POLITECNICO DI TORINO
Repository ISTITUZIONALE

A tool for IoT Firmware Certification

Original
A tool for IoT Firmware Certification / Bianco, G.M., Ardito, L., Valsesia, M.. - ELETTRONICO. - (2024), pp. 1-7. (ARES 2024: The 19th International Conference on Availability, Reliability and Security Vienna (AUT) 30 July 2024- 2 August 2024) [10.1145/3664476.3670469].

Availability:
This version is available at: 11583/2991671 since: 2024-08-12T13:38:30Z

Publisher:
Association for Computing Machinery

Published
DOI:10.1145/3664476.3670469

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
ACM postprint/Author's Accepted Manuscript

(Article begins on next page)

17 October 2026

A Tool for IoT Firmware Certification

Giuseppe Marco Bianco
Department of Control and Computer Engineering
Politecnico di Torino
Torino, Italy
giuseppe.bianco@polito.it

Luca Ardito
Department of Control and Computer Engineering
Politecnico di Torino
Torino, Italy
luca.ardito@polito.it

Michele Valsesia
Department of Control and Computer Engineering
Politecnico di Torino
Torino, Italy
michele.valsesia@polito.it

ABSTRACT

The IoT landscape is plagued by security and reliability concerns due to the absence of standardization, rendering devices susceptible to breaches. Certifying IoT firmware offers a solution by enabling consumers to easily identify secure products and incentivizing developers to prioritize secure coding practices, thereby fostering transparency within the IoT ecosystem. This study proposes a methodology centered on ELF binary analysis, aimed at discerning critical functionalities by identifying system calls within firmware. It introduces the manifest-producer tool, developed in Rust, for analyzing ELF binaries in IoT firmware certification. Employing static analysis techniques, the tool detects APIs and evaluates firmware behavior, culminating in the generation of JSON manifests encapsulating essential information. These manifests enable an assessment of firmware compliance with security and reliability standards, as well as alignment with declared device behaviors. Performance analysis using benchmarking tools demonstrates the tool's versatility and resilience across diverse programming languages and file sizes. Future avenues of research include refining API discovery algorithms and conducting vulnerability analyses to bolster IoT device security. This paper underscores the pivotal role of firmware certification in cultivating a safer IoT ecosystem and presents a valuable tool for realizing this objective within academic discourse.

CCS CONCEPTS

• **Software and its engineering** → **Software verification and validation**; *Embedded software*; *Software maintenance tools*; • **Computing methodologies** → Ontology engineering; • **Security and privacy** → **Domain-specific security and privacy architectures**.

KEYWORDS

Certification; IoT; IoT Firmware; Behaviour; Static analysis; Binary analysis; ELF file; IoT devices; Rust; Detection;

ACM Reference Format:
Giuseppe Marco Bianco, Luca Ardito, and Michele Valsesia. 2024. A Tool for IoT Firmware Certification. In *The 19th International Conference on Availability, Reliability and Security (ARES 2024), July 30-August 2, 2024, Vienna, Austria.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3664476.3670469

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ARES 2024, July 30-August 2, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1718-5/24/07
https://doi.org/10.1145/3664476.3670469

1 INTRODUCTION

The proliferation of Internet of Things (IoT) devices has introduced numerous benefits such as increased efficiency, connectivity, and automation. However, this rapid growth has also brought significant challenges, particularly in security and reliability. The lack of standardized practices for firmware development and certification has made IoT devices vulnerable to threats, undermining user trust and system integrity. Firmware, which interfaces directly with hardware, is crucial for IoT device functionality and security, making its integrity and reliability essential.

Current IoT security solutions focus on network protocols, encryption, and device authentication but often overlook firmware security. Without thorough scrutiny and certification, IoT devices are prone to breaches exploiting vulnerabilities or malicious code.

This study introduces a novel tool for IoT firmware certification, focusing on the analysis of ELF (Executable and Linkable Format) binaries. The tool uses static analysis to detect system calls and evaluate API behavior, generating comprehensive JSON manifests to ensure firmware compliance with security and reliability standards. Developed in Rust, this tool automates the certification process, promoting transparency and secure coding practices within the IoT ecosystem.

