POLITECNICO DI TORINO
Repository ISTITUZIONALE

A Rust library for behaviours assessment in software certification

Original
A Rust library for behaviours assessment in software certification / Aldini, A., Ardito, L., Bianco, G., Valsesia, M.. - In: IEEE INTERNET OF THINGS JOURNAL. - ISSN 2327-4662. - 12:20(2025), pp. 42303-42316. [10.1109/JIOT.2025.3592839]

Availability:
This version is available at: 11583/3002093 since: 2025-07-25T10:58:22Z

Publisher:
IEEE

Published
DOI:10.1109/JIOT.2025.3592839

Terms of use:

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

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2025 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

17 October 2026

A Rust library for behaviours assessment in software certification

Alessandro Aldini, Luca Ardito, *Member, IEEE*, Giuseppe Marco Bianco, and Michele Valsesia

***Abstract*—The majority of Internet of Things (IoT) devices available on the market nowadays present heterogeneity problems and security problems. These devices adopt different protocols to communicate among each other, hence the definition of a series of standards has been necessary to make their interaction possible. Moreover, certification processes have been proposed to analyse manufacturer's products in search of every vulnerability and security threat which might affect firmware and final devices. However, none of these methodologies fully considers the effects which might occur when executing an IoT firmware. Therefore, to fill in this gap, we propose the *behaviours assessment* as a technique which allows a certifier to evaluate whether the effects of firmware operations are in accordance with some established policies. Starting from this model, we have created a library which performs the behaviours assessment using a safe and secure programming language called Rust. We have named this library *manifest-producer*. This library provides APIs to analyse ELF binaries, extract their functions, disassemble machine code, and build call trees to support behaviour evaluation using reverse engineering techniques in an automated way. To demonstrate its potential as an additional tool for IoT certification, we present a proof of concept showing how it helps to assess behaviours in a mock IoT firmware.**

***Index Terms*—Behaviours Assessment, Certification, Firmware Analysis, IoT, Security, Reverse Engineering, Rust Library, Static Analysis**

I. Introduction

In the IoT industry, the lack of formal security rules is one of the main problems, especially between different manufacturers [1]. Furthermore, the demand for more inexpensive systems leads to an increase in heterogeneity between devices in addition to a generalized lack of security standards [2]. Therefore, it is necessary to promote the adoption of security measures, and IoT systems certification might be a possible solution to improve devices security mechanisms [3]. Ross *et al.* suggest a definition for security certification: an assessment to determine whether certain assumptions are correctly implemented, operating as intended, and producing the desired outcome to meet security requirements [4]. To evaluate the security of IoT systems' components, different kinds of assessments are necessary, namely risk, vulnerability, penetration testing, and fuzzing [1].

Risk assessment allows identifying internal and external threats and assess their likelihood of happening [5]. Each security certification must have a risk assessment to better express the specific security requirements of a target.

Vulnerability assessment aims to identify known vulnerabilities in a system [6]. However, it cannot discover unknown vulnerabilities.

Penetration testing is a technical assessment which simulates the behaviour of an attacker [1]. It differs from vulnerability assessment because it also discovers new vulnerabilities and evaluates their actual impact on a system. This kind of testing employs offensive attack techniques in order to discover vulnerabilities, and since malicious attacks need only a single exploit to be successful, improving this assessment coverage is crucial [7]. Since penetration testing is a manual time-consuming process, especially for IoT environments, security researchers are continuously developing tools to carry out these kinds of analysis.

Fuzzing assessment tests different input device interfaces against data generated automatically, combining random data and generation rules with each other. The main objective of this analysis is to discover unknown software vulnerabilities, therefore flaws or weaknesses in a system's design, implementation, or operation and management which an attacker could exploit to violate a system security policy [1]. Since this assessment is an intrusive procedure, a production environment should not perform fuzzing tests because of the possible service disruptions.

This paper focuses on analysing a library thought and developed to conduct a new type of assessment we have conceived to evaluate firmware effects on an environment: the behaviours assessment. This library, called manifest-producer, provides a series of APIs to extract from a binary all the necessary information for a certifier to describe IoT firmware behaviours. It is also able to obtain these data from general-purpose software. From a technical perspective, the manifest-producer library makes use of static analysis and reverse engineering techniques to process the binary and retrieve this information. A certifier then analyses these data and draws up a report containing a list of firmware behaviours, which a manufacturer must sign in to accept the analysis and conclude the certification process.

In brief, our work aims to answer these research questions: 1) What is a software behaviour? 2) What is the behaviours assessment? 3) Who is in charge of performing the behaviours assessment? 4) What are the challenges this kind of assessment can solve? 5) What are the problematics the behaviours assessment can introduce?

We have divided this paper in the following sections. Section II describes a series of security threats affecting IoT

This work has been funded by the European Union - NextGenerationEU within the framework of PNRR Mission 4 - Component 2 - Investment 1.1 under the Italian Ministry of University and Research (MUR) programme "PRIN 2022" - grant number 2022598LMZ - AsCoT-SCE (Assessing Compliance of IoT API for Security Critical Environments) - CUP: H53D23003430006.

L. Ardito and M. Valsesia are with the Department of Control and Computer Engineering, Politecnico di Torino, Torino 10129, Italy. (e-mail: luca.ardito@polito.it; michele.valsesia@polito.it).

A. Aldini and G.M. Bianco are with the Department of Pure and Applied Sciences, University of Urbino Carlo Bo, Urbino 61029, Italy. (e-mail: alessandro.aldini@uniurb.it; giuseppe.bianco1@uniurb.it).

[1] Shirey RW, Internet security glossary, 2000. [Online]. Available: https://tools.ietf.org/html/rfc2828. Accessed on: Nov 25, 2024.

systems and proposes some possible solutions. Section III addresses the certification process and some standards from which we have taken inspiration for our assessment model. Section IV introduces the behaviours assessment and its possible applications in certification processes. From Section V, the focus is on the manifest-producer library. Section VI outlines the library's main features, motivating our choices and pointing out possible limitations. Section VII describes the modular structure of the library, explaining the role played by each module and how they contribute to the overall analysis result. Finally, Section **??** illustrates a practical use case of the manifest-producer, detailing its application in analysing firmware behaviour and generating a comprehensive certification report.

II. Security Threats

An IoT system is the fusion of multiple technologies, hence all existing security threats have converged to IoT as well [8]. Over the years, numerous attacks have been recorded where IoT is being used as a tool to target an existing infrastructure [9]. Nowadays, IoT devices are mostly on and connected to the Internet, thus the potential exploitation of their vulnerabilities is quite plausible [10]. Furthermore, some of the oldest IoT devices have no update mechanism, hence only one firmware vulnerability can cause serious damages over a wide spectrum of devices [11].

In any device, a firmware is a type of software which interacts with the hardware [10]. A device stores the firmware binary in a volatile or a non-volatile memory, depending on its hardware characteristics. A firmware fully encapsulates an IoT device, including the application and the operating system. Indeed, depending on the type of hardware architecture in use, the firmware layer of an IoT system can be quite thin, in the form of an application, or it can be as thick as the entire machine software.

At firmware-level, attackers are focusing more on firmware vulnerability exploitation because the security on this part of a computer stack has often been overlooked. Therefore, the successful exploitation of any portion of firmware usually results in the complete device hijacking [9].

Unfortunate business trends can affect the security of IoT firmware. Manufacturers can make use of old and free codebases and libraries, which might lead to insecure firmware [12]. Vendors can ask to introduce hidden backdoors during the development process to update or manage devices remotely, thus leaving a remote login interface with an unsafe password for authentication [13]. For selling a more competitive product on the market, it is given major attention to performance than security [10]. Furthermore, there are no regulatory agencies which enforce security recommendations on IoT manufacturers.

A. Vulnerabilities

Since an IoT firmware is a composition of different technologies, some of the security threats which affect an IoT device come from these technologies [10].

Memory vulnerabilities such as buffer-overflow, pointer violations, integer overflows, uncontrolled format strings missing bound checks, and type confusion are among the most common due to memory constraints present in IoT devices [14]. All these memory corruptions can lead to system crashes, privilege escalation, and code injection exploits.

IoT devices communicate and share data with a certain number of external interfaces. Among them, there are web interfaces: APIs for connecting to other devices, which are part of an IoT ecosystem, such as mobile phones, peers, servers, cloud or edge computing devices. Therefore, these interfaces also suffer from a number of web insecurities [8].

