FRAMED: Toward Automated Identification of Embedded Frameworks in Firmware Images

Jorik van Nielen[1], Andreas Peter[2], and Andrea Continella[1]

[1] University of Twente, The Netherlands
[2] University of Oldenburg, Germany
{j.j.vannielen,a.continella}@utwente.nl
andreas.peter@uol.de

Abstract. In the era of the Internet of Things, firmware security analyses have become tremendously important to protect networks and guarantee safety-critical operations. Indeed, the firmware running on smart devices (which are increasingly adopted also in critical infrastructures) often contains security vulnerabilities, and delivering timely updates proved to be challenging, both from a technical perspective and due to a lack of support from device vendors. In particular, firmware images present difficulties that hinder automated analyses and patching, mostly because their code and data are opaquely intermixed and squashed together on top of embedded development frameworks. In this paper, we propose a new lightweight approach to automatically analyze firmware images and identify the embedded frameworks they are built upon. Our approach facilitates reverse engineering, reducing the scope for security analyses and assisting the vulnerability detection and patching process of embedded devices. We implement our approach in FRAMED, and we evaluate it on a dataset of 536 firmware images from different devices and vendors. Our system identifies embedded frameworks with an accuracy of 83%, and we perform a case study to combine FRAMED with an existing patch injection framework, demonstrating to be a helpful and effective tool for security analysts and reverse engineers.

Keywords: Embedded Frameworks · Firmware · Reversing · IoT

1 Introduction

The adoption of Internet of Things (IoT) devices has grown explosively over the last decade [31]. In fact, IoT devices present an affordable and accessible option to add connectivity to physical objects that were not able to cross the barrier to the digital world before. This feature is attractive for both individuals and organizations, and manufacturers meet this demand with a wide variety of devices, such as thermostats, lighting, and smoke detectors, but also more critical ones such as pacemaker monitors and smart locks. Moreover, smart devices are increasingly adopted within critical infrastructures, where they enable the remote management of cyber-physical systems, such as industrial robots and actuators [11].

While the extra connectivity of IoT devices brings value to the users, it also opens up new attack vectors. Attacks on smart devices over the Internet or Bluetooth are nowadays a real and practical scenario [11]. If vulnerable, threat actors can mount attacks against devices to alter their behavior or take full control of them. In fact, the number of cyber-attacks mounted against IoT devices has almost tripled in the last two years [42]. With the popularity of IoT devices in vital industrial processes and critical infrastructures, the security of their firmware is an area of utmost importance. To further highlight the state of (in)security in smart devices, new advances in vulnerability discovery for embedded firmware [6,7,26,35,33,40] has significantly increased the rate of security advisories released and disclosed by researchers and practitioners.

However, the disclosure of vulnerabilities to the vendors does not guarantee a fast deployment of patches to ensure the security of all vulnerable devices. One recurring cause of the long patch latency is vendors dropping support for older devices [44,16]—besides, update mechanisms are themselves often vulnerable [19,45]. Another critical cause is the complexity of producing and testing patches. Many IoT devices run custom firmware, and generating a patch often requires extensive testing. Since a low patch latency is crucial for IoT devices, especially when deployed in critical infrastructures, researchers investigated approaches to introduce quick "hot" patches before a fully-fledged patch is released by the vendor [16,17,30].

Third-party patching for embedded firmware without the cooperation of the vendor is a significantly challenging task that is made even harder by the wide adoption of development frameworks (known as *embedded frameworks*)—i.e., embedded-specific development frameworks that abstract the development and provide OS-like features to firmware developers. During compilation, the embedded framework and user application are squashed together into a single image (also known as *blob*). Such firmware blobs typically do not contain debugging symbols, are structured in custom binary formats, and have no clear distinctions between code and data segments, making automated analyses hard. Besides, device manufacturers usually do not disclose that their devices are vulnerable before releasing a patch. Therefore, potentially affected organizations need to track if their devices are vulnerable to newly disclosed vulnerabilities, especially taking into account any vulnerabilities that affect libraries and frameworks that the device firmware relies on. Thus, knowing what frameworks firmware images are built upon is of great use for security management. Moreover, with information about the adopted embedded framework, security analysts and reverse engineers know what functions to expect in the firmware. This helps analysts distinguish framework functions from user application functions, which, since frameworks introduce hundreds of functions, significantly eases the reverse engineering process. All in all, the automated identification of the embedded frameworks within firmware images can be of great use to reverse engineers for security analyses. In the academic community, the process of uncovering components and dependencies within software is part of a broader field known as Software Composition Analysis (SCA).

While existing SCA approaches have shown great results, they are unfortunately not directly applicable to the firmware domain. Binary-to-binary SCA approaches, such as FirmSec [49] and Asm2Vec [12], require compiled embedded framework object files. However, the compilation of embedded frameworks involves manual efforts and has shown to be time-consuming at scale [38], hindering automation. Binary-to-source techniques, such as BinaryAI [20] and OSSPolice [13], do not require the compilation. However, these approaches leverage information stored in the binary that is not always present in firmware blobs, such as exported function names. Besides, embedded frameworks often make use of macros to suit the many compilation targets, which can highly impact the semantics on a function level. Current binary-to-source SCA techniques do not account for this dynamic setting.

In this work, we propose a lightweight approach to analyze firmware images and identify the embedded frameworks they are built upon. By identifying the embedded frameworks, our approach assists security analyses by revealing the main firmware component and gives reverse engineers a starting point for recovering function symbols in firmware images. Additionally, our approach can assist in the patching of firmware. For instance, state-of-the-art firmware patching approaches such as HERA [30] and Shimware [16] require a hooking location where additional checks can be performed. Currently, the hooking location has to be manually provided by the analyst. Our approach eases the process of finding a hooking location by giving security analysts information about the structure of the adopted frameworks and facilitating the function symbols recovery process.