The remainder of this paper is organized as follows: Section 2 provides a comprehensive review of the background and related work in the IoT landscape and device firmware certification. Section 3 delves into the exploration of analysis methodologies employed for firmware analysis with the aim of identifying certifiable parameters. Section 4 provides an in-depth elucidation of the mainfest-producer tool's functionality, specifically tailored to aid in firmware certification analysis. Section 5 elucidates the data collected during the analysis, presented in the form of a JSON manifest. Section 6 aggregates a series of performance analyses of the tool concerning execution times and memory usage during the analysis of specific ELF files. Finally, Section 7 concludes the paper, outlining future work.

2 BACKGROUND AND RELATED WORK

The landscape of IoT, despite its advantages in terms of efficiency and convenience, is often plagued by concerns regarding the security and reliability of connected devices and systems. Particularly, the lack of standardization represents a significant gap in this context[1, 3, 7]. This deficit creates an environment where the security and reliability of IoT products can be compromised [5], as there is no formal guarantee regarding the quality and compliance

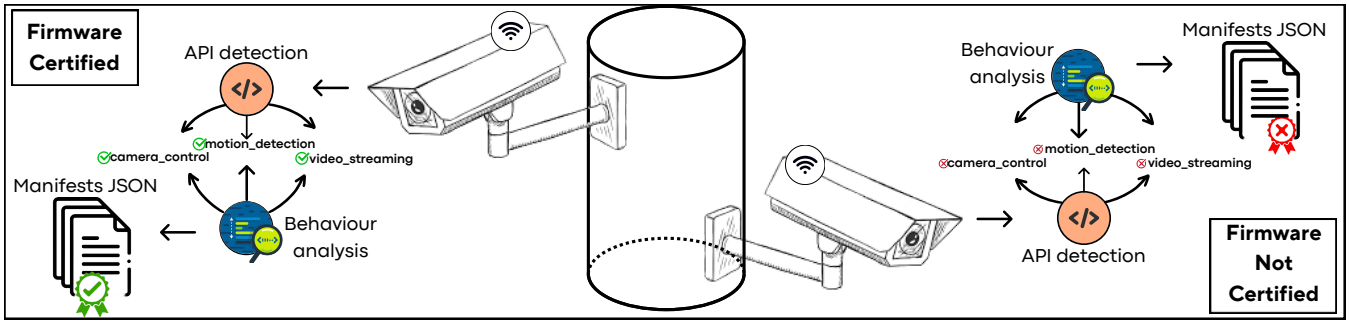

Figure 1: Example of the certification process

of the firmware used [2, 4]. **Certifying firmware** for IoT devices could address this issue, offering numerous advantages. Firstly, certification would allow consumers to identify products that meet certain security and reliability standards easily. This would help ensure greater peace of mind for end-users, while simultaneously reducing the risk of vulnerabilities and security breaches associated with uncertified firmware. Moreover, certification would positively impact the practice of IoT software development. Developers would be encouraged to allocate more resources to secure coding and code quality verification. This would enhance the robustness and resilience of firmware, reducing the likelihood of critical errors or security vulnerabilities. Lastly, firmware certification could contribute to promoting greater transparency and accountability in the IoT ecosystem. Developers would be required to accurately document the functionalities and behaviours of their firmware, enabling better understanding and evaluation by end-users and regulatory bodies. In the context of firmware certification for IoT devices, an important aspect is the analysis of **ELF** (Executable and Linkable Format) binary. This section aims to introduce such analysis, highlighting its advantages and disadvantages, as well as the rationale behind its consideration. IoT firmware represents a fundamental component for Internet of Things (IoT) devices, defining itself as the software incorporated directly into the hardware device. This firmware, closely linked to the peculiarities of the IoT environment, requires particular attention to ensure security and reliability [6], given the extensive interconnection and data collection involved.