B. Recommendations

As pointed out before, manufacturers tend not to follow standards on how to develop and integrate a secure IoT system for various reasons [10]. This attitude leads to poor security design. Therefore, deciding not to comply with standards might cause some problems such as data leakage, insecure booting, weak encryption algorithms, inadequate credential management, and outdated services.

Some recommendations have been proposed to solve these vulnerabilities [10]. Among the ones which can concern a great variety of IoT devices, there is the recommendation to choose an operating system which allows exploiting the underlying hardware architecture of an IoT device at fullest. Removing unnecessary libraries and packages also reduces the possibility of potential exploits. The second one is to adopt a secure wireless service to install and update firmware versions over the air: the OTA mechanism [15]. The underlying IoT devices perform regular checks with OTA servers for updates. Also, OTA servers notify all connected devices once new updates are available. However, OTA is not so common in many IoT devices [16], and since software for firmware updates is usually vulnerable [2], this mechanism might lead to improved system security. The last recommendation is to test a firmware with vulnerability finding software during its development and before its release [10].

Although Nadir *et al.* recommend following the guidelines above, they firmly believe IoT standardization might be the way to resolve the significant security threats present in IoT systems. Since manufacturers are not applying security by design, providing securing development guidelines is a fundamental building block for IoT security, hence certifications help advance security by design in the IoT ecosystem [1].

III. Certification

A product compliant with a standard represents an assurance of good quality because it will operate according to given specifications [17]. This assurance also gives consumers a certain amount of confidence regarding the safety and reliability of that product. Furthermore, the presence of standardization ensures interoperability, which enhances the integration and sharing of information between systems, significantly reducing

[2]ISTR ransomware & business 2018, *Broadcom*, 2018. [Online]. Available: https://docs.broadcom.com/doc/istr-23-2018-en. Accessed on: Nov 25, 2024.

the heterogeneity incompatibilities among IoT devices [2]. All these aspects constitute the main points which lead to certification [17].

The first definition of a security certification standard originates in the defence domain [18]. Indeed, military systems need a greater level of protection against security threats than commercial domain systems. However, the entire certification process was lengthy and costly; it could last for years and cost millions of dollars, hence an unacceptable condition in the IoT world, where fast placement in the market is one of the most important elements for business success [19].

In the following subsections, we will analyse two recent certification standards which helped us shape our behaviours assessment model.

A. Matter

Matter is an open-source, royalty-free connectivity standard which offers a secure, reliable, and seamless way for IoT devices to communicate and interact, regardless of the manufacturer [20]. Interoperability is a core strength, allowing all Matter-compliant devices to work together without conflict. This standard has been developed by the Connectivity Standards Alliance (CSA-IoT), which safeguards the communication between devices through reliable practices and protocols.

Matter implements a protocol stack which resides on the OSI application layer. Each stack level receives a package from a higher level, processes the package's data, creates a new package, and then passes it to a lower level. When an associated device begins to receive data, it moves backward up the protocol stack by performing a reverse of the packaging operations, delivering the unwrapped data to the receiver's application layer, which can thus consume the information.

The Matter certification process starts with a device manufacturer joining the Alliance as an Adopter, Participant, or Promoter profile. These membership profiles guarantee to a manufacturer different privileges and require the payment of an annual subscription fee to keep being part of the Alliance. Those who wish to obtain the certification for their products, without joining the Alliance, can take advantage of the Certification Transfer Program (CTP) as an Associate. The profile of an Associate does not require any subscription fee, but since the certification process is not cost-free, a subject must pay a one-time fee for each product to certify, in addition to a reduced annual fee to keep the certification active. Once registered, a manufacturer receives a Vendor ID and Product ID code, uniquely identifying a vendor and its products. The certification process consists of a series of steps performed through a web tool provided by the CSA.

During each step, the manufacturer must produce some documents to ensure its product meets the required certification standards. The first document is the Declaration of Conformity. This report attests whether the product properties declared by a manufacturer have passed a conformity assessment. The PICS, instead, is a structured document detailing the requirements met by implementing the Matter Protocol. The Final Test Report provides feedback on the overall state of the device to market. The Dependent Transport Attestation confirms whether a product developer has obtained the certification for the transport platform in use for the Matter component. The Security Attestation lists the security requirements and the level of compliance the product must guarantee based on the Matter standard. The whole documentation attests whether a Matter device meets all protocol requirements. Once submitted through the certification web tool, the CSA verifies the relevant criteria and if they are deemed valid, the device obtains the certification.

Authorized laboratories, contracted by the CSA, are in charge of performing the assessments described in Section I.

Even if Matter is a freely implementable standard, there is no assurance it will not evolve into a proprietary standard dominated by a few key players, potentially sidelining smaller IoT companies [20]. As evidence of this lack of trust, many Matter-compliant manufacturers still reserve advanced monitoring features for their official applications, not providing full access to third-party controllers.

B. PSA Certified

Chips in the hardware contain critical information, such as the encryption key [21]. Applications often rely on the secret keys embedded in these chips to perform their tasks. Therefore, Arm and other chip security research organizations jointly launched the Platform Security Architecture (PSA) certification program in 2019 to improve IoT chip security [3].

This program aims to foster better IoT security practices by certifying whether an IoT chip has reached industry security standards [21], offering manufacturers guidelines for designing trusted computing IoT systems [22].

The PSA certification consists of four levels, with Level 1 being a basic security requirement and Level 4 indicating the highest security possible [21]. Level 1 ensures IoT compliance to baseline security requirements for chip platforms [22]. Level 2 offers protection against various risks such as remote attacks, rogue device impersonation, and misuse of firmware updates and secure storage. Level 3 provides protection against hardware and data manipulation, undetected memory content alteration, side-channel attacks, and physical probing on the chip's surface. Level 4 offers enhanced protection of high-value assets which can secure valuable AI models and future-proofing systems against new attack methods [4].

Until now, a few chips have obtained the PSA Levels 1/2 certification, and only two chips have been able to obtain a Level 3 certification in 2021 [21].

As for the Matter case, a series of certification laboratories are in charge of performing all necessary assessments to assign to a chip the most suitable certification level [5].

[3]PSA Certified, *PSA*, 2021. [Online]. Available:https://www.psacertified.org/. Accessed on: Nov 14, 2024

[4]PSA Level 4, *PSA*, Apr. 2024 [Online]. Available: https://www.psacertified.org/blog/psa-certified-announces-new-psa-certified-level-4-certification-and-continued-partner-growth/. Accessed on: Nov 14, 2024.

[5]PSA Certified Evaluation Labs, *PSA*. [Online]. Available: https://www.psacertified.org/getting-certified/evaluation-labs/. Accessed on: Nov 14, 2024.

C. Challenges

1) Economic Constraints: The certification standards described previously face some challenges. The first is an unsustainable membership or certification cost, which forces small companies to sell their own products without adequate security guidelines and leads to the threats addressed in Section II. Moreover, any changes to a device configuration, or even an update to that, may invalidate some parts of the certification process, further extending costs and time to market a product.

2) Technical Limits: Not all components of a product can be entirely certified. For example, third-party public clouds are very difficult to analyse because of the scope, scale, and legal obstacles needed for testing [3]. Furthermore, some components are generally untestable due to their third-party libraries, black-box functionality, scalability, and heterogeneity.

3) Documentation Problems: Legal terms usually fill in certification documents, making the comprehension difficult for consumers not accustomed to a legal language [23]. Furthermore, when IoT system logs do not contain much technical language, they are more understandable, therefore consumers will be more and more confident in that system over time [3]. Since the quality of certificates can vary from one to the other, good documentation might also help a manufacturer in choosing the most viable one for the company's needs.

4) Process Challenges: A certification is always a balancing act between certifiers and manufactures, since the former can always project liability onto the latter, creating a certain amount of tension [3]. To avoid dealing with certification authorities, some manufacturers prefer self-certifying their own processes, thus becoming a certification authority themselves. This approach, though, has a limited and possibly questionable relevance due to the potential conflicts of interest. The great majority of certifications have been thought and conceived for manufacturers, so they do not take into account final users in their creation process [1]. Therefore, IoT systems do not receive any feedback about their security state or usability from users, probably hindering the improvement of some cumbersome parts of the certification process.