To identify embedded frameworks, our approach leverages the strings embedded within firmware blobs to produce fingerprints—the intuition is that strings carry significant semantic information to characterize software components. In fact, previous work showed the effectiveness of using strings for library detection [27,18,46,20,13], but to the best of our knowledge, we are the first to apply this method to embedded framework identification. More specifically, our approach first recognizes and filters out irrelevant strings, and then compares the remaining ones to string literals identified in the source code of embedded frameworks. To improve the accuracy, we take into account the existence of framework clones and third-party library reuse.

We implement a proof-of-concept of our approach in FRAMED. In our experiments, FRAMED shows promising results, correctly identifying 83% of the frameworks in a labeled dataset of 536 firmware images, spanning eight different frameworks. Moreover, for four of the most popular embedded frameworks, including Mbed [25], our system achieves correct identification for all of the tested firmware samples.

To demonstrate the real-world applicability and practicality of FRAMED, we describe a real-world use case. We combine FRAMED with Shimware [16], an existing patch injection tool, to identify the framework adopted by a firmware image and use this information to find a suitable location for a patch, ultimately fixing a security flaw in the firmware image.

In summary, our contributions are as follows:

- We propose a new lightweight, string-based approach to characterize embedded frameworks.
- We implement our approach in FRAMED, a pipeline to automatically identify the frameworks used in real-world firmware images.
- We evaluate our approach on a dataset gathered from four prior works, containing 536 labeled firmware samples. Additionally, we demonstrate a use-case where we combine FRAMED with an existing patch injection tool to ease the patching process.

In the spirit of open science, our source code is publicly available at `https://github.com/utwente-scs/frameD`.

2 Background & Challenges

In this section, we provide background information on the firmware structure and types, and how such structure and the adoption of embedded frameworks affect the analysis and reversing processes. Then, we present the challenges in the firmware domain for embedded framework identification.

2.1 Firmware Analysis

Firmware Structure and Types. The low processing power and long battery lifetime requirements of embedded devices make it sometimes infeasible to run full operating systems (OS) like Linux or Windows. Instead, firmware features embedded operating systems, or even embedded frameworks without a logical separation between kernel and application code. In the literature, these two types of firmware are referred to as Type-2 and Type-3 firmware, respectively [29]. While detecting the use of Linux in firmware images is an explored topic [4], the identification of Type-2 and Type-3 frameworks is not. In this paper, we refer to both embedded operating systems and embedded frameworks as embedded frameworks.

Embedded Frameworks. To take away the complexity of the hardware programming from the programmer, various abstractions are used in practice. As a basis, Hardware Abstraction Layers (HALs) provide functions to developers to perform hardware operations without knowledge of the underlying hardware. HAL functions are small functions that specifically target one hardware configuration. In most cases, HAL functions are provided by chip vendors directly. Besides ease of communication with the hardware, developers might want to simplify the development further with a higher level of abstraction. Another highly demanded feature is OS-like capabilities such as scheduling of processes or tasks. Embedded frameworks provide such functionalities in varying degrees of sophistication. Some frameworks, like FreeRTOS [15], solely provide a couple of simple operating system functionalities, while, for instance, RIOT [34] adds more functionalities such as a full system shell and a complete file system implementation.

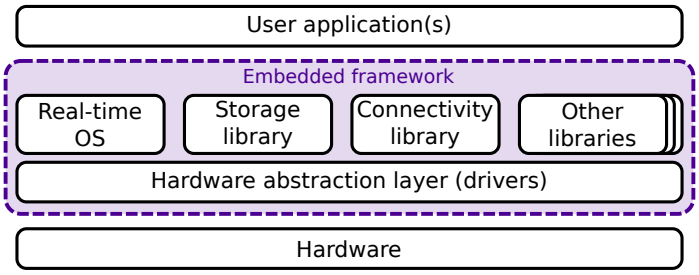

Fig. 1 – An embedded framework consists of several components or a subset thereof. The hardware abstraction layer features hardware-specific functions that provide access to hardware resources. The real-time OS component enables process scheduling and resource management. A storage library adds file system functionality to the firmware. A connectivity library implements a network protocol, such as Bluetooth. Additionally, embedded frameworks can include libraries for cryptographic functions, data handling, etc. The user application is not part of the embedded framework. It implements the process logic and operates on top of the provided embedded framework functionality.

To simplify the development process even further, developers turn to libraries for performing higher-level tasks. Popular libraries provide functionalities such as communication protocols (e.g., HTTP), data processing (e.g., compression), and user interface capabilities. In this research, we consider *embedded frameworks* in a broad sense as any embedded-specific development framework that aims to abstract the development and provide OS-like features to firmware developers. Some frameworks purely implement the HAL with the option to add third-party libraries, while others feature a more complete embedded OS with ad-hoc libraries. Figure 1 visualizes the different components of the embedded framework in the context of the complete firmware image.

Firmware Analysis and Reversing. Non-general-purpose OS-based firmware is difficult to analyze and patch. In fact, the projects are mostly compiled into a single executable file, without any distinction between data, operating system functions, and user application functions. Also, firmware images are compiled without debugging symbols. Thus, analyzing such firmware is a daunting task. The use of embedded frameworks further increases the difficulty of reversing firmware images. Not only introduce frameworks more functions, they also add custom abstraction layers and sometimes apply object-oriented programming techniques. Thus, it becomes more labor-intensive to determine the semantics of functions and identify the relevant ones, creating challenges for automation. However, if the framework can be identified, the process becomes much simpler. Equipped with knowledge of the framework, the analyst can consult the documentation of the framework to learn more about the firmware structure. Additionally, the analyst can use existing SCA techniques to locate important functions in the firmware image.

2.2 Challenges for Firmware Composition Analysis

The properties of the firmware domain pose several challenges for the identification of embedded frameworks.

Firmware blobs are stripped and use opaque binary formats. Existing binary-to-source SCA techniques leverage various features for fingerprinting. Two examples are the exported function names and the data present in data segments. However, with the opaque binary format used for firmware images, many of these normally easily retrievable fingerprinting sources are either very hard to extract or not present at all. For instance, OSSPolice [13] uses exported function names to identify present libraries. This information is however not present in firmware blobs. To successfully identify frameworks in firmware images, the solution must rely solely on features that can be reliably extracted.

