2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)

FirmProj: Detecting Firmware Leakage in IoT Update Processes via Companion App Analysis

Wenzhi Li*#, Jialong Guo*#, Jiongyi Chen†, Fan Li*, Yujie Xing*, Yanbo Xu‡, Shishuai Yang*, and Wenrui Diao*§¶(✉)

*School of Cyber Science and Technology, Shandong University, diaowenrui@link.cuhk.edu.hk

†National University of Defense Technology ‡Shanghai Jiao Tong University

§Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University

¶State Key Laboratory of Cryptography and Digital Economy Security, Shandong University

***Abstract*—The rapid growth of the Internet of Things (IoT) has led to the widespread use of companion apps for device management. However, these apps expose a critical vulnerability in the IoT ecosystem: insufficient verification procedures during device firmware updates (DFU), often resulting in firmware leakage. Once leaked, the firmware reveals sensitive design details, creating a straightforward path for attackers to reverse-engineer devices. To address this issue, we designed an automated analysis tool called FirmProj. It systematically evaluates firmware leakage risks by examining IoT companion apps. FirmProj combines advanced static analysis techniques with large language models to identify DFU modules, extract firmware files, and detect security vulnerabilities. In a large-scale study involving 10,047 IoT companion apps, FirmProj successfully retrieved 3,434 firmware files, uncovering severe flaws in DFU implementations that can lead to firmware leakage. These findings resulted in the assignment of 35 CVE IDs. Our results highlight the urgent need to strengthen firmware protection mechanisms throughout the IoT ecosystem.**

I. Introduction

The Internet of Things (IoT) has experienced explosive growth, permeating various aspects of modern life – from smart homes and wearable devices to industrial automation and critical infrastructure [31]. However, as the number of IoT devices surges, their security issues have increasingly drawn attention. Since these devices typically handle sensitive data and are connected to critical infrastructure, any security vulnerabilities could lead to privacy breaches, system failures, or even larger-scale cyberattacks. To address these challenges, manufacturers regularly release firmware updates to patch discovered vulnerabilities and improve the security of their devices. This update mechanism not only fixes existing security flaws but also strengthens the device's ability to defend against potential threats, thereby prolonging the device's secure lifespan.

Updating the firmware via Over-the-Air (OTA) is the mainstream method for device firmware update (**DFU**). It allows vendors to push new firmware remotely to update their devices without a physical connection. To make it easier for users to control devices and access device information, manufacturers typically provide companion apps on smartphones [15]. Users can send a request through companion apps to the server to check for new firmware versions. If a new version is available, the device automatically downloads the firmware over the network and updates its current version.

# Wenzhi Li and Jialong Guo contributed equally (co-first authors).

Unlike traditional system updates primarily managed by the device itself, firmware updates through companion apps introduce new attack vectors during checking, downloading, and transferring firmware. Ibrahim et al. [14] conducted a security analysis of firmware update mechanisms in IoT devices via companion apps, revealing severe vulnerabilities in several SDKs implementing update functions. This study highlights security risks linked to SDKs for transferring firmware from apps to devices. However, previous studies have not considered the risk of firmware leakage during the DFU process, particularly the risk of leakage exploiting companion apps. For example, in the IoT companion app (package name: `com.theswitchbot.switchbot`) developed by SwitchBot [28], the URL of a firmware information `XML` file is hard-coded, which allows attackers to reverse-engineer the app and download the latest firmware without the physical device, leading to firmware leakage (this issue has been responsibly reported to the relevant vendor).

As a vital intellectual property asset for device manufacturers, firmware often contains low-level implementation details that can, if leaked, give attackers direct access to perform vulnerability analysis. Recent studies have shown that leaked firmware can expose vulnerabilities that lead to real-world attacks such as data theft and unauthorized access [38], [11], [39]. Unlike indirect methods of inferring device security through apps [6], [35], direct access to firmware allows attackers to analyze systems much more efficiently. This enables faster matching of n-day vulnerabilities and can even facilitate the discovery of zero-day vulnerabilities.

Our Work. In this study, we developed an automated analysis tool, FirmProj, to systematically investigate firmware leakage vulnerabilities during DFU from the perspective of IoT companion apps. Specifically, in the design of FirmProj, we employ advanced static analysis techniques, such as value set analysis, to locate DFU modules within Android apps and extract network requests associated with firmware updates. To enhance automation capabilities, we develop a workflow that leverages Large Language Models (LLMs) for automated validation and analysis of the extracted network requests. Ultimately, FIRMPROJ generates a detailed report and extracts the obtained firmware files, enabling the evaluation of potential security vulnerabilities in apps, such as insecure firmware storage, unencrypted firmware transmission during download, or insufficient validation mechanisms.

To evaluate the effectiveness of FIRMPROJ and to understand the current state of firmware leakage risks, we conducted a series of experiments guided by the following research questions:

RQ1: *How effectively can* FIRMPROJ *detect firmware leakage risks in IoT companion apps?*

RQ2: *How prevalent are direct firmware leakage vulnerabilities caused by insecure update processes in IoT companion apps?*

RQ3: *How effective is* FIRMPROJ *in processing incomplete firmware update requests that lack dynamic information?*

To answer these questions, we collected 10,047 unique IoT companion apps and performed a large-scale security evaluation using FIRMPROJ. Through this process, we successfully recovered 3,434 firmware files from 428 apps. When firmware could not be directly obtained due to missing dynamic parameters (e.g., `UserToken`), FIRMPROJ still provided sufficient auxiliary information to facilitate manual dynamic analysis. To verify this auxiliary approach, we manually examined ten apps flagged by FIRMPROJ for lacking dynamic parameters and obtained 30 firmware files. Notably, these assessments exposed leakage issues in IoT systems from prominent manufacturers, including Yamaha and Panasonic. Our results indicate that many IoT products are insufficiently protected, potentially allowing attackers to retrieve firmware without restriction.

Responsible Disclosure. We reported our discovered vulnerabilities to the corresponding IoT vendors or the CVE Program (in cases, direct vendor contact was unavailable or no suitable reporting channel existed). Only five vendors responded to our reports, mostly with automated acknowledgments, while Yamaha was the only vendor that confirmed receipt and indicated further investigation. As a result, all confirmations came directly from the CVE Program. At the time of writing, **35 CVE IDs** have been assigned, with **9 critical**, **22 high**, and 4 medium severity ratings, including `CVE-2024-[48538~48542, 48544~48548, 48768~48778, 48784, 48786~48793, 48795~48799]`.

Contributions. Here we list the main contributions:

- ***Systematic Study.*** We conducted a systematic study on firmware leakage in the IoT ecosystem from the perspective of companion apps.
- ***New Analysis Tool.*** We developed an automated tool, FIRMPROJ, to detect firmware leakage vulnerabilities within the DFU process. FIRMPROJ takes IoT companion apps as input, automatically retrieves firmware, and generates analysis reports.
- ***Real-world Evaluation.*** Using FIRMPROJ, we evaluated 10,047 IoT companion apps and successfully extracted 3,434 firmware files. Also, our findings led to 35 CVE IDs being assigned, nine of which were rated as critical.

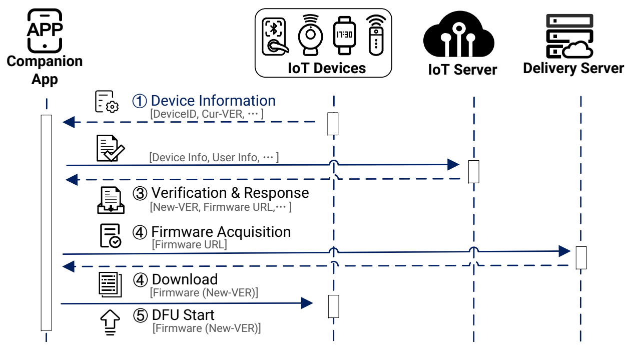


Fig. 1: Firmware update through IoT companion app.

Open Science. The source code of our tool is available at `https://github.com/GuoXBQ-Q/FirmProj`. The raw data contains sensitive information that could be exploited, so it cannot be released publicly.

II. BACKGROUND

In this section, we provide the necessary background on IoT device firmware updates and introduce the firmware acquisition process through the companion app.

