Unveiling IoT Security in Reality: A Firmware-Centric Journey

Nicolas Nino, *School of Computing, University of Georgia;* Ruibo Lu and Wei Zhou, *School of Cyber Science and Engineering, Huazhong University of Science and Technology;* Kyu Hyung Lee, *School of Computing, University of Georgia;* Ziming Zhao, *Khoury College of Computer Sciences, Northeastern University;* Le Guan, *School of Computing, University of Georgia*

https://www.usenix.org/conference/usenixsecurity24/presentation/nino

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

Unveiling IoT Security in Reality: A Firmware-Centric Journey

Nicolas Nino*
School of Computing
University of Georgia

Ruibo Lu*, Wei Zhou
School of Cyber Science and Engineering
Huazhong University of Science and Technology

Kyu Hyung Lee
School of Computing
University of Georgia

Ziming Zhao
Khoury College of Computer Sciences
Northeastern University

Le Guan
School of Computing
University of Georgia

Abstract

To study the security properties of the Internet of Things (IoT), firmware analysis is crucial. In the past, many works have been focused on analyzing Linux-based firmware. Less known is the security landscape of MCU-based IoT devices, an essential portion of the IoT ecosystem. Existing works on MCU firmware analysis either leverage the companion mobile apps to infer the security properties of the firmware (thus unable to collect low-level properties) or rely on small-scale firmware datasets collected in ad-hoc ways (thus cannot be generalized). To fill this gap, we create a large dataset of MCU firmware for real IoT devices. Our approach statically analyzes how MCU firmware is distributed and then captures the firmware. To reliably recognize the firmware, we develop a firmware signature database, which can match the footprints left in the firmware compilation and packing process. In total, we obtained 8,432 confirmed firmware images (3,692 unique) covering at least 11 chip vendors across 7 known architectures and 2 proprietary architectures. We also conducted a series of static analyses to assess the security properties of this dataset. The result reveals three disconcerting facts: 1) the lack of firmware protection, 2) the existence of N-day vulnerabilities, and 3) the rare adoption of security mitigation.

1 Introduction

The *Internet of Things* (IoT) is an ecosystem of interconnected devices. We are witnessing an increasing number of IoT devices being deployed in various domains, such as smart homes, smart cities, and industrial automation. While IoT makes our lives more convenient, it has also introduced new security vulnerabilities. Exploiting these vulnerabilities allows adversaries to launch large-scale attacks, including data exfiltration [21, 28], remote vehicle hijacking [41], illegal break-in [82], and even disabling medial treatment [58].

Firmware analysis can play a crucial step in understanding the security properties of IoT devices. Previous work has shown promising results in large-scale analysis of firmware, both statically [18, 81] and dynamically [13]. Analysis performed at that scale (e.g., thousands of firmware images) helps us understand the commonalities of security vulnerabilities across different devices and thus suggest effective mitigations. A prerequisite for conducting such research, however, is a huge collection of firmware images. Prior work addresses this challenge by developing vendor-specific crawlers to dump firmware images from top vendors' websites. The resulting datasets mainly comprise firmware for home routers, IP cameras, etc., which are all Linux-based.

Besides Linux-based embedded devices, our IoT space also encompasses a wide range of even smaller gadgets that are based on microcontrollers (MCUs). Due to hardware limitations, these devices do not run a full-fledged OS, but they easily outnumber their Linux-based counterparts. With the small form factor and built-in low-energy wireless communication capability, MCU-based devices have been widely adopted in IoT, such as smart homes, smart cities, and industrial automation. Since these devices do not have direct access to the Internet, they commonly rely on mobile companion apps to communicate with the user and the cloud.

Compared with research on Linux-based firmware analysis, less has been done to MCU-based devices, mainly due to the lack of a large dataset of real-world firmware images. Indeed, we rarely find the public release of firmware images for these devices [14]. To overcome this challenge, most existing works utilize the mobile companion apps to infer the security properties of the our IoT ecosystem, including the device, the remote cloud, and the mobile app itself [31, 32, 63, 78, 86]. For example, by analyzing the library artifacts in mobile companion apps, vulnerabilities in the device firmware can be inferred [78]. By reusing the logic of the mobile app, IoTFuzzer can fuzz the device more effectively [14]. Unfortunately, without access to the firmware, it is difficult, if not impossible, to reveal the low-level security features, such as the adoption of attack mitigations. If a bug is found, it is also hard to tell the root cause.

In this paper, we create a large dataset of firmware for

*Both authors contributed equally to this work.

MCU-based IoT products. This will benefit the IoT community in three aspects. First, this dataset will enrich existing firmware corpus [75], which mainly consists of homemade samples compiled from chip SDKs or only represents IoT devices with the BLE feature [79]. The proposed dataset broadly covers firmware of diverse formats, architectures, chips and application fields, enabling us to uncover common security vulnerabilities of real-world IoT devices and enhance manufacturers' awareness. Second, the dataset can unleash the full power of existing firmware analysis tools. For example, when IoTFuzzer finds a bug, if the firmware is available, we can possibly explain the root cause via binary analysis. Third, it will facilitate future research. For example, it can be used to evaluate the effectiveness of existing firmware testing techniques such as rehosting [62]. When these tools fail, investigation can be done to improve the state of the art. As another example, when common vulnerabilities are found in the dataset, it will push research on defense techniques.

To collect real-world firmware, we leverage the fact that IoT devices commonly rely on a companion mobile app to retrieve the latest firmware. Depending on whether the latest firmware is bundled with the APK release or downloaded from the cloud via over-the-air (OTA) update, we develop two tools to capture firmware. First, we retrofit FirmXray which extracts bundled firmware from APKs. Our tool additionally utilizes a list of firmware-related keywords and extensions, compiled based on our empirical study, to screen a more diverse set of candidate firmware. Second, we develop a static analysis tool called *OTACap* to automatically recover the URLs that point to the firmware update server and use them to crawl potential firmware. *OTACap* relies on Value Set Analysis (VSA) [11] to track down the source values used to construct URLs. However, traditional designs may miss many data flows due to the lack of run-time information and implicit data flow in Android apps. In particular, the URL may contain a substring that is dynamically requested from the device. Observing that although the needed information may not be directly on the data flow of the URL, it is sometimes referred to at other places, e.g., for version comparison. By collecting and solving the constraints of these references, we can infer the possible value range of the missing information. The reconstructed URLs do not always directly point to the firmware. Sometime, they can return an intermediate text file that contains the actual firmware distribution point. Since the text format is unpredictable and sometimes contains credentials associated with the links, we empower our crawler with large language models (LLM) to automatically reason about the text structure and extract firmware links (and associated credentials if any).

The candidate firmware images have a high false positive rate, making them unsuitable for reliable analysis. To confirm a firmware image, we look for the footprints left in the image during firmware compilation and packing. Particularly, we find that the chip architecture, vendor toolchain, and firmware bootloader/packer offer distinctive artifacts, making them excellent choices for firmware signatures. Thus, we manually identify 39 signatures to match 4 categories of firmware artifacts. Applying them to the candidate images, we obtained 8,432 confirmed firmware images, out of which 3,692 are unique. Our dataset contains firmware from at least 11 chip vendors across 7 known architectures and 2 proprietary architectures, offering a diverse firmware dataset.

Our analysis of this dataset reveals several disconcerting findings. First, 99.43% of these images are in plaintext, allowing for reverse engineering. Second, by library matching, we found 14 N-day vulnerabilities that affect 191 firmware images. Lastly, the adoption of security mitigations is rare. For example, the adoption rates of memory protection unit (MPU) and stack canary are only 1.56% and 0.11%, respectively.

In summary, our contributions are as follows:

- We developed *OTACap*, the first static analysis tool to recover URLs used in firmware update from Android apps.
- To fully exploit the recovered URLs, we developed an intelligent crawler to download potential MCU firmware.
- Leveraging the footprints left in the firmware compilation and packing process, we identified 39 signatures to reliably recognize real MCU firmware.
- With the developed tools, we collected 3,692 unique MCU firmware images, representing a diverse dataset of real-world firmware, facilitating future research.
- Our analysis of the dataset reveals several disconcerting findings, suggesting future remediation efforts.

2 Background

MCU-based IoT Device. A vital component of IoT is billions of small interconnected embedded devices that are powered by MCU chips. An MCU integrates a processor, RAM, and other peripheral functions into a single chip, aiming to reduce cost, form factor, and power consumption. The peripherals are memory-mapped into the system address space, and their registers are accessed via normal memory access instructions (i.e., memory-mapped I/O or MMIO). The software that runs on MCUs, named firmware, either runs on the bare-metal or with a real-time operating system (RTOS). To better utilize the limited resources (100+ MHz CPU, ~256 KB SRAM, ~1 MB flash), MCU firmware is typically programmed using the unsafe C/C++ programming languages. Over the past years, we have witnessed numerous vulnerabilities targeting the firmware of IoT devices [30, 33, 34, 44, 65].

An MCU chip typically implements a RISC instruction set architecture (ISA), with Arm being the dominating ISA. The device manufacturers (e.g., Amazon), after finding a suitable MCU chip to implement the designed functionality, assemble parts together, including the PCB, sensors, actuators, and most importantly, the MCU chip. In this process, the ISA vendor is at the top of the supply chain, followed by the chip vendor

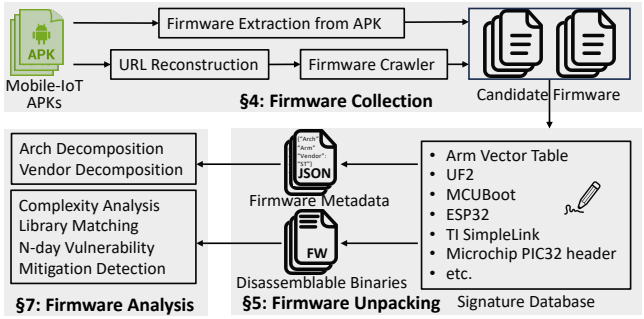

Figure 1: System Overview.

(e.g., Nordic), who fabricates chips that implement an ISA, and then the device manufacturer. While some chip vendors develop their own ISAs (e.g., PIC32 by Microchip), most others, such as NXP and Nordic obtain a license to use a certain ISA. It is worth noting that a chip vendor may offer chips across various series, each employing a different ISA.

Many MCU-based devices run low-energy wireless protocols such as Bluetooth. As such, they rely on companion mobile apps to interact with the user and the cloud.

