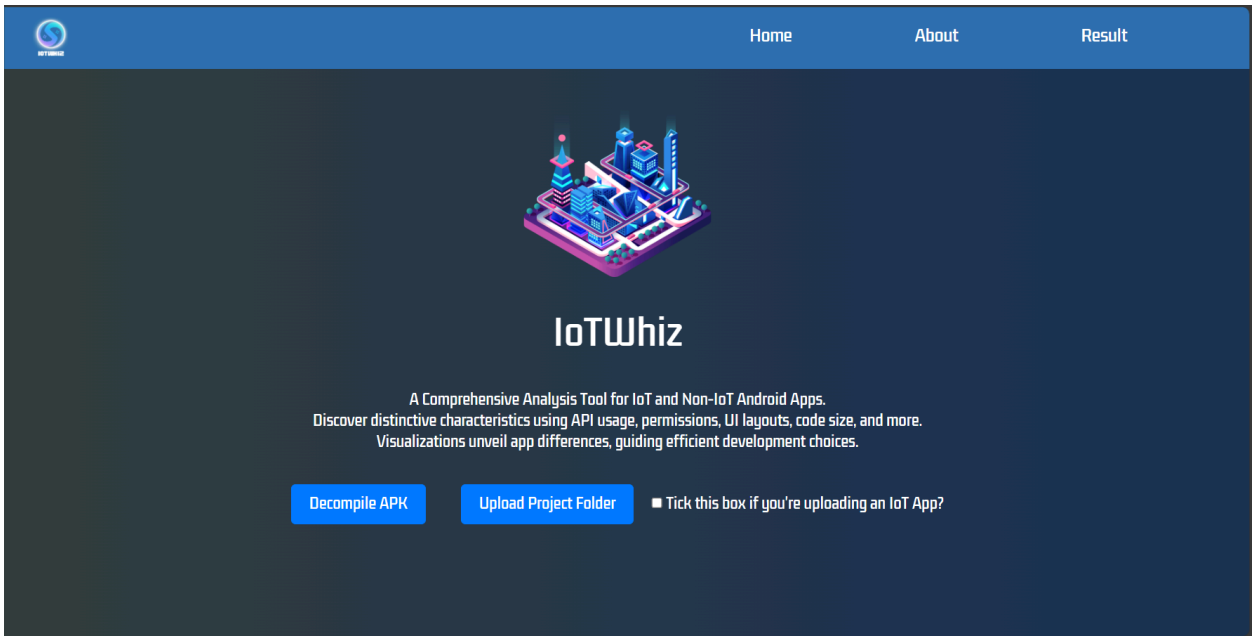


FIGURE: Homepage of IoTWhiz

User will view the home page after entering the tool. This screen accommodates *task 1*.

Task-1: Upload APK - Users can upload an APK file for the decompilation process to retrieve the application's source code.