2.1 Motivations for analyzing ELF binary

Analyzing ELF binary files offers numerous advantages in the context of IoT firmware certification. Firstly, this format is widely used in Unix-like operating systems, particularly Linux, making it a natural choice given the widespread adoption of such systems in the IoT ecosystem. Additionally, ELF binary files contain detailed information about program structures and functionality, providing a comprehensive overview of the firmware and potential points to examine. ELF binary analysis enables the examination of firmware at a lower level, providing a detailed view of program instructions and data structures. This approach allows for the identification of potential security vulnerabilities, understanding firmware behaviour, and ensuring compliance with security standards and development policies. Furthermore, ELF binary analysis can facilitate the creation of preventive measures and vulnerability correction, thereby improving the overall security of IoT firmware.
However, ELF binary analysis may present some disadvantages, including the complexity of program structures and the need for specialized skills to conduct a thorough analysis. Additionally, ELF binary analysis may not reveal all vulnerabilities present in the firmware, necessitating the adoption of complementary approaches to ensure a comprehensive security assessment.

3 EXPLORING STRATEGIES

3.1 Preliminary Analysis

The developmental trajectory of the manifest-producer tool started with a preliminary analysis aimed at probing the inherent challenges associated with analyzing ELF binaries within the context of certifying firmware for IoT devices. In this phase, the use of tools such as **radare2**[1] and **objdump**[2] was crucial. This preliminary analysis method facilitated comprehension of the ELF structure, enabling the identification of areas relevant to firmware certification. Specifically, the analysis, initiated through **code disassembly**, focused on system calls, deemed crucial in clarifying the firmware's authentic behaviour. Indeed, system calls allow user programs to access functionality that requires access to operating system privileges, such as file and memory management, communication with I/O devices, and many other operations. However, despite the granular control offered by the analysis with the cited tools, the imperative need to adopt an automated solution has emerged, given the laboriousness and impracticality associated with this approach. Furthermore, it is important to note that since Rust has been chosen as the programming language for the development of the manifest-producer tool, radare2 and objdump do not offer adequately supported crates for direct integration within a Rust program. Consequently, it was not possible to implement the manifest-producer directly using the workflow of these tools to obtain references to the various system calls during the analysis of the binaries. After a comprehensive preliminary analysis, two primary approaches for ELF binary analysis have been delineated: static analysis and dynamic analysis.

Static analysis entailed the exploration of two distinct methodologies.

[1] radare2 is a complete framework for reverse-engineering and analyzing binaries.
[2] objdump is a program for displaying various information about object files on Unix-like operating systems.

The first method conceived involved the use of **hexadecimal patterns**: they make it possible to identify and compare particular byte sequences in a hexadecimal representation, which is useful in this context for identifying specific behaviour representing system call instructions. However, this strategy was immediately recognized as complex and onerous in terms of computational resources, as it required the generation and management of a large corpus of hexadecimal models to cover the multiple architectures supported, as not all share the same syscall patterns. Moreover, the requirement to keep these models constantly updated, to adapt them to changing architectures, would involve considerable effort. However, recognition of this pattern alone may not be sufficient to reliably identify the syscall, as there may be other instructions in the code between the one that loads the syscall number into the appropriate register and the one that invokes it. Therefore, analysis based on hexadecimal patterns requires careful consideration of context and may be prone to errors if not implemented with attention and a thorough understanding of the binary code. The second method was to create a **system call mapping table**, which is a systematic approach to correlate system call numbers with their respective names. As explained in the previous point, by convention the operating system uses positive integers as identifiers for the various syscalls. This methodology inherently exploits the insights of the prior approach by focusing on the *.text section* of the ELF file, where the executable code is contained. Through an analysis of this section, the mapping process establishes a consistent association between the numerical identifiers of system calls and their semantic representations. Compared with the use of hexadecimal patterns, this approach significantly improves code readability and generalizability, as it can be applied to different architectures (such as x86, x86-64, ARM, ...) without requiring substantial modification or adaptation. However, despite the inherent advantages, it is important to note that the possible absence of a system call in the mapping table could result in incomplete categorization, compromising the integrity of the analysis.

