A tool for IoT Firmware Certification

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 rapid growth of the Internet of Things (IoT) has created a fragmented ecosystem, with no clear rules for security and reliability. This lack of standardization makes IoT devices vulnerable to attacks. IoT firmware certification can address these security concerns. It empowers consumers to make informed choices by readily identifying secure products. Additionally, it incentivizes developers to prioritize secure coding practices, ultimately promoting transparency and trust within the IoT ecosystem. Several existing IoT device certifications (e.g. Cybersecurity Assurance Program, British Standards Institution, ioXt Alliance) prioritise cybersecurity through risk and vulnerability assessments. This paper proposes a complementary approach. Our tool focuses on identifying firmware functionality by analysing system calls through static analysis. This allows to publicly identify APIs to assess the actual behaviour of a firmware. The analysis culminates in the generation of JSON manifests, which encapsulate the relevant information gathered during the case study. In particular, this analysis verifies whether the actual behaviour is in line with the developer's statements about the device's functionality, contributing to the security and reliability of a device. To evaluate tool's performance, we conducted a benchmarking analysis which has demonstrated efficient handling of binaries written in various languages, even those with large file sizes. Future will be based on refining the API search and syscall collection algorithms, other than incorporating vulnerability analysis to further strengthen the security of an IoT device.

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;

ARES 2024, July 30–August 02, 2024, Vienna, Austria

ACM ISBN 979-8-4007-1718-5/24/07
https://doi.org/10.1145/3664476.3670469

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 02, 2024, Vienna, Austria.* ACM, New York, NY, USA, 7 pages. 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, privacy and reliability[3, 7]. The lack of standardized practices for firmware development and certification has made IoT devices vulnerable to threats, undermining users' trust and systems integrity. A firmware is a software which interfaces directly with hardware and is crucial for IoT device functionality and security, making its integrity and reliability essential.

Current IoT security solutions mainly focus on network protocols, encryption and device authentication[5], relying on the IoT Security Testing Framework[8], but often ignoring firmware security. The resource-constrained nature of IoT devices makes them attractive targets for cyberattacks, and although advanced IoT platforms, such as Apple's HomeKit and Intel's IoTivity, offer varying degrees of security, most platforms do not adequately address firmware security[2]. Additionally, IoT security and privacy challenges are exacerbated by the interconnectivity of devices, exposing networks to threats from anonymous and untrusted actors[12]. Frequent security breaches have highlighted the vulnerabilities of IoT technologies, and a proper consideration of firmware security can help mitigate these risks.

Although established tools for static analysis of ELF (Executable and Linkable Format) binaries such as Radare2, Ghidra, and Binary Ninja already exist, the tool introduced in this paper is distinguished from them by its ability to perform automated reverse engineering, specifically aimed at detecting system calls. This static analysis highlights the actual behaviours of the public APIs contained in an IoT device firmware, providing the information needed to assess whether the behaviour of each one is consistent with what the developer has declared.

Another distinguishing feature of this tool is that we developed it entirely in Rust, a modern, efficient, and memory-safe programming language[14]. Rust's intrinsic characteristics contribute to improve the security and reliability of a software, minimising the risk of memory management vulnerabilities, as opposed to, for example, software written in C and C++.

The remainder of this paper is organized as follows: Section 2 provides a review of the state of the art in the IoT landscape and