Over-the-Air Update. To fix software bugs or add new features, many IoT devices support firmware updates. The new firmware can be distributed to the device via two methods. First, the companion app periodically checks for updates from the remote server. If an update is available, the companion app fetches the latest firmware and then pushes it to the device via the low-energy protocol. This process is commonly referred to as OTA update. Second, the firmware can be bundled with the companion app. When the app is updated, the device can also be updated. Different from traditional Linux-based embedded devices such as home routers, the firmware of MCU-based IoT devices is rarely released via vendor websites.

3 Overview

An overview of our system is illustrated in Figure 1. We first design and develop tools to automatically collect firmware images from mobile companion apps (§4). Besides extracting firmware images from APKs, following the method originally proposed in FirmXray [79] (§4.1), we develop an Android static analysis tool (named *OTACap*) to reconstruct URLs that are used to download the firmware via OTA update (§4.2). *OTACap* enhances state-of-the-art string value analysis techniques [63, 86] by incorporating more accurate and effective data flow tracking, such as support for inferring hardware-related fields. Utilizing the reconstructed URLs, our intelligent crawler simulates the OTA update process of real-world mobile-IoT applications to gather firmware images. Throughout this process, we successfully extracted 43,231 candidate firmware images from 40,675 Mobile-IoT APKs.

For the next step, we analyze the collected image files to identify and verify MCU firmware. To facilitate this, we developed a signature-based firmware unpacker. It processes images to detect firmware artifacts present within firmware images (e.g., magic numbers in firmware headers or the architectural metadata essential for firmware execution). This approach not only assists in confirming real MCU firmware but also retrieves metadata for further analysis (§5). In particular, we leverage the collected metadata to decompose the IoT market by major architecture vendors and chip vendors (§7.1). The firmware metadata, which includes information about the instruction set architecture (ISA), load address and entry point, is also crucial in disassembling the firmware and performing binary analysis. This analysis reveals low-level security properties. Specifically, we locate all the recognizable functions, match them with existing IoT libraries, and report whether they contain any known vulnerabilities. Besides, we examine the adoption of security mechanisms such as stack canary in the wild (§7.2).

4 Firmware Collection

Considering the common practice of distributing firmware through mobile apps by device manufacturers, we develop two methods for collecting firmware images: 1) extracting firmware images directly from APKs and 2) reconstructing URLs through APK analysis to collect firmware images during the OTA update process. This step is designed to maximize the collection of firmware images while simultaneously filtering out non-firmware images. We conduct additional verification of the firmware images using various artifacts, as detailed in §5.

4.1 Extracting Firmware from APKs

It has been observed that some IoT device manufacturers bundle firmware images in their companion apps [2, 52]. When the app is updated, the device firmware can also be updated with the latest version that comes with the app. For example, MiBand firmware is typically stored in the `assets` directory of unzipped APKs [52]. This way of firmware distribution has been leveraged by FirmXray [79] to collect firmware of BLE-related devices using Nordic or TI chips. Since our goal is to measure the general IoT security landscape with no bias towards any specific wireless protocol (e.g., BLE) or application field (e.g., home security), we develop general rules to recognize firmware beyond Nordic and TI based on our empirical study of how firmware is stored and named. Concretely, our tool examines the file name, including its path, name, and extension. We use both firmware-related keywords (e.g., "`firmware`" and "`ota`") and self-developed heuristics as filters. For example, we found that many firmware images are named after the hash values of the file content. Therefore, we also include files with

names that are comprised of consecutive hexadecimal digits (e.g., `3f4c38fc366c44d3b399d3951ab719b4.zip`). The list of keywords we use to match file names is provided in Table 11. For the file extension, we follow the common conventions of manufacturers (e.g., `.bin` and `.fw`). The complete list of firmware extensions and their explanations are provided in Table 12. If a file matches any of the rules, it is considered a candidate firmware image.

4.2 Firmware Capture via OTA Update Simulation

Another popular method for distributing firmware images is using OTA updates, where the firmware is downloaded from the server on demand. To tackle this, we develop *OTACap* to reconstruct the URLs for downloading firmware images by statically analyzing Android APKs. In essence, *OTACap* employs Value Set Analysis (VSA) [11] on the string parameters of Android network functions to obtain URLs, and then it applies keyword-based filters to exclude irrelevant links.

In this section, we first introduce the basic string reconstruction solution (§4.2.1). Then we discuss the challenges encountered in applying it to the URL reconstruction problem and propose solutions for a comprehensive and precise analysis of URL-related strings (§4.2.2). Finally, we present an AI-powered crawler that uses the reconstructed URLs to simulate OTA update for firmware downloading (§4.2.3).

4.2.1 String Analysis for Android

VSA is a powerful binary analysis technique to over-approximate the set of values that each data object can hold at each program point [11]. It has been used to analyze strings in Android apps for different purposes [20, 63, 86, 87]. For example, LeakScope [86] recovers URLs from the apps to uncover data leakage in the cloud. IoTFlow [63] analyzes the data flow among the app, the device, and the cloud via internet addresses reconstruction. Many of these works perform static analysis on top of the *Jimple* Intermediate Representation (IR) of the target APK, which simplifies the analysis by exposing to the analysis tools a non-stack-based and non-nested program representation [1]. These tools construct a call graph of the target APK in *Jimple* IR. Then, they perform a backward data flow analysis to trace how a string is constructed. Finally, based on the recorded backward execution trace, a forward simulation is conducted to recover the string. We describe further details on the backward analysis and forward simulation steps of the techniques mentioned above as follows.

Backward Analysis. Starting from the target string parameter, the tool locates the calling method (called a sink) and taints the string parameter, which is termed a `ValuePoint`. Each `ValuePoint` contains information such as the method it is located in, the beginning statement (e.g., the call to the sink method), and the target parameter(s). From this point, the tool moves backward, storing each statement (with its context) that uses or modifies the values of tainted parameters and propagates the taint to relevant variables in the new statements. When the first instruction of a method is reached, the tool finds all its call sites, branches the analysis to each of the callers, and continues backward analysis through each branch sequentially. When the definition of the tainted data (called a source) is reached, the analysis stops. Here, the definition could be an assignment from immediate values, a global variable with default values, or values from Android resource files. If the tool does not find the source and cannot move backward further, it fails to reconstruct the target string.

Forward Simulation. At the end of backward analysis, if the tool finds the sources of the target string, it also maintains a list of execution traces from the source to the sink. The tool then starts from the source, simulates the execution of statements, and updates the register values until the sink is reached. At this point, the tool has reconstructed the target string. Note that since one sink-source pair can have multiple execution traces, the tool may generate multiple values for the target string.

4.2.2 Enhancing Data Flow Tracking

We create a list of Android native or third-party network functions as sinks, focusing on the HTTP GET, FTP and MQTT protocols since they are the most common in OTA update. Important parameters such as URLs, credentials and MQTT topics are tainted, which are to be recovered via the aforementioned string analysis. We initially used the existing implementations of LeakScope [86] and IoTFlow [63]. However, while effective in their respective tasks, we found that they commonly fall short in recovering information needed in OTA update. A significant reason is that they may lose track of essential data flow information due to IoT specificity or incomplete implementation. Below, we present why existing tools fail to capture the intended data flow and how we address them with *OTACap*.

Missing Definition of Hardware-Related Variables. As mentioned before, the backward analysis succeeds when the definition of a tainted value (i.e., the source) is found. However, we found many failed cases where the tainted value cannot be further traced back statically because it is defined by information that is only known at run-time, potentially from the IoT device. We demonstrate such an example in Listing 1, which contains the decompiled Java Code of the app `vstc.vscam.client`. As shown in lines 14-18, part of the URL is a concatenation of three strings in `strArr`. Except for `"firmware"`, both `sFUA.this.language` and `sFUA.this.LocalSysver` are values obtained from the IoT device at run-time. Specifically, the function `"CallBack_CameraStatusParams"` queries the device about its current firmware version and stores it in `sFUA.this.LocalSysver` (line 22). Using the con-

structed URL, `VscamApi.runGet` is called to send the current firmware version to the cloud endpoint and obtain a `JSON` file containing real links to the updated firmware images. Since "`sFUA.this.LocalSysver`" is only available with a real device, it is impossible to recover it statically.

```
1  boolean checkSpecialFirmware(String str){
2      if (str.length() < 11)
3          return false;
4      String[] split = str.split("\\.");
5      if (split.length >= 3 && split[0].equals("48") &&
        split[1].equals("50") && split[2].equals("64") &&
         Integer.parseInt(split[3]) < 49)
6          return true;
7      return false;
8  }
9  void handleMessage(Message message){
10     if (!sFUA.checkSpecialFirmware(sFUA.LocalSysver))
11         return;
12 }
13 void run(){
14     String[] strArr = {"firmware", sFUA.this.
        LocalSysver, sFUA.this.language};
15     StringBuffer buffer = new StringBuffer();
16     for (int i = 0; i < 3; i++)
17         buffer.append(strArr[i]);
18     VscamApi.runGet(connectionURLNew + buffer.toString
        ());
19 }
20 void CallBack_CameraStatusParams(String str, String
        str2, String str3, String str4){
21     ...
22     this.LocalSysver = str2;
23 }
```

Listing 1: URL construction using hardware-related information.

Fortunately, we observe that sometimes these values—while cannot be directly recovered—can be inferred to some extent at other places. For example, the function `checkSpecialFirmware` takes in `LocalSysver` as input and splits it with the delimiter "`\\.`" (line 4). Then, it compares each part with a constant value (line 5). Although this function does not directly influence the value of `LocalSysver`, nor does it determine the execution of `VscamApi.runGet`, we can infer some constraints about it, with which we can guess its possible values. For example, the version number `48.50.64.48` would return `true` while `48.50.64.49` would return `false`.

Solution: For a tainted value whose source is determined at run-time, *OTACap* searches for its references beyond the recovered data flow. From there, *OTACap* performs a symbolic execution to build the constraints along the execution path. Each constraint will be solved to a concrete value, which is used as the source value of the tainted variable. In this example, we would have six constraints and correspondingly six value candidates for `LocalSysver`.

Implicit Data Flow in `AsyncTask`. Firmware downloading usually takes time. Therefore, app developers commonly use a background task to complete it without blocking the UI. In Android, the most popular library to achieve this is `AsyncTask`, which simplifies background task execution and allows updating the UI thread with the task results. `AsyncTask` must be subclassed, and the type information for the parameter of the background task is passed as the first parameter to the constructor of an `AsyncTask`. Then, the `doInBackground` method, which must be overridden, takes an array of parameters of the same type from the `execute` method and performs the background task.