Dynamic analysis is characterized by its ability to provide a probable and contextualized representation of firmware behaviour by focusing on **tracking** system calls during execution using the **strace** tool[3]. However, the effectiveness of this approach is closely related to the availability and functionality of strace in the target system. The presence of strace and its ability to provide interpretable output play a crucial role in determining the accuracy and usefulness of the dynamic analysis. Another significant aspect is that dynamic analysis records the execution flow of a *single firmware instance*. Therefore, it omits consideration of every possible alternative path that other instances might take in different contexts or with different inputs. This implies that although dynamic analysis provides realistic and immediate data, its coverage is inherently limited. To obtain a complete and thorough understanding of firmware behaviour, it may be necessary to run many instances to explore all possible combinations of scenarios and input configurations.

3.2 Definitive analysis

Preliminary analysis aimed at comprehending the structure of ELF files and configuring an analysis to identify suitable parameters for certification highlighted the necessity for a more precise and targeted methodology. In particular, greater emphasis has been placed on the implementation of individual public APIs rather than only relying on the entire firmware execution. This approach allows the analysis to focus on specific **code blocks**, thereby enhancing the granularity and precision of the evaluation, and enabling the division of firmware functionalities among different APIs. This orientation of the analysis towards a more static view, suggests the identification of the main functions and their memory addresses, thus enabling the correct disassembly of the code and a detailed analysis of system calls. Furthermore, the importance of identifying library function calls has also been recognised in the context of dynamically linked firmware.

4 HOW MANIFEST-PRODUCER WORKS

The manifest-producer tool, developed in Rust, is therefore designed to perform the firmware certification process through ELF binaries analysis. Its primary objective is to ensure firmware integrity and compliance through two key steps:

(1) **API Detection:** Firmware developers must provide the ELF binary with a list of used public APIs. This list forms the basis for the analysis, as each API is independently examined to assess its adherence to the intended behaviour.
(2) **Behavioral Analysis:** Once the APIs provided by the developer are identified, the tool disassembles its code and searches for system calls and external library functions, evaluating whether the APIs align with the expected behaviour or exhibit undesired characteristics.

Through this process, the manifest-producer tool enables validation of firmware compliance with security, reliability, and acceptable performance. Ultimately, it generates three distinct manifests in JSON format, which encapsulates the extracted and processed information. In essence, the manifest-producer serves as an instrument in binary firmware certification, offering a systematic approach to validate aspects of firmware behaviour and foster a safer IoT ecosystem.

4.1 API Detection

The first point of the analysis aims to carefully examine the public APIs provided by a firmware developer, to assess their adherence to specifications and ensure their integrity and compliance. Initially, the tool checks for the presence of the debug sections within the ELF file. The debug sections contain useful data for analysis purposes, including symbols representing variables, functions and other code entities. This information is essential for identifying and understanding the structure of a firmware. Once these sections have been confirmed, the tool proceeds to retrieve the list of APIs provided by the firmware developer. This list, consisting of a set of strings representing the names of the APIs, is essential for guiding the search process within the symbol table[4] of the ELF file. The symbol table in a binary file contains information on variables, functions and other code entities, along with their memory addresses. This information is used by the operating system to link program symbols to data and executable code during program execution.

[3] strace is a diagnostic and debugging utility for Linux.

[4] symbol table holds information needed to locate and relocate a program's symbolic definitions and references.

The tool then scans the symbol table, examining each symbol to determine whether it represents a function and if it is associated with a valid code section. These criteria are crucial for distinguishing valid functions within the firmware. For each symbol that meets these criteria, the tool checks if the function name matches one in the API list. If there is a match, it is an API to be analysed and then the tool obtains the starting address and size of the associated code block. Using this data, the end address of the code block can be calculated through a simple addition. At the end of this operation, an appropriate data structure contains the information for each API in the list:

- Name
- Starting address
- End address
- Vector of strings for syscalls

In addition, the structure provides the possibility of recording each system call associated with an API, thus offering a broader context for evaluating the behaviour and API usage in a firmware. In conclusion, the process above represents the first fundamental step in the firmware certification process, allowing the identification of public functions within the code, and thus providing a solid base for the subsequent stages of firmware analysis.