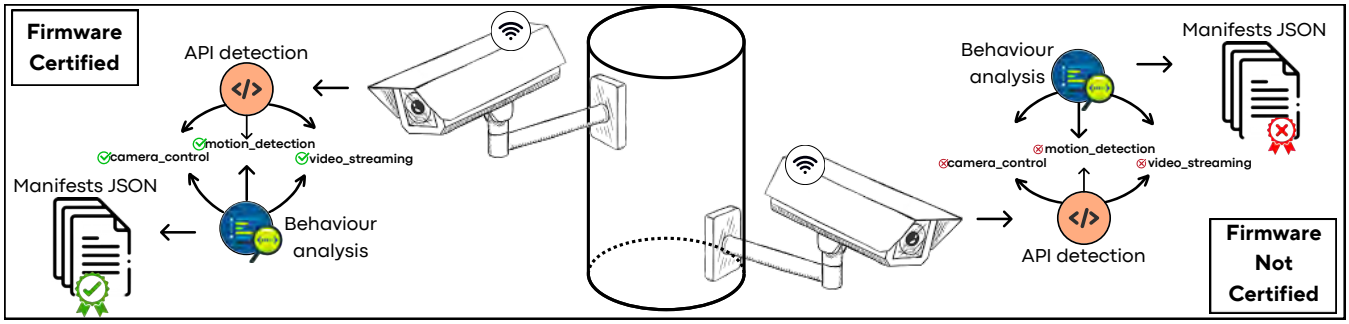

Figure 1: Example of the certification process

device firmware certification. Section 3 delves into the exploration of analysis methodologies employed for firmware analysis with the goal of identifying certifiable parameters. Section 4 provides an in-depth elucidation of the mainfest-producer[1] tool's functionality, specifically tailored to aid in firmware certification analysis. Section 5 elucidates on data collected during the analysis, presented in the form of a JSON manifest. Section 6 aggregates a series of performance analyses made on the tool concerning execution times and memory usage obtained during the analysis of specific ELF files. Finally, Section 7 concludes this paper, outlining possible future works.

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[1, 5, 11] represents a significant gap in this context. This deficit creates an environment where the security and reliability of IoT products can be compromised[9], also because there is no formal guarantee of the quality and conformity of the firmware used[4, 6]. The **Certification** of IoT devices could address this issue, offering numerous advantages.

Current certification systems often rely on several established techniques, such as risk analysis, assessment of known vulnerabilities, penetration testing, and code analysis[5, 13]. However, none of these techniques directly address the static analysis of a firmware and fully understand the **behaviour of the device**, ensuring compliance with the behaviour specifications defined by a developer. Firmware requires particular attention to ensure security and reliability [10], given the extensive interconnection and data collection involved in the IoT ecosystem. There is still a gap in the static evaluation of IoT firmware devices, which can hide issues[13, 15] that could allow non-compliant and potentially harmful behaviour, even if no threats are detected in the code.

For example, a home security camera has firmware which captures and saves images in local memory, through the API provided, without using an external server for storage. However, if not documented or improper behaviour emerges during firmware static analysis, such as sending the captured images to a third-party server, this could be the starting point for possible malicious behaviour.

[1]GitHub repository: https://github.com/SoftengPoliTo/manifest-producer

Although security certifications for IoT devices are based on many established cybersecurity techniques, the introduction of static firmware analysis also integrated into existing risk assessment steps, could improve the effectiveness of such certifications in identifying non-compliant behaviours and protecting users from potential threats.

2.1 Motivations for analyzing ELF binary

In firmware certification for IoT devices, an important aspect is the analysis of **Executable and Linkable Format (ELF)** binary. This section aims to introduce such analysis, highlighting its potential and problems, as well as the rationale behind its consideration. Firstly, this format is widely used in Unix-like operating systems, particularly Linux versions, making it a natural choice, given the widespread adoption of such systems in the IoT ecosystem. Additionally, ELF binary files contain detailed information about the structures and functionalities of a program, providing a comprehensive firmware overview. ELF binary analysis enables the examination of a firmware at a lower level, providing a detailed view of program instructions and data structures. This approach offers the opportunity to identify potential security vulnerabilities, enables an understanding of firmware behaviour and allows verification of compliance with security standards and development policies. Furthermore, ELF binary analysis can facilitate the creation of preventive measures and vulnerabilities correction, thereby improving the overall security of an IoT firmware.
However, ELF binary analysis may present some problems, including the complexity of program structure (such as division into sections) and the need for specialized skills to conduct an in-depth analysis. Additionally, an ELF binary analysis may not reveal all vulnerabilities present in a 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 the analysis of ELF binaries within firmware certification for IoT devices. In this phase, the use of tools such as