7.3.2 Download from AndroZoo

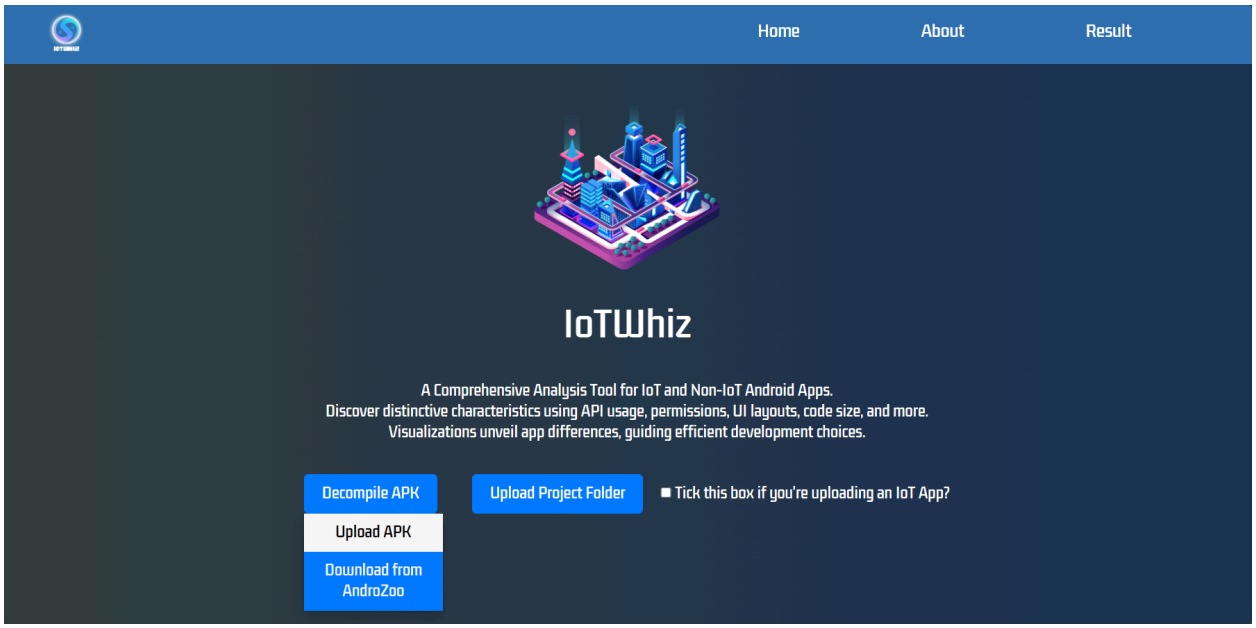


FIGURE: Dropdown of Decompile APK

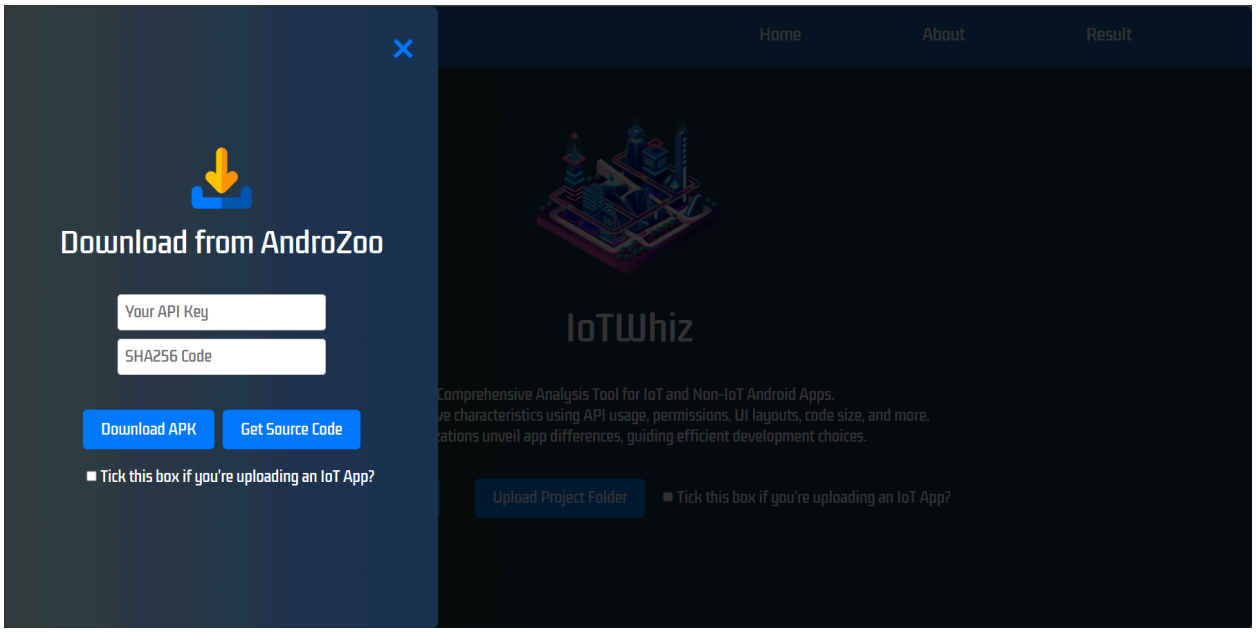


FIGURE: Modal View of Download from AndroZoo

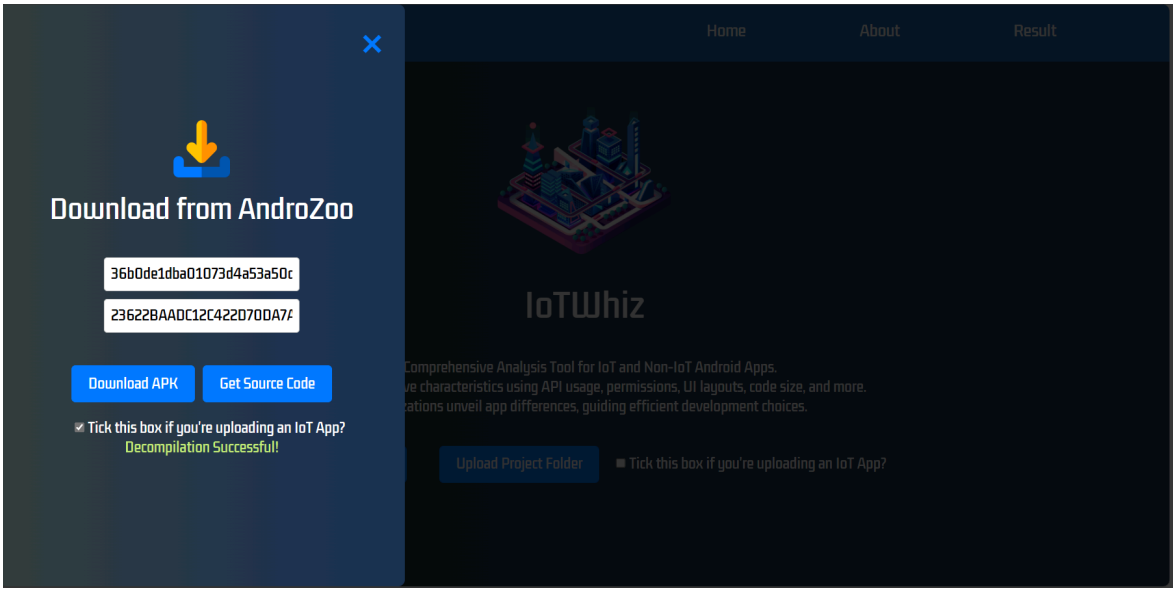


FIGURE: Successful Decompilation using JadX

Task-2: Download APK from AndroZoo - Entering the API key and SHA256 code enables users to download APKs from AndroZoo's repository. They can specify the app type (IoT/non-IoT) by ticking the respective checkbox before initiating the download.

Task-3: Get Source Code from AndroZoo - Similar to downloading APKs, users input the API key and SHA256 code. They can choose to decompile the downloaded APK file by selecting the IoT/non-IoT checkbox.

7.3.3 Project Upload for Analysis

Task-4: Upload Project Folder for Analysis - Users upload the decompiled or existing project source code folder for further analysis. They categorize the uploaded project as IoT or non-IoT by ticking the checkbox accordingly.

Then, an analysis dashboard will be generated containing the analysis output for the input project folder. The dashboard will contain data of - type of application (IoT/non-IoT), detected APIs, detected dynamic class loading, detected permissions, layout & widgets, LOC, number of classes & methods, reflection usage, database storage strategy etc.