In Listing 2, we show how `AsyncTask` is used in `com.rhodonite.hms_ione_6630` to download firmware. The class `HttpDownload` is a subclass of `AsyncTask`, and the `doInBackground` method takes a string array as download URLs. The type information (i.e., `String`) is specified by the parameter for the `HttpDownload` constructor (line 5). In line 3, the background task is started by calling the `execute` method using `MainActivity.this.file_url + "dir.txt"` as argument, which is passed to the `doInBackground` method (line 6). Unfortunately, the backward analysis does not recognize such implicit data flow since there is no edge in the call graph.

```
1  class BluetoothProcess extends Handler {
2      public void handleMessage(Message msg)
3          new HttpDownload().execute(MainActivity.this.
            file_url + "dir.txt");
4  }
5  class HttpDownload extends AsyncTask<String, ...> {
6      public String doInBackground(String... f_url){
7          URL url = new URL(f_url[0]);
8      }
9  }
```

Listing 2: Firmware Download in Background with `AsyncTask`.

Solution: After the initial call graph is built, we analyze all the `doInBackground` method definitions and look for any sink methods. If so, we ensure the tainted value can be traced back to its parameter. Then, all the invocations to the constructor of this class are located to find the matching `execute` invocations. Finally, an implicit edge is created in the call graph for each pair of `execute` and `doInBackground`.

It is worth mentioning that the `AsyncTask` class has been deprecated since Android API level 30 [29]. Google recommends using the low-level thread control API `java.util.concurrent` to handle asynchronous tasks. Based on our research, using the recommended `Executor` interface, the parameter of the worker thread is explicitly passed from the `createWorker` method. In other words, the recommended API does not need special treatment. Interestingly, our evaluation never encountered the use of recommended API in firmware downloading. We expect more uses of the `Executor` API as `AsyncTask` becomes abandoned entirely.

Additional Challenges. In addition to the aforementioned challenges and our solutions, we also identify further obstacles and describe our strategies to address them. 1) Implicit Call of `toString`: Many primitive types such as `Integer`, `Long`, `Boolean`, `Byte` and `Array` have a `toString` method that is implicitly called when they are concatenated to another string. Existing tools [63, 86] simply ignore these cases, causing incompletely constructed URLs or the premature termination of analysis. Our forward execu-

tion simulates such implicit conversions for common primitive types. 2) String Processing Methods: Neither IoTFlow nor LeakScope provides comprehensive support for string processing methods, leading to incorrect string value recovery. These missing methods such as `relpace` and `split` were frequently encountered during our experiments and are addressed in *OTACap*. 3) Loops: Existing solutions do not support loops. We partially address this by simply unrolling the loop when the number of iterations is statically known. 4) `Intent`: `Intent` is a heavily used *Inter-Component Communication* (ICC) mechanism in Android. Similar to `AsyncTask`, it involves implicit data flow. While IoTFlow [63] handles them, we found some implementation bugs which prevent it from capturing the correct data flow when the "`key`" and "`value`" parameters of `Intent.putExtra` are global variables. 5) URLs in Resource Files: Android apps commonly use resource files to store static content such as bitmaps, layout definitions, user interface strings, etc. Sometimes, resource strings are used as URLs. Although our string analysis supports recovering URLs from resource strings, it may miss some. Therefore, we explicitly include URL-format resource strings in the list of candidate URLs.

4.2.3 Simulating OTA Firmware Downloads

With a list of candidate URLs, we connect to the endpoints via the corresponding protocols (HTTP, FTP or MQTT) to download firmware. A connection may fail for three reasons. First, *OTACap* suffers from accuracy issues as with other static analysis methods. Second, sometimes the HTTP URL needs credential query strings which can only be obtained from a real registration (e.g., `?token=` and `?deviceid=`). Third, the endpoint might be temporarily or permanently down.

For a successful connection, we observe that sometimes it does not directly return a firmware image. Instead, a text-format datastream (e.g., XML, JSON, or even unstructured text) is obtained, pointing to the real OTA URL. The initial URL can be viewed as a permanent distribution point for new firmware, requiring the app to parse and process its dynamic response in a multi-round fashion. For example, when analyzing the app `com.linkiing.firesign`, *OTACap* recovers a URL pointing to a JSON file which in turn contains links to the newest firmware URLs. Emulating the same app logic to download firmware is challenging since it requires accurate static analysis to decide the code slices to run. We therefore approximate the problem to a crawling problem. Specifically, we treat the initial set of URLs as the seeds and recursively crawl the server with the returned new links. To avoid connecting to irrelevant endpoints, in each round, we use the keyword lists in Table 11 and Table 12 to filter the new links.

As aforementioned, the returned text might be unstructured and sometimes contains credentials to make a successful connection. Missing the original app logic to parse the text, we leverage the powerful reasoning capability of LLM to automatically extract the needed information from unforeseen text. In Appendix B, we provide a real example of unstructured text we captured, the prompt we used, and the result.

SDK Artifacts	Nordic [51]	(1)
	TI SimpleLink [72], [73]	(7)
	Espressif ESP-SDK [26]	(2)
	Microchip [37], [38], [39]	(3)
	Infineon/Cypress [19]	(2)
	Qualcomm CSR102x [55]	(1)
	Dialog SmartBond [61]	(2)
	TeLink [70]	(1)
	Renesas RX Core [60]	(3)
	CSR BlueCore [56]	(3)
	Opulinks [53]	(1)
	Silicon labs [66]	(1)
	STM32 [67], [68]	(2)
	Ubisys [77]	(1)
Bootloader Artifacts	UF2 [40]	(1)
	MCUBoot [35]	(1)
	Zigbee ZCL [85]	(1)
	UPG [4]	(1)
Encoding Artifacts	Intel HEX [8]	(1)
	Motorola S-Record [43]	(1)
	TI-TXT [74]	(1)
	Tektronix [74]	(1)
Architecture artifacts	Arm Interrupt Vector Table	(1)

Table 1: Firmware signatures in groups. The number in the parentheses indicates the number of signatures.

5 Signature-Based Firmware Recognization

Even though we applied filters during firmware collection, a high rate of false positives is expected due to the nature of the crawling process. To further reduce false positives, we implement a signature-based firmware recognition method. The idea is to collect the artifacts that are left to the firmware during the development process and then craft rules to recognize these artifacts. We categorize firmware artifacts into four groups: SDK artifacts, bootloader artifacts, encoding artifacts, and architecture artifacts. These artifacts can not only help us confirm firmware but also play an essential role in firmware analysis due to the firmware metadata contained in it. For example, some firmware headers include fields that indicate the chip information, encryption status, or even the SDK version. We summarize all 39 signatures that we have identified in 4 categories in Table 1. The number in the parentheses indicates the number of signatures for that artifact since a vendor SDK may have multiple versions.

SDK Artifacts. To attract developers, it is common practice that chip vendors provide Software Development Kits (SDKs) to ease the development. Consequently, device manufacturers using these chips often reuse the SDKs shipped with the chip. These SDKs typically include a full toolchain for building firmware images. The use of them leaves diverse footprints in the compiled firmware which are observable via different means. For example, TI designed a firmware update protocol called Over the Air Download (OAD) [71]. When a firmware

image supports OAD, its header must start with the magic string "CC26x2R1", "CC13x2R1", or "OAD IMG". Some chips have unique file extensions (e.g., .cyacd for Cypress), while some organize firmware in specific ways (e.g., Nordic's Device Firmware Update, or DFU image is compressed into a zip file along with some metadata [51]). More generally, a vendor's SDK includes bootstrap code that is commonly shared by all its chips. Since the bootstrap code handles low-level initialization process and has to be written in assembly, the translated machine code—regardless of the compiler being used—can reliably serve as fingerprint of the SDK. To collect SDK artifacts, we manually downloaded and investigated the SDKs of top MCU vendors [54]. Then we summarized the invariances.

Bootloader Artifacts. Beside chip-exclusive firmware formats, we observe a trend of adopting open-source bootloaders to wrap the real firmware. These initiatives will greatly simplify the maintenance and distribution of firmware. Similar to how the U-Boot Image wrapper (uImage) is used to support different OSs (e.g., Linux) that are loaded by U-Boot [64], several bootloaders for MCU firmware are gaining popularity. For example, the USB Flashing Format (UF2) [40] is developed by Microsoft for flashing MCUs over MSC (Mass Storage Class). It has been adopted by the Adafruit community and gains native in-silicon support on the popular Raspberry Pi RP2040 chips. Another example is MCUBoot [35], an open-source secure bootloader for 32-bit MCUs, which can be used in by Zephyr, Mbed OS, RIOT, etc. Finally, the ZigBee alliance also standardized a bootloader for different ZigBee-compatible devices to cooperate with each other with its ZigBee Cluster Library (ZCL) specification [85].

Encoding Artifacts. The firmware conveys binary data that can be understood by the MCU hardware. However, when transmitted or stored, the binary format cannot be easily carried on certain media such as paper tape, punch cards, etc. To address this issue, specially designed encoding schemes are used to represent the binary data in a human-readable ASCII text form. These encodings, as exemplified by Intel HEX [8] and Motorola S-record [43], almost certainly suggest firmware.

Architecture Artifacts. The processor architecture sometimes necessitates specific structures of code or data that can be used as signatures. A classic example is the Arm Cortex-M series's *Nested Vectored Interrupt Controller* (NVIC). According to the Arm architecture manual [7], the NVIC needs a vector table that is mapped at the beginning of the firmware. This table contains the reset value of the stack pointer followed by exception vectors that specify the entry points of the exception handlers, including the reset handler, the entry point of the firmware. Since the stack pointer must point to a valid RAM range, it should be within 0x20000000-0x20200000 and aligned on a word boundary. Besides, each entry in the vector table has to point to a valid address and must be odd since Arm MCUs only support Thumb mode. These simple constraints can quickly screen NVIC vector tables, thus identifying images that are likely to be Arm firmware. To further confirm the result, we ran a chip-generic version of FirmXRay [79], similar to FirmXRay-M used in FirmLine [5], to try to calculate the base address of the firmware. If no meaningful base address can be found, the firmware is discarded. We discuss our improved FirmXRay in §6.1.

6 Firmware Collection Results

6.1 Implementation

The data flow analysis module of *OTACap* is based on IoTFlow [63], which itself is based on LeakScope [86]. The symbolic execution engine reuses the tainting system of data flow analysis and relies on JavaSMT [80] to solve the collected constraints. All the new features were implemented with 2.4k lines of Java code.

Our intelligent crawler is based on Scrapy [88], an open-source web crawling written in Python. We improved it with better FTP support and the integration of the Paho MQTT client [22]. To extract URLs and credentials from unstructured text, our prototype uses a quantized 8B Llama 3 LLM [36] but it can be easily replaced by other more advanced models.