radare2[2] and **objdump**[3] was crucial. This preliminary analysis provided an understanding of the structure of an ELF, enabling the identification of areas relevant for firmware certification. Specifically, the analysis, initiated through **code disassembly** and focused on system calls, deemed crucial in clarifying the authentic behaviour of a firmware. Indeed, system calls allow user programs to obtain functionality that requires access to operating system privileges, such as file and memory management, communication with I/O devices, and many other kernel operations. However, despite the granular control offered by the analysis with radare2 and objdump, it became imperative to adopt an automated solution, given the laboriousness and impracticality of using these tools in this direction. 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 adequate support for direct integration within a Rust program. Consequently, it was not possible to implement into the manifest-producer these tools to obtain references to the various system calls during the analysis of the binaries. After a comprehensive preliminary study, two primary approaches for ELF binary analysis have been delineated: static analysis and dynamic analysis.

Static analysis entailed the exploration of two distinct methodologies.

The first conceived method involved the use of **hexadecimal patterns**: they make it possible to identify and compare particular byte sequences in a hexadecimal representation, which is useful for detecting specific behaviour concerning 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 every supported architectures since most of them do not share the same syscall patterns. Moreover, the requirement to keep these models constantly updated, thus adapting them to changing architectures, would involve a considerable effort. However, recognition of this pattern alone may not be sufficient to reliably identify syscalls, as there may be other instructions involved in the code, those which load the syscall number into the appropriate register and those which invoke it. Therefore, analysis based on hexadecimal patterns requires careful context consideration and may be prone to errors if not implemented with attention and a thorough understanding of binary code. The second method was to create a **system call mapping table**, 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 an ELF file, where a program executable code is contained. Through an analysis of this section, the mapping process establishes a consistent association between the system calls numerical identifiers 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 these inherent advantages, it is important to note that the possible absence of a system call in the mapping table could result in an incomplete categorization, compromising the whole 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 its execution using the **strace** tool[4]. However, the effectiveness of this approach depends on the **availability** and **functionality** of strace in the target system. Specifically:

(1) **Aviability**: Strace must be installed on the target system. If strace cannot be installed, this method of analysis cannot be used.
(2) **Functionality**: Strace must be able to operate correctly. Some systems may have security restrictions or configurations that prevent strace from monitoring system calls. Additionally, appropriate permissions (such as administrator privileges) might be necessary to use strace effectively.

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 any other possible alternative path that other instances might travel 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 in-depth understanding of firmware behaviour, it may be necessary to run a great number of instances to explore all possible combinations of scenarios and input configurations.

3.2 Definitive analysis

A preliminary study 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 an 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 static analysis helps identify the main functions and their memory addresses, allowing for correct code disassembly and detailed analysis of system calls. Additionally, it is important to identify library function calls, even for dynamically linked firmware.

4 HOW MANIFEST-PRODUCER WORKS

The manifest-producer tool, developed in Rust, has been designed to perform the firmware certification process through the analysis of ELF binaries. Its primary objective consists of ensuring firmware integrity and compliance through two key steps:

(1) **API Detection:** For firmware certification, developers are required to submit an ELF binary along with a documented

[2] radare2 is a complete framework for reverse-engineering and binary analyzer.
[3] objdump is a program for displaying various information about Unix-like operating systems object files.
[4] strace is a diagnostic and debugging utility for Linux.

list of public APIs. These APIs represent the externally accessible functions offered by the firmware's libraries. This list is the analysis starting point since each API is independently examined to assess its adherence to the intended behaviour.

(2) **Behavioral Analysis:** Once the APIs provided by a 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 JSON format manifests, which encapsulates the extracted and processed information. In essence, the manifest-producer serves as an instrument for binary firmware certification, offering a systematic approach to validate every aspect of a firmware behaviour.