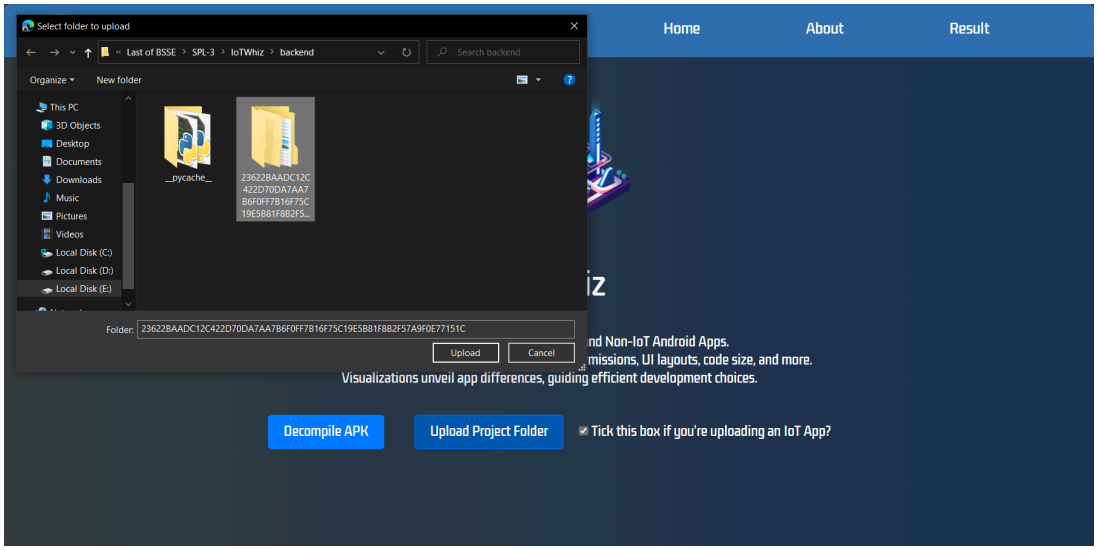


FIGURE: Upload Project Folder for Analysis

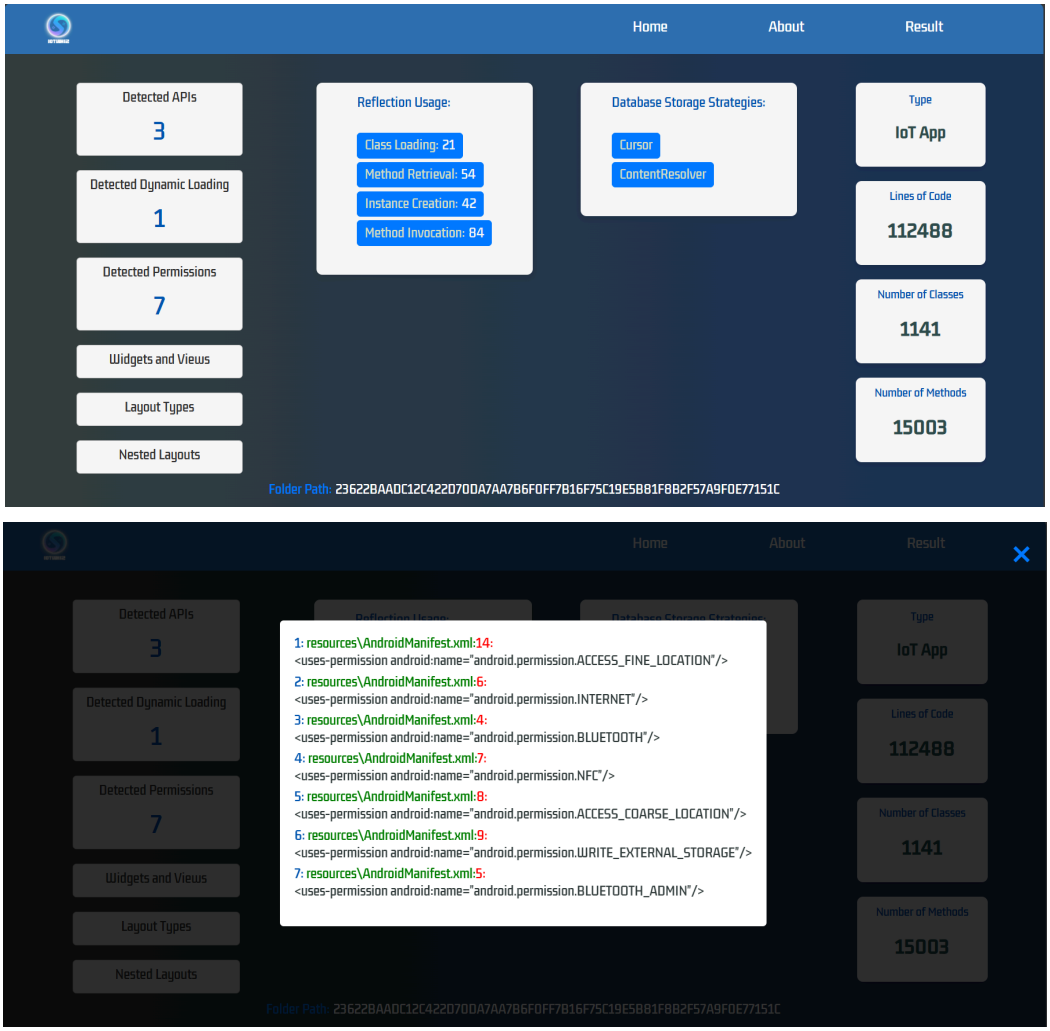


FIGURE: Analysis Dashboard of the given app

7.3.4 Generate & Download Report as PDF

Task-5: Generate Report PDF - Users can generate a comprehensive report summarizing the analysis conducted on both IoT and non-IoT apps.

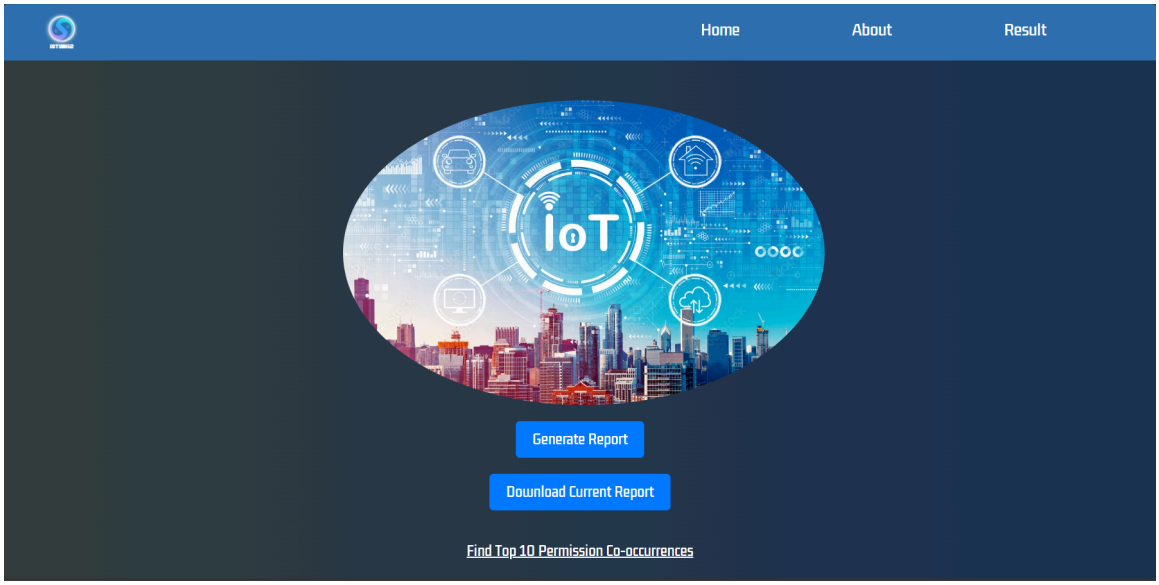


FIGURE: Result Page of IoTWhiz

Task-6: Download Generated Report - It allows users to download the generated report for detailed insights and comparisons between IoT and non-IoT app analyses.

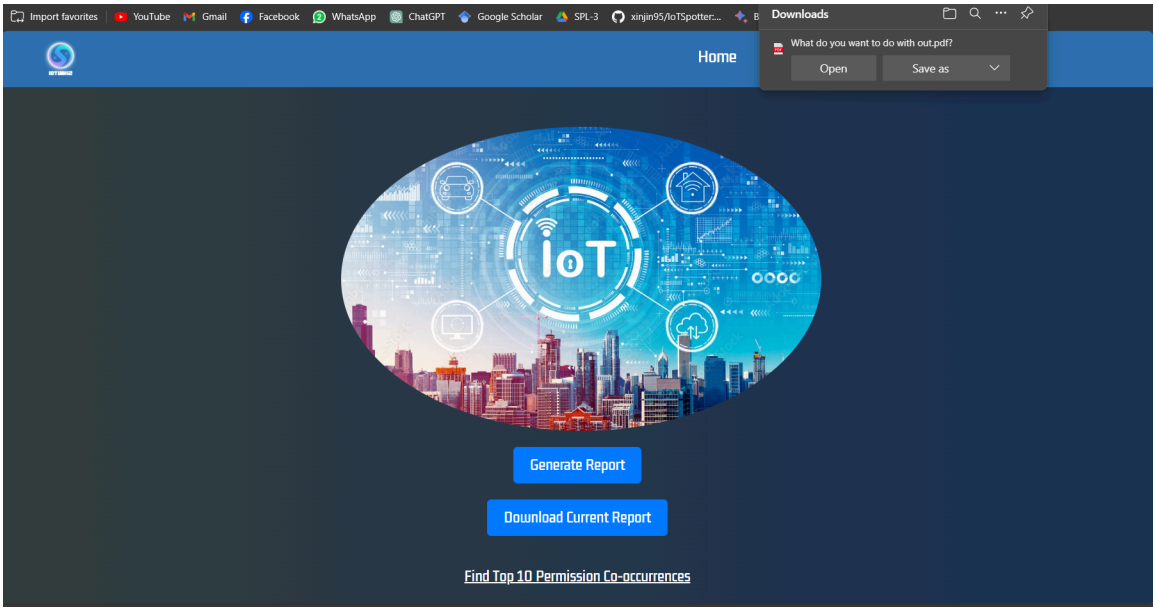
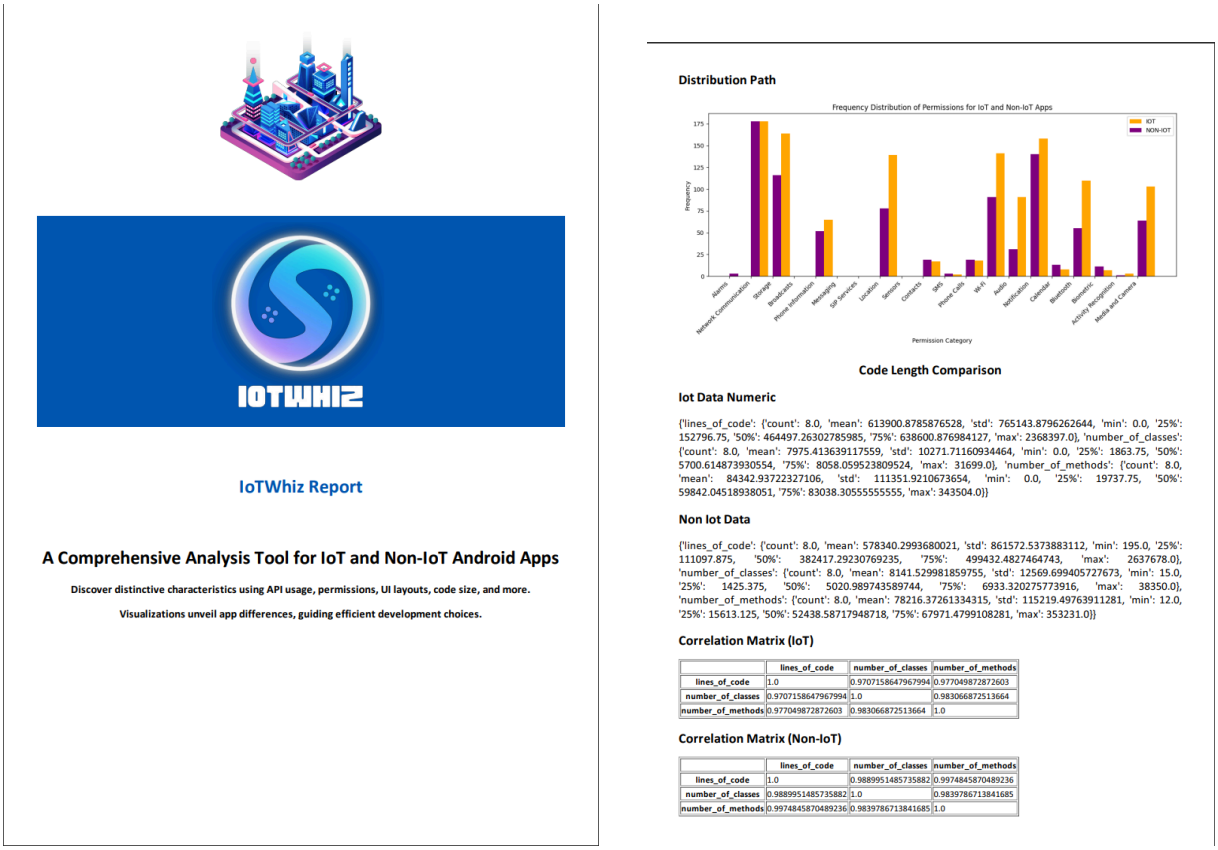