Embedded framework functions are semantically different, depending on the build target. Embedded frameworks have been designed to support many architectures and microcontrollers. To achieve this portability, embedded frameworks widely use macros to differentiate the behavior for each platform. Unfortunately, existing binary-to-source SCA techniques do not take this into account. As a result, many of the features used by the state-of-the-art cannot be used reliably for identifying the frameworks present in firmware images. For instance, SCA techniques frequently use the number of basic blocks contained in a function as a fingerprint for that function. However, in the firmware domain this might not be a reliable feature. For example, TCP/IP implementation functions are semantically different for boards that support IPv6, and as a result have different basic block counts. Consequently, matching functions based on features such as basic blocks is more challenging in the domain of embedded frameworks.

Embedded frameworks are hard to compile in an automated fashion. Binary-to-binary SCA techniques require the compilation of all components that you want to identify. In our scenario, this requires the compilation of all embedded frameworks. Previous works mentioned the extensive manual effort required for the compilation task [38]. With hundreds of embedded frameworks available on GitHub and many of them receiving continuous updates, manual compilation becomes a significant limitation that hinders automation.

Embedded frameworks might contain third-party libraries. It is common for embedded frameworks to use open-source libraries to facilitate certain features. For example, Contiki [8] uses a third-party library for the FAT file system implementation. A solid approach to identifying embedded frameworks should ensure that the identification of a third-party library does not result in the incorrect identification of the embedded framework.

Existing SCA approaches have not been designed to address the challenges that occur when identifying embedded frameworks. Therefore, a new, embedded framework-specific SCA technique is required.

3 Methodology

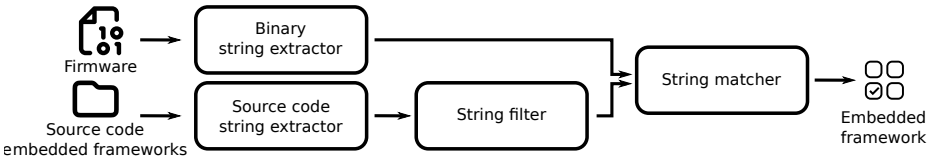

Fig. 2 – Simplified representation of FRAMED. The arrows represent the flow of data between the different components.

We present FRAMED, our approach to identifying embedded frameworks in firmware images. At a high level, FRAMED takes as input an embedded firmware image and a set of embedded frameworks (including their source code) and evaluates whether the given image is built on top of any of the user-provided frameworks. Figure 2 shows a high-level overview of the complete pipeline.

3.1 Frameworks Collection and Processing

The first step of our methodology is to build a large collection of embedded frameworks to identify within firmware images. Embedded frameworks are generally released as open-source code to the public, and they can be easily found online. Two popular examples are Mbed [25] and RIOT [34]. After building a satisfactory complete collection, the `Source code string extractor` extracts the hardcoded strings from each project.

As a second step, the `String filter` applies multiple filters to the strings extracted from the open-source projects. A first filter removes common and short strings. Very short strings are removed since they are not strong indicators and would increase the number of false positives. They include omnipresent strings such as a single format string specifier, e.g. *"%d"*. Moreover, very common strings, like *"error"*, should have a lower impact on matches than for example *"Error: new serial object is using same UART as STDIO"*, a string only found in Mbed. Furthermore, some strings are present across firmware samples, also when no framework is used. Examples are *"0123456789abcdef"* and *"localhost"*. Thus, we filter out popular strings to reduce false positives. Additionally, FRAMED filters out strings that are found in common firmware libraries. Since many open-source frameworks use common third-party libraries in their source code, these strings could result in false positives when a firmware image uses a library that is also used by an embedded framework. By filtering out the strings found in common embedded libraries the number of false positives is reduced.

While the construction of a complete string collection of all embedded frameworks is easily automated, it is still a time-consuming process. Luckily, this process only has to be conducted once. Thereafter, we can use the local string database to evaluate any number of firmware samples.

3.2 Framework Identification

After the construction of the embedded framework string collection, the `Binary string extractor` extracts the strings from the given firmware image. The extracted firmware strings and filtered source code strings are then handed to the `String matcher`. For each framework, the `String matcher` computes how many exact string matches are present. An additional step is required to accurately identify embedded frameworks. This is because many open-source frameworks are clones of more popular frameworks, with minimal changes. These clones often contain the same signatures as the original repository, impacting the performance of our tool. To limit the negative impact of clones, we cluster the potentially matching frameworks based on whether the matching strings are the same. A `String score` is calculated for each cluster using Equation (3). To calculate the `String score`, we first calculate the `String popularity` for each string present in the cluster, by counting how many clusters in total contain this string (Equation (2)). The `String score` of the cluster is then calculated as the sum of the inverse `String popularity` of all the strings present in the cluster. The `String score` ensures that unique strings have a bigger impact on the final result, since unique strings are strong indicators for an embedded framework. Finally, the cluster with the highest `String score` is selected as the matching cluster. If the `String score` of the selected cluster is higher than a predetermined threshold, the most popular framework in that cluster is returned as the identified framework.

$$I(\text{string s}, \text{cluster c}) = \begin{cases} 1 & \text{if } s \in c \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

$$\text{String popularity}(\text{string } s) = \sum_{c \in \text{clusters}} I(s, c) \tag{2}$$

$$\text{String score}(\text{cluster } c) = \sum_{\text{string } s \in c} \frac{1}{\text{String popularity}(s)} \tag{3}$$

Employing the clustering technique in a more general SCA approach might result in clustering together two libraries that are both present, and returning only one. In the domain of embedded framework identification such cases of false negatives are not expected, since only one framework is used to build the firmware image around. The embedded framework can be seen as the operating system of the firmware image, thus it is not practical to use multiple frameworks in the same image. This is why FRAMED looks for one framework, and not multiple ones.