4.1 API Detection

The first analysis step aims to examine each public API provided by a firmware developer, to assess its adherence to specifications and ensure its integrity and compliance. Initially, the tool checks for the presence of the debug sections within the ELF file. Debug sections contain symbols which represent variables, functions and other code entities. This information is essential for identifying and understanding a firmware structure. Once these sections have been checked, the tool proceeds to detect the APIs provided by a firmware developer. This list, consisting of a set of strings representing API names, guides the search process within the symbol table[5] of an ELF file. A binary symbol table contains symbols, thus information on variables, functions and other code entities, along with their memory addresses which an operating system links to data and executable code necessary for executing a program. Subsequently, the tool then scans the symbol table, examining each symbol to determine whether it is a function and if it can be associated with a valid code section. For each symbol that satisfies the previous criteria, the tool evaluates whether a function name matches one of the API list names. If there is a match, the tool obtains the starting address and size of the code block associated with that API. Starting from this code block information, the end address of the code block is calculated through a simple addition. Each API in the list has the following data structure once all operations have been completed:

- Name
- Starting address
- End address
- Sequence of invoked syscalls

In addition, the structure allows recording each system call associated with an API, offering a broader context for evaluating the behaviour and an API in a firmware. To summarize, the process above represents the first fundamental step in the firmware certification process, leading to the detection of public functions within the code, and thus providing a solid base for the subsequent steps of a firmware analysis. Figure 2 shows the workflow of the first phase performed by the tool.

[5] symbol table holds all necessary information to locate and relocate program's symbols definitions and references.

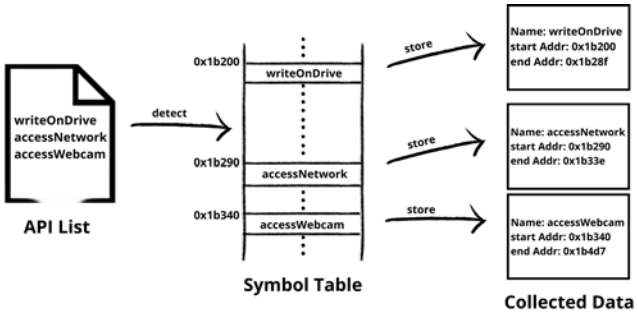

Figure 2: API Detection workflow.

4.2 Behavioural analysis

The second phase of the process begins with the information stored in the previous phase. As shown in figure 3, this new phase is mainly divided into two sub-phases:

(i) **Code Disassembly:** In this first sub-phase, the process analyzes the executed instructions to list 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 execution flow analysis and makes easier the system calls retrival. In particular, the attention is focused on identifying two specific instructions: *call* and *lea*. The **call instruction** is designed to invoke a function. It takes a single operand, which specifies the target function's address. This address can be provided in two ways:

- **Direct Addressing**: The address of the function is directly encoded in the instruction, represented as a hexadecimal value (e.g., **call 0x1352**).
- **Register Indirect Addressing**: The address is stored in a specific register (e.g., rax) and passed through that register during the call instruction (**call rax**).

The **lea instruction**, short for **load effective address**, loads the address of a function into a specific register. For example, a lea statement might have the following syntax: **lea 0x6452(%rip), %rax**. These two instructions are fundamental for the analysis of the disassembled code since they allow to identify all system calls invoked by a function.

(ii) **System Call Identification:** In this second sub-phase, system calls invoked by a function are detected and logged. These calls are significant for the analysis since they offer insights into an API behaviour. For example, the detection of a **sendto** system call implies potential involvement in network operations, as sendto is typically employed to transmit data within a network environment. In this context, it is very important to acknowledge how 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 function name present in the external library called by an API. Once every address has been obtained from lea or call instructions, it is necessary to consider how external dependencies are managed during a building process. Two different alternatives can be considered: **static linking**, where identifying the function invoked is a relatively simple process since the function is contained in the *.text section* of the binary. This

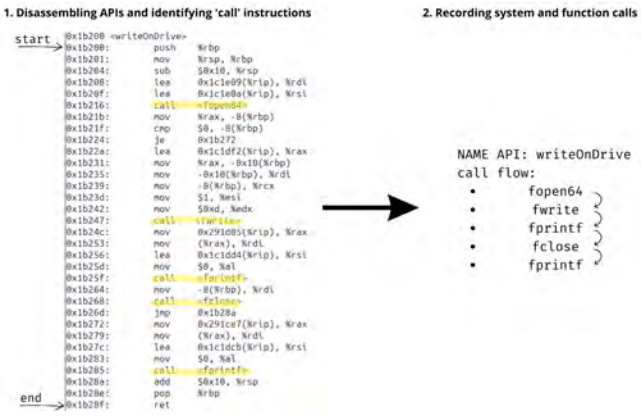

Figure 3: Code disassembly and syscall identification.