FIGURE: Downloading Report of IoTWhiz



IOTWHIZ

IoTWhiz Report

A Comprehensive Analysis Tool for IoT and Non-IoT Android Apps

Discover distinctive characteristics using API usage, permissions, UI layouts, code size, and more.

Visualizations unveil app differences, guiding efficient development choices.

Distribution Path

Code Length Comparison

Iot Data Numeric

{'lines_of_code': {'count': 8.0, 'mean': 613900.8785876528, 'std': 765143.8796262644, 'min': 0.0, '25%': 152796.75, '50%': 464497.26302785985, '75%': 638600.876984127, 'max': 2368397.0}, 'number_of_classes': {'count': 8.0, 'mean': 7975.413639117559, 'std': 10271.71160934464, 'min': 0.0, '25%': 1863.75, '50%': 5700.614873930554, '75%': 8058.059523809524, 'max': 31699.0}, 'number_of_methods': {'count': 8.0, 'mean': 84342.93722327106, 'std': 111351.9210673654, 'min': 0.0, '25%': 19737.75, '50%': 59842.04518938051, '75%': 83038.30555555555, 'max': 343504.0}}

Non Iot Data

{'lines_of_code': {'count': 8.0, 'mean': 578340.2993680021, 'std': 861572.5373883112, 'min': 195.0, '25%': 111097.875, '50%': 382417.29230769235, '75%': 499432.4827464743, 'max': 2637678.0}, 'number_of_classes': {'count': 8.0, 'mean': 8141.529981859755, 'std': 12569.699405727673, 'min': 15.0, '25%': 1425.375, '50%': 5020.989743589744, '75%': 6933.320275773916, 'max': 38350.0}, 'number_of_methods': {'count': 8.0, 'mean': 78216.37261334315, 'std': 115219.49763911281, 'min': 12.0, '25%': 15613.125, '50%': 52438.58717948718, '75%': 67971.4799108281, 'max': 353231.0}}

Correlation Matrix (IoT)

	lines_of_code	number_of_classes	number_of_methods
lines_of_code	1.0	0.9707158647967994	0.977049872872603
number_of_classes	0.9707158647967994	1.0	0.983066872513664
number_of_methods	0.977049872872603	0.983066872513664	1.0

Correlation Matrix (Non-IoT)

	lines_of_code	number_of_classes	number_of_methods
lines_of_code	1.0	0.9889951485735882	0.9974845870489236
number_of_classes	0.9889951485735882	1.0	0.9839786713841685
number_of_methods	0.9974845870489236	0.9839786713841685	1.0

FIGURE: Downloaded Report as PDF

Task-7: View Top 10 Permission Co-Occurrences - This task presents the top 10 permission co-occurrences for both IoT and non-IoT apps, derived from the app permissions analysis. It shows the occurrence frequency based on the app type.

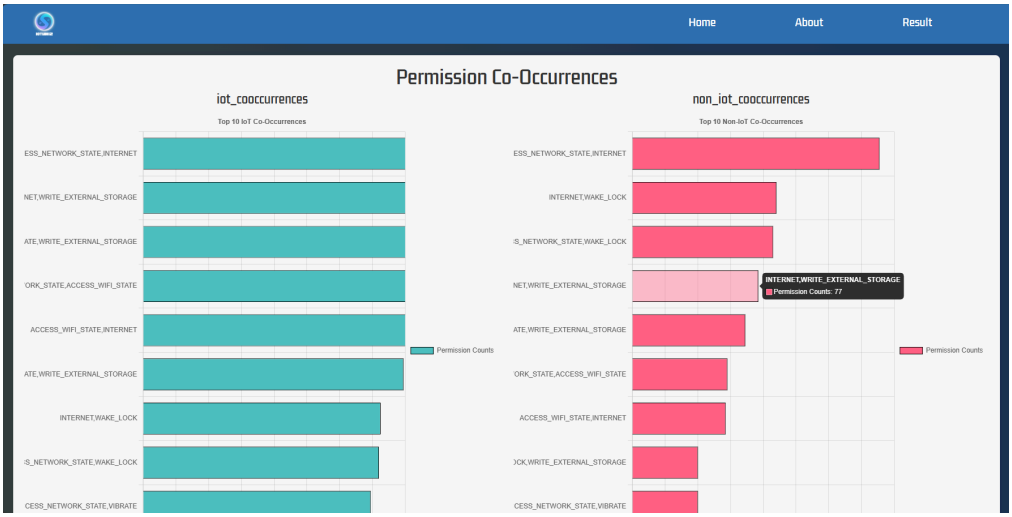


FIGURE: Top 10 Permission Co-occurrences

Chapter 8

Results & Insights

This detailed analysis provides comprehensive insights into the similarities and differences between IoT and non-IoT applications across various aspects like API usage, permissions, code characteristics, database storage, and reflection usage. These insights can guide development choices and help in understanding the distinctive features of both IoT and non-IoT apps.

8.1 Dataset

8.1.1 Initial Dataset Selection (51 IoT & 51 Non-IoT Apps)

The selection of 51 IoT and 51 non-IoT apps laid the foundation for the comparative analysis. This initial dataset likely aimed to provide an exploratory understanding of key metrics and characteristics distinguishing these two categories. It allowed for a focused examination, possibly revealing preliminary trends and patterns.

8.1.2 Expanded Dataset (195 IoT & 195 Non-IoT Apps at 15 MB Size Threshold)

Expanding the dataset to 195 apps in each category, filtered by a size threshold of 15 MB, represents a significant enhancement to the initial sample size. The imposition of the size threshold ensured a more standardized and refined selection of apps, potentially filtering out outliers and maintaining a more homogeneous dataset for comparative analysis.