4.2 Behavioural analysis

The analysis process continues using the previously collected information stored in the dedicated data structure. This process is mainly divided into two phases:

(i) **Code Disassembly:** During this phase, the process focuses on analyzing the executed instructions to understand the operations performed by a function. This step translates the machine code into readable and understandable instructions, i.e. assembly code. This translation simplifies the analysis of the program execution flow and facilitates the identification of system calls. In particular, attention is focused on the identification of two specific instructions: *call* and *lea*. The **call instruction** receives a single piece of information: the address of the function to invoke. This can happen in two ways, either by directly passing the address or by loading it into a register. For example, we can express a call instruction as call 0x1352 or call rax, where the function's address is either directly specified (0x1352) or has been previously loaded into the RAX register. The **lea instruction**, short for **load effective address**, is used within the code to load the address of a function which will be engraved by the program into a specific register. For example, a lea statement might have the following syntax: lea 0x6452(%rip), %rax. In this context, lea instruction loads into the destination operand, the rax register, an offset arithmetically added to the rip register.
These two instructions are fundamental in the analysis of the disassembled code since they allow the identification of all system calls made by a function. They provide a fundamental overview of the operations performed by an API and its interactions with system libraries and the operating system.

(ii) **System Call Identification:** During this phase of the analysis, system calls occurring within a function are detected and logged. These calls are significant for the analysis as they offer insights into the API's behaviour. For instance, the detection of the sendto system call implies potential involvement in network operations, as sendto is typically used for transmitting data within a network environment. In this context, it is essential to acknowledge the potential scenario where certain system calls are not identified. This could occur when API operations are conducted within functions called from external libraries. Even in these situations, the tool can obtain the name of the function associated with the external library called by the API. Once the addresses have been obtained from lea or call instructions, it becomes crucial to consider how the external dependencies are managed in a building process. This analysis highlights two alternatives: for the **static linking**, identifying the name of the called function associated with the address is a relatively simple process. This is because the code of the functions is contained within the *.text section* of the binary. This structure simplifies the access to the various function allocations, allowing the tool to directly consult the symbol table. From there, it is possible to retrieve the index corresponding to the entry in the string table, providing the exact name of the function invoked by an API. In the **dynamic linking** case, the process is theoretically more cumbersome. During dynamic linking, not all addresses are resolved at compile time. This necessitates accessing the **Procedure Linkage Table (PLT)**[5] to retrieve the names of functions from external libraries. These addresses are dynamically resolved at runtime, making the process of identifying function names more intricate and dynamic. To simplify this process, the tool adopts a different strategy. First, identify the .plt section containing the PLT table. Next, it loads all the addresses associated with their names into a hash table. This design choice significantly speeds up the search because the tool can perform a simple query against the hash table rather than performing more complex operations in terms of time and number of operations.

5 MANIFESTS GENERATION

The generation of **JSON manifests** represents the last phase in the binary analysis, as it allows the essential information obtained from previous firmware analysis steps to be represented in a structured way. These manifests provide an important overview of the salient features of the analyzed ELF file and its interactions with system calls and library functions.
Manifest for basic information is a starting point for understanding the firmware. It provides general information about the ELF file, such as its file name, its programming language used, its target architecture, and its dependency linking type, static or dynamic. Additionally, it lists all public APIs identified in the code, providing a preliminary indication of the functionalities offered by the firmware.
Manifest for syscall flow provides a detailed overview of the system calls and library functions associated with each API identified in the firmware. This document provides a sequence of the operations performed during the execution of the various public functions. The peculiarity of this static analysis lies in its ability to comprehensively capture the API interactions with the system libraries and operating system, considering all possible execution paths that may not be explored whether a single dynamic instance of the program is used. By representing this information, this manifest contributes to a detailed and comprehensive understanding of