3.3 Implementation details

To implement the `Source code string extractor`, FRAMED uses the `ANTLR` lexer [1]. For our proof-of-concept implementation, we provide `ANTLR` with C and C++ grammars. However, FRAMED can easily be extended to other languages by

providing additional grammars. For example, adding Rust support only requires providing a Rust grammar, which can be easily found online.

For the implementation of `Binary string extractor`, FRAMED uses the Linux utility `strings`. This utility scans through the binary looking for subsequent bytes that represent valid ASCII characters, and returns them as strings.

In the final step of the `String matcher`, FRAMED clusters similar frameworks together. We then use the GitHub star count as a popularity metric to select the final framework from the cluster.

4 Evaluation

In this section, we describe the process we follow to build our database of frameworks, and we evaluate the performance of our tool in identifying frameworks in embedded firmware.

4.1 Database Creation

To assess the performance of FRAMED, we start by constructing the local string database. We use the GitHub API to query for open-source embedded frameworks and collect their source code. To search for embedded frameworks, we use six different queries:

- Embedded operating system;
- Operating system microcontrollers;
- Internet of Things operating system;
- OS IoT;
- OS Internet of Things;
- RTOS.

These specific queries have been selected to ensure that all firmware framework projects we found on GitHub through a manual search are present in the final list. For each query, we search in the description of all GitHub repositories and select the top 100 C/C++ repositories based on star count. We clone the resulting 404 repositories locally. We add one additional framework manually, namely the Arduino framework. While Arduino is a widely adopted framework, it cannot be found on GitHub using our queries since it only has a very short description. The `String filter` reduces the number of strings extracted from the source codes in multiple steps. First, we remove string duplicates and decode escape sequences, to match how strings are stored in the firmware images. Then, we remove strings that are less than 6 characters long. This filters out strings that do not contain a lot of information about the repository, such as *"bytes"*, *"done"*, and also single characters. Next, we filter out the most popular strings. The popularity of strings in our dataset is shown in Figure 3. In our dataset, we notice many strings occurring 22 times. This turned out to be strings from Mbed clones, which we do not want to filter out. This is why we decided to filter out strings with more than 22 occurrences. A few strings that are filtered out this way are *"failed"*,

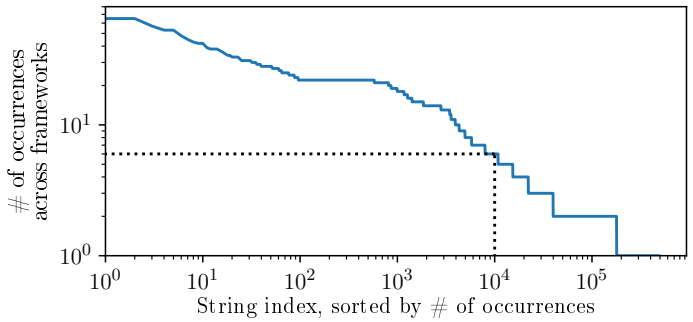

Fig. 3 – String popularity in our framework string database, with double logarithmic axes. E.g., as depicted with the black dotted line, the 10,000th most frequently occurring string is present in 6 embedded frameworks in our collection.

"abcdefghijklmnopqrstuvwxyz", and *"%s%s%s"*. Filtering out popular strings has shown to be effective in prior works [27]. The final step we perform is filtering out strings from common libraries that are used for embedded systems. We found that many frameworks use common third-party libraries. Without filtering out library strings, FRAMED would raise many false positives. To filter out library strings, we locally cloned all library repositories listed in a popular reference collection called "awesome-embedded-software" [3]. We extracted the strings in a similar fashion to how we extracted the framework strings.

4.2 Pipeline Performance

We evaluate the performance of FRAMED on a dataset of 536 labeled firmware images. We construct the dataset by combining the datasets from four prior works. We specifically select collections with firmware samples that use popular embedded frameworks, to ensure the experiments are similar to real-world scenarios. Our dataset features eight different embedded frameworks and a variety of target microcontrollers. Besides, twelve of the samples do not use a framework. Each firmware sample has either been labeled with a framework by the source, or we manually reverse engineer the firmware images to determine the adopted frameworks. The complete list of sources, frameworks, and sample counts of the firmware samples is shown in Table 1.

To examine the capabilities of our pipeline, we measure its performance on the 536 labeled firmware images. The framework identification takes 3 seconds per firmware sample on average, in a single-threaded environment.

Figure 4 shows the performance for the framework detection compared to different string score thresholds. At a threshold of 3.5, 83% of the samples are

Dataset	Framework	No. Samples
Feng et al. (P2IM) [14]	Arduino [2]	5
	RIOT [34]	1
	FreeRTOS [15]	1
	None	3
Clements et al. (HALucinator) [7]	Contiki [8]	2
	None	9
Scharnowski et al. (Fuzzware) [35]	Zephyr [48]	10
	Contiki-NG [9]	2
	Mbed [25]	10
Shen et al. [37]	Zephyr [48]	193
	Mbed [25]	32
	NuttX [32]	188
	FreeRTOS [15]	80
Total		536

Table 1 – Sources, frameworks, and counts of firmware samples that we used in our evaluation.

labeled correctly. At a lower threshold of 2, 18% of the framework identifications are false positives, while at a higher threshold of 5 results in only 1% false positives, but more than 20% false negatives. For Mbed the accuracy is 100% at lower thresholds. Only at thresholds starting from 6, do false negatives start occurring. Both NuttX and Zephyr achieve 90%+ accuracy at String score thresholds under 4. Interestingly, for both frameworks we notice that from the threshold of 4, false positives are reducing, and false negatives are increasing.

The detection results per framework and for three different threshold values are shown in Table 2.