A. *IoT Device Firmware Update*

Device Firmware Update (DFU) is vital for sustaining the functionality, security, and performance of IoT devices. Over-the-air (OTA) updates have become the primary method for delivering firmware remotely, removing the need for physical access. In IoT systems, companion apps typically manage OTA updates in one of two roles: (1) as a DFU initiator, triggering the update while the device handles the download and installation, or (2) as a DFU manager, handling the entire process, from checking for updates to transmitting and confirming firmware installation. The latter role involves more complex implementation. Therefore, analyzing how these apps retrieve, download, and store firmware is crucial for identifying potential leakage vulnerabilities.

We manually collected 230 companion apps from third-party app markets (Wandoujia [33], PC6 [22], and APKPure [4]). We first filtered apps in the "IoT Smart Home" category of these markets and then selected those that offered DFU functionality. Afterward, we performed a detailed study of their DFU processes using a combination of static and dynamic methods. Each APK was decompiled into Java source code with JADX [26]. By filtering activities and identifying update-related methods through keyword matching, we located the DFU implementation, including firmware acquisition, storage, and connection setup to initiate updates. This analysis revealed a typical DFU process for companion apps, as shown in Figure 1.

1) **Obtaining Device Information.** Companion apps first retrieve essential device information, such as `DeviceID` and current firmware version (`Cur-VER`), typically during pairing or subsequent interactions.
2) **Sending DFU Network Request.** With this information, the companion app sends a network request to the IoT

server to check for firmware updates, including device and user data for access control and validation.

3) **Server Verification & Response.** The IoT server validates the request and, if a new firmware is available, responds with details like the version number, download URL, and update description. If not, it returns an error (e.g., "*the firmware version is already up-to-date*" or "*illegal request*").
4) **Firmware Acquisition & Download.** If an update is available, the app prompts the user. Upon confirmation, it downloads the firmware to a temporary cache and notifies the user that the DFU is ready.
5) **DFU Start.** The app checks pre-update conditions (e.g., connectivity, battery level), then prompts the user to begin. Upon confirmation, it sends the firmware to the device. After completion and reboot, the app verifies the update's success by retrieving device information.

Firmware updates delivered through companion apps offer convenience but pose significant security risks. Because these apps are easily accessible, skilled attackers can reverse-engineer them to uncover the firmware retrieval process, potentially leading to leakage. If update requests in Step (2) lack proper authentication or authorization, unauthorized users may directly access the firmware. To investigate these risks, we further examine the firmware acquisition mechanisms employed by companion apps in the following section.

B. *Firmware Acquisition via Companion App*

Insecure implementation of IoT companion apps can expose vulnerabilities that allow attackers to access sensitive assets like firmware. To assess firmware leakage risks, it is essential to investigate how these apps acquire and store firmware. We manually analyzed 230 IoT companion apps to study their DFU processes and identified three leakage scenarios. These scenarios were categorized based on whether the firmware is locally embedded or remotely downloaded, and whether proper access controls are applied:

Scenario 1: App-bundled Firmware Leakage. Some companion apps, such as AWOX (`com.awox.smart.control`), lack network request functionality for updates. Instead, new firmware is bundled directly within the APK's resource directories and released with each app update. While this reduces server maintenance, it increases the risk of firmware leakage, as attackers can decompile the app to extract the embedded firmware.

Scenario 2: Firmware Leakage without Controls. Some IoT companion apps, such as smartplus (`com.ledvance.smartplus`), check for firmware updates via simple HTTPS GET requests without query parameters or access controls. Instead of verifying specific device information, these apps retrieve a generic configuration file (e.g., versionInfo.json) containing firmware details for all supported devices, including version, release date, download URL, and checksum. The app parses this file, compares versions, and prompts users to update if needed. Attackers can reverse-engineer the app to locate and access the configuration file URL, exposing firmware download links for all devices and leading to significant firmware leakage risks.

Scenario 3: Firmware Leakage via Access Control Flaws. While most companion apps require device or user information in firmware update requests, some servers implement weak access control on the server side. For example, in the DreamCatcher app (`com.dc.dreamcatcherlife`), requests include `DeviceID` and `UserToken` fields, but the server does not verify their binding. Any registered user can combine their `UserToken` with a valid `DeviceID` to bypass access control and access firmware.

This scenario allows attackers to collect valid `DeviceIDs` (e.g., by enumerating if `DeviceIDs` follow sequential integers, as in DreamCatcher), and use any user's credentials to download firmware at scale. Furthermore, when firmware update requests require dynamic data like `UserToken` or `DeviceID`, the static information within the app is insufficient to reconstruct the complete request. Since these values are generated at runtime (e.g., during user registration or device binding), we supplemented our automated analysis with manual inspection to identify leakage risks from dynamic request construction (see §IV-D).

III. Design of FirmProj

We designed FirmProj, an automated analysis tool, to identify firmware leakage vulnerabilities in IoT companion apps. Figure 2 illustrates the overall workflow of FirmProj, which consists of two main phases: (1) *Automated Request Reconstruction* (P1-ReqRec for short, §III-A); and (2) *LLM-Guided Firmware Retrieval* (P2-FirmRetr for short, §III-B).

FirmProj analyzes IoT companion apps to generate both an analysis report and the acquired firmware. In the P1-ReqRec phase, it employs static analysis to extract firmware update-related network requests. In the P2-FirmRetr phase, FirmProj utilizes LLMs to format and classify these requests by completeness. For complete requests, it attempts firmware downloads; for incomplete ones, typically requiring device binding or dynamic parameters, it generates detailed reports to support manual analysis.

A. *Phase 1: Automated Request Reconstruction*

To detect firmware leakage via companion apps, we first identify the methods responsible for firmware updates and reconstruct their network requests using Value Set Analysis (VSA). VSA is a program analysis technique that determines possible values at specific program points through backward slicing and forward value propagation. Previous work [24], [44], [43] has leveraged VSA to recover backend URLs and Bluetooth UUIDs. Building on the design of IoTFlow [24], we extend this approach to the firmware update scenario. Specifically, we locate the firmware update request methods, trace their parameter sources, and use VSA to reconstruct firmware acquisition requests. This reconstructed information supports our subsequent firmware retrieval.

FirmProj begins by decompiling the target companion app and constructing its call graph. Due to variations in firmware

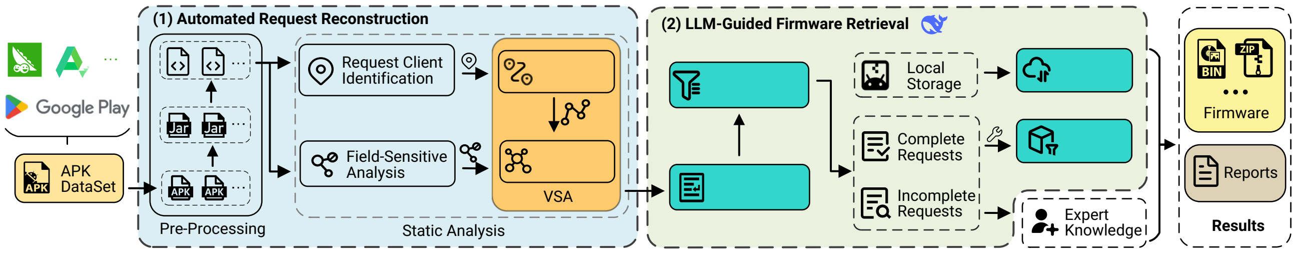


Fig. 2: Overview of FIRMPROJ.

update implementations, it first identifies key entry points, typically class methods that send firmware update requests, for VSA. Notably, the required fields in these requests often originate from class-level variables. FIRMPROJ performs a field-sensitive analysis to determine their possible values, enabling precise backward tracing and forward rebuilding of update requests. The detailed steps of our approach are as follows:

(1) Pre-Processing. FIRMPROJ decompiles the input APK into JAR files using dex2Jar, then converts them into Jimple, Soot's intermediate representation [32]. It builds a call graph by analyzing all non-library methods and statements to facilitate parameter source tracing in later stages.