[5] PLT is a table used to manage calls to functions present in dynamically linked libraries.

the API's behaviour and its impact on the execution environment. Such static analysis is essential for revealing dependencies and interactions of the API with the underlying system, providing a solid base for evaluating the security and performance of a firmware.
Manifest for features, classifies APIs according to their functionality offering a structured overview of firmware's capabilities. This categorization occurs through a systematic process that evaluates the system calls and library functions associated with each API, identifying the tasks performed and grouping them into meaningful categories. The categorization process is based on a predefined set of functional categories, such as file manipulation, network access, device management, encryption. Each category is associated with a set of keywords or substrates that indicate the presence of specific functionality within system calls and library functions. This approach helps to categorize APIs based on what they can do, giving a clear picture of the firmware's main features.

6 PERFORMANCE ANALYSIS

The performance analysis of the manifest-producer tool aims to evaluate its effectiveness in analysing a series of ELF files written in C/C++ and Rust. Some of them simulate the firmware behaviour of an IoT device, others are well-known projects such as FFmpeg, xi-core and OpenCV. The aforementioned projects are open-source, which means that their source code is publicly available. **FFmpeg**[6] has been chosen for its broad utility in digital media manipulation. The complexity of the source code, primarily written in C with some critical parts optimized in assembly, provides an opportunity to assess the tool's performance in scenarios where complexity may impact the analysis of ELF files, making it a relevant study subject to evaluate the performance of the manifest-producer tool in practical contexts. **OpenCV**[7], written in C++, has been included in the analysis to examine the performance of the manifest-producer on ELF files involving complex computational calculations and intensive processing. The **xi-core**[8] project, being the core of the Xi text editor, represents an opportunity to evaluate the tool's capabilities in analyzing ELF binaries from projects that require optimal performance and efficient management of system resources, written in Rust.

This broad range of ELF binaries provides a comprehensive methodology for evaluating the tool's performance in real-world contexts, allowing for a detailed understanding of its strengths and possible areas for improvement.

6.1 Selected tools

Two performance analysis tools, **Hyperfine** and **Heaptrack**, have been used to conduct a thorough analysis. **Hyperfine**[9], a benchmarking tool, plays a role in analyzing the performance of the manifest-producer. It measures the execution times of programs, providing valuable insights into the duration of the analysis for each considered ELF binary file. Hyperfine's repeated benchmark runs offer an overview of the manifest-producer tool's performance, particularly regarding its speed and responsiveness. Hyperfine comes with a default configuration, including a warmup of 100 iterations followed by 1000 actual runs. This setup ensures a stable execution environment, minimizing the impact of any initial performance variations due to initialization processes or caching. Consequently, the data obtained through Hyperfine offers a dependable understanding of the manifest-producer tool's performance, effectively eliminating disturbances and providing a solid base for comparative analysis of execution times among different ELF binaries. **Heaptrack**[10] is a performance analysis tool designed to provide an insight into memory usage during program execution. This tool plays a role in the performance analysis of the manifest-producer tool. It enables monitoring and evaluating memory allocation in the software under examination by recording information about memory consumption peaks, temporary allocations, and any memory leaks. This ability to identify memory management issues is essential for accurately and thoroughly assessing the efficiency of memory allocation in a software.

6.2 Programming language comparisons

Through the utilization of binaries generated from a library designed to simulate potential firmware for IoT devices[11], a comparative analysis was conducted among the various programming languages under consideration. rust-dynamic shows the largest size at 54.0 MB, followed by C-dynamic at 18.2 MB and Cpp-dynamic at 7.3 MB. This variation may indicate differences in code optimization among the programming languages. Regarding **execution times**, Cpp-dynamic is the fastest at 10.7 ms, followed by C-dynamic at 15.8 ms and rust-dynamic at 43.6 ms. Interestingly, there is no direct correlation between file size and execution times. While the file size-to-execution time ratio in the case of the Cpp version may suggest increasing times with larger file sizes, this is not confirmed in the versions written in C and Rust, which exhibit accessible execution times despite their larger sizes. This suggests that factors such as code complexity and resource management significantly influence performance. Figure 2 shows the relationship between file size and execution time just described.