Using a low threshold (e.g., 2) results in few false negatives for all frameworks. This can be a useful metric to detect the use of a framework. Although the wrong framework might be detected, the security analyst can use the result to further research the used framework. Samples that do not use a framework can be discarded early. Using a higher threshold such as 5, on the other hand, can be useful in scenarios where one wants to be sure that the detected framework is correct. For example, at the threshold of 6 only 4 of the 536 samples were incorrectly labeled with a framework. However, there are 110 false negatives at this high threshold. As a result, a firmware blob that has not been identified with a framework might still use a framework.

For the cases where the framework identification is incorrect, we see a few trends. First, we notice that some frameworks use a low number of hard-coded strings, limiting the effectiveness of FRAMED for these specific frameworks. As a result, Arduino and FreeRTOS are harder to detect with FRAMED, and have high false negative rates independent of the string-score threshold. On the other

	Framework	TPs	FPs	TNs	FNs
Threshold = 2	FreeRTOS	27	54	N/A	0
	Zephyr	193	10	N/A	0
	Mbed	42	0	N/A	0
	NuttX	171	17	N/A	0
	Arduino	0	5	N/A	0
	RIOT	1	0	N/A	0
	Contiki	2	0	N/A	0
	Contiki-NG	2	0	N/A	0
	None	N/A	11	1	N/A
	Total	438	97	1	0
Threshold = 3.5	FreeRTOS	27	35	N/A	19
	Zephyr	193	3	N/A	7
	Mbed	42	0	N/A	0
	NuttX	171	17	N/A	0
	Arduino	0	1	N/A	4
	RIOT	1	0	N/A	0
	Contiki	2	0	N/A	0
	Contiki-NG	2	0	N/A	0
	None	N/A	4	8	N/A
	Total	438	60	8	30
Threshold = 6	FreeRTOS	24	0	N/A	57
	Zephyr	174	0	N/A	29
	Mbed	41	0	N/A	1
	NuttX	167	3	N/A	18
	Arduino	0	0	N/A	5
	RIOT	1	0	N/A	0
	Contiki	2	0	N/A	0
	Contiki-NG	2	0	N/A	0
	None	N/A	1	11	N/A
	Total	411	4	11	110

Table 2 – The results of FRAMED when running over 536 labeled firmware samples, with different string score thresholds.

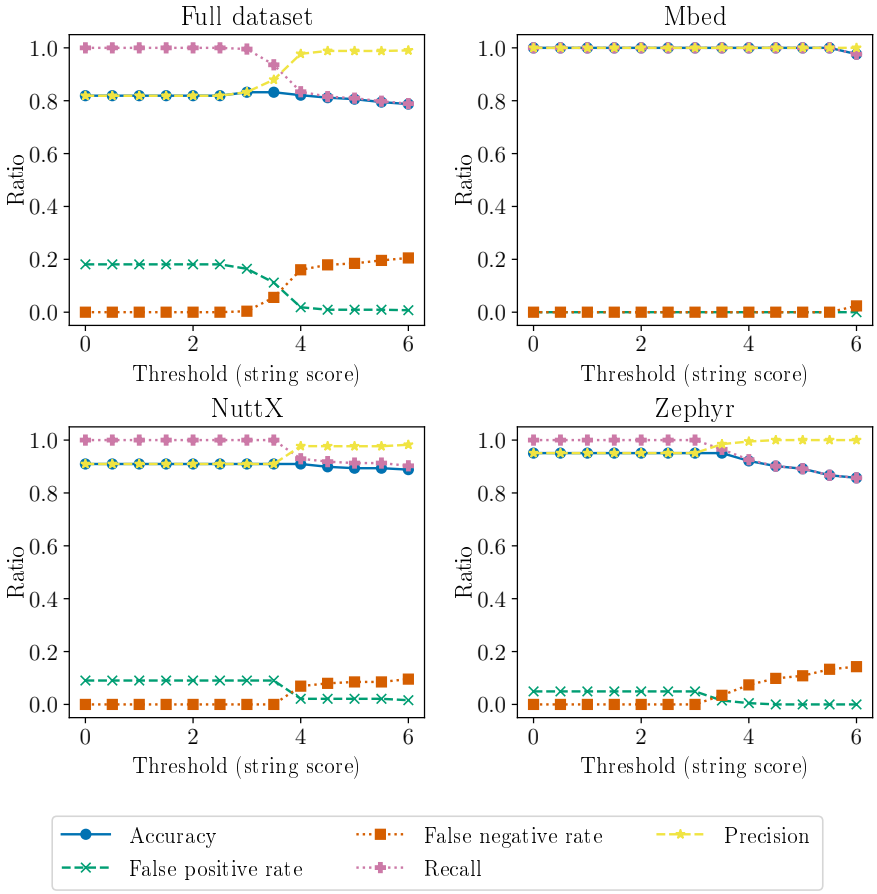

Fig. 4 – FRAMED detection performance metrics on the 536 firmware samples.

hand, larger frameworks that feature many strings are accurately identified by FRAMED, across firmware samples. Second, we observe a low number of false positives with a balanced threshold (3.5) in place. We manually investigated the false positive cases. We found that the matches mostly occur due to the use of libraries, which are not present in our library collection that is used for string filtering.

We recognize that using an absolute cut-off threshold is a rather naive method. However, we found that for our dataset, which is diverse in nature, it performs well. Nonetheless, we envision future research on the design of a more accurate approach to identifying embedded frameworks.

5 Case Study

We conduct a case study to demonstrate the application of FRAMED to assist in reversing and patching IoT firmware. Specifically, we focus on the Atmel 6LoWPAN Sender firmware image from the HALucinator paper [7] and study how FRAMED can be integrated with Shimware [16] to enhance its I/O function tracking, ultimately leading to patch injection. Shimware uses an approach called IOFinder to determine I/O functions in firmware images. IOFinder combines static and dynamic analysis to find functions that perform I/O functions. One limitation of IOFinder is that it is not designed to process firmware images that use a larger embedded framework. In the case of the Atmel image, due to the use of a large embedded framework Contiki, IOFinder is unable to detect several relevant I/O functions. As a result, Shimware fails to locate the function where the patch should be introduced [16].