8.1.3 Impact of Dataset Expansion for Enhanced Comparative Analysis

The increased dataset size and refined criteria would have likely strengthened the ability to draw meaningful conclusions. It offered a more comprehensive view of the landscape, enabling a more confident identification of nuanced differences and similarities between IoT and non-IoT applications across various attributes, such as API usage, code length, permissions, and database storage strategies.

8.2 API Usage Comparison

In the initial dataset, where 51 IoT and 51 non-IoT apps were analyzed, a stark disparity in API usage was evident. The mean API usage for IoT apps stood notably higher at approximately 23.88 compared to non-IoT apps, which averaged around 8.61. The standard deviation for both categories indicated a considerable variability in API usages, being 22.52 for IoT and 9.78 for non-IoT apps.

The expanded dataset, encompassing 195 apps in each category and employing a size threshold of 15 MB, reflected a different scenario. The mean API usages for IoT and non-IoT apps were approximately 17.61 and 19.46, respectively. The standard deviation for API usages remained high for both, around 17.52 for IoT apps and notably higher at 24.50 for non-IoT apps.

	Initial Dataset		**Expanded Dataset**	
Type	**IoT Apps**	**non-IoT Apps**	**IoT Apps**	**non-IoT Apps**
Count	51	51	195	195
Mean	23.88	8.61	17.61	19.46
Std	22.52	9.78	17.52	24.50
Median	22	6	14	10
Min	0	0	0	0
Max	121	38	121	167

TABLE: Descriptive Statistics of API Usage

In the initial dataset, the difference in mean API usage between IoT and non-IoT apps was significant, but with the expanded dataset, this distinction diminished. The means for both categories converged closer together, indicating a smaller discrepancy in API usage between IoT and non-IoT apps in the larger dataset.

The variability in API usage remained relatively high for both categories in both datasets, suggesting a wide range of API utilization across the apps studied. Despite this, the analysis of the expanded dataset suggests a less pronounced difference between IoT and non-IoT apps in terms of API usage, contrary to the substantial gap observed in the initial dataset.

Verdict	
Initial Dataset	There is a significant difference between IoT and Non-IoT API usages.
Expanded Dataset	There is no significant difference between IoT and Non-IoT API usages.

TABLE: Verdict of API Usage

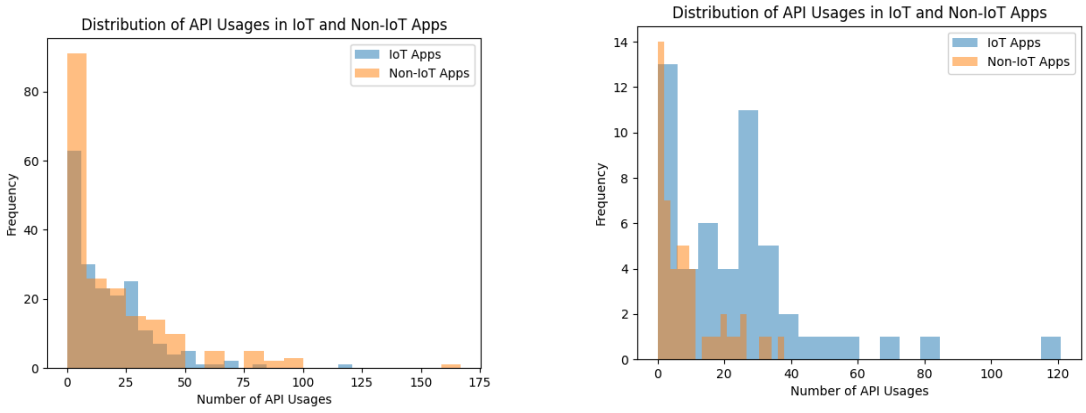


FIGURE: API Usage Distribution (Initial vs Expanded)

The graph shows a clear difference between the distribution of API usage in IoT and non-IoT apps.

Non-IoT apps: The distribution is more spread out, with a peak frequency around 25-50 API usages. This suggests that there's a wider range of API usage in non-IoT apps, with some apps using many APIs and others using only a few.
IoT apps: The distribution is skewed towards lower API usage, with the peak frequency around 0-25 API usages. This indicates that most IoT apps tend to use fewer APIs than non-IoT apps.

IoT apps often focus on collecting and processing data from sensors and other devices. This might involve fewer external interactions compared to non-IoT apps that deal with user interfaces, social interactions, or other activities.

8.3 Dynamic Class Usage Comparison

In the initial dataset, consisting of 51 IoT and 51 non-IoT apps, the analysis of dynamic class loading usage revealed a substantial disparity. The mean dynamic class loading for IoT apps averaged around 6.35, notably higher than the average of approximately 3.1 for non-IoT apps. Both categories exhibited relatively high variability in dynamic class loading, with standard deviations of 5.68 for IoT and 3.75 for non-IoT apps.

In the expanded dataset, encompassing 195 apps in each category and maintaining a size threshold of 15 MB, a different trend emerged. The mean dynamic class loading for IoT and non-IoT apps was approximately 5.5 and 5.68, respectively. Both categories continued to demonstrate high variability in dynamic class loading, with standard deviations of around 5.23 for IoT and 6.77 for non-IoT apps.

	Initial Dataset		Expanded Dataset	
Type	**IoT Apps**	**non-IoT Apps**	**IoT Apps**	**non-IoT Apps**
Count	51	51	195	195
Mean	6.35	3.10	5.50	5.68
Std	5.68	3.75	5.23	6.77
Median	5	2	4	4
Min	0	0	0	0
Max	23	17	23	46

TABLE: Descriptive Statistics of Dynamic Class Usage

In the initial dataset, a statistically significant difference was observed in the mean dynamic class loading between IoT and non-IoT apps. However, in the expanded dataset, this significant difference diminished. The means for both categories converged closer together, indicating a smaller discrepancy in dynamic class loading between IoT and non-IoT apps in the larger dataset.

Verdict	
Initial Dataset	There is a statistically significant difference in the mean dynamic class loading usage between IoT and non-IoT apps.
Expanded Dataset	There is no statistically significant difference in the mean dynamic class loading usage between IoT and non-IoT apps

TABLE: Verdict of Dynamic Class Loading

The majority of data points in the IoT boxplot are clustered within a smaller range on the x-axis (number of classes). This means that most IoT apps tend to use a similar, relatively low number of dynamic classes. This central cluster forms the "box" portion of the boxplot.

In contrast, the non-IoT boxplot has data points spread out over a wider range, indicating a greater variety in how many classes different non-IoT apps use.

There are a few data points in the non-loT app boxplot that extend beyond the whiskers (the upper and lower bars). These are outliers, indicating a small number of non-loT apps that use a significantly higher number of dynamic classes than the majority.

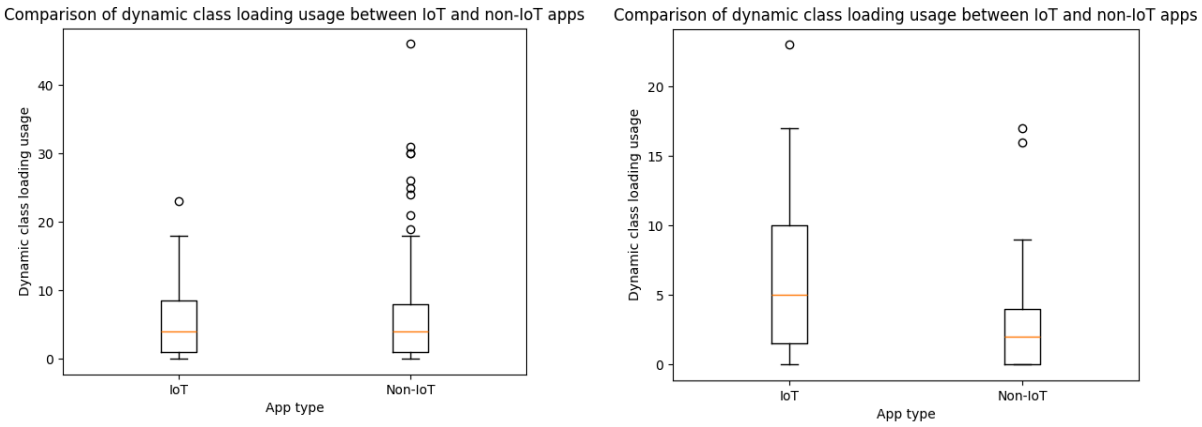


FIGURE: Dynamic Class Usage Box Plots (Initial vs Expanded)

8.4 App Permissions Comparison

In the initial dataset consisting of 51 IoT and 51 non-IoT apps, the analysis of permissions revealed a notable difference between the two categories. On average, IoT apps requested around 16.7 permissions, significantly higher than the approximately 8.1 permissions requested by non-IoT apps. Both categories exhibited relatively high variability in permission requests, with standard deviations of approximately 7.4 for IoT and 7.1 for non-IoT apps.