structure streamlines access to a function's memory allocations. By referencingg the symbol table directly, the tool can efficiently retrieve the function's index within the string table. This index directly translates to the function's name invoked by the API. In the **dynamic linking** case, the process is a bit more cumbersome. During dynamic linking, not all addresses are resolved at compile time. So it is necessary to access the **Procedure Linkage Table (PLT)**[6] to retrieve the functions name present in external libraries. These addresses are dynamically resolved at runtime, making the entire process more intricate and dynamic. To simplify this process, the tool adopts a simpler strategy. At first, it identifies the .plt section containing the PLT table. Next, it loads all the addresses associated with functions name into a hash table. This choice significantly speeds up the search because the tool can perform a simple query against the hash table rather than performing more heavy operations.

5 MANIFESTS GENERATION

Json manifests generation is the last in the binary analysis phase, where the crucial information gathered from the firmware analysis is presented in a textured way. These manifests provide an important overview of the information extracted from an analyzed ELF file and its interactions with system calls and library functions.

Manifest for basic information provides general information about an ELF file, such as its file name, its programming language, its target architecture, and its dependency linking type: static or dynamic. Additionally, it lists all public library APIs, providing a preliminary indication of the functionalities offered by a firmware.

Manifest for syscall flow provides a detailed overview of system calls and library functions of each firmware API. This report provides a sequence of the operating systems functions performed during the execution of the various public user-space functions. The peculiarity of this static analysis lies in its ability to capture the API interactions with operating system libraries, considering all possible execution paths that may not be explored when a single dynamic instance of the program is executed. This manifest contributes to a detailed and comprehensive understanding of an API behaviour and provides a solid base for evaluating the security and performance of a firmware.

Manifest for features classifies APIs according to their functionalities offering a structured overview of of a firmware capabilities. This categorization occurs through a systematic process that evaluates the tasks performed by system call and groups them into meaningful categories. The categorization process is based on a predefined set of functional categories, such as file manipulation, network access, device management, and encryption. This approach facilitates the classification of APIs according to their capabilities, providing a clear view of the main features of a firmware.

6 PERFORMANCE ANALYSIS

The performance analysis of the manifest-producer tool aims to assess its capability in analysing a series of ELF files written in C/C++ and Rust languages. Some of of these binaries simulate the IoT device firmware behaviour, while others are well-known projects such as FFmpeg, xi-core and OpenCV. The aforementioned projects are open-source, thus their source code can be consulted publicly. **FFmpeg**[7] is a library for digital media manipulation and it has been chosen for its variegated code. The complexity of its source code, primarily written in C with some of its critical parts optimized in assembly, allows to assess the manifest-producer performance in scenarios where complexity may impact the analysis of ELF files, making it a relevant case study to evaluate the tool performance in real contexts. **OpenCV**[8] is a tool written in C++ and it has been in this analysis to examine the manifest-producer performance on ELF files involving complex computational calculations and intensive processing. The **xi-core**[9] project is written in Rust and represents an opportunity to evaluate the tool's capabilities on analyzing ELF binaries from projects that require optimal performance and efficient management of system resources.

This small but specific range of ELF binaries provides a comprehensive methodology for evaluating the performance of this tool in real-world contexts, allowing it to detect its strengths and possible areas for improvement.

6.1 Selected tools

Two performance analysis tools, **Hyperfine** and **Heaptrack**, have been used to conduct a time and memory analysis. **Hyperfine**[10], is a benchmarking tool that plays a role in analyzing the performance of the manifest-producer. It measures the execution times of a program, providing valuable insights into the duration of the analysis for each considered ELF binary file. Hyperfine's repeated benchmark offers an overview of the speed and responsiveness of the manifest-producer tool. By default, Hyperfine warms up the system with 100 iterations before performing 1000 actual measurements. This two-step process guarantees a stable execution environment, eliminating the influence of initial setup or caching on performance. Consequently, Hyperfine results offer clean data, ideal for comparing execution times across different ELF binaries. **Heaptrack**[11] is a tool designed to provide a memory introspection

[6] PLT is a table which manages function calls present in dynamically linked libraries.