(2) Request Client Identification. FIRMPROJ identifies methods that send firmware update requests, referred to as "Request Clients" (e.g., `getNewVersion` in Listing 3). These methods collect parameters like the `URL` and `Request Body`, and invoke network APIs (e.g., `OkHttpClient`, `java.net.URL`). Serving as entry points for VSA, they allow FIRMPROJ to determine potential parameter values during firmware update requests.

To identify these methods, FIRMPROJ focuses on common network APIs used in companion apps. It traces calls to these APIs via the call graph from pre-processing. Then it filters for update-related methods using DFU-related keywords (e.g., `checkFirmwareUpdate`) found in fields, constants, and parameters. Subsequently, FIRMPROJ employs program slicing to locate the construction process of network requests, and applies a depth-first strategy to analyze parameter passing and assignments. When a method is identified as a potential request client, FIRMPROJ maps its parameters to the components of a network request (e.g., URL, Header) and traces their sources, even across multiple construction layers. The parameters of these Request Clients are then used as targets for VSA.

(3) Field-Sensitive Analysis. Firmware update requests often include device- or user-specific parameters, such as query strings in GET requests (e.g., `deviceID=1001&userToken=abcd`). These key-value pairs typically originate from class fields, where the field name serves as the key and its content as the value. VSA starts at the request client, which retrieves these fields and formats them accordingly. To improve accuracy and reduce path explosion, FIRMPROJ performs a field-sensitive analysis to pre-compute possible values for each class field, streamlining backward tracing and forward rebuilding in VSA.

In this step, FIRMPROJ analyzes the value sets of all class fields and tracks the methods that operate on them, building a Field-Method Mapping to identify the sources and possible values of each field. Most non-static fields have corresponding setter and getter methods, identified by backtracking through all non-library classes and methods to analyze the method behavior. Setter methods assign values to fields from parameters, while getter methods return field values. This mapping is many-to-many, as methods can interact with multiple fields and vice versa. FIRMPROJ also monitors assignment behaviors, such as static assignments in class `init` functions, to capture the complete value set for each field.

FIRMPROJ also considers parameter transfer during function calls in its backtracking traversal. If a setter method's parameter is associated with a field, FIRMPROJ maps the parameter to that field, helping identify their relationships. When method calls pass constant or resolved values, these values are added directly to the field's value set.

(4) Backward Tracing. In this step, FIRMPROJ traces the origins of the target parameters within each identified Request Client. With the field value sets precomputed through the sensitivity analysis in the previous step, backward tracing can focus on the construction of parameters without repeatedly inferring possible field values, which simplifies the process. FIRMPROJ then identifies the target parameters within the Request Client and performs backward tracing by analyzing method calls, parameter passing, class initializations, and object instantiations to determine their sources and construction. This step is essential for accurately establishing parameter value sets in the subsequent forward rebuilding phase.

During backward tracing, FIRMPROJ follows the invocation paths in the call graph until it either reaches a definitive parameter construction, exceeds a predefined backtracking limit[1], or encounters an absence of explicit invocations. For each method encountered, FIRMPROJ analyzes a forward code slice up to the relevant invocation statements. If a parameter's value still depends on the current method's parameters, the backward tracing continues; otherwise, FIRMPROJ considers the source identified.

To further expand the coverage of request information, FIRMPROJ also addresses complex request construction methods, such as array iteration for formatting or the use of encryption functions for parameter generation. These complexities can hinder static analysis from capturing complete value sets. To

[1]Empirical testing set this cap at five steps, as longer traces led to path explosion without improving accuracy.

```
[SolveClient]:
<com.gooclient.anycam.activity.settings.update.UpdateFirmwareActivity2: void getNewVersion(java.lang.String)>
Client = [{
  Body = ["application/octet-stream",
    format(
      0 = {"AppCom": "%s", "SolCom": "%s", "ReleaseTime": "%s", "HDVersion": "%s"},
      1 = [
        <com.gooclient.anycam.activity.settings.update.MyHttp: java.lang.String encryption(java.lang.String,
java.lang.String)> {0=[InvokeArray, <com.gooclient.anycam.activity.settings.update.MyHttp: java.lang.String[]
parseJSON(java.lang.String, java.lang.String[])> {0=UNKNOWN, 1=[AppCom, SolCom, ReleaseTime, HDVersion]},
get(0)], 1=Goolink2014},
        <com.gooclient.anycam.activity.settings.update.MyHttp: java.lang.String encryption(java.lang.String,
java.lang.String)> {0=[InvokeArray, get(1)], 1=Goolink2014},
        <com.gooclient.anycam.activity.settings.update.MyHttp: java.lang.String encryption(java.lang.String,
java.lang.String)> {0=[InvokeArray, get(2)], 1=Goolink2014},
        <com.gooclient.anycam.activity.settings.update.MyHttp: java.lang.String encryption(java.lang.String,
java.lang.String)> {0=[InvokeArray, get(3)], 1=Goolink2014}
  ])],
  url = "https://app.login.yunyis.com/g_version_match.php"}]
```

Fig. 3: Example output (Neye3c) of P1-ReqRec phase.

mitigate this, FirmProj records unresolved method signatures and related parameter value sets, especially those involving encryption or complex formatting, during backward tracing. When these signatures appear in the VSA results, FirmProj performs additional code slicing on the corresponding methods and generates method queries that include both code slices and parameter value sets. These queries are then analyzed by the LLM in P2-FirmRetr, enabling FirmProj to access encryption or formatting logic used by companion apps in update requests.

(5) Forward Rebuilding. FirmProj begins with the identified source of specific parameters and simulates code execution to reconstruct their value sets. During this simulation, the sources of parameters mainly fall into three categories:

- ***Resolved value sets from fields and methods.*** FirmProj simulates forward execution of code slices to propagate known value sets and reconstruct the complete values of target parameters. For example, when building request parameters, it simulates common classes like Map and String to leverage resolved values. If a Request Client directly uses class instances and converts their fields into "`key=value`" pairs, FirmProj retrieves those fields and their value sets as the target parameter values.
- ***Unresolved values from methods and fields.*** When FirmProj encounters an unresolved method, it treats the return value as a target and applies VSA to attempt resolution. If successful, the method is resolved; otherwise, FirmProj retains information about the method call, such as the method name and the parameter value sets. Similarly, if a field remains unresolved after tracing via its setter, the tool preserves the field's name and data type as its value set.
- ***Parameters with unidentified sources.*** FirmProj verifies whether parameters originate from callback methods within the identified Request Client. If so, it maps them to the corresponding client, ensuring traceability of their sources.

Using the parameter-to-request mapping, FirmProj reconstructs the necessary components for the target Request Client and outputs the reconstructed information (see Figure 3).

B. Phase 2: LLM-Guided Firmware Retrieval

To verify the potential firmware leakage risks in the firmware update methods identified from companion apps, FirmProj examines the extracted requests using a multi-stage interactive workflow powered by LLMs, called P2-FirmRetr. By decomposing the process into sequential steps and invoking LLMs, we automate tasks that would otherwise require extensive manual effort. Two main factors motivate the use of LLMs in this request verification phase:

- ***Diverse request structures.*** IoT companion apps rely on varied methods to construct firmware update requests, resulting in different formats from the P1-ReqRec phase. Script-based solutions struggle to handle such diversity, whereas LLMs excel at semantic parsing, code comprehension, and generalization, enabling them to normalize requests into a unified format as specified by the prompt.
- ***Function-calling capabilities.*** Once the LLM extracts parameters in the required format, it can automatically send requests to download firmware, eliminating the need for custom scripts for each request type.

The workflow takes network requests from P1-ReqRec as input and outputs an analysis report on IoT companion apps and the obtained leaked firmware. Following OpenAI's prompt design principles, the P2-FirmRetr process is divided into three sequential subtasks: (1) network request information formatting; (2) network request classification; and (3) firmware acquisition. Each subtask's output feeds into the next, enabling a structured and automated analysis.