However, in the expanded dataset, encompassing 189 IoT and 195 non-IoT apps and maintaining a size threshold of 15 MB, the difference in permission requests between the two categories reduced. The mean permissions requested by IoT apps decreased slightly to around 16.4, while for non-IoT apps, it increased to approximately 9.4. The standard deviations remained high for both categories, approximately 10.4 for IoT and 7.2 for non-IoT apps.

	Initial Dataset		**Expanded Dataset**	
Type	**IoT Apps**	**non-IoT Apps**	**IoT Apps**	**non-IoT Apps**
Count	51	51	195	195
Mean	16.71	8.14	16.36	9.38
Std	7.38	7.13	10.43	7.18
Median	16	6	15	7
Min	4	0	0	0
Max	37	31	90	33

TABLE: Descriptive Statistics of App Permissions

Verdict	
Initial Dataset	IoT apps require significantly more permissions than non-IoT apps.
Expanded Dataset	IoT apps require significantly more permissions than non-IoT apps.

TABLE: Verdict of App Permissions

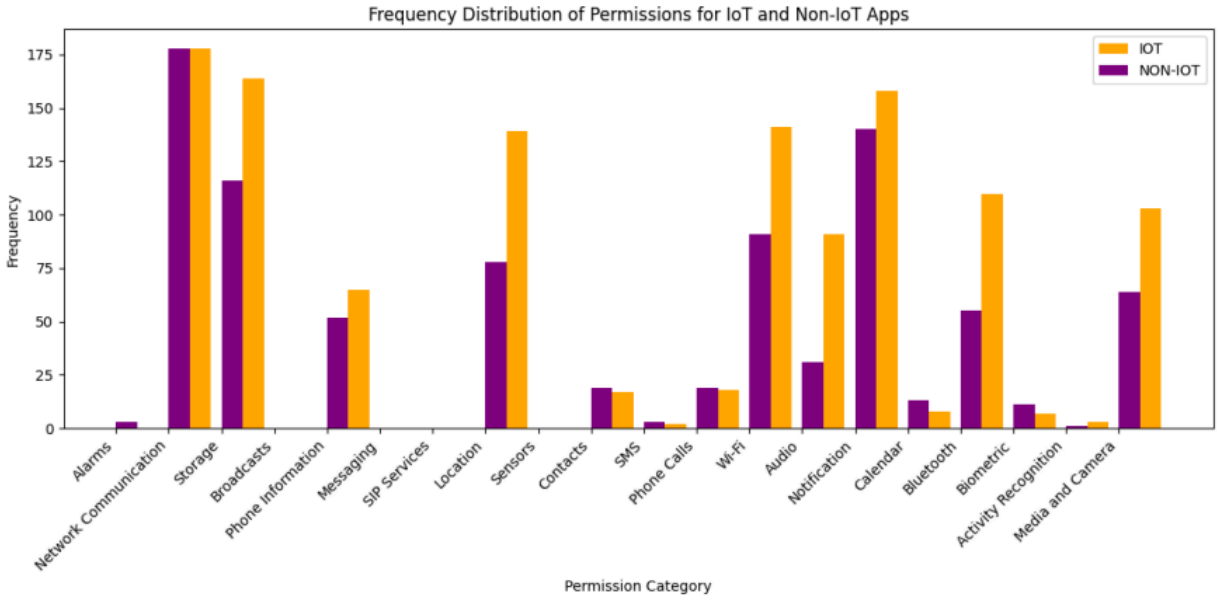


FIGURE: Distribution Path of App Permissions

As we can see, the top five app permissions needed for IoT & non-IoT-

IoT	**Non-IoT**
1. Network Communication	1. Network Communication
2. Storage	2. Notification
3. Notification	3. Storage
4. Sensors	4. Wi-Fi
5. Wi-Fi	5. Phone Information

TABLE: Top Five Occurrences

The top 10 permission co-occurrences for both datasets should also be considered. These patterns might highlight specific permissions that frequently appear together in IoT and non-IoT apps, offering insights into the distinct permission needs of each category.

It shows that IoT apps tend to give more permissions to network and connectivity, more than non-IoT apps. Overall, IoT apps require a lot more permissions than non-IoT apps.

8.5 Code Length Comparison

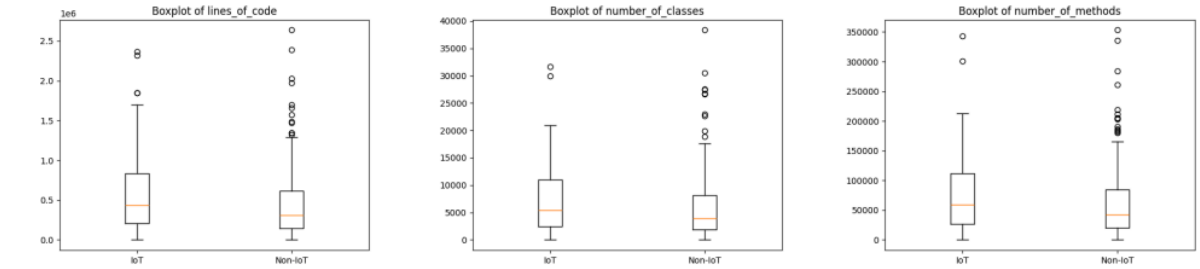
In the comparison of code length metrics between IoT and non-IoT datasets, several key observations were made.

IoT Data:
Lines of Code: The mean code length was approximately 612,476, with a standard deviation of 765,197. The range varied from 0 to 2,368,397 lines of code.
Number of Classes: The mean count of classes was around 7,966, with a standard deviation of 10,274. The range spanned from 0 to 31,699 classes.
Number of Methods: On average, there were approximately 84,378 methods, with a standard deviation of 111,314. The range varied from 0 to 343,504 methods.

Non-IoT Data:
Lines of Code: The mean code length was about 578,340, with a standard deviation of 861,572. The range spanned from 195 to 2,637,678 lines of code.
Number of Classes: The mean count of classes was approximately 8,142, with a standard deviation of 12,570. The range extended from 15 to 38,350 classes.
Number of Methods: On average, there were about 78,216 methods, with a standard deviation of 115,219. The range varied from 12 to 353,231 methods.