Regarding the firmware signature, our tool now supports 39 firmware signature rules, corresponding 30 SDK artifacts, 4 bootloader artifacts, 4 encoding artifacts, and 1 architecture artifact. We implemented these rules as custom signatures for Binwalk [59], the *de facto* standard of firmware recognition. In particular, the majority of newly developed signatures were written in the libmagic file format. However, when the libmagic is not expressive enough to support a signature (e.g., in the case of Arm vector table recognition), we used the plugin system of Binwalk, which is written in Python. Since the detection of Arm vector table requires FirmXRay [79], our Python plugin also invokes FirmXRay as an external program. FirmXRay was originally implemented for Nordic and TI firmware; however, the idea of using absolute addresses to resolve the base address is general. Our implementation removes all the chip-specific knowledge, similar to FirmLine [5], but retains all the original constraints as much as possible, including indirect calls, string pointers and vector table entries. If all these constraints are found useless during analysis, our tool outputs an invalid base address and discards the firmware. In contrast, the original FirmXray would output a default base value (0x00) since it was built on the assumption that the firmware is either TI for Nordic. Therefore, our modification not only finds the base address for general Arm firmware, but also verifies the Arm signature to reduce false positives. In addition, we fixed some bugs of FirmXRay, which for example might resolve an odd base

address occasionally[1].

To facilitate firmware analysis, we also developed a pipeline to extract firmware metadata from its artifacts and prepare a firmware copy ready to be loaded into Ghidra. If a compressed image is detected, we decompress it using the built-in extraction rules of Binwalk and only analyze the decompressed files. When a candidate image matches a signature, we extract the metadata and store it in a `JSON` file, with entries containing information about firmware architecture, chip vendor/model, base address, entry point, CRC protection, encryption status, etc. Note that not all of these metadata are available on every firmware image. When unsure, the corresponding field is left empty. The text-based firmware (e.g., Intel HEX) is also converted into binary format. The implementation is integrated into Binwalk as extraction rules using an external tool, `srec_cat` [42].

6.2 Dataset

The mobile-IoT Android apps were collected from three sources. In IoTSpotter [32], the authors developed a machine learning-based system that collected a dataset of 37,783 mobile-IoT Android apps to conduct a market-scale security analysis of consumer IoT products. We supplemented this dataset with the APKs used in IoTProfiler [45] (6,208) and another study [47] (455), resulting in a combined dataset of 40,675 unique mobile-IoT APKs. After retrieving the package names of these apps, we downloaded a snapshot of these APKs from Androzoo [6] on July 28, 2023.

6.3 Performance of *OTACap*

As far as we know, there is no prior work that can reconstruct firmware download URLs by analyzing mobile companion apps. The closest work to *OTACap* is IoTFlow [63], which analyzes mobile apps to automatically reveal how the app communicates with IoT devices and remote cloud backends. It, therefore, necessitates reconstructing the Internet addresses of the involved entities, including the cloud backends. In fact, our implementation is based on it. In this section, we evaluate how solving the challenges explained in §4.2.2, including inferring hardware information—a piece of critical information for the OTA protocol—can improve our capability in URL reconstruction. We used the same configuration for both tools to ensure a fair comparison, including the timeout, maximum number of backward iterations/contexts, and the network sinks. These configurations are necessary because they prevent the analysis from spending too much time on a single backward path. No filtering was performed since we focused on evaluating the URL recovery capability instead of the firmware collection capability.

Our experiments ran on a server equipped with two Intel Xeon Silver 4110 CPUs, 46 GB of RAM, and a 14 TB hard disk. The used dataset is a subset of the IoTSpotter dataset, comprising 1,000 randomly selected Mobile-IoT APKs. Since a URL might be incorrectly constructed, we also tested communication via the constructed URLs and checked the status code to verify the result. As shown in Table 2, *OTACap* and IoTFlow are on par regarding the execution speed despite the overhead caused by the symbolic execution engine. There was a high deviation (indicated by standard deviation or SD) in execution time, ranging from five seconds to an hour. The time breakdown indicates that the majority of the time was spent on control flow graph (CFG) construction and backward analysis. *OTACap* found 11,091 more URLs and 422 more reachable URLs compared to IoTFlow. This means *OTACap* can potentially find 25.10% more firmware.

	Time Breakdown in Seconds (mean/SD)				URL #	
	CFG	Backward	Forward	Symbolic	All	Reachable
IoTFlow	152.7 *(253.7)*	283.4 *(597.3)*	4.33 *(7.4)*	–	50,538	1,681
OTACap	155.0 *(253.0)*	297.8 *(628.3)*	4.74 *(8.3)*	20.54 *(45.4)*	61,629	2,103

Table 2: Performance of URL Recovery.

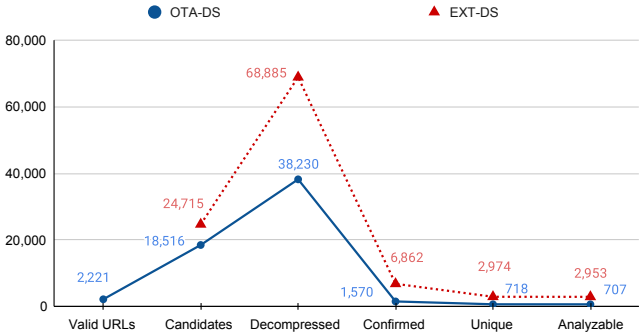

Figure 2: Firmware collection progress.

6.4 Result Summary

Considering the massive number of 40,675 APKs, we threw the workload of *OTACap* to six AWS nodes, each having four CPU cores and 32 GB of RAM. It took six days to complete the URL reconstruction. Other processing steps, including validating the URLs, crawling the firmware images, extracting firmware from APKs, and signature recognition, took negligible time compared with *OTACap*, typically within a few hours. In total, we recovered over 100k URLs after filtering. Among them, 2,221 are valid, with which we crawled 18,516 candidate firmware images. On the contrary, with the APK extraction method, we collected another 24,715 unique candidates. As we will discuss soon, the majority of them are false positives with a `.bin` extension name. We distinguish the two firmware datasets collected via different methods.

- ***OTA-DS*:** Firmware collected via *OTACap*.
- ***EXT-DS*:** Firmware collected via APK extraction.

The majority of the *OTA-DS* dataset consists of firmware from the HTTP GET or FTP protocol. But we did obtain 130

[1] https://github.com/OSUSecLab/FirmXRay/pull/4

images thanks to the 44 valid MQTT endpoints we recovered. Ideally, firmware download should be protected via the unique topic names to subscribe to or additional authentication [76]. However, we found 3 MQTT endpoints that allowed us to connect to the wildcard topic `#` without any authentication, returning trunks of firmware data or the real firmware distribution points. Additionally, one developer used the test broker Mosquitto [23] to send OTA URLs.

Figure 2 shows the progress of processing the candidate firmware. For each dataset, starting with the candidate firmware images (marked as **Candidates** in the figure), we first scanned for zipped files and decompressed them. This leads to a rise in numbers (marked as **Decompressed**). Then, the signature matching was conducted to get the confirmed number of firmware (marked as **Confirmed**). We found that this set of firmware images contains some Linux/Solaris-based images, which were identified by the existing signatures of Binwalk. We excluded them from our dataset since our focus is on MCU firmware. Also, lots of duplications were observed. After excluding Linux/Solaris-based and duplicated images, we obtained the number of unique MCU firmware images (marked as **Unique**). Finally, with the metadata from signature matching, we identified the number of analyzable firmware images to the best of our capability (marked as **Analyzable**). Here, analyzable means that the instruction set of the firmware is known, and the firmware is not encrypted or compressed. Eventually, we obtained 718 and 2,974 unique MCU-based firmware images in *OTA-DS* and *EXT-DS*, respectively.

As shown in the figure, *EXT-DS* includes way more confirmed images than *OTA-DS* (6,862 vs 1,570). We attribute this to two reasons. First, to reduce the cost of maintaining an OTA server, device manufacturers tend to choose bundling firmware within the APKs and rely on Google Play Store to distribute firmware. Second, *OTACap* suffers from the inherent limitation of static analysis (see §8).

We can see a significant drop from **Candidates** to **Confirmed**, indicating the effectiveness of our signature matching mechanism. A high number of `.bin` files were discarded since they are used to store application data instead of code. The developed signatures are extremely accurate. We did not find any false positives in our experiments, as explained in §7.2.

We also found a non-negligible duplication rate in both datasets. The most common reason is shared bootloaders among different devices. For example, the same Nordic SoftDevice firmware image [50], which includes drivers for Nordic radio peripherals, is found multiple times. Some developers release apps with very similar functionality under different names, which in turn lead to duplicated firmware images. For example, both `com.nextys.mobile.powermaster` and `com.nextys.dcu20.mobile.powermaster` are developed by `Nextys`. The two APKs contain identical firmware under `/com/nextys/lib/resources/firmwares/_nef210-bootloader-rel.hex`. Interestingly, we did not find any duplication across the two datasets, indicating that developers tend to stick with one approach of firmware distribution.

Protection	Encrypted or Compressed (High Entropy)	CRC	Not Protected
OTA-DS	10	6	702
EXT-DS	11	157	2,806

Table 3: Firmware protection level.

Regarding the slight drop from **Unique** to **Analyzable**, it is because of encryption or compression that is applied to the firmware or the use of proprietary instruction sets. We found many images with CRC integrity protection (4.41%), but only 0.57% are crypto- or compression-protected (Table 3). This means that at least 99.43% of the collected images are in plaintext and subject to firmware analysis. Moreover, there are 21 images with proprietary instruction sets.

7 Characterizing the IoT Ecosystem

With the large firmware dataset, we conducted two types of analyses. In §7.1, we aim to decompose the IoT market by analyzing the major players in the market. §7.2 is security oriented binary-level analysis. First, we reveal the complexity of firmware datasets, including size and function count. Second, we applied function signature recognition to detect the utilization of IoT libraries and to reveal N-day vulnerabilities. Third, we scanned for the adoption of common attack mitigation. The results are alarming.

7.1 IoT Decomposition

With the signature metadata, we analyze major players in the IoT market by chip vendors and architectures over the **unique** images. Since artifacts follow diverse formats conveying custom information defined by the respective developers, we procured the needed information on a best-effort basis.