In conclusion, the possibility of certifying an IoT device to a safe and reliable level of confidence usually requires a great amount of time and money, making this one of the biggest problems for IoT systems.

IV. Behaviours Assessment

A software can perform a series of operations during its execution. With the term software behaviour, we intend all possible effects which may arise from executing an operation. Therefore, an operation determines none, one or more effects. For an IoT system, we can define as effects: the state in which a device resides after an operation, the interactions among a device and external resources such as network and standard streams, stdout and files, or an effective action, such as turning a light on. If we extend this definition to a general purpose software, some effects remain the same of an IoT system, in addition to all the others possible states, events, and interactions which might occur among the processes interacting on an operating system. Therefore, behaviours are all the effects which influence a system.

The behaviours assessment is the analysis of software behaviours. It evaluates and describes the effects produced by each software operation on a system.

A. Software Analysis

The analysis of a binary is often the main way to detect behaviours which rarely occur at runtime and might lead to unknown effects. This evaluation makes use of static analysis techniques to parse and extract information from a binary. In an operating system, this approach translates into retrieving all kernel's system calls which APIs need to invoke to perform one or more operations. From this list of system calls, it is then possible to describe the majority of software behaviours. Starting from the data collected, we can locate which APIs open a file and which ones access a network. In an embedded environment, instead, the analysis can detect behaviours from the assembly lines which compose a firmware, thus allowing the evaluation of IoT devices as well. For clarity, henceforth, we will use the term manufacturer when we are referring to a firmware developer.

B. Certification Process

This assessment has been thought and conceived as a process which starts from a developer and ends with a certifier. A developer executes a behaviours assessment tool, which requires as input a software or firmware binary. Most of the time, a series of guidelines direct the development of a software, and some of them might lead to accepting or discarding behaviours. For example, if a rule imposes a software not to access the network, but some of its APIs present this behaviour in their implementation, so these APIs need an immediate modification. Therefore, a developer must provide an alternative implementation in order to comply with the guidelines. On the other hand, a certifier can execute the same tool to evaluate whether the behaviours declared by a developer are reliable. If a software or firmware behaves differently from what it has been declared, a certifier stops the certification process and asks for explanations from a developer.

C. Idea Conception

The idea of a new kind of assessment was born from the analysis of Matter and PSA Certified standards. None of them considers in great detail how software modifies the surrounding environment. As addressed in Section III, Matter defines a protocol stack which manufactures should use to implement IoT devices firmware. This approach increases interoperability and makes an IoT system safer, but it does not provide any tool to describe what software does within that system. Even the certification process does not fully consider these situations. A manufacturer has to declare whether a software complies with the Matter standard filling in some documents. Subsequently, a certifier attests whether the declarations contained in these documents are valid and, if they are, approves and releases the certification. However, this procedure is prone to human errors. A manufacturer can forget to add some important

device aspects, whereas a certifier might not pay attention to some parts during testing. The behaviours assessment might improve a certification process identifying unknown, or taken for granted, behaviours through the automation of some of its parts. From PSA Certified, we have been inspired by its level concept to shape the layer idea of our behaviours assessment model. Layers differ from levels because they do not presuppose an order. A layer identifies all those behaviours belonging to a same category. For example, the network layer will contain all those behaviours which produce an effect on a system network. Therefore, a certifier can describe a software in a more human-readable way through the composition of layers.

D. Strengths

Our assessment tries to avoid incurring as much as possible in the challenges described in Section III. Since the behaviours assessment requires a sharing of information among manufacturers and certifiers, hence the general trust level increases significantly. Indeed, both of the parts clearly know the role of the other one because the assessment procedure, and its inputs, are always the same. Only the output interpretation might be different. A manufacturer knows what a certifier must evaluate: whether the declared behaviours are correct. On the other hand, a certifier must only certify whether there are no mistakes or missing behaviours. Due to automatization, the behaviours assessment reduces the time necessary to certify a product. A long-term certification is indeed one of the causes which bring a manufacturer to rely on self-certification.

E. Problematics

1) Impossible Assessment: If we consider a general purpose software running on an operating system, we can describe nearly perfectly the behaviours which can arise from its execution. Since each API calls precise system calls to perform an operation, we always know their goal and outcome either from their results or from their interfaces. The situation is different for IoT devices firmware. In that case, the behaviours assessment can only verify whether an operation, which interacts with other devices, has been performed, such as a network access. However, it cannot provide any information about the behaviour of an operation which interacts with an external component connected to a board. For example, if a firmware sets a pin impedance to a determined value, we cannot verify what will be the final device behaviour. We can only state the pin value has been changed. Therefore, a software operation cannot always determine the final effect on a device. These kinds of situations stand on firmware definition: a strict bond between hardware and software. As of now, we cannot solve this problem from a software perspective, hence it is necessary to consider the entire device to evaluate with certainty the correspondence among a behaviour and the relative hardware response.

2) Lack of Trust: This assessment cannot even start without a binary. If a manufacturer decides not to share firmware due to privacy or commercial concerns, the entire analysis is impossible to make. As we have pointed out in Section III, one of the challenges concerning a certification process is the trust a manufacturer puts in a certification authority. Distributing a software binary after a theft, or due to poor internal management, is simpler than doing the same with a final product. Therefore, there must be a legal contract which obliges a certifier to assume all responsibilities in case these kinds of acts happen. A possible solution to safeguard both parties might be the creation of a platform which guarantees common security policies for sharing information between a manufacturer and a certifier. This proposal might also increase the level of trust in a certification authority and, at the same time, simplify the entire process.

In conclusion, the behaviours assessment provides a new approach to investigating software behaviours. It can help manufacturers during development, but also certifiers in evaluating a product. Due to its automatization, the behaviours assessment can significantly reduce time and increase the trust among parties in a certification process.

V. MANIFEST-PRODUCER LIBRARY

As seen in the previous sections, certifying IoT devices is a complex challenge, mainly due to the difficulty of performing a systematic analysis of firmware behaviour. By systematic analysis, we mean an organised and structured process that evaluates how firmware interacts with hardware and other system components.

In this context, analysing a binary requires reverse engineering practices. However, these practices require an in-depth knowledge of processor architectures, low-level languages, such as assembly, and the ability to use specialised tools, which often have a steep learning curve. Therefore, there is a need for an alternative that reduces the necessity for advanced reverse engineering knowledge to describe software behaviour.

To meet these needs, we developed a manifest-producer library based on the tool of the same name produced in our previous work [24]. Initially, the manifest-producer was designed to perform, through a flow of concatenated operations, an analysis of ELF binaries, a format widely used in IoT-constrained devices since based on the Linux operating system [24]. We have evolved the manifest-producer into a modular library. Its functionalities automate processes related to reverse engineering. Examples include code disassembly, analysis of ELF sections, extraction of functions from the symbol table, and creation of a call tree structure.

We define a call tree as a structure representing the interactions between binary functions. In this structure, each function appears as a node in a tree. Starting from the tree's root, usually the main function, it expands to the leaves, which are terminal nodes representing low-level functions and system calls. System calls are essential for this analysis because they request a service at the kernel level, which handles interaction with the underlying hardware. This structure helps certifiers and developers to visualise the interaction between functions within the IoT device's firmware or general-purpose software. Since the library applies to devices with an operating system and a kernel, we will use the term software to refer to both firmware and general-purpose software.

Knowing how these functions interact allows discriminating among expected or unexpected behaviour. We will call expected behaviour what the software should be doing according to the intentions declared by the developer, whereas unexpected behaviour refers to those software behaviours that are not expected or predictable. This separation is important since unexpected behaviour can reflect security problems or malfunctions.

We develop the manifest-producer library in Rust, a programming language known for its safety and performance. Rust is distinguished by its rigorous approach to memory management and concurrency, which makes it ideal for developing tools that must be both safe and efficient at the same time.

VI. Library Features

The following section describes each of the main library features by introducing its key concepts, exploring its technical motivations and discussing its limitations.