(1) Network Request Information Formatting. In this step, the LLM processes network requests extracted during the P1-ReqRec phase (see Figure 3). Since the output may include verbose Jimple-formatted signatures and redundant data (e.g., empty fields or raw method signatures), the LLM refines and standardizes the information. The formatting focuses on four key properties: `Method Signature`, `URL`, `Body`, and `Converter`. The LLM extracts all the fields needed for building network requests and analyzes method queries from the Backward Tracing step to determine the functional roles of parameters (e.g., encryption methods or keys). Unresolved method signatures and their LLM-derived answers are stored under the Converter attribute. An example output of this step is shown in Listing 1.

```
"Request Method Sig": "void getNewVersion(
    java.lang.String)",
"url": {
  "base_url": "https://app.login.yunyis.com",
  "path": "/g_version_match.php" },
"method": "POST",
"headers": {
  "Content-Type":"application/octet-stream"},
"body": {
  "AppCom": "<dynamic from encryption>",
  "SolCom": "<dynamic from encryption>",
  "ReleaseTime": "<dynamic from encryption>",
  "HDVersion": "<dynamic from encryption>" },
"converter": {
  "function_calls": [
    {"function_name": "encryption",
    "parameters": ["<dynamic>","Goolink2014"
        ]}]}
```

Listing 1: Example Output (Neye3c) of P2-FirmRetr-Step (1).

(2) Network Request Classification. In this step, the LLM takes the formatted network requests from the previous step and classifies them according to their completeness. Specifically, it checks whether dynamic fields are present or if any parameters are missing. Based on this analysis, each request is categorized as either complete or incomplete.

To improve classification stability, we adopt a dynamic majority voting mechanism. Each request is first classified by five independent LLM instances, and a consistency score is calculated. Classification stops once the score reaches 0.8 or higher (at least 4 out of 5 votes agree). If the threshold is not met, additional voting rounds are conducted until the score reaches 0.8 or the preset maximum number of votes is exhausted. The final classification result is then determined by majority voting. Note that all LLM instances operate independently without sharing historical messages, which prevents contextual interference in the classification results.

(3) Firmware Acquisition. In this step, the LLM checks the requests classified as complete in the previous step. Using its function-calling capability, it constructs the API request parameters, sends the request, processes the server response, and attempts to retrieve the firmware. In practice, two main types of server responses are encountered:

- The firmware is downloaded directly from the server, where the request itself provides the download link.
- The server returns valid firmware information that includes a download link (see Listing 2).

```
{"SYLVANIA A19 C1 SA": {
   "fwFile":"https://apps.ledvanceus.com/
     firmware/***/***/A19C1_SA-M_2.2.05.bin",
   "version":"2.2.05", "deviceTypeId":
   "sylvania-a19-c1-sa"}, ......
}
```

Listing 2: Download link (`com.ledvance.smart.plus`). Some sensitive information is redacted (`***`) for ethical reasons.

In the second case, the LLM extracts the firmware link from the response and invokes the network request API again to complete the download. If neither situation applies, the request and response are saved for manual review.

For Scenario 1, some companion apps store firmware locally rather than downloading it in real time. To detect such cases, FIRMPROJ scans the APK resource directory and filters files using common firmware-related extensions from the ChkUp dataset [37] (e.g., `ZIP`, `BIN`, `HEX`) to identify potential candidates. The filenames and paths of these candidates are then provided to an LLM, which determines whether they match typical firmware naming patterns (such as including device models and version numbers) or contain relevant keywords like "firmware" or "ota". For firmware obtained through network downloads, the same extension-based filtering and LLM-based identification process is also applied.

Each step uses carefully designed few-shot prompts to guide the LLM, and outputs follow a standardized format for seamless execution. The prompts are available in our online repository. A low model temperature of 0.5 ensures consistent responses. Once the P2-FirmRetr phase is complete, FIRMPROJ compiles results from all stages into a final report.

IV. Evaluation and Findings

This section outlines the experimental procedure and the findings. We aim to answer three research questions:

RQ1: *How effectively can* FIRMPROJ *detect firmware leakage risks in IoT companion apps?*

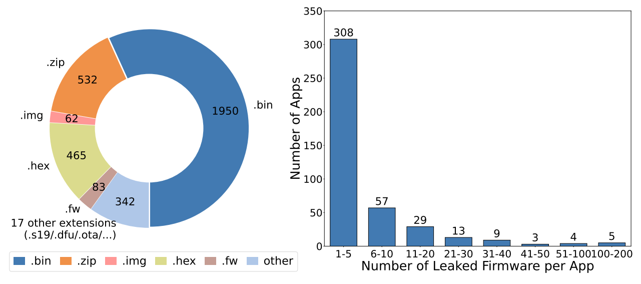
RQ2: *How prevalent are direct firmware leakage vulnerabilities caused by insecure update processes in IoT companion apps?*

RQ3: *How effective is* FIRMPROJ *in processing incomplete firmware update requests that lack dynamic information?*

A. Experiment Setup

We implemented a prototype of FIRMPROJ (building upon IoTFlow [24]), consisting of 8,140 lines of Java code and 1,700 lines of Python code. The LLM we used is DeepSeek-V3 [8].

IoT Companion App Dataset. Our experiments used a dataset of 10,047 unique IoT companion apps. The dataset consists of two parts: the first includes 230 apps manually downloaded from Wandoujia [33], PC6 [22], and APKPure [4], which were also used for manual analysis in Section II; the second consists of 9,889 verified apps from IoT-VER [24]. After identifying and removing 72 duplicates by package name, we obtained a final



(a) Distribution of Firmware File Extensions (b) Distribution of Leaked Firmware per App

Fig. 4: Distribution of leaked firmware.

dataset of 10,047 distinct apps. To ensure timeliness, we cross-checked the IoT-VER apps against the continuously updated AndroZoo repository [1] and replaced outdated versions.

Execution Environment. FIRMPROJ was deployed on an Ubuntu 22.04 server with 96 AMD EPYC™ 9654 2.40 GHz CPU cores and 1,024 GB RAM, allocating 100 GB of memory per app for analysis. Dynamic testing was conducted on a rooted Pixel 3a running Android 12 (chosen for compatibility considerations), with network traffic captured using Fiddler Everywhere [29] on Windows and HTTP Canary [13] on Android. HTTPS packet decryption was enabled by installing CA certificates and using Xposed [23].

Selection of LLM. For the P2-FirmRetr phase, we chose the open-source DeepSeek-V3 [8] due to its Function Calling capability, which DeepSeek-R1 [9] lacks. DeepSeek-R1 also exhibited longer inference times for complex tasks, reducing efficiency. While SOTA closed-source models like GPT-4o [20] and Claude 3.5 Sonnet [3] performed better, they were cost-prohibitive for large-scale use. Thus, DeepSeek-V3 provided the best balance between performance and cost.

B. *RQ1: Effectiveness of Firmware Leakage Detection*

Our study evaluated FIRMPROJ's performance on a large-scale dataset, dividing the process into two phases: P1-ReqRec and P2-FirmRetr. Across both phases, FIRMPROJ extracted 3,434 firmware files from 428 apps within the dataset of 10,047 IoT companion apps. These findings indicate that approximately 4.23% of the apps exhibited firmware leakage risks.

The distribution of firmware file types is shown in Figure 4(a). FIRMPROJ retrieved firmware files with 22 different extensions, among which `.bin`, `.zip`, and `.hex` were the most common. Figure 4(b) illustrates the number of leaked firmware files per app. Most apps (308, over 70%) leaked 1-5 files, while a small number leaked more than 50. Further analysis showed that these apps typically manage multiple device models. In particular, they either embed firmware for several devices within a single app or use a centralized intermediate file to distribute firmware across different models. Such designs account for the disproportionately high number of leaked firmware files.

Accuracy of Request Reconstruction. In the P1-ReqRec phase, FIRMPROJ uses static analysis to extract firmware update-related network requests. For 4,833 apps, FIRMPROJ

TABLE I: Accuracy of sample verification.

	TP	TN	FP	FN	Prec	Rec	F1	FPR	FNR
FIRMPROJ	47	48	3	2	94%	95.92%	94.95%	5.88%	4.08%

TABLE II: Time and tokens consumed by FIRMPROJ.

Token / Time	Single App second(s)	Single App token	Phase 1 second(s)	Phase 1 token	Phase 2 second(s)	Phase 2 token
Min.	1.7	0	0.9	-	0.8	0
Max.	4,680.0	609,431.0	3,142.0	-	4,226.0	609,431.0
Avg.	377.4	6,873.0	153.1	-	224.2	6,873.0

produced empty results, suggesting that these apps likely do not support DFU functionality. In addition, 250 apps failed during decompilation with dex2jar [21] (mainly due to packing protection) and were therefore excluded from the analysis. As a result, 4,964 apps produced valid results.