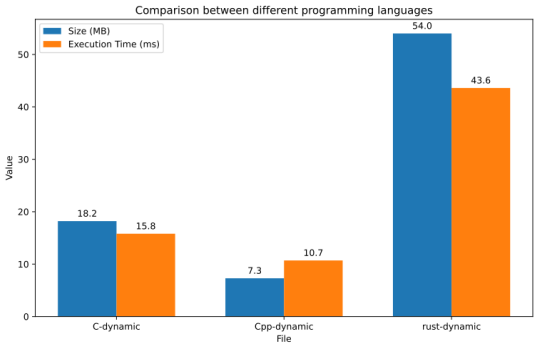

Figure 2: Execution time vs. file size for IoT library variants.

[6]Github repository:https://github.com/FFmpeg/FFmpeg
[7]GitHub repository: https://github.com/opencv/opencv
[8]GitHub repository: https://github.com/xi-editor/xi-editor
[9]GitHub repository: https://github.com/sharkdp/hyperfine
[10]GitHub repository: https://github.com/KDE/heaptrack
[11]GitHub repository: https://github.com/SoftengPoliTo/dummy-firmware-device

The comparison between FFmpeg, OpenCV, and xi-core files aims to contrast their different implementations and functionalities, along with their programming languages. FFmpeg, with a file size of 409.9 kB, exhibits an execution time of 6.3 ms, while OpenCV, with a smaller file size of 177.3 kB, shows a faster execution time of 4.0 ms. In contrast, xi-core stands out with a significantly larger file size of 74.6 MB, resulting in a longer execution time of 34.7 ms. This disparity suggests that larger file sizes generally correspond to longer execution times. However, it is interesting to note that, similar to previous analyses, there is no significant increase in execution times as file sizes increase, contrary to the trend observed for FFmpeg and OpenCV files. The graph in Figure 3 shows this comparison.

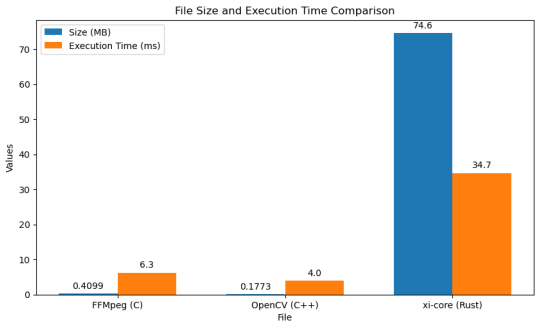

Figure 3: Execution time vs. file size for open-source projects.

From the point of view of memory usage, it is interesting to note the trend represented by the data collected through the memory peak and peak RSS parameters. Despite the obvious differences in file sizes, a consistent pattern emerges showing uniform growth ratios in both heap memory usage and maximum physical memory usage. This trend is precisely illustrated in the graph in Figure 4. For example, although FFmpeg and OpenCV are relatively compact file sizes, their memory usage growth ratio is approximately equal to that of the largest file size, xi-core. This suggests that, regardless of file size, the tool tends to use memory consistently and predictably.

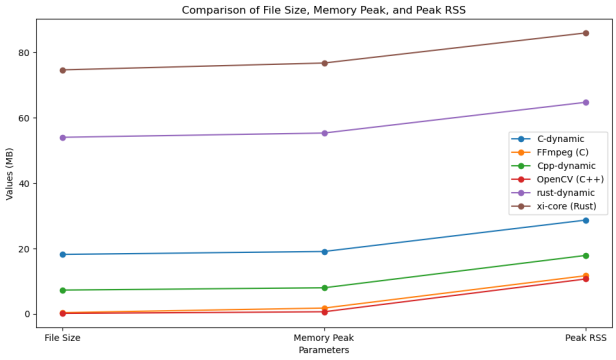

Figure 4: Memory usage comparison.