A. Rust Programming Language

Rust is a programming language that has gained popularity in recent years because it combines security and performance. Due to its different memory management, Rust prevents critical vulnerabilities such as buffer overflows, estimated to account for about 70% of reported CVE vulnerabilities [25]. At the same time, Rust efficiently provides direct low-level control over resources. Unlike most other languages, it does not rely on garbage collection, thus avoiding extra overhead associated with automatic memory management [26], [25], [27]. These properties make Rust an ideal choice for developing our manifest-producer library, where security and high performance must be guaranteed for reverse engineering operations, such as code disassembly.

Rust's modular design revolves around crates. A crate is a compilation unit and the basic building block of the modular structure of a program. Crates can be compiled into either binaries or libraries. This modularity aligns well with the requirements of our manifest-producer, allowing for clear separation of functionality and streamlined maintenance.

However, Rust comes with some initial challenges, especially for developers who come from other programming languages. According to Zhu *et al.*, Rust's unfamiliar semantics and security model make it quite different from classic programming languages such as C or C++. Furthermore, the security-focused approach, which imposes rigorous checks during compilation, makes it difficult for developers to write code that respects all language rules [28].

B. Reverse Engineering with Static Analysis

According to E. Chikofsky and J. Cross, reverse engineering (RE) consists of analysing a system to identify its components and interrelationships, creating a series at a higher level of abstraction [29]. In software, RE is typically performed on binaries, when source code is unavailable, to understand their internal logic and structure. This spans from the recovery of software design details to the identification of interdependencies or the understanding of the requirements implemented in a system.

On the other hand, static analysis involves examining the compiled code of a binary, or its source code if available, in order to extract such information as function calls, control flows, and system-level interactions [30]. The manifest-producer library uses the ELF binary reverse engineering approach, based on static analysis techniques. It will enable the extraction of meaningful information without executing the binary itself, thereby ensuring files, which are poorly documented or potentially unsafe [31].

Static analysis-based reverse engineering suits our use case, which aims to detect firmware behaviour in support of certification. It extracts structural information from binaries, such as function interactions, in order to infer behaviours and verify developer self-declarations. Being non-intrusive, it avoids firmware execution and thus eliminates risks of crashes or malware activation. While dynamic analysis, widely adopted in IoT for runtime monitoring in sandboxed environments [32], [33], offers detailed behavioural insights, it also requires complex environment setups and hardware emulation, limiting scalability and early-stage adoption in certification. Conversely, static analysis provides a reasonable reconstruction of a program control flow and logic, making identifying dead or unreachable code possible. While such code does not necessarily affect runtime behaviour, its presence may indicate inconsistencies or discrepancies with the developer's self-declared functionalities, supporting a more comprehensive behavioural assessment.

Despite these strengths, static analysis has inherent weaknesses when applied to reverse engineering. Indeed, dynamic constructs, such as runtime generated addresses, or obfuscated code, represent weaknesses for static analysis. Compilers, especially the ones with optimisations enabled, do not always resolve all addresses at compile time. In that case, recovering complete information without a runtime context is impossible. Moreover, debug symbols map memory addresses to high-level constructs, such as function and variable names. Without this abstraction, binaries appear as a raw sequence of addresses and instructions, further complicating the analysis. Another weakness is the Position-Independent Executables (PIE) effect, which introduces address randomisation, a technique designed to prevent static address-based attacks such as buffer overflows. Although this improves security, it poses challenges in reconstructing control flows and function interconnection, making static analysis very complicated.

C. Disassembly

Disassembly is a necessary step in reverse engineering, as it changes machine code into something which is human-readable, such as assembly language [34]. We have designed our library to disassemble statically compiled ELF binaries. This configuration ensures that the binary includes the code of functions belonging to external dependencies. As a result, the manifest-producer can disassemble these functions, enabling a more complete and detailed comprehension of the software under analysis.

Disassembling a binary requires two widely used algorithms to translate its machine code: Linear Sweep and Recursive Descent. The first one processes the binary linearly, instruction by instruction, starting from the first byte of the .text section, which contains the code to be executed [35]. Linear Sweep assumes all .text section bytes correspond to valid instructions. It does not consider the control flow, so the logical order of execution determined by branch and jump instructions. This simplicity makes it efficient with statically compiled binaries, which are typically linear in their structure and rarely include breaks which jump to other code sections. In contrast, Recursive Descent maps the control flow by scanning branches and jump instructions, hence following all possible execution paths [35].

This library adopts the Linear Sweep algorithm and analyses one function at a time. The symbol table helps to locate each function, disassembling its binary code block separately [36]. For practical implementation, the manifest-producer integrates the *Capstone* disassembly engine, a widely adopted framework known for its portability and multi-architecture support. Capstone also provides Rust bindings, which enable seamless integration within the manifest-producer pipeline.

The disassembly process operates by iterating over the set of instructions of each function extracted from the symbol table. Capstone decodes the machine code sequentially, producing the mnemonic and operands for each instruction. The library attempts to resolve known function names through symbol analysis when encountering call instructions.

Although the current implementation targets x86_64 firmware, using Capstone ensures future extensibility to other architectures, such as ARM, MIPS, or RISC-V, often found in IoT systems. Moreover, the manifest-producer library does not provide a public disassembly API because it is a private function of the Functional Interaction module, described in the next section.

D. Mangling

In compiled programs, a symbol represents either a function, variable or other program element of a code [36]. Compilers use a process called mangling, to make every symbol unique and identifiable. The importance of mangling comes from its significant contribution to features such as function overloading, where multiple functions in the same program may have the same name, but different lists of parameters [37].

Mangled names allows a linker to correctly resolve function calls or variable references. However, they encode names in a verbose format, hence making the symbols much more complex to read, especially those which have been reverse engineered.

To improve the readability of symbols during analysis, the manifest-producer library integrates a demangling algorithm to reverse a mangling process. It transforms the encoded names to their original or similar form, making them more readable. Demangling is especially useful for reverse engineering and static analysis tools, where it simplifies the output for binaries compiled in C++ and Rust languages.

The programming language used to compile the firmware guides the demangling process. The Binary Inspection module, detailed in the next section, infers this language through its APIs during the preliminary analysis. This language information proves crucial to determine whether mangling exists and which decoding strategy to apply. The manifest-producer focuses primarily on three languages: C, C++, and Rust. Among these, only C++ and Rust typically apply name mangling. Therefore, the library integrates two Rust crates: *cpp_demangle*, specialised in C++ symbols, and *rustc-demangle*, analysing Rust ones.

The manifest-producer does not expose a public API for demangling. Instead, the Function Detection module uses this functionality internally, where it demangles all function symbols before further processing. By automating the demangling process, the manifest-producer generates human-readable and accessible analysis reports. This way, it keeps the output simple without adding complexity.

E. Data integrity verification

Another feature associated with this library is the computation of a digest. The digest represents a digital signature of a file. Calculated from the binary content, it changes its value at the slightest binary modification, immediately revealing any tampering attempt [6].

Due to this mechanism, the certifier might verify whether the binary provided by a developer has not been tampered with. In other words, this mechanism assures a certifier that the examined binary is identical to the one supplied by a developer.

In any case, for an adequate level of security, it is necessary to use modern and secure hash algorithms, such as SHA-256 or higher. This process, which detect any change, creates a level of trust between a developer and a certifier, strengthening the security and reliability of the entire certification process.

VII. Library Layout

The manifest-producer library performs static analysis over ELF binaries, with a focus on automating reverse engineering mechanisms, producing structured results through the use of various crates. This modular approach leads to flexibility over our library architecture.

As Figure 1 illustrates, we have divided the library into two main crates. The first is the Backend crate, which processes ELF binaries through four analysis modules: Binary Inspection, Function Detection, Functional Interaction, and Finding Entries. Each of them can operate independently or sequentially. The second is the Frontend crate, which renders the Backend's JSON outputs as static web pages.