To evaluate P1-ReqRec results accuracy, we randomly sampled 100 reports from our experiments for manual verification, with 50 positive and 50 negative samples. A "*positive*" sample means that FIRMPROJ successfully identified and extracted firmware update request information, while a "*negative*" sample means no such information was detected. False positives arise when FIRMPROJ incorrectly classifies a non-firmware-related request as firmware-related, whereas false negatives occur when a genuine firmware update request is missed.

In manual verification, we decompiled each app into Java code using JADX and inspected the code to determine whether the app contained a firmware update module. We then compared these results with the modules identified by FIRMPROJ based on Jimple code. Table I presents our accuracy evaluation results. Manual verification identified that false positives (3 out of 50) were mainly caused by LLM misclassifications of irrelevant content as firmware-related. False negatives (2 out of 50) resulted from complex or obfuscated firmware update request structures that FIRMPROJ could not fully analyze.

Accuracy of Firmware Retrieval. In the P2-FirmRetr phase, FIRMPROJ retrieves firmware files based on the extracted requests. We manually inspected all 3,434 retrieved files, checking whether their names matched typical firmware naming patterns and contained firmware update-related keywords to determine potential false positives in the LLM results. In total, we identified 148 false positives (4.3%). Among these, 22 were `.zip` files, 85 were `.bin` files, five were `.dat` files, and 36 were `.hex` files. These false positives mainly consisted of SDK packages, voice resource packages, and app-related data. The primary causes were limited filename information or firmware-like names, which led to misclassification by the LLM-based identification process.

Runtime Performance. We next evaluate the runtime efficiency of FIRMPROJ. As shown in Table II, the maximum runtime for a single app was 4,680 seconds, the minimum was 1.7 seconds, and the average was 377.4 seconds (excluding preprocessing and firmware download). The longest runtime resulted from complex DFU logic and numerous network requests, while the shortest was for an app without a DFU module. Most of the runtime was spent in the P2-FirmRetr phase, averaging 224.2 seconds. P1-ReqRec time primarily depended on APK complexity and machine performance, whereas P2-FirmRetr was influenced by LLM inference speed and the number of requests requiring dynamic voting. A 60-minute timeout was set for P1-ReqRec, causing 30 apps to be terminated upon reaching this limit.

Token Usage and Cost. We also measured the token consumption and associated cost. The highest token usage for a single app was 609,431, the lowest was 0, and the average was 6,873. Apps with many network requests required the most tokens, while those without requests used none. Based on DeepSeek's pricing, processing 100 APKs costs an average of $0.649.

Answer to RQ1

Our study analyzed 10,047 IoT companion apps with FIRMPROJ and extracted 3,434 firmware files from 428 unique apps, showing that about 4.2% of apps leak firmware. The evaluation confirmed reliable detection accuracy. In terms of efficiency, FIRMPROJ required an average runtime of about 6 minutes per app, with the cost of analyzing 100 apps below $1. These results demonstrate that FIRMPROJ is accurate, efficient, and scalable for large-scale IoT companion app analysis.

C. RQ2: Direct Firmware Leakage Risks in DFU

We now discuss results categorized under *App-Bundled Firmware Leakage* and *Firmware Leakage without Controls* (see §II-B), both of which allow direct access to firmware.

Results for Scenario 1: App-Bundled Firmware Leakage. For this scenario, FIRMPROJ extracted 2,559 firmware files from 356 IoT companion apps in formats such as `bin`, `zip`, `img`, and `hex`, with an average file size of 291.72 KB. As shown in Figure 5, 97.3% of these files are under 1 MB, likely to avoid inflating app size and impacting user experience. On average, each app contained 7.19 firmware files, with one app (com.awox.smart.control) containing up to 241.

In addition to the primary dataset, we analyzed older IoT-VER app versions to compare firmware distribution across releases. From these, we extracted 849 firmware files from 123 apps. New firmware typically appeared in recent app releases, while older firmware was found in earlier versions, confirming that developers update firmware through app version upgrades.

For example, both the Kerialed (`com.awox.kerialed`) and Smart Control (`com.awox.smart.control`) apps from the same vendor contain over 100 LED-related firmware files, averaging 0.1 MB in size. Firmware filenames typically include device models and versions (e.g., `smart_ercu_4_0.bin`). During updates, the app uses regex matching to identify device models in filenames, compares firmware versions, and notifies users if a newer version is available. Storing firmware locally eliminates server-side hosting and reduces development overhead, but also greatly increases the risk of firmware leakage.

Results for Scenario 2: Firmware Leakage without Controls. Recall the workflow of FIRMPROJ: In the P2-FirmRetr phase,

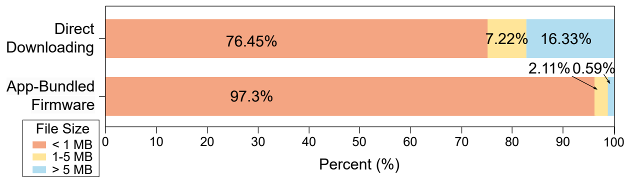


Fig. 5: Overview of Direct Firmware Leakage.

it formats and classifies extracted firmware update network requests, then attempts to acquire the firmware.

During the P1-ReqRec phase, FIRMPROJ analyzed 10,047 apps and identified firmware update-related network requests in 4,964 apps. In the P2-FirmRetr phase, it reconstructed complete requests from 801 apps and successfully downloaded 875 firmware files from 72 of them, averaging 12.15 files per app with a mean size of 5.67 MB.

The absence of server-side access control enabled bulk firmware downloads. The complete network requests included (a) direct URLs to specific firmware and (b) URLs for retrieving the latest firmware details. Direct URLs provided individual firmware files, while the latter often returned details for multiple devices, increasing the risk of widespread leakage.

For the latter type, for instance, in the LEDVANCE (`com.ledvance.smartplus`), firmware update requests were sent via four separate GET requests (using `Retrofit` [27]), without additional headers or parameters, indicating a lack of server-side access control. FIRMPROJ identified and extracted these requests, invoked external network APIs, and obtained four responses containing links to multiple firmware files. It parsed and downloaded 182 unique bin-format firmware files, with filenames specifying device model, firmware version, format, and release date (e.g., `AI9W-G 2.0.64.bin`) for device-specific matching. Attackers could exploit these exposed URLs to repeatedly download new firmware versions until proper access controls are implemented, leading to ongoing firmware leakage.

Answer to RQ2

FIRMPROJ identifies two key leakage scenarios: (1) *App-Bundled Firmware Leakage*, where 2,559 firmware files are directly embedded in 356 apps (most under 1 MB), suggesting that developers bundle firmware to limit app size; and (2) *Firmware Leakage without Controls*, where 72 apps allow FIRMPROJ to download 875 firmware files due to weak server-side controls, with some exposing bulk firmware access through unprotected URLs.

D. RQ3: Effectiveness on Incomplete Update Requests

In contrast to RQ2 (which dealt with complete network requests), we found that 3,772 apps produced incomplete requests missing dynamic elements such as `DeviceID`, `UserToken`, or encrypted parameters. Notably, some apps exhibited both request types. Due to the inherent limitations of static analysis, FIRMPROJ cannot automatically process these incomplete

Fig. 6: Devices used in the dynamic analysis evaluation.

requests. To evaluate FIRMPROJ's usefulness for firmware leakage investigations, we manually performed dynamic analysis on ten sampled apps, seven of which had devices available for purchase (see Figure 6). The devices were used solely to identify the format of dynamic parameters, not to obtain firmware. Our aim was to exploit vulnerabilities for bulk firmware access without needing the devices themselves.

We installed the ten apps on a rooted Pixel 3a and used HTTP Canary to capture network packets. For each app, we registered users, logged in, and bound devices to obtain the device and user information for firmware updates. `DeviceID` was typically collected during device binding, while `UserToken` was captured during login. For encrypted fields, FIRMPROJ used P2-FirmRetr results to retrieve encryption logic and keys, enabling manual encryption and decryption of required data.