Correlation:
IoT: Strong correlations existed among all code length metrics, with values ranging from 0.97 to 1.0.
Non-IoT: Similarly strong correlations were observed among code length metrics, ranging from 0.98 to 1.0.
Boxplots and Scatterplots:
Boxplots and scatterplots comparing IoT versus non-IoT code length metrics revealed:

Boxplots (IoT vs Non-IoT)



Scatterplots (IoT vs Non-IoT)

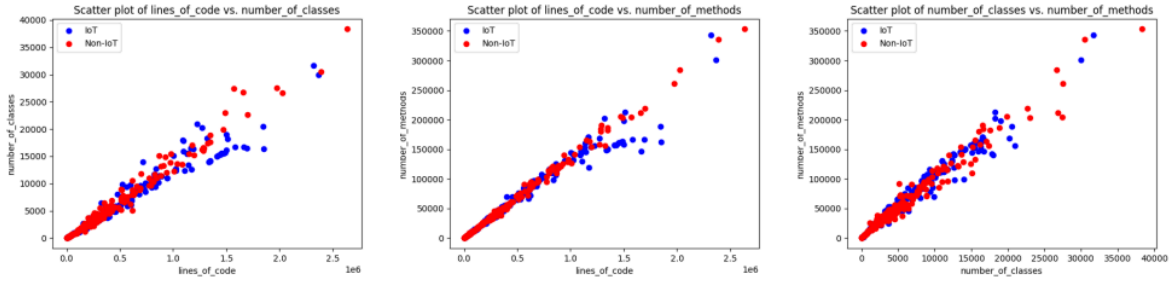


FIGURE: Box Plots & Scatter Plots of LOC, class & methods

Box Plots: Showed overlap in the distribution of code length metrics between IoT and non-IoT datasets.

Scatterplots: Displayed a trend of similarity in code length characteristics between IoT and non-IoT data points, with some scattering but no clear separation between the two categories.

Overall, the analysis suggests a considerable resemblance in code length characteristics between IoT and non-IoT datasets. While there are variations within each category, the correlations and visualizations indicate a notable similarity in code structure, class counts, and method counts between the two sets of applications, pointing toward convergence in coding practices or styles across IoT and non-IoT app development.

8.6 DB Storage Comparison

The analysis delves into the distribution and correlations between different database strategies employed within applications, specifically focusing on nine distinct strategies: Cursor, ContentResolver, MediaStoreQueries, SQLiteOpenHelper, RoomDatabasePatterns, RealmDatabase, FirebaseDatabase, ObjectBoxDatabase, and SQLiteDatabase. The analysis

involved percentage distributions, t-tests, chi-square tests, and correlation matrices for these database strategies.

Strategy	IoT (%)	Non-IoT (%)
Cursor	28.42051	25.82564
ContentResolver	4.40769	4.23026
MediaStoreQueries	1.11846	0.62872
SQLiteOpenHelper	2.57538	2.85026
RoomDatabasePatterns	0.9487	1.8821
RealmDatabase	8.98821	6.93333
FirebaseDatabase	0.24513	0.22667
ObjectBoxDatabase	0.00	0.00
SQLiteDatabase	11.28718	11.34974

TABLE: Percentage of DB Usage

Statistical Tests:

1) **T-tests:** Showed varying degrees of statistical significance for different database strategies between initial and expanded datasets. For instance, MediaStoreQueries displayed a statistically significant difference ($p = 0.0002$), while SQLiteDatabase showed no significant difference ($p = 0.9690$).
2) **Chi-square tests:** Indicated the degree of association between different strategies within the datasets. MediaStoreQueries demonstrated a significant association ($p = 0.0139$), while other strategies didn't show significant associations.
3) **Correlation Matrix:** The correlation matrix revealed the relationships between database strategies. High positive correlations were observed between SQLiteDatabase and Cursor (0.8168), SQLiteDatabase and SQLiteOpenHelper (0.9253), and Cursor and SQLiteOpenHelper (0.7699). These correlations suggest a potential interdependence or common usage patterns between these strategies within the apps.

The analysis underscores both similarities and disparities in the utilization of database strategies between the initial and expanded datasets. While some strategies like MediaStoreQueries exhibited significant differences and associations, others remained relatively consistent across datasets. The high correlations between certain strategies point towards their frequent combined usage within applications, indicating potential common functionalities or complementary usage patterns.

8.7 Reflection Comparison

The analysis compares the usage patterns of different reflection-related actions between IoT and non-IoT applications. This comparison involves various metrics such as counts, means, percentiles, and statistical tests for each reflection activity.

Reflection Metric	App Type	Count	Mean	Std Dev	Min	Median	Max
Class Loading	IoT	195	84.86	69.52	0.0	72.0	341.0
	Non-IoT	195	81.99	120.21	0.0	47.0	1289.0
Method Retrieval	IoT	195	160.81	114.70	0.0	152.0	630.0
	Non-IoT	195	167.18	196.90	0.0	115.0	2008.0
Instance Creation	IoT	195	274.79	460.82	0.0	184.0	5782.0
	Non-IoT	195	191.78	242.12	0.0	118.0	1572.0
Method Invocation	IoT	195	960.57	2017.40	0.0	217.0	17695.0

	Non-IoT	195	881.81	1662.48	0.0	164.0	8830.0
Field Retrieval	IoT	195	6.74	6.88	0.0	5.0	35.0
	Non-IoT	195	6.80	8.86	0.0	4.0	55.0
Access Control	IoT	195	0.0	0.0	0.0	0.0	0.0
	Non-IoT	195	0.0	0.0	0.0	0.0	0.0
Annotations Retrieval	IoT	195	1.49	4.28	0.0	0.0	31.0
	Non-IoT	195	1.95	5.55	0.0	0.0	35.0

TABLE: Descriptive Statistics of Various Reflection Types

This table presents the comparison of metrics for different reflection categories between IoT and non-IoT applications

Statistical Tests:
T-Tests:

No significant differences were observed in Class Loading, Method Retrieval, Method Invocation, Field Retrieval, Access Control, Annotations Retrieval, and Total Reflections between IoT and non-IoT apps.

However, there's a significant difference in Instance Creation ($p = 0.0265$), signifying disparate usage patterns between IoT and non-IoT apps regarding creating instances.

The analysis of reflection-related actions across IoT and non-IoT applications indicates similarities in most activities. However, Instance Creation stands out as significantly different

between the two categories. This suggests that while many reflection activities are similarly utilized across both IoT and non-IoT apps, the creation of instances exhibits notable variance.

Reflection (IoT vs Non-IoT)

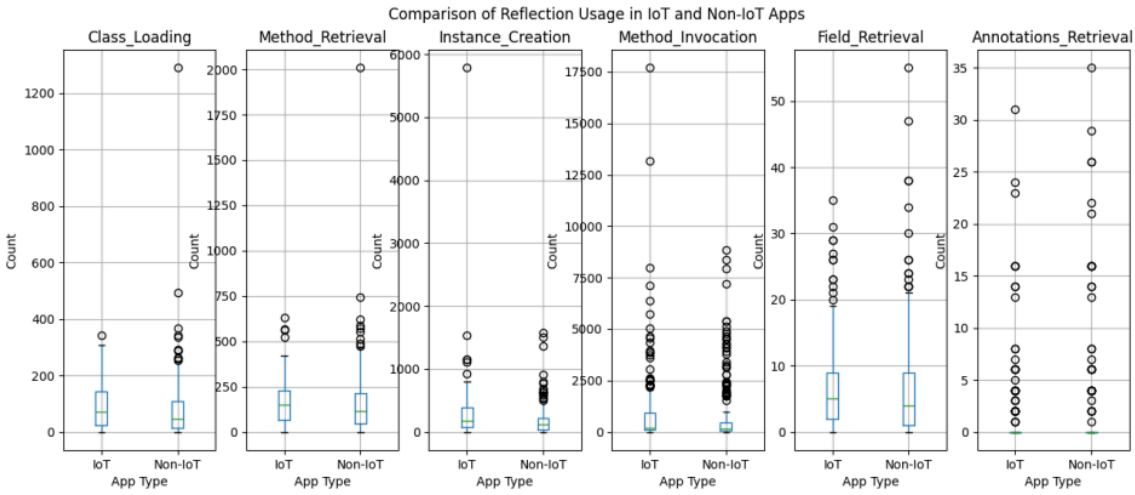


FIGURE: Box Plots of Reflection Usages

Chapter 9

Preliminary Test Plan

In this chapter, a high level description of testing goals and summary of features to be tested are presented.