A. Backend

The Backend crate is in charge of analysing ELF binaries. Each internal module exposes public APIs that return data structures designed to capture the results of the analysis. For

[6]NIST and Q. Dang, "Secure Hash Standard," in *Federal Information Processing Standards (FIPS)*, National Institute of Standards and Technology, Gaithersburg, MD, 2015, doi: 10.6028/NIST.FIPS.180-4. [Online]. Available: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=919060 Accessed on: Dec 5, 2024

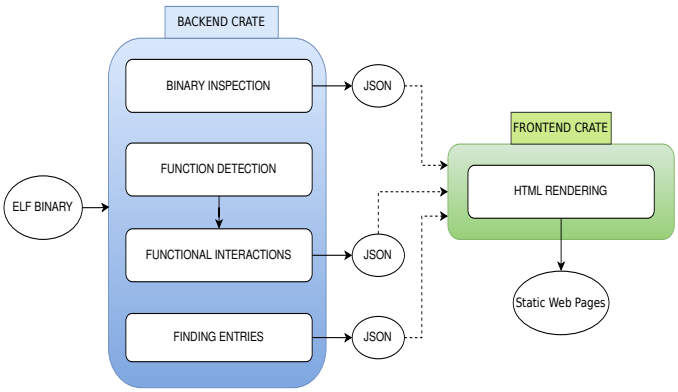

Fig. 1. Architecture and components interactions of the manifest-producer

example, the Function Detection module generates a vector of data structures that contain information about the detected functions, such as their name and memory entry address. In turn, the Function Interactions module enriches with additional data the obtained functions, such as disassembled code, direct calls to other functions or any syscalls. The Backend also serialises every data structure into a JSON files, which the Frontend crate can directly use to generate interactive visualisations. This output supports various other uses, such as providing useful information for security audits and additional certification reports. In fact, from a certification perspective, Backend helps a certifier verify whether software contains undeclared functions or any unexpected dependencies not declared by a developer.

The following subsections provide detailed descriptions of the functionality and implementation of each module.

1) Binary Inspection: The first module offers APIs to verify specific binary aspects, ensuring it meets the requirements for the manifest-producer to perform reverse engineering. Firstly, it checks for the presence of debug symbols, indispensable for correctly interpreting a binary. Then, it checks whether a binary has been statically compiled to make the identification of all embedded functions possible. It also checks whether the Position-Independent Executable option has not been enabled to avoid some complexities in the resolution of function addresses. Finally, it checks all critical sections of the ELF binary, such as .text, to ensure whether they have been correctly reported, possibly signalling problems caused by compilation errors.

2) Function Detection: The second module automatically detects binary functions by scanning its symbol table. This table contains metadata on each function, such as the virtual memory address of the first instruction and the block size of the executable code. By knowing this information, the APIs within this module map the executable code block to the memory range associated with each function. As discussed in the previous section, when our system identifies a binary which have been compiled from either a C++ or a Rust source code, this module performs name demangling to transform obfuscated symbols into human-readable function identifiers. We subsequently store all retrieved functions as structured entries, which incorporates raw metadata and enriched attributes needed for subsequent analysis phases. Notably, the memory layout information of this module serves as a fundamental input for the disassembly process performed by the Functional Interactions module.

3) Functional Interactions: The Functional Interactions module analyses how functions interact within the binary by examining their machine code in order to identify their connections direct relationships. Starting from the memory layout with the data structure provided by the Function detection module, it disassembles each function using the Capstone engine. It scans the instructions of a function to detect two main types: function and system calls. When the module encounters a call instruction, it resolves the destination address to a known function name if possible; otherwise, it records the raw address. The module stores this information in a function node's field, representing all directly invoked functions. If the module detects a system call, it extracts the syscall number from the *rax* register, on x86_64 architectures, and flags the function accordingly. Therefore, the module enriches each function with its disassembly code, the list of directly called functions, and all metadata of detected syscalls. However, the module creates only a single-level hierarchy: each function node maintains direct references to its immediate callees stored as function names, but these child functions do not store further references to their children. This lightweight design keeps the memory footprint manageable while preserving the ability to traverse deeper levels by retrieving child functions from the main structure as needed. The Functional Interactions module serialises this structure into a JSON file. The Frontend crate can then process the JSON to generate the call tree of a selected function. External tools can also use the file to build custom control flow graph tailored to specific analysis needs.

4) Finding Entries: A significant improvement compared to the previous version of our manifest-producer is the possibility to automatically identify entry points without any developer intervention [24]. We define an entry point as a function which has never been called by any other function. In order to identify an entry point, the backend only keeps those functions which do not receive an input call. This approach even finds out those functions which have been optimised away by a compiler, which can optimise part of the code by eliminating unused functions. Although such functions are invisible in the assembly code because they do not lie in the main flow of execution, they are nevertheless identified the same as entry points by the Backend, since they do not receive entry calls.

B. Frontend

The Frontend crate converts data generated by the Backend crate into static web pages, offering an intuitive way to explore binary analysis results. It organises data through predefined templates, which simplify page navigation, other than providing a better comprehension of binary behaviours. The view is interactive, allowing users to smoothly investigate the structure, such as functions, assembly instructions, and functional dependencies. Minijinja, a lightweight Rust-based templating engine, populates these HTML templates injecting values which have been extracted from the JSON output files

produced by the Backend crate. This approach ensures that layout and styling remain decoupled from the analysis logic, while allowing consistent and reusable formatting across all view pages.

Below, we present all HTML templates used for the Frontend outputs:

1) Functions Table: It contains a more visualisable version of a symbol table, showing all data collected during the analysis. This web page has been represented as a table, where each row is a function. Each column contains its demangled name, its initial virtual address, which locates its machine-code block, the number of times this function has been invoked by other ones, and its invocations to child functions. Child functions are all those functions explicitly called during the execution of a function, creating a parent-child interaction in the call hierarchy.

2) Assembly Code Viewing: This is a dedicated template for displaying the assembly code of the symbol table's functions. We have designed its structure to allow direct access to the instructions of each function and the relative calls.

3) Interactive Call Tree: The Frontend provides an API for constructing a call tree from the JSON output provided by the Backend crate. This API parses JSON data retrieved from Finding Entries module to establish tree roots. To manage large or deeply nested binaries, the implementation employs iterative traversal approach rather than recursion, preventing stack overflows and maintaining stable performance in resource-constrained environments. Additionally, we use the lazy loading technique to optimise performance, particularly for large trees. This technique helps to reduce initial page load times and ensures smooth navigation throughout the tree.

C. Integration Workflow

The manifest-producer provides two distinct integration approaches that accommodate diverse analysis requirements. For development environments, external tools can directly invoke specific APIs exposed by the internal Backend module, enabling the selective execution of individual analysis steps based on workflow needs. Moreover, the manifest-producer offers a standalone command-line interface for automated scenarios, orchestrating the complete analysis pipeline, accepting ELF binaries as input and generating JSON as output files.

VIII. Certification Process Example

This section integrates the manifest-producer into a firmware certification process involving a smart light developed by a small company. As part of this process, a certifier analyses the behaviour of the firmware and the output is a report describing the results of the evaluation.

We can summarise the structure of the report in four parts:

- Binary Identification, which establishes a baseline for the entities involved in the process and the binary under analysis;
- Behaviours Assessment, where the certifier documents the high-level functions and system calls detected during the analysis;
- Behaviours Comparison, which evaluates the alignment between the developer's declarations and the binary behaviours observed by the certifier;
- Parties Confirmation, where a certifier and developer validate the assessment by placing a signature on the report.

A. Light Firmware

To demonstrate the functioning of our library, we chose the firmware of a smart light as input for the manifest-producer. The firmware has been specifically designed by a small company for smart home environments. At the time of writing, the product has not yet been released to the market and currently being tested in a company-owned house.

The firmware has been written in Rust, so it embodies all language features described in Section VI, and supports all microcontrollers with x86_64, ARM, and RISC-V hardware architectures.

From an operational point of view, it acts as a server where each REST API triggers a task. For example, the */light/on* route, invoked by an external agent, triggers a series of operations on the device which turn the light on. The firmware also implements other additional tasks: the first turns the light off, while the second toggles it at a certain frequency.

B. Developer Role and Self-Declaration

As mentioned before, we have considered, as the main actors involved in the process, a software developer and a certifier.

In our case, the developer has implemented specific functionalities for a smart light device and is preparing the firmware for market release. Before this can happen, a certifier must analyse the firmware's behaviour to ensure it complies with the expected requirements. A developer computes the digest from the binary and sends both the binary and its digest to the certifier. Moreover, a developer also writes a self-declaration document listing the intended behaviours for that firmware.

The self-declaration is a very important document for the certification process, complementary to the analysis performed by the manifest-producer. Although the manifest-producer extracts some data from a binary, it may not always provide complete visibility over all firmware behaviours. By comparing the declared behaviours with the ones identified during a certifier's assessment, any discrepancies or undocumented actions can be discovered, ensuring a more comprehensive and accurate assessment.