7 CONCLUSIONS AND FUTURE WORKS

The manifest-producer tool emerges as a viable alternative in the firmware certification process for IoT devices, providing a systematic approach to analyse ELF binaries and generate comprehensive manifests encapsulating essential information. This certification process offers insights into firmware functionality, system dependencies, and interactions with the underlying environment. Performance analysis of the tool has provided a comprehensive overview of its capabilities in analysing a range of ELF files. The gathered data demonstrates that file sizes do not solely dictate program execution times. Notably, significant variability in execution times, not directly proportional to file sizes, was observed, particularly in files written in the C language. Moreover, memory allocation analysis revealed distinct resource utilization patterns among different types of ELF files, indicating varying efficiency levels. Despite differences in programming languages and file sizes, the tool exhibits uniform performance across various contexts, suggesting a high level of adaptability and robustness. Looking towards future developments, efforts could be directed towards further enhancing the tool's effectiveness in firmware certification for IoT devices. This may involve improving the API discovery algorithm to allow more comprehensive searches for public functions without explicitly requesting a list of API names from the firmware developer. A thorough analysis of external library functions could enhance the tool's functionality in retrieving system calls. Currently limited to retrieving the names of these external library functions, there is potential to extend the tool's capabilities to recursively resolve the code of various functions from external dependencies, thereby capturing system calls that would otherwise go unnoticed. A comprehensive study of potential vulnerabilities in IoT device firmware code could be conducted, aiming to implement an analysis focused on highlighting device security. Such an initiative could significantly contribute to bolstering the overall security of IoT devices.

ACKNOWLEDGMENTS

This study was carried out within the AsCoT-SCE project – funded by the European Union – Next Generation EU within the PRIN 2022 program (D.D. 104 - 02/02/2022 Ministero dell'Università e della Ricerca). This manuscript reflects only the authors' views and opinions and the Ministry cannot be considered responsible for them

REFERENCES

[1] Sarah A. Al-Qaseemi, Hajer A. Almulhim, Maria F. Almulhim, and Saqib Rasool Chaudhry. 2016. IoT architecture challenges and issues: Lack of standardization. In *2016 Future Technologies Conference (FTC)*. 731–738. https://doi.org/10.1109/FTC.2016.7821686

[2] Taimur Bakhshi, Bogdan Ghita, and Ievgeniia Kuzminykh. 2024. A Review of IoT Firmware Vulnerabilities and Auditing Techniques. *Sensors* 24, 2 (2024). https://doi.org/10.3390/s24020708

[3] André Cirne, Patrícia R. Sousa, João S. Resende, and Luís Antunes. 2022. IoT security certifications: Challenges and potential approaches. *Computers & Security* 116 (2022), 102669. https://doi.org/10.1016/j.cose.2022.102669

[4] Xiaotao Feng, Xiaogang Zhu, Qing-Long Han, Wei Zhou, Sheng Wen, and Yang Xiang. 2023. Detecting Vulnerability on IoT Device Firmware: A Survey. *IEEE/CAA Journal of Automatica Sinica* 10, 1 (2023), 25–41. https://doi.org/10.1109/JAS.2022.105860

[5] Basem Ibrahim Mukhtar, Mahmoud Said Elsayed, Anca D. Jurcut, and Marianne A. Azer. 2023. IoT Vulnerabilities and Attacks: SILEX Malware Case Study. *Symmetry* 15, 11 (2023). https://doi.org/10.3390/sym15111978

[6] Ibrahim Nadir, Haroon Mahmood, and Ghalib Asadullah. 2022. A taxonomy of IoT firmware security and principal firmware analysis techniques. *International Journal of Critical Infrastructure Protection* 38 (2022), 100552. https://doi.org/10.1016/j.ijcip.2022.100552

[7] Jibran Saleem, Mohammad Hammoudeh, Umar Raza, Bamidele Adebisi, and Ruth Ande. 2018. IoT standardisation-Challenges, perspectives and solution. In *ACM International Conference Proceeding Series*.

Received 15 May 2024; revised 15 May 2024; accepted 30 May 2024