After populating all dynamic field values in FIRMPROJ's firmware update requests, we used Fiddler Everywhere [29] to send the complete requests and process the server responses. Four main outcomes were observed: (1) can access various firmware without devices; (2) can access limited firmware related to the devices; (3) no firmware updates were released; and (4) cannot access the firmware without binding the devices.

- ***Unrestricted firmware access via predictable identifiers.*** In some cases, firmware update requests rely on dynamic fields (e.g., `DeviceID`, `UserToken`) that are simple sequential values. This simplicity allowed us to infer other `DeviceID`s, granting firmware access across multiple device types.
- ***Firmware access limited by device-specific restrictions.*** Some manufacturers use complex, unpredictable `DeviceID`s (e.g., `IC264C52UV2B524563Z7ZCJB`) to prevent enumeration attacks, restricting firmware access to specific devices or models.
- ***Firmware update discontinuation.*** For some devices (e.g., Qingniu Body Fat Scale), although they support firmware updates, vendors no longer provide updates, so update checks do not return newer firmware versions.
- ***Firmware access restricted by device binding.*** Some manufacturers require user-device binding verification during update requests, rejecting attempts from unbound devices. We validated this by submitting identical requests for both bound and unbound devices.

Table III summarizes the experimental results. Two apps (No.2 and No.4) were vulnerable to unrestricted firmware access due to predictable identifiers; for example, DreamCatcher used sequential `DeviceID`s, enabling attackers to enumerate and access all associated firmware. In contrast, only two apps (No.1 and No.9), which used the Danale [7] and Tuya [30] SDKs, enforced access control via device-user bindings. Seamooncloud (No.8) did not expose updated firmware during testing, likely due to no new releases, but its device binding relied on a simple sequential SN code, allowing any unbound device to be bound by iterating through SN codes. These results highlight the need for robust access controls and secure device binding to prevent firmware leaks and unauthorized access. In §IV-E, we present two case studies for an in-depth examination of firmware leakage.

Answer to RQ3

While FIRMPROJ cannot automatically process requests with missing dynamic fields, its effectiveness improves significantly when paired with expert analysis. Testing on ten apps showed that this combined approach reliably detects firmware access vulnerabilities, highlighting the value of FIRMPROJ's reports in supporting dynamic analysis.

E. Vulnerability Discovery and Case Studies

Following the Responsible Disclosure policy, we reported the discovered vulnerabilities to the corresponding IoT vendors or to the CVE Program. Since most vendors did not provide official security reporting channels, we submitted the reports by email. In total, only five responses were received: four were automated acknowledgments, and only Yamaha manually confirmed receipt and forwarded the report to its security team for further review. Given the limited vendor feedback, we escalated all reports directly to MITRE (CVE Program).

MITRE recognized firmware leakage as a significant security threat and assigned 35 CVE IDs across 32 vendors. Of these, 9 were rated critical, 22 high, and 4 medium severity. In terms of leakage scenarios, 27 CVEs correspond to Scenario 2 (Firmware Leakage without Controls), and 8 correspond to Scenario 3 (Firmware Leakage via Access Control Flaws). These vulnerabilities are mainly linked to weaknesses in access control and information disclosure, including (1) CWE-863: Incorrect Authorization, (2) CWE-306: Missing Authentication for Critical Function, and (3) CWE-200: Exposure of Sensitive Information to an Unauthorized Actor.

To further illustrate these findings, we next present two case studies that highlight representative firmware leakage vulnerabilities identified by FIRMPROJ.

Case 1: Neye3c, `CVE-2024-48539`, critical severity. Neye3c app (`com.gooclient.anycam.neye3ctwo`) uses RC4 encryption for network requests, implemented directly in the app code rather than external libraries, securing parameters in update requests and server responses. However, the encryption key is hard-coded in the function parameters, creating a critical vulnerability. This allowed FIRMPROJ to identify the encryption method and key using VSA and LLM assistance.

TABLE III: Dynamic analysis results by apps.

No.	Package Name	Version	Device Type	Result	Firmware #
1	`com.danale.video`	5.9.36	No device*	×	0
2	`com.dc.dreamcatcherlife`	1.8.3	No device*	✓	25
3	`com.elinkthings.ailink.kingbeifit`	1.2.0	Body Fat Scale	○	0
4	`com.gooclient.anycam.neye3ctwo`	4.5.2.0	Wi-Fi Camera	✓	2
5	`com.hle.china.smarthome.xiaohe`†	4.3.1	Light Strip	○	1
6	`com.ivyiot.IvySmart`	4.5.0	Wi-Fi Camera	●	1
7	`com.qingniu.plus`	3.18.3	Body Fat Scale	○	0
8	`com.seamooncloud.cloudsmartlock`	2.0.1	Smart Lock	○	0
9	`com.tcl.smarthome`	5.9.36	No device*	×	0
10	`com.yingsheng.nadai`	1.2.0	Smart Watch	●	1

✓ Unrestricted firmware access via predictable identifiers. × Cannot access the firmware without binding the devices. ○ Firmware update discontinuation. ● Firmware access restricted by device binding. ∗ Device information was obtained through alternative methods, such as online searches. † No firmware update available; however, the flashing toolchain was accessible.

Additionally, this app uses `OkHttp` POST for firmware update requests. During analysis, FIRMPROJ identified the `getNewVersion` method (see Listing 3), but static analysis could not resolve its input parameters, which were set dynamically via a handler call. FIRMPROJ marked these parameters as dynamic and used Forward Rebuilding to reconstruct the request. In this stage, it encountered the encryption function, preserving the complete function signature, call chain, and parameter values. FIRMPROJ extracted relevant code slices and saved key parameters (such as the hard-coded key) in a `JSON` file for further analysis. In the P2-FirmRetr phase, unresolved function signatures triggered a lookup in this `JSON` file, which contained encryption algorithm details and parameter value sets, allowing the LLM to thoroughly analyze the function.

```
class UpdateFirmwareActivity{
  public void getNewVersion(String str) {
    String[] parseJSON = MyHttp.parseJSON(
      str, new String[]{"AppCom", "SolCom",
      "ReleaseTime", "HDVersion"});
    parseJSON[0] = MyHttp.encryption(
      parseJSON[0], "KEY");
    parseJSON[1] = MyHttp.encryption(
      parseJSON[1], "KEY");
    parseJSON[2] = MyHttp.encryption(
      parseJSON[2], "KEY");
    parseJSON[3] = MyHttp.encryption(
      parseJSON[3], "KEY");
    MyHttp.okHttpPost(String.format(
      "{\"AppCom\":\"%s\",\"SolCom\":\"%s\",
      \"ReleaseTime\":\"%s\",\"HDVersion\":
      \"%s\"}", parseJSON[0], parseJSON[1],
      parseJSON[2], parseJSON[3]), "https://
      app.login.yunyis.com/g_version_match.
      php", new Callback());}}
class MyHttp {
  public static String encryption(String str
  , String str2) {  ......
    return new String(Base64.encode(RC4Test.
    RC4  (str.getBytes(), str2)));}}
```

Listing 3: Code snippet of Neye3c.

Further manual verification confirmed that Neye3c's firmware update requests use RC4 encryption with a hard-coded key, as indicated by LLM outputs. By intercepting and modifying encrypted parameters like `HDVersion` and `ReleaseTime`, then re-encrypting them, we bypassed server-side checks and accessed both the latest and current firmware versions. Additionally, the absence of device-user binding allowed repeated unauthorized downloads of new firmware by simply providing a device model and any valid user information, significantly increasing the risk of firmware leakage.