For this use case, the self-declaration includes:

- Turn the light on
- Turn the light off
- Toggle the light at a certain frequency

C. Assessment and Report Creation

A behaviour assessment process starts with a certifier comparing the digest computed by a developer with the one calculated from the binary. If the two digests are equal, the process

can continue the first step. In case of a discrepancy, due to network problems that could have corrupted the download or an effective binary difference, the certifier must reject the binary and request the correct one.

The analysis begins with the execution of the Backend crate. The module identifies all information a certifier needs to conduct a comprehensive assessment of firmware behaviours. After Backend has generated all data, Frontend creates a call tree and displays its data in static web pages. Depending on a binary size, this automatic process ranges from a few seconds to several minutes. As the last operation, the certifier obtains sufficient data to assess firmware behaviours and create a detailed report.

Binary Identification: The first part of the report provides a common and verifiable reference between the entities involved in the certification process. To ensure the authenticity of the evaluated file, it specifies the name, size, and digest of the binary. It also identifies the parties involved in the process: who is the certifier assigned to the analysis and who is the developer providing the binary to assess.

This approach ensures that we are going to base all the following assessment steps on a clear identification of the binary and entities involved.

Behaviours Assessment: This part links data automatically generated by the Backend crate to the observations made by a certifier, presenting concisely and transparently through two tables the operations executed by the binary.

The first table lists all functions which a certifier considers essential to define binary behaviours. A function reports the name, its address in memory, and the number of times this function has been called. For each function, a certifier adds a short description summarising its presumed behaviour detected during static analysis. As a result, these descriptions reflect the observations made by a certifier, but they may not be consistent with the intent of a developer.

The second table focuses on the system calls identified during the analysis. These calls represent the interface between the binary and its underlying hardware. They support the interpretation made by the certifier for the high-level functions, since each system call produces the effects, therefore the behaviours of these functions.

In this paragraph, we can summarise the possible content of the first table as filled in by the certifier, omitting the one with the system calls, as it depends on the context. Indeed, an open system call may have different meanings, such as opening a file, but the term file might also refer to physical devices, pipes, sockets, and other resources in a system environment. Therefore, interpreting and presenting these calls would be complex without a more in-depth analysis of the disassembled code. In the analysis of the smart light firmware, several high-level functions have been identified that play a role in controlling the light status and managing internal parameters. The `turn_light_on` function updates three fields in its internal data structure indicated by %rdi register. Specifically, it stores the values passed through %rsi, %rdx, and %cl registers at offsets 0x10, 0x00, and 0x08, respectively. Additionally, it clears the field at offset 0x50. This pattern suggests that the function sets up multiple parameters required to activate the light. The `turn_light_off` function also writes the input from %rsi register to the base address of the data structure, and clears the field at offset 0x30, possibly indicating a deactivation or shutdown signal. No additional configuration steps are performed, suggesting a simpler control flow than `turn_light_on`. The `toggle` function exhibits a memory access pattern similar to `turn_light_off`. It stores the value in %rsi register at the base of the structure and writes zero at offset 0x30. However, based on its semantic role and naming, the value in %rsi is likely chosen to invert the currently active state. This behaviour aligns with a toggling mechanism, possibly reflecting a low-to-high or high-to-low transition on a control pin.

In addition to the functions that directly manage the state of the light, the certifier has identified two additional functions worthy of attention. The `update_energy_efficiency` function copies the values received in the %rsi and %rdx registers to specific positions of the data structure indicated by %rdi, also zeroing a field at offset 0x28. This behaviour indicates a direct update of internal parameters, probably related to energy efficiency. Finally, the `info` function copies the value in %rsi to the beginning of the data structure and clears the field at offset 0x30. This minimal update pattern may reflect a metadata change triggered by an internal state transition.

These functions represent fundamental elements for understanding the behaviour of the firmware and require specific clarifications from the developer to confirm the functional assumptions and the design intention. However, a complementary analysis at the hardware level must validate some of these interpretations, particularly those related to interactions with the physical environment, such as turning a control pin on and off. This verification phase, while remaining part of the certification process conducted by the same body, involves specialised hardware figures who provide the context needed to confirm how the identified operations impact the actions of the physical device.

Behaviours Comparison: In this part of the report, the certifier compares the data collected in the high-level functions table and the system calls table with the self-declaration provided by the developer. Several scenarios can occur during this analysis, which guide the certifier in assessing firmware behaviours and managing their discrepancies.

The first scenario occurs when the certifier identifies the behaviour declared by the developer during an assessment. In this case, no cause for concern or further action is required. Secondly, the developer declares behaviour, but the certifier does not identify it because probably the manifest producer's work may have failed. In such cases, the certifier must make further investigations; if these are inconclusive, the certifier asks the developer for an explanation. Another possible situation is the detection of behaviour not declared by the developer. This scenario indicates that the developer has not documented the behaviour(s) identified by the certifier, who needs further explanation to resolve the discrepancy. Finally, there are cases where behaviours are neither declared by the developer nor detected by the certifier. These cases cannot affect the ongoing analysis because they have not affected the

certifier's assessment, the identified discrepancies or any other necessary action.

Parties Confirmation: The report's final part confirms the validity of analysis results for both a certifier and a developer. It requires signatures from both parties to formalize the report. By signing in the report, a certifier assures correct firmware behaviours. Consequently, even the data extracted from the binary analysis are valid.

The developer receives the signed report from the certifier and, if everything is correct, countersigns the report as well. The developer must commit to addressing requests for clarification or modification within the time written in the report. Otherwise, the certification will not be valid. In case of disagreement with the analysis results, the developer must submit objections within the time specified in the report, and the relative documentation must be attached.

IX. Performance Analysis

To evaluate performance of the manifest-producer in a realistic context, we subjected the smart light-firmware to the analysis of two separate tools: manifest-producer and an established reverse engineering framework. This comparison has been conducted under controlled conditions using consistent and repeatable measurement tools. We conducted experiments in an isolated WSL2 environment running *Ubuntu 22.04.5 LTS* on a system with an *Intel Core i7-10750H (6cores, 12 threads)* and *16 GB of DDR4-2666 RAM*. We disabled CPU frequency scaling and non-essential services, minimised background processes, and cleared filesystem caches before benchmarks. The tools include manifest-producer *v0.1.0*, Ghidra *v11.3.2*, Hyperfine *v1.12.0*, and Heaptrack *v1.3.0*.

A. Ghidra

We chose Ghidra as our comparison baseline, excluding commercial solutions such as IDA Pro due to licensing limitations. Ghidra, written in Java, represents a mature open-source static analysis framework with National Security Agency maintenance to ensure technical credibility. Moreover, Ghidra supports a headless mode which enables automated binary processing and custom script execution without any GUI overhead, in order to obtain reproducible benchmarks and objective comparisons.

For methodological validity, we configured Ghidra with a custom post-analysis script to extract comparable data: ELF metadata, function enumeration, syscall identification, and disassembled code. However, before running our extraction script, Ghidra executes a complete internal analysis routine, including intermediate representation (IR) population, cross-reference construction, and semantic model creation.

B. Limitation of the Comparison

We have found several limitations in this comparative evaluation:

1) **Underutilisation of Ghidra's capabilities**: We deliberately extract only a limited subset of the data produced by Ghidra, ignoring advanced semantic representations such as a pseudo-C code to represent the decompiled binary, control flow graphs, and type annotations. This basic approach leaves Ghidra's advanced analytical features unexploited.
2) **Reduced experimental coverage**: We base our evaluation on a single test binary, which limits the generalisability of results across different architectures, binary sizes, and code complexity.
3) **Configuration limitations**: We could not selectively identify a method to disable specific Ghidra analysis steps. The `-noanalysis` option completely disables automatic analysis, making comparison impossible, whereas the default configuration runs all available analysers regardless of their relevance to our use case.

C. Profiling Tools

To compare the performance of Ghidra and manifest-producer scenarios, we collected a series of data on the time and memory consumption of their executions using two open source tools called **Hyperfine** and **Heaptrack**.

Hyperfine[7], is a benchmarking tool which estimates the execution time of a process by running in series multiple instances and averaging their execution times into a single value. We chose this tool to compare the timing of our scenarios because it allows to configure benchmarks through a single command line interface and its releases are regular.