By Chip Vendor. Recall that we developed four kinds of firmware artifacts (§5). If an image is recognized by a vendor artifact, it is naturally counted towards that vendor. If a bootloader artifact is used, we examine the fields contained in the bootloader header. Typically, it contains fields indicating the chip information. For example, in the UF2 bootloader, there is a `familyID` field at offset 28, which clearly reveals the device chip information[2]. If an encoding artifact is used, we first convert the firmware into binary format and then try to match it with SDK artifacts and bootloader artifacts. Architecture artifacts typically do not provide vendor information.

We summarize the results in Table 4. The vendor distributions in the two datasets are not consistent. We found more uses of Microchip and Renesas chips in *OTA-DS* while *EXT-DS* has more Nordic, TI, Espressif, and Telink chips. Top Arm MCU vendors such as NXP and STMicroelectronics do not

[2] https://github.com/microsoft/uf2/blob/master/utils/uf2families.json

Vendor	Nordic	TI	Espressif	Telink	Cypress/Infineon	Microchip	Dialog	Qualcomm	Renesas	CSR	Opulinks	Arm*	Total
OTA-DS	16	4	150	1	18	29	6	10	31	1	1	451	718
EXT-DS	875	164	244	450	78	14	24	9	1	1	-	1,114	2,974

Table 4: Firmware distribution by chip vendor (unique images). Arm* means firmware that matches with the general Arm architecture signature but without any vendor signature. This potentially includes other major players such as NXP and STM32.

Arch.	Arm	MIPS	PIC	Xtensa	RISC-V	RX Core	Prop.	Total
OTA-DS	496	18	11	150	1	31	11	718
EXT-DS	2,255	12	2	244	450	1	10	2,974

Table 5: Firmware distribution by architecture (unique images). PIC includes PIC18/24/32. Prop. refers to proprietary.

have an entry in the table since we did not find many reliable artifacts from their SDKs. Instead, we observe a large number of general Arm images in both datasets, which were detected by the architecture artifact. We speculate that a large portion of this group should be attributed to NXP or STM32 chips. Conversely, we also observed some Arm firmware (mainly TI chips) that is not detectable by the Arm architecture artifact.

By Architecture. To get architecture information, we mainly use the chip information. As expected, Arm dominates the market (74.51%), followed by RISC-V and Xtensa. Surprisingly, no TI firmware corresponds to the MSP430 chip, which was popular and runs a proprietary instruction set developed by TI. Xtensa's high share mainly comes from the popularity of ESP32 series chips by Espressif, while RISC-V mainly comes from Telink. Microchip is responsible for the PIC18/23/32 architectures. We also observe 21 images using proprietary architectures. They come from Qualcomm's `CSR102x` chips [55] and the legacy Cambridge Silicon Radio (CSR) BlueCore chips [56]. Both follow a proprietary 16-bit RISC architecture.

7.2 Binary-Level Analysis

To understand the IoT security landscape, unlike previous work that uses the companion apps as the analysis target [14, 31, 32, 63, 86], we directly study the firmware to reveal its binary-level properties, thanks to the large real-world firmware dataset we have collected. Due to their popularity in the market, we selectively conducted static analysis for 2,631 Arm and 394 Xtensa firmware images.

We primarily employed general static analysis techniques to cover a broad range of firmware and identify common properties. While it is feasible to conduct more sophisticated analysis to study a particular firmware property, this requires extensive domain knowledge to properly set up the environment. For instance, techniques like firmware rehosting necessitate an understanding of the device's memory map. If the configuration is wrong, the result might be unreliable. Although this is not done in this paper, domain experts can utilize advanced firmware analysis projects, such as Fuzzware [62] and HEAPSTER [30], to study firmware samples of interest.

	Arm			Xtensa		
	Mean	Median	SD	Mean	Median	SD
Func. #	789.96	539.00	3,317.15	3,799.04	3,106.50	3,337.91
Size (KB)	541.90	85.54	6,912.09	951.82	648.88	863.88

Table 6: Statistics of function count and firmware size.

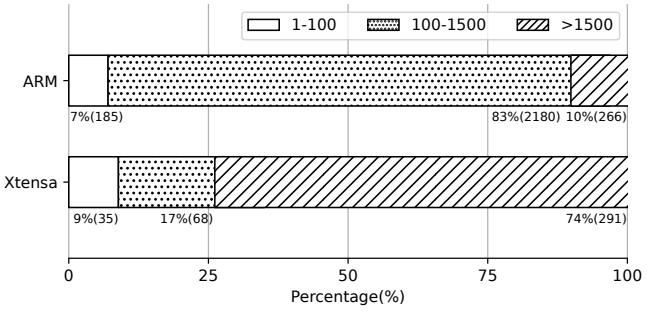

Figure 3: Distribution of function number.

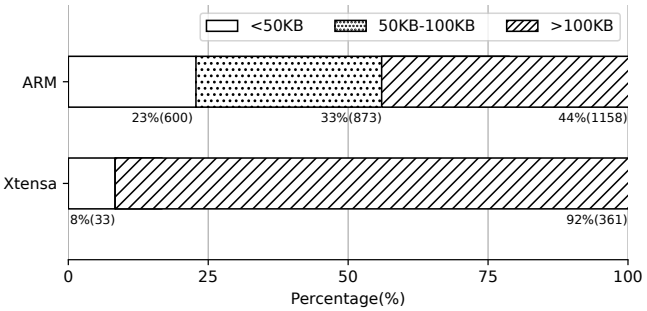

Figure 4: Distribution of firmware size.

7.2.1 Complexity Analysis

We loaded firmware images to Ghidra for automatic analysis and count the firmware size and the number of functions in each image. The distribution of function count in Figure 3 and Figure 4 indicates that Arm-based firmware samples generally have fewer functions and smaller sizes, with more than 90% having less than 1,500 functions. In contrast, over 74% of Xtensa-based firmware images have more than 1,500 functions. We hypothesize that this is because Arm chips are more diverse with customizable peripheral configurations. As a result, many of them are not as powerful as Espressif's ESP32 series chips, which provide a one-stop wireless solution suitable for a wide range of applications. Consequently, firmware running on Xtensa-based ESP32 MCUs often contains more functionalities. Table 6 shows statistics of function count and

firmware size, including mean, median, and standard deviation. We can see that the deviation in Arm-based firmware is more significant than that in Xtensa-based firmware, agreeing with our hypothesis before.

7.2.2 Library Adoption Analysis

To reduce time to market and streamline firmware development, it is common practice for device manufacturers to integrate open-source middleware (e.g., communication protocol libraries), RTOS libraries (e.g., FreeRTOS, Mbed OS), and chip SDKs/BSPs into the firmware stack. By fingerprinting these libraries in our dataset, we can uncover the prevalent libraries used in the IoT ecosystem.

Library Dataset. For Arm, we collected commonly used MCU middleware, RTOS libraries and chip SDKs/BSPs. For Xtensa, we mainly used the Espressif IoT Development Framework (ESP-IDF) [25], which provides a one-stop repository for developing firmware for ESP32 SoCs. Notably, it conveniently maintains not only Espressif's own driver code but custom copies of third-party libraries. We collected all the libraries of latest versions at the time of writing. In total, our dataset includes 1,435,788 Arm functions and 1,239,283 Xtensa functions.

Fingerprinting Libraries. Matching a function is the first step towards library matching. There are two flavors for function matching. The ID-based solution calculates a unique identification signature for each function. For example, Ghidra's Function ID [46] implementation hashes the masked function bytes to generate the ID. Here, masked means that the address information is removed to make the result independent of the function address. The other flavor is based on similarity. That is, for a pair of functions, a similarity score ranging from `0-1` is calculated. Graph isomorphism is widely adopted in this direction, e.g., BinDiff [27].

The Function ID method can be fast and accurate. However, it cannot tolerate trivial differences in library version and compiler selection. That is, to match a library, we would have to build a dataset for all the historical versions using all possible compilers (and options). With the similarity-based method, we can use only the latest library version to potentially match outdated versions. This is particularly relevant for matching functions in IoT firmware, which rarely updates its third-party libraries.

Conducting pairwise similarity-based matching is time-consuming, especially when comparing millions of functions. Inspired by LibMatch [17], we propose SimMatch, a two-step similarity-based function matching approach to overcome the scalability issue. In step 1, we calculate the product of three fundamental metrics for a function: the number of basic blocks, the number of edges in the CFG, and the count of jump instructions. We only proceed to step 2 if the step-1 scores for the two functions are close. In step 2, we compare the context of their basic blocks using the string comparison algorithm based on Ratcliff/Obershelp pattern [12]. We consider two function a match when the similarity score exceeds 95%.

	ARM		Xtensa	
	Function # (%)	Time (hh:mm:ss)	Function # (%)	Time (hh:mm:ss)
SimMatch	99,844 (6.95%)	01:59:56	293,341 (23.67%)	03:10:29
Function ID	72,102 (5.02%)	00:27:45	87,514 (07.06%)	00:15:27

Table 7: Function matching results.

	SDK		Middleware			RTOS
	STM32Cube	Nordic nRF5	Arduino	HARDWARIO	Mbedtls	FreeRTOS
SimMatch	482	615	92	619	8	235
Function ID	366	440	76	380	1	76

Table 8: Library matching results for Arm firmware.

	HAL	lwIP	WiFi	MQTT	BLE	RF	MbedTLS	FreeRTOS
SimMatch	100	343	345	1	144	352	62	266
Function ID	5	195	329	0	86	340	0	79

Table 9: Library matching results for Xtensa firmware.

Results. We ran both SimMatch and Function ID to find matching functions in our firmware datasets and show the results in Table 7. If we consider a library to be matched if two or more library functions are found in the firmware, the library adoption results are shown in Table 8 and Table 9. As expected, Function ID outperforms SimMatch significantly in terms of time consumption. However, it also found less matched libraries because it is more strict and we only used the lasted libraries to perform matching. The gap is bigger for Xtensa firmware. Our investigation shows that the Xtensa compiler tends to generate more diverse machine code even for the same piece of C code, making ID-based solutions more sensitive to little changes.

To understand the false positive rate of both solutions, we manually checked all the Xtensa firmware with matched BLE library. SimMatch incurred 2 false positives out of 144 matches, while Function ID incurred 1 out of 86. The counterintuitive false positive happened in Function ID is because it masks the otherwise different MMIO addresses in `load/store` instructions, leading to a collision. We cannot check false negatives since there is no ground truth about the library adoption.

7.2.3 N-Day Vulnerability Analysis

We identify N-day vulnerabilities using the library adoption information obtained in §7.2.2. If a library has ever had a vulnerable version, it is likely to impact the firmware. Therefore, we first searched for the involved libraries in the CVE database [3] and identified a list of CVEs affecting these libraries. Then, we manually collected the version information of the impacted libraries and the functions involved. Finally, we checked for the presence of vulnerable functions in the firmware.