In the practical scenario where we want to patch the Atmel 6LoWPAN Sender firmware image, we can leverage FRAMED to discover the use of the Contiki embedded framework. As we detect the use of a large framework, we know that the results of IOFinder are likely to be incomplete. Therefore, we decide not to use IOFinder. Instead, to recover the function symbols, we can now compile a Contiki example project and use existing binary-to-binary SCA techniques to recover symbols in the binary. For this example, we use Libmatch from HALucinator [7] and reveal the locations and names of 170 functions. Next, the analyst looking to apply the patch can now go over the functions, determine which functions handle the relevant input, and subsequently identify the location for Shimware to inject the patch. Depending on the specific vulnerability, it is possible to apply patches in higher-level functions than the I/O functions. For example, assume that we want to apply a patch to incoming network traffic to filter malicious payloads. We can locate the `tcpip_input` function, which is a Contiki function called by the network driver when a new TCP/IP packet is received, and which represents a perfect hook for our patch. With the knowledge of the address of the `tcpip_input` function, in combination with the Shimware tooling, we insert the patch into the original firmware image.

6 Discussion

We discuss the limitations of FRAMED and directions for future research.

Frameworks with few strings. The fundamentals of our approach rely on the presence of strings in the framework and in the firmware image. Generally, all frameworks do contain these strings, but this does not guarantee that they will be present in the final firmware image. To reduce the final image size, unused functions are often left out during the linking phase. As a result, also the strings used in these functions are not included in the final firmware image. This explains why FRAMED has low detection rates for samples using some specific frameworks. An example is FreeRTOS [15], a minimal framework that does not

contain much more than HAL functions and some basic scheduling. Since minimal frameworks mostly provide the HAL functions, other approaches are potentially a better fit to detect these types of frameworks. Compiling HAL functions automatically is less challenging than more advanced frameworks, as has been shown and used for a similar approach before (e.g., Libmatch [7]). Thus, one could automatically compile a database of HAL objects and apply Libmatch or a similar approach to detect the HAL functions in the firmware. In the context of firmware patching, IOFinder as used by Shimware [16] is a technique to find relevant IO functions, but only works for smaller frameworks. To summarize, FRAMED provides an approach to aid reverse engineers recover symbols and assist in the patching process of larger embedded frameworks, whereas prior works have shown approaches that are effective for smaller embedded frameworks.

Completeness of Framework Database. To construct our list of frameworks, we used various GitHub queries. While this covers a large portion of the available frameworks, it is not complete. For instance, Software Development Kits (SDKs) for embedded firmware often come with their own frameworks preinstalled. While these frameworks are often open source, you will not necessarily find them on GitHub, or we found them to have short descriptions and are hard to query for. FRAMED does currently not have the capability to detect these frameworks. However, they can be added manually to the database with ease. One other set of frameworks that are currently absent and more difficult to add are private frameworks developed and used by device manufacturers.

Completeness of Library Database. To reduce false positives, we filtered out strings present in commonly used third-party libraries. Selecting the libraries for this list has to be done carefully, as it can easily remove true positives. We chose to use a curated list, as described in Section 4. However, in our evaluation, a false positive occurred due to a library missing from our collection. The accuracy of FRAMED relies partly on creating a good list of libraries.

Automation in Function Matching. Our pipeline is fully automated when it comes to creating the string database and identifying the used frameworks. However, the next step in the pipeline would be to recover the symbols fully automatically. To achieve this goal, we have explored the idea of automated compilation of the framework, followed by binary-to-binary SCA. However, fully automating this process requires more research. One of the main challenges for achieving this automation is the nature of the framework projects. They have been developed to ease the development for embedded devices, and provide configuration files and build specifications to adjust the environment for specific hardware. Compiling the object files without the configuration enabled will often fail. Thus, it is required to manually set up a build environment. The alternative is binary-to-source matching. While this is a more complicated approach and no existing projects match our requirements, we see this as the most feasible way forward. Additionally, binary-to-source matching could also be used to resolve the limitations of the lower performance for frameworks that use fewer strings.

Compressed firmware images. Like any other static firmware analysis approach, FRAMED struggles with compressed firmware images, as compression

can result in failure to extract strings from firmware images. To resolve this, one could integrate FRAMED with existing solutions that unpack known formats (e.g., Binwalk [5]).

7 Related work

Many previous works have been conducted with similar goals to ours. In this section, we discuss their benefits, downsides, limitations, and assumptions.

There have been various efforts to simplify the reverse engineering and analysis of firmware. Muench et al. [29] focussed on grouping firmware images into types of operating systems. Thomas et al. [43] use both static and dynamic analysis to improve the disassembly quality. Firmline [4] provides a pipeline to automatically detect the architecture and base addresses of non-Linux-based firmware images. With FRAMED, we contribute to the area of firmware analysis by identifying embedded framework usage.

Several studies have been conducted in the area of patching of embedded devices. HERA [30] and RapidPatch [17] both apply hot-patches to running embedded devices, without rebooting the device. However, the focus of both HERA and RapidPatch is solely on how to inject the patch, not where. Dispatch [23] proposes an approach to patching the firmware of Robotic Aerial Vehicles. They target controller component functions and locate them through the identification of trigonometric functions. This approach does not transfer to a more generic case of function identification. Shimware [16] presents a set of tools to ease the patching of non-Linux based firmware images. The main contributions are the automated locating of I/O functions, finding a location to insert the patch without breaking the functionality of the device, and finding and fixing firmware validity checks. To locate the I/O functions, a hybrid approach combining static and dynamic analysis is used. While it has proven effective for general firmware images, it tends to miss functions if the firmware uses a larger embedded framework. Moreover, the identified I/O functions are not labeled and still require manual reverse engineering to reveal their exact purpose.