Case 2: DreamCatcher, `CVE-2024-48547`, high severity. Requests from DreamCatcher app (`com.dc.dreamcatcherlife`) required dynamic fields. Following FIRMPROJ's guidance, we purchased the relevant devices to capture `DeviceID` and `chipname`, and used our own valid `UserToken` to obtain firmware information. Since `DeviceID` was a five-digit number, we used Burp Suite to brute-force possible combinations, successfully downloading firmware for other devices. FIRMPROJ's output (see Listing 4) also revealed an alternative request method using a single-digit `pid` and the `chipname`. FIRMPROJ identified a potential `chipname` value as "wifi" and a corresponding `pid` value as "0". Using these values along with a valid `UserToken`, we successfully obtained the latest firmware version for this specific `pid`. This vulnerability exposes a significant security risk, enabling attackers to systematically steal firmware from vendor devices.

```
public final void sendGetLatestFwInfo(String
    str,int i,String str2,int i2,String str3
    ,final AmCallBack amCallBack) {
  FuelKt.httpGet(
    AM_user_latest_fwinfo_URL( str, Integer.
    valueOf(i) ), CollectionsKt.listOf( (
    Object[]) new Tuples[]{ TuplesKt.m137to(
    "token",str2), TuplesKt.m137to( "pid",
    Integer.valueOf(i2) ),TuplesKt.m137to(
    "chipname",str3 )})
).header(getHeaderData(str2)).responseString
(new CALLBACK FUNCTION);}
```

Listing 4: Code snippet of DreamCatcher.

V. DISCUSSION AND LIMITATIONS

This section discusses possible mitigations and the limitations of this study.

Mitigation. To effectively address firmware leakage in IoT ecosystems, mitigation strategies should involve both developers and vendors. Developers should avoid embedding firmware files in companion apps. Instead, firmware should be securely downloaded through protected channels such as HTTPS. If embedding is unavoidable, the firmware files must be encrypted, weak cryptographic algorithms should be avoided, and secure key management should be applied rather than using hard-coded keys. Vendors should implement strict access controls on cloud servers, including dynamic authentication tokens with short lifespans, and enforce device–user binding to ensure that only authorized users and devices can access firmware. Cloud servers should distribute firmware based on the user's device model rather than providing all device firmware through a single generic configuration file. Device- or user-specific credentials (e.g., `DeviceID` and `UserToken`) should be sufficiently complex and random to prevent enumeration attacks.

Limitations of Static Analysis. During the P1-ReqRec phase, FIRMPROJ relies on static analysis, which can occasionally produce overestimated or inaccurate value propagation when reconstructing target fields. For instance, if multiple values intended for different uses share the same variable name in the intermediate representation, unrelated values may be merged into the same set. In the forward rebuilding step, parameters that reference arrays can also lead to incorrect assignments of all associated values, compounding the inaccuracy. Furthermore, many companion apps implement protective measures such as code packing and extensive obfuscation, limiting FIRMPROJ's capability to detect firmware update mechanisms. Future improvements could include partial dynamic analysis or hybrid methods that reduce the reliance on static analysis alone.

Limitations of LLMs. In the P2-FirmRetr phase, the choice of LLM significantly impacts the consistency of FIRMPROJ's results. While DeepSeek-V3 meets basic performance needs, more advanced models could reduce errors in complex or edge cases. We apply prompt engineering techniques, such as few-shot examples and optimized prompts, to improve output stability and accuracy. However, limitations inherent to the model persist. Future work may incorporate Retrieval-Augmented Generation (RAG) or fine-tuning to further improve the accuracy and reduce dependence on handcrafted prompts, enabling FIRMPROJ to better handle a broader range of companion apps and firmware update mechanisms.

VI. RELATED WORK

Here, we review previous studies on IoT companion apps and firmware vulnerabilities.

IoT Companion Apps. Previous research [19], [16], [5], [14], [15], [6], [17], [35], [44], [10], [25], [36], [24] has primarily investigated security risks within the IoT ecosystem from the perspective of companion apps. Among these, the most relevant work is by Ibrahim et al. [14], who examined DFU security by identifying flawed SDKs used by the companion apps and focusing on security issues after the firmware is transferred to the device. In contrast, our study focuses on firmware leakage risks stemming from how firmware is acquired and stored within the companion app itself. Additionally, rather than relying on SDK analysis, our approach directly analyzes the firmware update process across different apps. Although FirmXRay [36] also extracts firmware from companion apps to detect Bluetooth firmware vulnerabilities, it only retrieves firmware embedded in the APK and does not analyze the firmware update process.

Jin et al. [15] introduced IoTSpotter, a framework that analyzed 37,783 mobile IoT apps and uncovered widespread cryptographic and library vulnerabilities. Chen et al. [6] proposed IoTFuzzer, an automated fuzzing framework that detects memory corruption vulnerabilities in IoT devices via companion apps without requiring firmware access. Mallojula et al. [16] examined 125 Android automotive companion apps, revealing that 70% have vulnerabilities that could leak private data or pose driving risks. Nan et al. [17], using IoTProfiler, analyzed 6,208 apps and found that 1,973 expose user data without proper disclosure. Wang et al. [35] identified 324 potentially vulnerable smart home devices across 73 vendors by analyzing their apps. Zuo et al. [44] found a BLE protocol flaw enabling device fingerprinting and unauthorized access, identifying 1,757 vulnerable apps.

Firmware Vulnerabilities. Recent studies proposed techniques for detecting firmware vulnerabilities in IoT devices. Wu et al. [37] developed ChkUp, which detected vulnerabilities in the updates of 12,000 firmware images, leading to 25 CVE disclosures. Zhao et al. [41] designed FirmSec, analyzing 34,136 images to uncover 584 components and 128,757 vulnerabilities. Xiao et al. [38] presented FirmRec, a recurring vulnerability detection approach for IoT firmware, efficiently identifying 642 vulnerabilities and 53 CVEs in 320 images. Zheng et al. [42] proposed EQUAFL, which automatically constructs an execution environment for embedded applications. Angelakopoulos et al. [2] introduced FirmSolo, adding kernel space analysis and uncovering 19 vulnerabilities in 1,470 images. Nino et al. [18] built a dataset of MCU firmware and assessed its security via static analysis. Gibbs et al. [12] developed MangoDFA, which found 56 vulnerabilities in seven firmware sets using dependency analysis. Wang et al. [34] and Zhang et al. [40] studied cryptographic misuse in IoT firmware.

VII. CONCLUSION

Insecure firmware update processes in IoT companion apps pose critical risks, enabling attackers to extract firmware and access sensitive system information. This study introduced FIRMPROJ, an automated tool for detecting firmware leakage vulnerabilities in the DFU process of IoT apps. Applying FIRMPROJ to 10,047 apps, we extracted 3,434 firmware files from 428 unique apps, including products from major manufacturers such as Yamaha and Panasonic. Our responsible disclosure led to 35 CVE IDs being assigned, highlighting the urgency of strengthening firmware update mechanisms to better protect IoT ecosystem.

ACKNOWLEDGEMENTS

Thanks to anonymous reviewers and shepherd. This work was partially supported by Shandong Provincial Natural Science Foundation (Grant No. ZR2023MF043 and ZR2025MS991).

References

[1] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR), Austin, TX, USA, May 14-22, 2016*, 2016.

[2] Ioannis Angelakopoulos, Gianluca Stringhini, and Manuel Egele. FirmSolo: Enabling dynamic analysis of binary Linux-based IoT kernel modules. In *Proceedings of the 32nd USENIX Security Symposium (USENIX-SEC), Anaheim, CA, USA, August 9-11, 2023*, 2023.

[3] Anthropic. Claude 3.5 Sonnet, 2024. URL: https://www.anthropic.com.

[4] APKPure. APKPure, 2024. URL: https://apkpure.com.

[5] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS) , San Diego, California, USA, February 21-24, 2016*, 2016.

[6] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 18-21, 2018*, 2018.

[7] Danale. Danale Introduce, 2023. URL: https://cn.danale.com/introduce.html.

[8] DeepSeek-AI. DeepSeek-V3 Technical Report, 2024. URL: https://arxiv.org/abs/2412.19437, arXiv:2412.19437.

[9] DeepSeek-AI. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning, 2025. URL: https://arxiv.org/abs/2501.12948, arXiv:2501.12948.

[10] Jianqi Du, Fenghao Xu, Chennan Zhang, Zidong Zhang, Xiaoyin Liu, Pengcheng Ren, Wenrui Diao, Shanqing Guo, and Kehuan Zhang. Identifying the BLE Misconfigurations of IoT Devices through Companion Mobile Apps. In *Proceedings of the 19th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON), Stockholm, Sweden, September 20-23, 2022*, 2022.

[11] Zicong Gao, Chao Zhang, Hangtian Liu, Wenhou Sun, Zhizhuo Tang, Liehui Jiang, Jianjun Chen, and Yong Xie. Faster and Better: Detecting Vulnerabilities in Linux-based IoT Firmware with Optimized Reaching Definition Analysis. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 26 - March 1, 2024*, 2024.