Since our library datasets were built from the latest releases, we also need to compile the old vulnerable ones to match the function. Fortunately, for Xtensa firmware, the version of the

MPU	TrustZone	Stack Limit	Stack Canary
41 (1.56%)	0 (0.00%)	0 (0.00%)	3 (0.11%)

Table 10: Mitigation features found in Arm firmware.

used ESP-IDF framework is already provided in the firmware header. Therefore, we only need to confirm if the function in firmware is close enough with the one in the latest library.

In total, the Arm dataset contains vulnerable libraries with 2 unique CVEs, while the Xtensa firmware dataset includes 12 unique CVEs. These CVEs were identified twice in 1 Arm firmware image and 552 times across 190 Xtensa firmware images, respectively. The details of these findings are listed in Appendix C. We have reported our findings to 31 manufacturers, and 3 have acknowledged and planned SDK updates.

7.2.4 Mitigation Detection

Modern Arm MCUs enable several security mechanisms to mitigate attacks. We developed Ghidra scripts to search for the adoption rates of these features.

MPU. An *Memory Protection Unit* (MPU) allows programmers to specify the permissions (`R/W/X`) for memory ranges based on the current privilege level. Not using MPU means no data execution protection (DEP) and no isolation between the kernel and user code, allowing code injection, illegal memory access, etc. Since MMIO is the only way to configure MPU on Cortex-M chips, we check whether there exist load/store instructions targeting the memory range of MPUs, i.e., `0xE000ED90 - 0xE000EDBB` for Arm's standard MPU and `0x40000528 - 0x4000060F` for Nordic's simplified MPU (sMPU) [49]. To recover the target addresses in these ranges, the Ghidra plugin finds the base register and offset (register or immediate value) that are known statically. As shown in Table 10, only 1.56% of the images use the MPU feature, agreeing with previous research [16]. There are two limitations of our prototype. First, it does not perform reachability analysis; therefore, the detected MPU code may not be invoked at all. Second, it may miss real MPU accesses due to unavoidable disassemble errors.

TrustZone. Arm announced its Trusted Execution Environment (TEE) solution for MCUs in 2016 [10]. By providing an isolated execution environment, it ensures robust protection even if the normal-world firmware is fully compromised. Nearly eight years following its announcement, we are curious about its real-world adoption. Our tool detects the footprint of TrustZone in the secure-world firmware, since TrustZone implementation details are hidden from the non-secure part. For the secure firmware, we search for the pair of Secure Gateway veneer (i.e., the SG instruction following a branch) and return to the normal world instructions (i.e., `bxns lr`) [9]. We did not obverse any sign of TrustZone usage.

Stack Limit. The stack limit feature introduces two new registers to define the lower limit for the two stacks on Arm. Configured properly, it ensures that stacks do not overrun. Unfortunately, we did not observe any usage of this feature.

Stack Canary. Stack canary is a pure-software mitigation mechanism. It detects stack overflow by checking whether a canary value that is placed by the function prologue is corrupted. We created instruction patterns to match both function prologues and epilogues of canary-enabled functions. Only 0.11% of the images adopt stack canary.

8 Discussion and Future Work

Limitations of *OTACap*. Currently, *EXT-DS* contains much more confirmed images than *OTA-DS*. Besides the fact that APK-bundled firmware distribution is more popular, *OTACap* may also miss a lot of firmware images for three reasons. First, we found many URLs containing a `?token=` or `?deviceid=` query string. These apps typically require user credentials to access the firmware, which is impossible without real user registration. Second, although we tried to infer information about real IoT device to reconstruct URLs as explained in §4.2.2, many apps do not include any clue in the static code. Third, our backward data flow analysis can miss data flows. For example, we cannot unroll loops when the iteration number is unknown.

Threats to Validity. Although we aim to measure the IoT landscape in a general sense, due to the unsoundness and incompleteness of our implementation, bias may exist. 1) We may miss many firmware images due to incomplete keywords/signatures and *OTACap* limitations, thus overrepresenting a specific vendor or architecture. 2) Our measurements presented in §7.1 rely on metadata extracted from known signatures, which may miss some information. For example, we did not find clear signatures for some major Arm vendors, such as NXP. Therefore, we cannot properly decompose the IoT by chip vendors other than roughly putting them to a general Arm group (Table 4). 3) The measurement was conducted over unique images. It does not necessarily reflect the number of chip shipments. 4) Some URLs may become inaccessible over time, leading to reduced firmware.

Future Work. Signature development and identification is largely manual work, which we plan to augment over time. We will also enhance our crawler's user authentication for MQTT and HTTP endpoints. An idea is to emulate user registration using a phantom IoT device, as described in existing work [15, 83]. The main observation is that many device secrets are easily guessable.

More sophisticated firmware analysis tools such as Fuzzware [62] and HEAPSTER [30] can be used to find deeper and even zero-day firmware bugs. Our initial attempt to use Fuzzware to fuzz firmware was not very successful, attributable to three reasons. 1) Some firmware is incomplete, lacking device initialization instructions found in bootloader that is

not part of the OTA update process. 2) Some firmware checks chip information during bootstrapping, which Fuzzware failed to infer. 3) The complex and varied peripheral usage in the firmware cannot be properly modeled. These failed cases will foster new ideas to advance the state of the art.

9 Related Work

Analysis Targeting Firmware. In FirmXray [79], the authors extracted 793 unique firmware from the companion apps. By analyzing this dataset, many BLE-link layer vulnerabilities are identified. In FIoT [84], the authors manually emulated fake IoT devices to capture the OTA update traffic. Then, symbolic execution and fuzzing were adopted to analyze the 318 obtained firmware images. The FirmXRay dataset has been widely used in other firmware analysis research. For example, HEAPSTER [30] runs symbolic execution against the FirmXRay dataset to find the presence of dynamic allocators and uses bounded model checking to find vulnerabilities. Tan et al. [69] systematizes research on Arm MCU firmware. They also analyzed 1,797 Arm firmware samples, which were mainly collected via the FirmXray method and web crawlers to the vendor websites. These works mainly target the Arm instruction set and the results overrepresent devices built on Nordic and TI chips. Our dataset contains more firmware images, covering at least 11 different chip vendors across 7 different architectures. Recently, Firmline [5] proposes a generic pipeline for analyzing non-Linux firmware. An impressive 21,755 samples were obtained from prior research, the Linux firmware repository, Android dumps, and other sources. However, the majority of the samples come from Linux/Android firmware for PC/smartphone peripherals. In contrast, our dataset primarily consists of firmware for consumer IoT devices. Another distinction is that the Firmline analysis pipeline relies on statistical analysis to determine the possible architecture, whereas we leverage firmware signatures to reliably determine the architecture and other metadata.

IoTFuzzer [14], while targeting firmware, relies on analysis of the companion app. It fuzzes device firmware to find memory corruption bugs without having to access the firmware. However, it lacks scalability as real devices are needed. Also, without firmware, it becomes hard to pinpoint the root cause of the bug. By integrating the firmware retrieval capabilities of *OTACap* with the bug detection power of IoTFuzzer, we anticipate more effective tools to combat firmware bugs.

Analysis Targeting Companion Apps. Due to the lack of real firmware, prior work uses mobile IoT companion apps to study the security of the IoT ecosystem on a larger scale. AoT-Scout [31] focuses on finding vulnerabilities involved in the OTA process. The authors analyzed 23 popular IoT devices and corresponding companion apps to locate six popular SDKs with security flaws. Then, they fingerprint each vulnerable SDK in a large dataset of apps, revealing many others that are prone to the same vulnerabilities. However, this approach requires a real device to analyze the APK, thus suffering from the scalability issue. In [78], the authors analyze the library artifacts in mobile companion apps and infer vulnerabilities in the device firmware. IoTSpotter [32] constructs a large IoT-Mobile companion app repository and evaluates the security of each app by analyzing known vulnerabilities in third-party libraries, identifying misuses of cryptographic APIs, and the signing scheme of the APK.

Leakscope [86] finds data leaks from mobile apps to the Internet by reconstructing paths from potentially sensitive sources to network sinks using backward VSA. Similarly, IoTFlow [63] reconstructs the communication of IoT devices with their companion apps and cloud-based backends. BleScope [87] leverages string reconstruction to identify hard-coded UUIDs in the companion app to fingerprint BLE devices. Stringoid [57] reconstructs URLs to analyze Web requests in Android apps. All these tools rely on string analysis. However, they are not specifically engineered towards OTA link recovery, and thus cannot yield the volume of firmware that *OTACap* can. Particularly, *OTACap* outperforms existing tools due to 1) improved VSA accuracy via cross-referencing hardware information and better implicit data flow tracking, and 2) an intelligent crawler that better utilizes the recovered URLs. Moreover, all these works rely on mobile apps to infer the security of the device. They generally cannot get low-level security properties of IoT firmware.

10 Conclusions

We present a novel approach for the large-scale collection, validation, and analysis of MCU-based IoT firmware to improve the understanding of the security landscape of the IoT ecosystem. Specifically, we introduce *OTACap*, a tool designed to automatically extract URLs for OTA firmware update by analyzing companion Android apps. Additionally, we utilize a keyword-based search strategy for rough firmware screening while leveraging a signature-based firmware unpacking pipeline to validate firmware and extract essential information for subsequent analysis.

In our experiments with 40,675 mobile-IoT Android APKs, *OTACap* has successfully extracted over 2,221 valid URLs related to firmware updates, allowing us to download 18,516 candidate firmware images by crawling these URLs. Moreover, we extracted 24,715 unique candidate firmware images directly from APKs that were shipped with the APKs. Using signature matching, we identified 3,692 unique firmware images, and 3,660 of them were analyzable.

Our security analysis indicates a lack of firmware protection, existence of N-day vulnerabilities, and rare adoption of security mitigation in these firmware images. The results presented in this paper establish an extensive, heterogeneous, and annotated dataset of bare-metal firmware currently available.

Acknowledgments

We thank the anonymous reviewers for their valuable comments to improve our paper. This work was partially supported by U.S National Science Foundation (NSF) grants (2238264, 1916500, 2237238, 2329704, 2422242), National Natural Science Foundation of China (NSFC) grant (62202188), and a grant from the University of Georgia Research Foundation, Inc. Opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies

Availability

We have made the source code for our tools available at https://github.com/MCUSec/RealworldFirmware to facilitate the replication of our work. However, releasing the firmware dataset poses potential risks, including copyright infringement and misuse. Despite these concerns, ethical security researchers can leverage the dataset to benefit the IoT community as discussed in §1. To balance these considerations, we will carefully evaluate access requests and share the dataset privately with research teams committed to responsible and ethical practices.