9.1 High-level description of the testing goals

High-level testing goals for the tool can be summarized as follows:

1. Validate the tool's ability to perform comprehensive analysis on Android app projects, including code metrics, insights, and structural elements.

2. Ensure that users can create, save, and access analysis reports easily, maintaining data integrity and accessibility.

3. Verify that the tool accurately identifies and reports on various software metrics, such as lines of code, code complexity, library usage, permissions, and more.

4. Confirm that users can effectively compare and analyze findings between different projects through clear visual representations.

5. Test the tool's user interface for user-friendliness, efficiency, and error handling in various scenarios.

6. Assess the tool's overall performance, stability, and accuracy in handling Android app analysis tasks across different project sizes and complexities.

9.2 Summary of items and features to be tested

Test ID: T1
Test Case: Test if the tool calculates the total number of lines in the codebase correctly. **Input Test Data:** Source code folder **Steps To Be Executed:** 1. Run the tool. 2. Calculate the lines of code.

Expected Result: The calculated lines of code should match the actual number of lines in the input source code folder files. **Target Item:** Code Metrics Module **Pass/Fail:** Pending
Test ID: T2
Test Case: Test if the tool accurately counts the number of classes, functions, and methods in the codebase. Input Test Data: Source code folder Steps To Be Executed: 1. Run the tool. 2. Count the number of classes, functions, and methods. Expected Result: The counted structural elements should match the actual count in the input Java source code file. Target Item: Code Metrics Module Pass/Fail: Pending
Test ID: T3
Test Case: Test if the tool correctly calculates the percentage of code that is commented. **Input Test Data:** Source code folder **Steps To Be Executed:** 1. Run the tool. 2. Calculate the percentage of code that is commented. **Expected Result:** The calculated percentage of code comments should match the actual percentage in the input source code file. **Target Item:** Code Metrics Module **Pass/Fail:** Pending
Test ID: T4
Test Case: Test if the tool accurately assesses the code complexity using cyclomatic complexity. **Input Test Data:** Source code folder **Steps To Be Executed:** 1. Run the tool. 2. Calculate the cyclomatic complexity. **Expected Result:** The calculated cyclomatic complexity should match the actual complexity of the input source code file. **Target Item:** Code Metrics Module

Pass/Fail: Pending
Test ID: T5
Test Case: Test if the tool correctly identifies and counts external libraries used in the codebase. **Input Test Data:** Source code folder files with library references **Steps To Be Executed:** 1. Run the tool. 2. Identify and count external libraries. **Expected Result:** The identified libraries and their count should match the libraries referenced in the input source code file. **Target Item:** Dependency Analysis Module **Pass/Fail:** Pending
Test ID: T6
Test Case: Test if the tool accurately examines and reports permissions in AndroidManifest.xml. **Input Test Data:** AndroidManifest.xml file with permissions **Steps To Be Executed:** 1. Run the tool. 2. Examine and report permissions. **Expected Result:** The reported permissions should match the permissions declared in the input AndroidManifest.xml file. **Target Item:** Permission Analysis Module **Pass/Fail:** Pending
Test ID: T7
Test Case: Test if the tool correctly detects dynamic code loading mechanisms. **Input Test Data:** Source code files with dynamic code loading **Steps To Be Executed:** 1. Run the tool. 2. Detect dynamic code loading. **Expected Result:** The detected dynamic code loading mechanisms should match those present in the input source code file. **Target Item:** Dependency Analysis Module **Pass/Fail:** Pending
Test ID: T8

Test Case: Test if the tool quantifies reflection and class loading usage accurately.
Input Test Data: Source code files with reflection and class loading
Steps To Be Executed:
1. Run the tool.
2. Quantify reflection and class loading usage.
Expected Result: The quantified usage should match the reflection and class loading present in the input source code file.
Target Item: Dependency Analysis Module
Pass/Fail: Pending

Test ID: T9

Test Case: Test if the tool determines the data serialization format correctly.
Input Test Data: Source code files with data serialization
Steps To Be Executed:
1. Run the tool.
2. Determine the data serialization format.
Expected Result: The determined format should match the actual data serialization format used in the input source code file.
Target Item: Data Analysis Module
Pass/Fail: Pending

Test ID: T10

Test Case: Test if the tool identifies network protocols used accurately.
Input Test Data: Source code files with network communication
Steps To Be Executed:
1. Run the tool.
2. Identify network protocols used.
Expected Result: The identified network protocols should match those used in the input source code file.
Target Item: Data Analysis Module
Pass/Fail: Pending

Test ID: T11

Test Case: Test if the tool performs static analysis for security vulnerabilities correctly.
Input Test Data: Source code files with security vulnerabilities
Steps To Be Executed:
1. Run the tool.
2. Perform static analysis for security vulnerabilities.

Expected Result: The tool should correctly identify and report security vulnerabilities in the input source code files. **Target Item:** Data Analysis Module **Pass/Fail:** Pending
Test ID: T12
Test Case: Test if the tool accurately analyzes layout files for UI components. **Input Test Data:** Android layout XML files **Steps To Be Executed:** 1. Run the tool. 2. Analyze layout files for UI components. **Expected Result:** The tool should correctly identify and report UI components and their properties in the input layout XML files. **Target Item:** UI Analysis Module **Pass/Fail:** Pending
Test ID: T13
Test Case: Test if the tool determines the data storage strategy correctly. **Input Test Data:** Source code files with data storage operations **Steps To Be Executed:** 1. Run the tool. 2. Determine the data storage strategy. **Expected Result:** The determined data storage strategy should match the actual strategy used in the input source code files. **Target Item:** Data Storage Analysis Module **Pass/Fail:** Pending

These test cases cover each of the mentioned metrics and their respective modules. You can execute these tests to ensure that your tool functions correctly and produces accurate results.

Chapter 10

Conclusion

In conclusion, IoTWhiz stands as a powerful and specialized desktop tool dedicated to the intricate world of IoT (Internet of Things) Android applications. Through its innovative program slicing approach, IoTWhiz enables in-depth characterization of IoT apps, providing insights into their structure, behavior, and dependencies. By dissecting code structures, evaluating dependencies, analyzing user interfaces, assessing data handling, identifying network protocols, and scrutinizing security and permissions, IoTWhiz offers a comprehensive toolkit for developers, project managers, and security analysts in the IoT application domain.

With a focus on optimizing software maintenance, enhancing performance, conducting program analysis, and fortifying security, IoTWhiz empowers professionals to navigate the unique challenges presented by IoT app development. By addressing these challenges head-on, IoTWhiz contributes to the continued growth and success of the IoT ecosystem, ensuring that IoT applications are not only efficient and user-friendly but also secure and reliable.

As the IoT landscape continues to expand and evolve, tools like IoTWhiz play a crucial role in supporting developers and analysts in their quest to create cutting-edge IoT solutions. In an ever-connected world, IoTWhiz stands as a beacon of knowledge and insight, helping to shape the future of IoT application development and ensuring that IoT-powered devices and services continue to enrich our lives in meaningful and secure ways.

References

adwaitnadkarni.com/downloads/manandhar-ccs22.pdf

IEEE Xplore Full-Text PDF:

Download APK on Android with Free Online APK Downloader - APKPure

Androzoo home (uni.lu)

What is Hypothesis Testing in Statistics? Types and Examples | Simplilearn

What You Need to Know About Inferential Statistics to Boost Your Career in Data Science (simplilearn.com)