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

into memory usage during the execution of a program. It monitors and evaluates the memory allocation of the software under examination by recording information about memory consumption peaks, temporary allocations, and possible memory leaks. The ability to identify memory management issues is essential for assessing memory allocations in a software.

6.2 Programming language comparisons

Using binaries generated from libraries conceived to simulate IoT devices firmware behaviours[12], a comparative analysis was conducted among the various considered programming languages. 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 among programming languages building optimizations. 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 may suggest increasing times with larger file sizes in the case of the Cpp version, this fact is not confirmed in C and Rust versions, which exhibit accessible execution times despite their larger sizes. This suggests that factors, such as code complexity and resource management, significantly influence performance. Figure 4 shows the relationship between file size and execution time in the considered binaries.

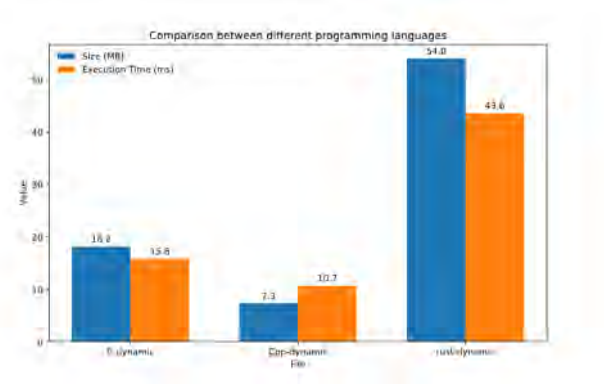

Figure 4: Execution time vs. file size for IoT library variants.

The comparison takes an in-depth look at the functionality offered by FFmpeg, OpenCV and xi-core, exploring how each library implements its functionality, beyond the programming languages used. 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 the analysis conducted previously, there is no significant growth in execution times as file sizes increase, contrary to the trend observed for FFmpeg and OpenCV files. The graph in Figure 5 shows this comparison.

[12]GitHub repository: https://github.com/SoftengPoliTo/dummy-firmware-device

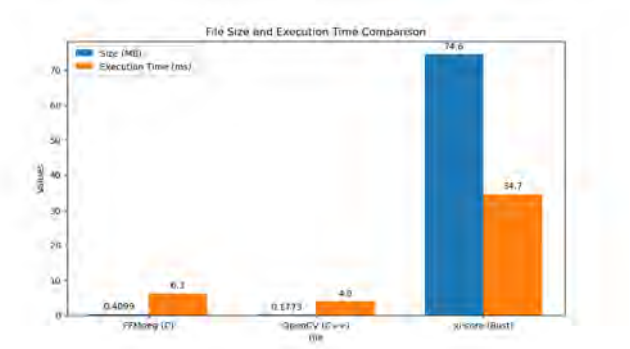

Figure 5: Execution time vs. file size for open-source projects.

Analyzing the trends in memory usage, as reflected by both peak heap memory consumption and peak RSS (Resident Set Size), offers valuable insights into a program's resource management efficiency. Despite the obvious differences in file sizes, a consistent pattern emerges, showing uniform growth ratios in both heap and maximum physical memory usage. This trend is illustrated in Figure 6. Despite having smaller file sizes, FFmpeg and OpenCV exhibit a memory usage growth pattern similar to the largest xi-core file. This suggests the tool's memory consumption primarily depends on factors other than the input file size, leading to consistent and predictable memory usage behaviour.