[12] Wil Gibbs, Arvind S. Raj, Jayakrishna Menon Vadayath, Hui Jun Tay, Justin Miller, Akshay Ajayan, Zion Leonahenahe Basque, Audrey Dutcher, Fangzhou Dong, Xavier J. Maso, Giovanni Vigna, Christopher Kruegel, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. Operation Mango: Scalable Discovery of Taint-Style Vulnerabilities in Binary Firmware Services. In *Proceedings of the 33rd USENIX Security Symposium (USENIX-SEC), Philadelphia, PA, USA, August 14-16, 2024*, 2024.

[13] GuoShi. HTTP Canary. URL: https://apkpure.net/httpcanary-%E2%80%94-http-sniffer-capture-analysis/com.guoshi.httpcanary.

[14] Muhammad Ibrahim, Andrea Continella, and Antonio Bianchi. AoT - Attack on Things: A security analysis of IoT firmware updates. In *Proceedings of the 8th IEEE European Symposium on Security and Privacy (EuroS&P), Delft, Netherlands, July 3-7, 2023*, 2023.

[15] Xin Jin, Sunil Manandhar, Kaushal Kafle, Zhiqiang Lin, and Adwait Nadkarni. Understanding IoT Security from a Market-Scale Perspective. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS), Los Angeles, CA, USA, November 7-11, 2022*, 2022.

[16] Prashanthi Mallojula, Fengjun Li, Xiaojiang Du, and Bo Luo. Companion Apps or Backdoors? On the Security of Automotive Companion Apps. In *Proceedings of the 29th European Symposium on Research in Computer Security (ESORICS), Bydgoszcz, Poland, September 16-20, 2024*, 2024.

[17] Yuhong Nan, Xueqiang Wang, Luyi Xing, Xiaojing Liao, Ruoyu Wu, Jianliang Wu, Yifan Zhang, and XiaoFeng Wang. Are You Spying on Me? Large-Scale Analysis on IoT Data Exposure through Companion Apps. In *Proceedings of the 32nd USENIX Security Symposium (USENIX-SEC), Anaheim, CA, USA, August 9-11, 2023*, 2023.

[18] Nicolas Nino, Ruibo Lu, Wei Zhou, Kyu Hyung Lee, Ziming Zhao, and Le Guan. Unveiling IoT Security in Reality: A Firmware-Centric Journey. In *Proceedings of the 33rd USENIX Security Symposium (USENIX-SEC), Philadelphia, PA, USA, August 14-16, 2024*, 2024.

[19] T. J. OConnor, Dylan Jessee, and Daniel Campos. Through the Spyglass: Towards IoT Companion App Man-in-the-Middle Attacks. In *Proceedings of the 14th Cyber Security Experimentation and Test Workshop (CSET), Virtual, CA, USA, August 9, 2021*, 2021.

[20] OpenAI. GPT-4o, 2024. URL: https://openai.com/index/hello-gpt-4o/.

[21] Bob Pan. Dex2jar, 2023. URL: https://github.com/pxb1988/dex2jar.

[22] PC6. PC6, 2024. URL: https://www.pc6.com/.

[23] rovo89. Xposed, 2023. URL: https://github.com/rovo89/Xposed.

[24] David Schmidt, Carlotta Tagliaro, Kevin Borgolte, and Martina Lindorfer. IoTFlow: Inferring IoT Device Behavior at Scale through Static Mobile Companion App Analysis. In *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security (CCS), Copenhagen, Denmark, November 26-30, 2023*, 2023.

[25] Pallavi Sivakumaran, Chaoshun Zuo, Zhiqiang Lin, and Jorge Blasco. Uncovering Vulnerabilities of Bluetooth Low Energy IoT from Companion Mobile Apps with Ble-Guuide. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security (ASIACCS), Melbourne, VIC, Australia, July 10-14, 2023*, 2023.

[26] skylot. JADX, 2023. URL: https://github.com/skylot/jadx.

[27] Square. Retrofit: A type-safe HTTP client for Android and Java, 2020. URL: https://square.github.io/retrofit/.

[28] SwitchBot. SwitchBot - Smart Home in One Tap, 2024. URL: https://www.switch-bot.com.

[29] Telerik. Fiddler Everywhere, 2023. URL: https://www.telerik.com/fiddler-everywhere.

[30] Tuya. App SDK, 2023. URL: https://www.tuya.com/platform/appdev/app-sdk.

[31] L. Vailshery. Number of Internet of Things (IoT) connections worldwide from 2022 to 2023, with forecasts from 2024 to 2033, 2024. URL: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/.

[32] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In *Proceedings of the 9th International Conference, Held as Part of the European Joint Conferences on the Theory and Practice of Software (ETAPS), Berlin, Germany, March 25 - April 2, 2000*, 2000.

[33] Wandoujia. Wandoujia, 2024. URL: https://www.wandoujia.com/.

[34] Jianing Wang, Shanqing Guo, Wenrui Diao, Yue Liu, Haixin Duan, Yichen Liu, and Zhenkai Liang. CrypTody: Cryptographic Misuse Analysis of IoT Firmware via Data-flow Reasoning. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID), Padua, Italy, 30 September 2024 - 2 October 2024*, 2024.

[35] Xueqiang Wang, Yuqiong Sun, Susanta Nanda, and XiaoFeng Wang. Looking from the Mirror: Evaluating IoT Device Security through Mobile Companion Apps. In *Proceedings of the 28th USENIX Security Symposium (USENIX-SEC), Santa Clara, CA, USA, August 14-16, 2019*, 2019.

[36] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities From Bare-Metal Firmware. In *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS), Virtual Event, USA, November 9-13, 2020*, 2020.

[37] Yuhao Wu, Jinwen Wang, Yujie Wang, Shixuan Zhai, Zihan Li, Yi He, Kun Sun, Qi Li, and Ning Zhang. Your Firmware Has Arrived: A Study of Firmware Update Vulnerabilities. In *Proceedings of the 33rd USENIX Security Symposium (USENIX-SEC), Philadelphia, PA, USA, August 14-16, 2024*, 2024.

[38] Haoyu Xiao, Yuan Zhang, Minghang Shen, Chaoyang Lin, Can Zhang, Shengli Liu, and Min Yang. Accurate and Efficient Recurring Vulnerability Detection for IoT Firmware. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS), Salt Lake City, UT, USA, October 14-18, 2024*, 2024.

[39] Yuting Xiao, Jiongyi Chen, Yupeng Hu, and Jing Huang. FIRMRES: Exposing Broken Device-Cloud Access Control in IoT Through Static Firmware Analysis. In *Proceedings of the 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Brisbane, Australia, June 24-27, 2024*, 2024.

[40] Li Zhang, Jiongyi Chen, Wenrui Diao, Shanqing Guo, Jian Weng, and Kehuan Zhang. CryptoREX: Large-scale Analysis of Cryptographic Misuse in IoT Devices. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID), Beijing, China, September 23-25, 2019*, 2019.

[41] Binbin Zhao, Shouling Ji, Jiacheng Xu, Yuan Tian, Qiuyang Wei, Qinying Wang, Chenyang Lyu, Xuhong Zhang, Changting Lin, Jingzheng Wu, and Raheem Beyah. A Large-Scale Empirical Analysis of the Vulnerabilities Introduced by Third-Party Components in IoT Firmware. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Virtual Event, South Korea, July 18 - 22, 2022*, 2022.

[42] Yaowen Zheng, Yuekang Li, Cen Zhang, Hongsong Zhu, Yang Liu, and Limin Sun. Efficient Greybox Fuzzing of Applications in Linux-Based IoT Devices via Enhanced User-Mode Emulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Virtual Event, South Korea, July 18 - 22, 2022*, 2022.

[43] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. Why Does Your Data Leak? Uncovering the Data Leakage in Cloud from Mobile Apps. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, May 19-23, 2019*, 2019.

[44] Chaoshun Zuo, Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. Automatic Fingerprinting of Vulnerable BLE IoT Devices with Static UUIDs from Mobile Apps. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS), London, UK, November 11-15, 2019*, 2019.