Software Component Analysis can be divided into approaches using binary-to-binary matching or binary-to-source matching:

Binary-to-binary SCA. Libmatch [7] combines static and dynamic analysis to locate Hardware Abstraction Layer functions. Asm2Vec [12] uses machine learning to match functions across architecture and compiler optimization levels. FirmSec [49] detects third-party component usage in both Linux-based and non-Linux-based firmware images. However, none of these approaches is directly applicable to the challenge of framework identification, due to the manual effort required to compile embedded frameworks.

Binary-to-source SCA. BAT [18] uses strings to get an estimate of code reuse, but does not address further challenges such as cloning. OSSPolice [13] leverages strings, in combination with exported function names to detect the reuse of open source software. Exported function names are however not present in firmware

images. Bigmatch [27] identifies the use of open-source libraries in binary executables solely based on strings, but does not take into account the presence of shared third-party libraries between repositories, nor does it tackle software clones that have not been marked as such on GitHub. B2SFinder [46] matches source code to binaries by using 7 binary features that can be extracted with extra knowledge of the binary format. Most of these features are however not present in firmware images.

When it comes to finding vulnerabilities in embedded firmware, various research directions have been explored. First, fuzzing with devices in the loop was a popular topic of research a couple of years ago [10,21,24,47]. While these approaches have good results, they are limited to small-scale experiments and analyses since the device has to be both available and set up. Recently, many rehosting approaches have been proposed. There is a clear divide between papers targeting Linux-based firmware images [16,41,22,28] and monolithic firmware images [7,14,50,35,39,36]. Most of these approaches focus on providing valid inputs for MMIO addresses in memory or hardware abstraction layer functions.

8 Conclusion

With FRAMED, we presented an automated pipeline to identify embedded frameworks within firmware images. FRAMED combines string matching and insights in embedded frameworks to provide meaningful semantic annotations that facilitate reverse engineering and security analyses. We evaluated the performance of FRAMED on a labeled dataset of 536 images, obtaining an accuracy of 83%. By changing the threshold, our tool can be adjusted to fit scenarios where either a low number of false positives or a low number of false negatives are required. We envision our tool being used to assist both reverse engineers and automated security pipelines that perform firmware analyses such as vulnerability discovery and patching.

Acknowledgements

We would like to thank our reviewers for their valuable comments and inputs to improve our paper. This work has been partially supported by the Dutch Ministry of Economic Affairs and Climate Policy (EZK) through the AVR project "FirmPatch" and by the project P6 (Open Technology Programme No. 20475) funded by the Dutch Research Council (NWO).

References

1. ANTLR. `https://www.antlr.org/`, accessed: 2024-02-14
2. Arduino. `https://www.arduino.cc/`, accessed: 2024-06-17
3. Awesome embedded resources for developers. `https://github.com/iDoka/awesome-embedded-software`, accessed: 2024-02-14

4. Balgavy, A., Muench, M.: Firmline: a generic pipeline for large-scale analysis of non-linux firmware. In: Proceedings of the Workshop on Binary Analysis Research (BAR) (2024)
5. Binwalk. https://github.com/ReFirmLabs/binwalk, accessed: 2024-02-14
6. Chen, L., Wang, Y., Cai, Q., Zhan, Y., Hu, H., Linghu, J., Hou, Q., Zhang, C., Duan, H., Xue, Z.: Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In: Proceedings of the USENIX Security Symposium (2021)
7. Clements, A.A., Gustafson, E., Scharnowski, T., Grosen, P., Fritz, D., Kruegel, C., Vigna, G., Bagchi, S., Payer, M.: HALucinator: Firmware re-hosting through abstraction layer emulation. In: Proceedings of the USENIX Security Symposium (2020)
8. Contiki. http://www.contiki-os.org/, accessed: 2024-06-17
9. Contiki-NG. https://www.contiki-ng.org/, accessed: 2024-06-17
10. Corteggiani, N., Camurati, G., Francillon, A.: Inception: System-wide security testing of real-world embedded systems software. In: Proceedings of the USENIX Security Symposium (2018)
11. Cyber risks to critical infrastructure are on the rise, https://news.microsoft.com/en-cee/2023/06/26/cyber-risks-to-critical-infrastructure-are-on-the-rise/, accessed: 2024-02-14
12. Ding, S.H., Fung, B.C., Charland, P.: Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: Proceedings of the IEEE Symposium on Security and Privacy (2019)
13. Duan, R., Bijlani, A., Xu, M., Kim, T., Lee, W.: Identifying open-source license violation and 1-day security risk at large scale. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2017)
14. Feng, B., Mera, A., Lu, L.: P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In: Proceedings of the USENIX Security Symposium (2020)
15. FreeRTOS. https://www.freertos.org/index.html, accessed: 2024-02-14
16. Gustafson, E., Grosen, P., Redini, N., Jha, S., Continella, A., Wang, R., Fu, K., Rampazzi, S., Kruegel, C., Vigna, G.: Shimware: Toward practical security retrofitting for monolithic firmware images. In: Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID) (2023)
17. He, Y., Zou, Z., Sun, K., Liu, Z., Xu, K., Wang, Q., Shen, C., Wang, Z., Li, Q.: RapidPatch: Firmware hotpatching for real-time embedded devices. In: Proceedings of the USENIX Security Symposium (2022)
18. Hemel, A., Kalleberg, K.T., Vermaas, R., Dolstra, E.: Finding software license violations through binary code clone detection. In: Proceedings of the Working Conference on Mining Software Repositories (2011)
19. Ibrahim, M., Continella, A., Bianchi, A.: AoT - attack on things: A security analysis of IoT firmware updates. In: Proceedings of the IEEE Symposium on Security and Privacy (2023)
20. Jiang, L., An, J., Huang, H., Tang, Q., Nie, S., Wu, S., Zhang, Y.: BinaryAI: Binary software composition analysis via intelligent binary source code matching. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE) (2024)
21. Kammerstetter, M., Platzer, C., Kastner, W.: Prospect: peripheral proxying supported embedded code testing. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2014)