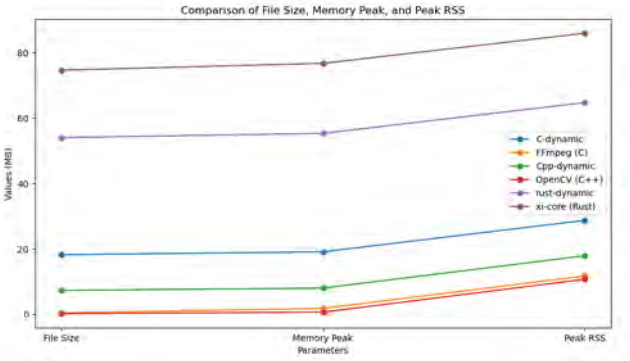

Figure 6: Memory usage comparison.

7 CONCLUSIONS AND FUTURE WORKS

The manifest-producer emerges as a complementary tool in the firmware certification for IoT devices. It analyzes ELF binaries in a structured way and creates detailed reports (manifests) summarizing key information. These reports reveal how the firmware works, what system components it relies on, and how it interacts with the device's hardware. Performance analysis reveals the tool's strengths in handling various ELF files. Interestingly, the data shows that file size alone does not determine how long a program takes to run. Notably, significant variability in execution times, not directly proportional to file sizes, was observed, particularly in C

language files. 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 IoT device firmware certification. This may involve improving the API discovery algorithm to allow a comprehensive search for public functions without explicitly requesting a list of API names from a firmware developer. A thorough analysis of external library functions could also enhance the tool's functionality in retrieving system calls. While the tool currently only retrieves names of external library functions, it has the potential to become even more powerful. By recursively resolving the code of these functions, it could also capture system calls hidden within them. This would provide a more complete picture of the firmware behaviour. A comprehensive study of potential vulnerabilities in IoT device firmware code could be conducted, aiming to implement an analysis focused on highlighting device security problems. 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] Stefan-Ciprian Arseni, Simona Halunga, Octavian Fratu, Alexandru Vulpe, and George Suciu. 2015. Analysis of the security solutions implemented in current Internet of Things platforms. In *2015 Conference Grid, Cloud & High Performance Computing in Science (ROLCG)*. 1–4. https://doi.org/10.1109/ROLCG.2015.7367416

[3] Abeer Assiri and Haya Almagwashi. 2018. IoT Security and Privacy Issues. In *2018 1st International Conference on Computer Applications & Information Security (ICCAIS)*. 1–5. https://doi.org/10.1109/CAIS.2018.8442002

[4] 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

[5] 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

[6] 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

[7] Fatima Hussain, Rasheed Hussain, Syed Ali Hassan, and Ekram Hossain. 2020. Machine Learning in IoT Security: Current Solutions and Future Challenges. *IEEE Communications Surveys & Tutorials* 22, 3 (2020), 1686–1721. https://doi.org/10.1109/COMST.2020.2986444

[8] Bryer Jeannotte and Ali Tekeoglu. 2019. Artorias: IoT Security Testing Framework. In *2019 26th International Conference on Telecommunications (ICT)*. 233–237. https://doi.org/10.1109/ICT.2019.8798846

[9] 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

[10] 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

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

[12] Lo'ai Tawalbeh, Fadi Muheidat, Mais Tawalbeh, and Muhannad Quwaider. 2020. IoT Privacy and Security: Challenges and Solutions. *Applied Sciences* 10, 12 (2020). https://doi.org/10.3390/app10124102

[13] Chin-Wei Tien, Tsung-Ta Tsai, Ing-Yi Chen, and Sy-Yen Kuo. 2018. UFO - Hidden Backdoor Discovery and Security Verification in IoT Device Firmware. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 18–23. https://doi.org/10.1109/ISSREW.2018.00-37

[14] Paul C. van Oorschot. 2023. Memory Errors and Memory Safety: A Look at Java and Rust. *IEEE Security & Privacy* 21, 3 (2023), 62–68. https://doi.org/10.1109/MSEC.2023.3249719

[15] Muhammad Shaharyar Yaqub, Haroon Mahmood, Ibrahim Nadir, and Ghalib Asadullah Shah. 2022. An Ensemble Approach for IoT Firmware Strength Analysis using STRIDE Threat Modeling and Reverse Engineering. In *2022 24th International Multitopic Conference (INMIC)*. 1–6. https://doi.org/10.1109/INMIC56986.2022.9972941

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