References

[1] Jimple. https://soot-oss.github.io/SootUp/v1.1.2/jimple/. (Retrieved: 02/02/2024).

[2] mi band firmware analyse. https://github.com/flycodepl/mi-band-firmware-analyse. (Retrieved: 02/02/2024).

[3] Mitre CVE program. https://cve.mitre.org/. (Retrieved: 02/02/2024).

[4] Upg file. https://filext.com/file-extension/UPG. (Retrieved: 02/02/2024).

[5] Marius Muench Alexander Balgavy. Firmline: a generic pipeline for large-scale analysis of non-linux firmware. In *Workshop on Binary Analysis Research*, 2024.

[6] 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*, 2016.

[7] Arm Holdings. Armv7-m architecture reference manual. https://developer.arm.com/documentation/ddi0403/latest/. (Retrieved: 02/02/2024).

[8] Arm Holdings. Intel hex file format. https://developer.arm.com/documentation/ka003292/latest/. (Retrieved: 02/02/2024).

[9] Arm Holdings. Switching between Secure and Non-secure states. https://developer.arm.com/documentation/100690/0201/Switching-between-Secure-and-Non-secure-states. (Retrieved: 02/02/2024).

[10] Arm Holdings. TrustZone technology for the Armv8-M architecture Version 2.1. https://developer.arm.com/documentation/100690/latest/. (Retrieved: 02/02/2024).

[11] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2010.

[12] Paul E. Black. Ratcliff/obershelp pattern recognition in dictionary of algorithms and data structures. https://www.nist.gov/dads/HTML/ratcliffObershelp.html.

[13] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, 2016.

[14] 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 *NDSS*, 2018.

[15] Jiongyi Chen, Chaoshun Zuo, Wenrui Diao, Shuaike Dong, Qingchuan Zhao, Menghan Sun, Zhiqiang Lin, Yinqian Zhang, and Kehuan Zhang. Your iots are (not) mine: On the remote binding between iot devices and users. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019.

[16] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *IEEE Symposium on Security and Privacy*, 2017.

[17] Clements, Abraham and Gustafson, Eric and Scharnowski, Tobias and Grosen, Paul and Fritz, David and Kruegel, Christopher and Vigna, Giovanni and Bagchi, Saurabh and Payer, Mathias. Halucinator: Firmware re-hosting through abstraction layer emulation. 2020.

[18] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014.

[19] Cypress Semiconductor. Bootloader and bootloadable. https://www.infineon.com/dgdl/Infineon-Component_BootloadableBootloader_V1.60-Software%20Module%20Datasheets-v01_06-EN.pdf?fileId=8ac78c8c7d0d8da4017d0e9cbad0230c. (Retrieved: 02/02/2024).

[20] Justin Del Vecchio, Feng Shen, Kenny M Yee, Boyu Wang, Steven Y Ko, and Lukasz Ziarek. String analysis of android applications. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[21] Christian J. D'Orazio, Kim-Kwang Raymond Choo, and Laurence T. Yang. Data exfiltration from internet of things devices: ios devices as case studies. *IEEE Internet of Things Journal*, 2017.

[22] Eclipse Foundation. Eclipse paho mqtt python client. https://eclipse.dev/paho/files/paho.mqtt.python/html/client.html. (Retrieved: 02/02/2024).

[23] Eclipse Foundation. Mosquitto test server. https://test.mosquitto.org/. (Retrieved: 02/02/2024).

[24] Espressif Systems. Esp8266 nonos sdk. https://github.com/espressif/ESP8266_NONOS_SDK. (Retrieved: 02/02/2024).

[25] Espressif Systems. Espressif iot development framework. https://www.espressif.com/en/products/sdks/esp-idf. (Retrieved: 02/02/2024).

[26] Espressif Systems. Firmware image format. https://docs.espressif.com/projects/esptool/en/latest/esp32/advanced-topics/firmware-image-format.html. (Retrieved: 02/02/2024).

[27] Halvar Flake. Structural comparison of executable objects. *DIMVA*, 2004.

[28] Ankit Gangwal, Shubham Singh, Riccardo Spolaor, and Abhijeet Srivastava. Blewhisperer: Exploiting ble advertisements for data exfiltration. In *European Symposium on Research in Computer Security*, 2022.

[29] Google. Asynctask reference. https://developer.android.com/reference/android/os/AsyncTask. (Retrieved: 02/02/2024).

[30] Fabio Gritti, Fabio Pagani, Ilya Grishchenko, Lukas Dresel, Nilo Redini, Christopher Kruegel, and Giovanni Vigna. Heapster: Analyzing the security of dynamic allocators for monolithic firmware images. In *2022 IEEE Symposium on Security and Privacy*, 2022.

[31] M. Ibrahim, A. Continella, and A. Bianchi. Aot - attack on things: A security analysis of iot firmware updates. In *2023 IEEE 8th European Symposium on Security and Privacy*, 2023.

[32] 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*, 2022.

[33] Ori Karliner. FreeRTOS TCP/IP Stack Vulnerabilities – The Details. https://blog.zimperium.com/freertos-tcpip-stack-vulnerabilities-details/. (Retrieved: 02/02/2024).

[34] M Kol and S Oberman. 19 Zero-Day Vulnerabilities Amplified by the Supply Chain. *JSOF, White Paper*. (Retrieved: 02/02/2024).

[35] mcu tools. Mcuboot secure boot for 32-bit microcontrollers! https://docs.mcuboot.com/. (Retrieved: 02/02/2024).

[36] Meta. Meta llama 3. https://llama.meta.com/llama3/. (Retrieved: 02/02/2024).

[37] Microchip. Microchip over-the-air updates. https://www.microchip.com/en-us/products/wireless-connectivity/over-the-air-updates. (Retrieved: 02/02/2024).

[38] Microchip. Migrating from pic18f to pic18fxxj flash devices. https://ww1.microchip.com/downloads/en/DeviceDoc/01021a.pdf. (Retrieved: 02/02/2024).

[39] Microchip. Mplab harmony 3 is an extension of the mplab ecosystem for creating embedded firmware solutions for 32-bit microchip devices. https://github.com/Microchip-MPLAB-Harmony/bootloader_apps_ota/blob/master/tools/ota_host_mcu_header.py. (Retrieved: 02/02/2024).

[40] Microsoft. Usb flashing format (uf2). https://github.com/microsoft/uf2. (Retrieved: 02/02/2024).

[41] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015.

[42] Peter Miller. srec_cat. https://srecord.sourceforge.net/man/man1/srec_cat.1.html. (Retrieved: 02/02/2024).

[43] Motorola. Motorola s-records. http://www.amelek.gda.pl/avr/uisp/srecord.htm. (Retrieved: 02/02/2024).

[44] MSRC Team. BadAlloc – Memory allocation vulnerabilities could affect wide range of IoT and OT devices in industrial, medical, and enterprise networks. https://msrc-blog.microsoft.com/2021/04/29/badalloc-memory-allocation-vulnerabilities-could-affect-wide-range-of-iot-and-ot-devices-in-industrial-medical-and-enterprise-networks/. (Retrieved: 02/02/2024).

[45] 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 *32nd USENIX Security Symposium*, 2023.

[46] National Security Agency of the United States. Ghidra feature: Function id. https://github.com/NationalSecurityAgency/ghidra/tree/master/Ghidra/Features/FunctionID. (Retrieved: 02/02/2024).

[47] Shradha Neupane, Faiza Tazi, Upakar Paudel, Freddy Baez, Merzia Adamjee, Lorenzo De Carli, Sanchari Das, and Indrakshi Ray. On the data privacy, security, and risk postures of iot mobile companion apps. *SSRN Electronic Journal*, 2022.

[48] Nordic SemiConductor. nrf5 sdk for mesh. https://www.nordicsemi.com/Products/Development-software/nRF5-SDK-for-Mesh/Download. (Retrieved: 02/02/2024).

[49] Nordic Semiconductor. nrf51 series reference manual. https://infocenter.nordicsemi.com/pdf/nRF51_RM_v3.0.pdf. (Retrieved: 02/02/2024).

[50] Nordic Semiconductor. SoftDevices. https://infocenter.nordicsemi.com/topic/ug_gsg_ses/UG/gsg/softdevices.html, 2021.

[51] Noridc Semiconductor. Device firmware update process. https://infocenter.nordicsemi.com/topic/sdk_nrf5_v17.0.2/lib_bootloader_dfu_process.html. (Retrieved: 02/02/2024).

[52] Yogesh Ojha. I hacked miband 3, and here is how i did it part ii — reverse engineering to upload firmware and resources over the air. https://medium.com/@yogeshojha/i-hacked-miband-3-and-here-is-how-i-did-it-part-ii-reverse-engineering-to-upload-firmware-and-b28a05dfc308. (Retrieved: 02/02/2024).

[53] Opulinks Tech. Opl1000a1-sdk. https://github.com/Opulinks-Tech/OPL1000A1-SDK/tree/master. (Retrieved: 02/02/2024).

[54] Precedence Research. Microcontroller market size to reach usd 69.08 bn by 2032. https://www.precedenceresearch.com/microcontroller-mcu-market. (Retrieved: 02/02/2024).

[55] Qualcomm Technologies International. Otau csr102x. https://developer.qualcomm.com/qfile/34081/csr102x_otau_overview.pdf. (Retrieved: 02/02/2024).

[56] Rami ramikg. Csr dfu file format. https://github.com/ramikg/csr-dfu-parser. (Retrieved: 02/02/2024).

[57] Marianna Rapoport, Philippe Suter, Erik Wittern, Ondrej Lhotak, and Julian Dolby. Who you gonna call? analyzing web requests in android applications. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017.

[58] Betsy Reed. Hacking risk leads to recall of 500,000 pacemakers due to patient death fears. https://www.theguardian.com/technology/2017/aug/31/hacking-risk-recall-pacemakers-patient-death-fears-fda-firmware-update. (Retrieved: 02/02/2024).

[59] ReFirm Labs. Binwalk. https://github.com/ReFirmLabs/binwalk. (Retrieved: 02/02/2024).

[60] Renesas Electronics. Firmware update module using firmware integration technology. https://www.renesas.com/us/en/document/apn/rx-family-firmware-update-module-using-firmware-integration-technology-application-notes. (Retrieved: 02/02/2024).

[61] Renesas Electronics. Suota memory layout. https://lpccs-docs.renesas.com/Tutorial_SDK6/suota_memory.html. (Retrieved: 02/02/2024).