Heaptrack[8] monitors and evaluates the memory allocation of a running process by recording information about temporary allocations, peak heap usage, peak RSS (Resident Set Size), and leaked memory. We chose this tool to evaluate the memory performance of our scenarios because it makes use of coloured charts to emphasize key data and, even in this case, its releases are regular.

D. Benchmarking

The results in Table I show a significant in terms of time: the manifest-producer completes the analysis in 5.22 ± 0.24 seconds on average, while Ghidra requires 468.79 ± 5.30 seconds, approximately 7 minutes and 49 seconds. This gap does not indicate inefficiency: Ghidra performs a deep semantic analysis of the binary: it builds complex data structures, it performs control flow analysis, and it executes constant propagation, function, and symbol recognition, even when launched with simple scripts. This architecture reflects a deliberate design choice, more oriented towards information richness rather than pure speed.

Moreover, both tools demonstrate predictable performance with stable execution times. The manifest-producer shows a relative standard deviation of 4.6%, while Ghidra presents 1.1%, benefiting from the stabilisation effect typical of long-running and computationally intensive processes. The CPU usage analysis reveals a massively parallel computation (406.1% CPU utilisation) for Ghidra, exploiting multiple core. instead, the manifest-producer operates with near-linear single-thread

[7]GitHub repository: https://github.com/sharkdp/hyperfine
[8]GitHub repository: https://github.com/KDE/heaptrack

TABLE I
TIME PERFORMANCE COMPARISON.

Metric	Manifest-Producer	Ghidra
Mean Time	5.22 s	468.79 s
Std. Dev.	±0.24 s	±5.30 s
Median	5.16 s	496.75 s
User Time	1.39 s	1829.56 s
System Time	3.60 s	74.71 s
Min/Max	4.94–5.76 s	459.41–475.77 s

execution (95.6% CPU utilisation). This architectural difference reflects comprehensive semantic analysis adopted by Ghidra versus the more focused extraction approach conducted by the manifest-producer.

E. Memory Profiling

Table II presents the memory metrics obtained with Heaptrack, revealing significant architectural differences between the two tools.

TABLE II
MEMORY PROFILING COMPARISON.

Metric	Manifest-Producer	Ghidra
Temp. Allocations	1,094,273 (42.55%, 169,549/s)	603,604 (42.31%, 1,276/s)
Peak Heap Mem.	43.6 MB after 01.273s	173.0 MB after 5min16s
Peak RSS	57.6 MB	1.3 GB
Tot. Mem. Leaked	336 B	100.6 MB (36.7 kB suppressed)
Tot. Runtime	6.454s	7min52s

Ghidra's heap consumption is 4× higher than manifest-producer (*173.0 MB vs 43.6 MB*), attributable to complex IR structures, internal decompilation caches, in-memory symbols databases, and JVM runtime overhead. The RSS difference, hence the physical memory occupied by the process, is even more pronounced at 22.6× (1.3 GB vs 57.6 MB), reflecting JVM overhead and a shared memory for Java libraries. Memory management reveals contrasting approaches: manifest-producer shows minimal leakage (336B), thus indicating a controlled explicit deallocation. Conversely, Ghidra exhibits 100.6 MB of leaked memory, which is typical of garbage collection environments where objects persist until execution ends.

F. Implications for Operational Use

The results obtained from the benchmarks highlight some relevant operational implications, which differentiate the two libraries depending on the context of use. The difference in execution times makes manifest-producer highly suitable for automated pipelines, where the speed of analysis is a critical factor. The small memory footprint and the limited execution time make the manifest-producer ideal for environments with limited computational resources. Conversely, Ghidra stands out for the depth of semantic analysis and the structural richness of internal representations, which justify the computational overhead and make the tool more suitable for in-depth investigations, manual reverse engineering, and integration in interactive or investigative environments.

X. DISCUSSION

The manifest-producer supports the job of a certifier, providing essential information to evaluate firmware behaviours. However, some limitations and possibilities for improvements might hinder its usefulness and applicability.

The current implementation has a notable limitation: whereas the library automates binary analysis, the manual intervention required by the certifier introduces the risk of human errors and leads to extended validation times. Implementing Machine Learning (ML) algorithms could solve, in part, this limitation [38]. AI models might process the collected data, identify known patterns, and automatically classify behaviours. This approach would provide assistance to a certifier, who might analyse the machine learning results instead of raw data extracted by a binary. The entire process would be much simpler and less error-prone.

As we said in Section VI, the manifest-producer can only analyse x86_64 binaries. This is a remarkable constraint since other architectures are occupying a large share of the IoT market, such as the ARM one. Moreover, the RISC-V architecture, due to its open-source nature, is becoming more and more popular nowadays [39]. Therefore, as future objective, we plan to provide a library version which also consider the binaries built for these architectures. In this way, the manifest-producer will cover the majority of devices present in the IoT world, offering the same functionalities of the x86_64 architecture.

To address the lack of trust among manufacturers and certifiers described in Section IV, a future improvement might be the creation of a platform designed with the aim of replacing the certification process example described in Section VIII. Both manufacturers and certifiers can access the main page of the platform through their personal accounts, but each party can only perform a certain set of actions. A manufacturer is the only one who can upload and delete a firmware from the platform, while a certifier can only select the binary as input for the manifest-producer, without the ability to copy or distribute its contents in any way. The certification process consists of an instance of the manifest producer running on the platform with the binary as input. As a final step, the static web pages will be visible to both parties on the main page of the platform. Therefore, the platform takes the role of a sandbox.

The platform is a container runnable on machines with determined hardware architectures. Manufacturers and certifiers choose a virtual machine service on which the platform will have to run by mutual agreement. This service will be accessible for a limited period of time. To start the certification process, one of the two parties loads the platform container onto the virtual machine, hence starting its execution. The

counterpart can follow the entire configuration phase and, in case of disagreement, ask to stop the whole procedure. When the platform is running on the machine, the certification process can actually start. Using a temporary virtual machine provides a higher level of reliability, as the platform is running on a service external to both parties. In addition, the certification costs include the rental prices of this service.

XI. Conclusion

Most Internet of Things devices present interoperability problems and security threats. Over the years, standards and certifications have introduced some methods to thin out these problematics, with the objective of creating more reliable systems. However, neither standards nor certifications fully consider the effects which might occur during a device firmware execution. Therefore, we have created the behaviours assessment to fill in this gap. In this paper, we have explained the reasons behind this new kind of assessment and its description. We have also conceived and developed a new software library written in the Rust language, called manifest-producer, as a proof of concept for our assessment.

By automating binary inspection through a static reverse engineering approach, the manifest-producer supports certifiers in identifying expected and unexpected behaviour. To demonstrate its practicality, we conducted a proof of concept by simulating the certification of a mock firmware. This use case highlighted the library's ability to analyse binaries, generating sufficient data to help the certifier assess the firmware's behaviour and produce a detailed report.

References

[1] A. Cirne, P. R. Sousa, J. S. Resende, and L. Antunes, "Iot security certifications: Challenges and potential approaches," *Computers & Security*, vol. 116, p. 102669, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404822000682

[2] S. A. Al-Qaseemi, H. A. Almulhim, M. F. Almulhim, and S. R. Chaudhry, "Iot architecture challenges and issues: Lack of standardization," *2016 Future Technologies Conference (FTC)*, pp. 731–738, 2016.

[3] J. Voas and P. A. Laplante, "Iot's certification quagmire," *Computer*, vol. 51, no. 4, pp. 86–89, 2018.

[4] R. S. Ross, S. W. Katzke, and L. A. Johnson, "Minimum security requirements for federal information and information systems," 2006. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.200

[5] G. Purdy, "Iso 31000: 2009—setting a new standard for risk management," *Risk Analysis: An International Journal*, vol. 30, no. 6, pp. 881–886, 2010. [Online]. Available: https://doi.org/10.1111/j.1539-6924.2010.01442.x

[6] K. Scarfone, M. Souppaya, A. Cody, and A. Orebaugh, "Technical guide to information security testing and assessment," pp. 2–25, 09 2008. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-115

[7] C.-K. Chen, Z.-K. Zhang, S.-H. Lee, and S. Shieh, "Penetration testing in the iot age," *Computer*, vol. 51, no. 4, pp. 82–85, 2018.