22. Kim, M., Kim, D., Kim, E., Kim, S., Jang, Y., Kim, Y.: Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC) (2020)
23. Kim, T., Ding, A., Etigowni, S., Sun, P., Chen, J., Garcia, L., Zonouz, S., Xu, D., Tian, D.: Reverse engineering and retrofitting robotic aerial vehicle control firmware using dispatch. In: Proceedings of the ACM International Conference on Mobile Systems, Applications, and Services (MobiSys) (2022)
24. Koscher, K., Kohno, T., Molnar, D.: SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In: Proceedings of the USENIX Workshop on Offensive Technologies (WOOT) (2015)
25. Mbed. https://os.mbed.com/, accessed: 2024-02-14
26. Melotti, D., Rossi-Bellom, M., Continella, A.: Reversing and fuzzing the Google Titan M chip. In: Proceedings of the Reversing and Offensive-Oriented Trends Symposium (ROOTS) (2021)
27. Montesel, P.: Big Match - how i learned to stop reversing and love the strings, https://conference.hitb.org/hitbsecconf2023hkt/materials/D1%20COMMSEC%20Big%20Match%20-%20How%20I%20Learned%20to%20Stop%20Reversing%20and%20Love%20the%20Strings%20-%20Paolo%20Montesel.pdf, accessed: 2023-12-10
28. Muench, M., Nisi, D., Francillon, A., Balzarotti, D.: Avatar 2: A multi-target orchestration platform. In: Proceedings of the Workshop on Binary Analysis Research (BAR) (2018)
29. Muench, M., Stijohann, J., Kargl, F., Francillon, A., Balzarotti, D.: What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In: Proceedings of the Symposium on Network and Distributed System Security (NDSS) (2018)
30. Niesler, C., Surminski, S., Davi, L.: Hera: Hotpatching of embedded real-time applications. In: Proceedings of the Symposium on Network and Distributed System Security (NDSS) (2021)
31. Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030, https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/, accessed: 2024-03-20
32. NuttX. https://nuttx.apache.org/, accessed: 2024-06-17
33. Redini, N., Machiry, A., Wang, R., Spensky, C., Continella, A., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: KARONTE: Detecting insecure multi-binary interactions in embedded firmware. In: Proceedings of the IEEE Symposium on Security and Privacy (2020)
34. RIOT. https://www.riot-os.org/, accessed: 2024-02-14
35. Scharnowski, T., Bars, N., Schloegel, M., Gustafson, E., Muench, M., Vigna, G., Kruegel, C., Holz, T., Abbasi, A.: Fuzzware: Using precise MMIO modeling for effective firmware fuzzing. In: Proceedings of the USENIX Security Symposium (2022)
36. Seidel, L., Maier, D.C., Muench, M.: Forming faster firmware fuzzers. In: Proceedings of the USENIX Security Symposium (2023)
37. Shen, M., Davis, J.C., Machiry, A.: Towards automated identification of layering violations in embedded applications. In: Proceedings of the ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES) (2023)
38. Shen, M., Pillai, A., Yuan, B.A., Davis, J.C., Machiry, A.: An empirical study on the use of static analysis tools in open source embedded software. arXiv preprint arXiv:2310.00205 (2023)

39. Srinivasan, J., Tanksalkar, S.R., Amusuo, P.C., Davis, J.C., Machiry, A.: Towards rehosting embedded applications as linux applications. In: Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE (2023)
40. Tay, H.J., Zeng, K., Vadayath, J.M., Raj, A.S., Dutcher, A., Reddy, T., Gibbs, W., Basque, Z.L., Dong, F., Smith, Z., et al.: Greenhouse: Single-service rehosting of Linux-based firmware binaries in user-space emulation. In: Proceedings of the USENIX Security Symposium (2023)
41. Tay, H.J., Zeng, K., Vadayath, J.M., Raj, A.S., Dutcher, A., Reddy, T., Gibbs, W., Basque, Z.L., Dong, F., Smith, Z., et al.: Greenhouse: Single-service rehosting of linux-based firmware binaries in user-space emulation. In: Proceedings of the USENIX Security Symposium (2023)
42. The tipping point: Exploring the surge in IoT cyberattacks globally, `https://blog.checkpoint.com/security/the-tipping-point-exploring-the-surge-in-iot-cyberattacks-plaguing-the-education-sector/`, accessed: 2024-02-14
43. Thomas, S.L., Van den Herrewegen, J., Vasilakis, G., Chen, Z., Ordean, M., Garcia, F.D.: Cutting through the complexity of reverse engineering embedded devices. IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES) (2021)
44. What happens when the sun sets on a smart product?, `https://www.ftc.gov/business-guidance/blog/2016/07/what-happens-when-sun-sets-smart-product`, accessed: 2023-12-22
45. Wu, Y., Wang, J., Wang, Y., Zhai, S., Li, Z., He, Y., Sun, K., Li, Q., Zhang, N.: Your firmware has arrived: A study of firmware update vulnerabilities. In: Proceedings of the USENIX Security Symposium (2023)
46. Yuan, Z., Feng, M., Li, F., Ban, G., Xiao, Y., Wang, S., Tang, Q., Su, H., Yu, C., Xu, J., et al.: B2sfinder: Detecting open-source software reuse in cots software. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE) (2019)
47. Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D., et al.: Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In: Proceedings of the Symposium on Network and Distributed System Security (NDSS) (2014)
48. Zephyr. `https://www.zephyrproject.org/`, accessed: 2024-06-17
49. Zhao, B., Ji, S., Xu, J., Tian, Y., Wei, Q., Wang, Q., Lyu, C., Zhang, X., Lin, C., Wu, J., et al.: One bad apple spoils the barrel: Understanding the security risks introduced by third-party components in iot firmware. IEEE Transactions on Dependable and Secure Computing (2023)
50. Zhou, W., Guan, L., Liu, P., Zhang, Y.: Automatic firmware emulation through invalidity-guided knowledge inference. In: Proceedings of the USENIX Security Symposium (2021)