[62] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using precise MMIO modeling for effective firmware fuzzing. In *31st USENIX Security Symposium*, 2022.

[63] 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 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[64] Semihalf. U-boot image interface. https://source.denx.de/u-boot/u-boot/-/raw/e4dba4ba6f/include/image.h. (Retrieved: 02/02/2024).

[65] Ben Seri, Gregory Vishnepolsky, and Dor Zusman. Critical vulnerabilities to remotely compromise VxWorks, the most popular RTOS. https://www.armis.com/research/urgent11/. (Retrieved: 02/02/2024).

[66] Silicon Labs. Silicon labs gecko bootloader. https://www.silabs.com/documents/public/user-guides/ug266-gecko-bootloader-user-guide.pdf#page=10. (Retrieved: 02/02/2024).

[67] STMicroelectronics. Over-the-air application and wireless firmware update for stm32wb series microcontrollers. https://www.st.com/resource/en/application_note/an5247-overtheair-application-and-wireless-firmware-update-for-stm32wb-series-microcontrollers-stmicroelectronics.pdf. (Retrieved: 02/02/2024).

[68] STMicroelectronics. Secure boot & secure firmware update software expansion for stm32cube. https://www.st.com/en/embedded-software/x-cube-sbsfu.html. (Retrieved: 02/02/2024).

[69] Xi Tan, Zheyuan Ma, Sandro Pinto, Le Guan, Ning Zhang, Jun Xu, Zhiqiang Lin, Hongxing Hu, and Ziming Zhao. Sok: Where's the "up"?! a comprehensive (bottom-up) study on the security of arm cortex-m systems. In *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*, 2024.

[70] Telink Semi. Secure boot application note. https://doc.telink-semi.cn/index/index/detail/id/231/type/telinksales@telink-semi.com. (Retrieved: 02/02/2024).

[71] Texas Instruments. Over the air download (oad). https://software-dl.ti.com/lprf/simplelink_cc2640r2_sdk/1.35.00.33/exports/docs/ble5stack/ble_user_guide/html/oad/oad.html. (Retrieved: 02/02/2024).

[72] Texas Instruments. Simplelink cc13xx_cc26xx sdk documentation. https://software-dl.ti.com/simplelink/esd/simplelink_cc13xx_cc26xx_sdk/7.10.02.23/exports/docs/Documentation_Overview.html. (Retrieved: 02/02/2024).

[73] Texas Instruments. Simplelink msp432 software development kit. https://dev.ti.com/tirex/explore/node?node=A__AC4fea1GvZOsPa2Xi4z3Gw__com.ti.SIMPLELINK_MSP432E4_SDK__J4.hfJy__LATEST. (Retrieved: 02/02/2024).

[74] Texas Instruments. Tms320c6000 assembly language tools v8.3.x, description of object formats. https://downloads.ti.com/docs/esd/SPRUI03/#viewer?document=%257B%2522href%2522%253A%2522%252Fdocs%252Fesd%252FSPRUI03%2522%257D&url=description-of-the-object-formats-stdz0792390.html%23STDZ0792390. (Retrieved: 02/02/2024).

[75] The Computer Security Group at UC Santa Barbara. Repository for monolithic firmware blobs. https://github.com/ucsb-seclab/monolithic-firmware-collection, 2022. (Retrieved: 02/02/2024).

[76] Tuya Inc. Tuya mqtt topics. https://developer.tuya.com/en/docs/iot/MQTT_Topic?id=Kbt4ezpeko2rz. (Retrieved: 02/02/2024).

[77] Ubisys Technologies. Ubisys technologies. https://www.ubisys.de/en/main-page/. (Retrieved: 02/02/2024).

[78] Xueqiang Wang, Yuqiong Sun, Susanta Nanda, and XiaoFeng Wang. Looking from the mirror: Evaluating IoT device security through mobile companion apps. In *28th USENIX Security Symposium*, 2019.

[79] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. Firmxray: Detecting bluetooth link layer vulnerabilities from bare-metal firmware. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[80] Philipp Wendler, Karlheinz Friedberger, and George Karpenkov. Javasmt. https://github.com/sosy-lab/java-smt. (Retrieved: 02/02/2024).

[81] Ruotong Yu, Francesca Del Nin, Yuchen Zhang, Shan Huang, Pallavi Kaliyar, Sarah Zakto, Mauro Conti, Georgios Portokalidis, and Jun Xu. Building embedded systems like it's 1996. *arXiv preprint arXiv:2203.06834*, 2022.

[82] Yin Zhang and Vern Paxson. Detecting backdoors. In *9th USENIX Security Symposium*, 2000.

[83] Wei Zhou, Yan Jia, Yao Yao, Lipeng Zhu, Le Guan, Yuhang Mao, Peng Liu, and Yuqing Zhang. Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms. In *28th USENIX Security Symposium*, 2019.

[84] Lipeng Zhu, Xiaotong Fu, Yao Yao, Yuqing Zhang, and He Wang. Fiot: Detecting the memory corruption in lightweight iot device firmware. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, 2019.

[85] ZigBee Alliance. Zigbee cluster library specification. https://zigbeealliance.org/wp-content/uploads/2021/10/07-5123-08-Zigbee-Cluster-Library.pdf, 2019. (Retrieved: 02/02/2024).

[86] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy*, 2019.

[87] 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 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[88] Zyte. Scrapy, a fast high-level web crawling & scraping framework for python. https://scrapy.org/. (Retrieved: 02/02/2024).

A Lists of Keywords and File Extensions

Table 11 and Table 12 list the firmware-related keywords and file extensions used in *OTACap*.

Keyword	Regular Expression	Description
firmware	(?i)firmware	Firmware
download	(?i)download	Download
release	(?i)release	Release
image	(?i)image	Image
update, upgrade	(?i)update (?i)upgrade	Update
fw	(?i)(?<![a-zA-Z])fw(?![a-zA-Z])	Firmware
ota	(?i)(?<![a-zA-Z])ota(?![a-zA-Z])	Over-The-Air
oad	(?i)(?<![a-zA-Z])oad(?![a-zA-Z])	Over-The-Air Download, mainly TI
dfu	(?i)(?<![a-zA-Z])dfu(?![a-zA-Z])	Device Firmware Update, mainly Nordic
ble	(?i)(?<![a-zA-Z])ble(?![a-zA-Z])	Bluetooth Low Energy
Consecutive hex values	[a-fA-F0-9]{16}	File content hash value as file names

Table 11: Keywords used in firmware collection.

Format	Extensions	Description
Intel HEX	.hex, .mcs, .int, .ihex, .ihe, .ihe, .ihx, .h80,.h86, .a43, .a90, .hxl-.hxh, .h00-.h15, .p00-.pff, .obj, .obl, .obh, .rom,.eep, .bex	Intel hex object file format conveys binary informationin in ASCII text form. Each text line is called a record that represents machine code/data and its address information
Motorola S-Rec	.s19, .s28, .s37, .s1, .s2, .s3, .s, .sx, .srec, .exo, .mot, .mxt	Similar to Intel HEX and created by Motorola
TI-TXT	.txt	Similar to Intel HEX and created by TI
Tektronix	.tek	Similar to Intel HEX and created by Tektronix
Cypress	.cyacd, .cyacd2	Proprietary file format developed by Cypress, now acquired by Infineon
CSR	.csr	The firmware format used in CSR BlueCore bluetoothchips. Developed by Cambridge Silicon Radio and now acquired by Qualcomm
RSU	.rsu	Renesas Secure Update format
UF2	.uf2	A file format developed by Microsoft that is particularly suitable for flashing microcontrollers
MISC.	.bin, .fw, .img, .upg	File extensions generally selected by manufacturers

Table 12: File extensions used in firmware collection.

B An Example of URL Analysis using LLM

When crawling the initial URLs from the app `com.lorexcorp.lorexping`, our crawler captured the following text (actual URL redacted).

```
# FTP tag|FTP server|FTP username|FTP password|
FTP|[redacted]|upgrade|upgrade|
# clientModelID|FW version|FW filename|Config version|Config
filename|
G1|030509|IC717w-gen-030509-n.f|2|g1.cfg|
G2|030418|IC717g-gen-030418-n.f|2|g2.cfg|
...
```

We developed a *system prompt* to extract information.

```
I have some unstructured text. It might contain information about
firmware distribution URLs with credentials. Can you extract such
information, including the protocol, server address, username
and password? If you can, please format your answer in JSON.
Otherwise, just say no.
...
```

LLM returned the following correct result.

```
After analyzing the text, I was able to extract the following
information about firmware distribution URLs with credentials:

```json
[
    {
        "protocol": "FTP",
        "server_address": "[redacted]",
        "username": "upgrade",
        "password": "upgrade"
    }
]
```
```

CVE	Version	Component	FW #
CVE-2022-35623	v4.2.0-v5.0.0	Mesh	1
CVE-2022-35624	v4.2.0-v5.0.0	Mesh	1
CVE-2019-12588	v2.2.0-v3.1.0	WiFi (Beacon)	45
CVE-2019-12586	v2.0.0-v4.0.0	WiFi (WPA2)	70
CVE-2019-12587	v2.0.0-v4.0.0	WiFi (WPA2)	70
CVE-2020-12638	v4.0.0-v4.2.0	WiFi (WPA2)	74
CVE-2020-13594	v4.0.0-v4.2.0	BLE	13
CVE-2020-13595	v4.0.0-v4.2.0	BLE (HCI)	4
CVE-2020-16146	v2.0.0-v4.0.1*	BLE (Blufi)	3
CVE-2021-28135	v4.0.0-v4.4.0	Bluetooth (LMP)	46
CVE-2021-28136	v4.0.0-v4.4.0	Bluetooth (LMP)	46
CVE-2021-28139	v4.0.0-v4.4.0	Bluetooth (LMP)	46
CVE-2021-33028	v4.0.0-v4.2.0	WiFi (QoS)	94
CVE-2021-34173	v4.0.0-v4.2.0	WiFi (Beacon)	4
		Total (deduplicated)	191

*: v3.1.8, v3.2.4, and v3.3.3 are not subject to the CVE since these minor versions backported the patch released in v4.0.2.

Table 13: Detected N-day Vulnerabilities. Except for the first two CVEs found in nRF5-Mesh [48] and the third found in ESP8266_NONOS_SDK [24], all other CVEs were found in ESP-IDF [25].

C List of N-day Vulnerabilities

We identified 14 N-day vulnerabilities across 191 firmware samples. The details including the affected SDKs, versions, and components are shown in Table 13.