[8] D. Miessler, "Securing the internet of things: Mapping attack surface areas using the owasp iot top 10," in *RSA Conference*, 2015.

[9] I. Makhdoom, M. Abolhasan, J. Lipman, R. P. Liu, and W. Ni, "Anatomy of threats to the internet of things," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1636–1675, 2019.

[10] I. Nadir, H. Mahmood, and G. Asadullah, "A taxonomy of iot firmware security and principal firmware analysis techniques," *International Journal of Critical Infrastructure Protection*, vol. 38, pp. 100–552, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1874548222000373

[11] A. Hezam, D. Konstantas, and M. Mahyoub, "A comprehensive iot attacks survey based on a building-blocked reference mode," *International Journal of Advanced Computer Science and Applications*, vol. Vol. 9, no. 3, pp. 355–373, 04 2018.

[12] A. Gupta, *The IoT Hacker's Handbook: A Practical Guide to Hacking the Internet of Things*. Walnut, CA, USA: Springer, 2019. [Online]. Available: https://doi.org/10.1007/978-1-4842-4300-8

[13] C.-W. Tien, T.-T. Tsai, I.-Y. Chen, and S.-Y. Kuo, "Ufo - hidden backdoor discovery and security verification in iot device firmware," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2018, pp. 18–23.

[14] N. G. Tsoutsos and M. Maniatakos, "Anatomy of memory corruption attacks and mitigations in embedded systems," *IEEE Embedded Systems Letters*, vol. 10, no. 3, pp. 95–98, 2018.

[15] M. Ammar, G. Russello, and B. Crispo, "Internet of things: A survey on the security of iot frameworks," *Journal of Information Security and Applications*, vol. 38, pp. 8–27, 2018.

[16] M. Lezzi, M. Lazoi, and A. Corallo, "Cybersecurity for industry 4.0 in the current literature: A reference framework," *Computers in Industry*, vol. 103, pp. 97–110, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166361518303658

[17] J. Saleem, M. Hammoudeh, U. Raza, B. Adebisi, and R. Ande, "Iot standardisation - challenges, perspectives and solution," in *Proceedings of the 2nd International Conference*. Association for Computing Machinery, 06 2018, pp. 1–9.

[18] R. Anderson and S. Fuloria, "Certification and evaluation: A security economics perspective," in *2009 IEEE Conference on Emerging Technologies & Factory Automation*, 09 2009, pp. 1–7.

[19] G. Baldini, A. Skarmeta, E. Fourneret, R. Neisse, B. Legeard, and F. L. Gall, "Security certification and labelling in internet of things," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, 2016, pp. 627–632.

[20] D. Belli, P. Barsocchi, and F. Palumbo, "Connectivity standards alliance matter: State of the art and opportunities," *Internet of Things*, vol. 25, p. 101005, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2542660523003281

[21] F. Chen, D. Luo, J. Li, V. C. M. Leung, S. Li, and J. Fan, "Arm psa-certified iot chip security: A case study," *Tsinghua Science and Technology*, vol. 28, no. 2, pp. 244–257, 2023.

[22] V. Tenentes, A. Xynos, C. Zonios, A. Koutra, C. Dilopoulou, K. Tsampiras, Y. Tsiatouhas, and D. Rossi, "Embedded platforms for trusted edge computing towards quality assurance along the supply chain," in *2023 8th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM)*, 2023, pp. 1–6.

[23] S. P. Kaluvuri, M. Bezzi, and Y. Roudier, "A quantitative analysis of common criteria certification practice," in *International Conference on Trust, Privacy and Security in Digital Business*, vol. 8647, 09 2014, pp. 132–143.

[24] G. M. Bianco, L. Ardito, and M. Valsesia, "A tool for iot firmware certification," in *Proceedings of the 19th International Conference on Availability, Reliability and Security (ARES '24)*. New York, NY, USA: Association for Computing Machinery, 2024, pp. 59–65. [Online]. Available: https://doi.org/10.1145/3664476.3670469

[25] W. Bugden and A. Alahmar, "Rust: The programming language for safety and performance," *arXiv*, vol. abs/2206.05503, Jun 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:249626023

[26] N. D. Matsakis and F. S. Klock, "The rust language," *Ada Letters*, vol. 34, no. 3, pp. 103–104, 10 2014. [Online]. Available: https://doi.org/10.1145/2692956.2663188

[27] T. Uzlu and E. Şaykol, "On utilizing rust programming language for internet of things," in *2017 9th International Conference on Computational Intelligence and Communication Networks (CICN)*, 2017, pp. 93–96.

[28] S. Zhu, Z. Zhang, B. Qin, A. Xiong, and L. Song, "Learning and programming challenges of rust: A mixed-methods study," in *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*, 2022, pp. 1269–1281.

[29] E. Chikofsky and J. Cross, "Reverse engineering and design recovery: a taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

[30] M. D. Ernst, "Static and dynamic analysis: synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, 2003, pp. 24–27. [Online]. Available: https://api.semanticscholar.org/CorpusID:1477502

[31] S. Hanov, "Static analysis of binary executables," in *Proc. of the Semantic Scholar API Conference*, 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:15766577

[32] G. Kachare, G. Choudhary, S. K. Shandilya, and V. Sihag, *Sandbox Environment for Real Time Malware Analysis of IoT Devices*. Springer, 01 2022, pp. 169–183.

[33] D. Jang, T. Kim, and D. Kim, "Dynamic analysis tool for iot device," in *2020 International Conference on Information and Communication Technology Convergence (ICTC)*, 2020, pp. 1864–1867.
[34] M. Popa, "Binary code disassembly for reverse engineering," *Journal of Mobile, Embedded and Distributed Systems - JMEDS*, vol. 4, 12 2012.
[35] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 833–851.
[36] Y. Lee and Y. Son, "A study on verification and analysis of symbol tables for development of the c++ compiler," *International Journal of Multimedia and Ubiquitous Engineering*, vol. 7, no. 4, pp. 175–186, 01 2012.
[37] R. Szalay, Z. Porkoláb, and D. Krupp, "Measuring mangled name ambiguity in large c/c++ projects," in *Sixth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications*, Belgrade, Serbia, 09 2017.
[38] R. Saravanan and P. Sujatha, "A state of art techniques on machine learning algorithms: A perspective of supervised learning approaches in data classification," in *2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS)*, 2018, pp. 945–949.
[39] M. Sharma, E. Bhatnagar, K. Puri, A. Mitra, and J. Agarwal, "A survey of risc-v cpu for iot applications," *SSRN Electronic Journal*, 2022.

Alessandro Aldini received the PhD degree in Computer Science at the University of Bologna, Italy, and is currently Associate Professor at the University of Urbino Carlo Bo, Italy. His research interests are focused on the study and application of automated methodologies for the design and analysis of computer and network systems, with an emphasis on foundations of security, trust, and performance verification. He served as Vice-Chair of the IFIP TC11 (Security and Privacy Protection in Information Processing Systems) Working Group 11.14 on Secure Engineering (2021-2024) and of the International Summerschool on Foundations of Security Analysis and Design (FOSAD) for the last 20 years.

Luca Ardito is an Assistant Professor at Dept. of Control and Computer Engineering of Polytechnic of Turin, working in the Software Engineering research group. He received BSc, MSc, and PhD in Computer Engineering from Politecnico di Torino. His research interests are mobile development and testing, gamification in software engineering, IoT, industrial and home automation, static and dynamic analysis, green software and empirical software engineering methodologies.

Giuseppe Marco Bianco received the M.S. degree in Computer Engineering, specializing in Cybersecurity, from the Politecnico di Torino, Italy, in 2024. He is currently an Assistant Researcher at the University of Urbino Carlo Bo, Department of Pure and Applied Sciences, Italy. His research focuses on API security in the IoT ecosystem, particularly on constrained devices and their firmware.

Michele Valsesia received the M.S. degree in Computer Science from University of Milan, Milan, Italy, in 2019. He is currently working as an Assistant Researcher for the Politecnico di Torino, Turin, Italy. His current research interests include IoT, industrial and home automation, and static and dynamic analysis. His other interests include AI, multimedia processing, and Rust language development. He is active in some rust-based open-source organizations, such as Burn and rust-av. He also contributes to some Mozilla projects, in addition to other open-source libraries and tools, always related to his interests.