Lich: Enhancing IoT Supply Chain Security Through Automated Firmware Analysis

Lich: Enhancing IoT Supply Chain Security Through Automated Firmware Analysis

Alessandro Aldini*, Luca Ardito†, Giuseppe Marco Bianco* and Michele Valsesia†
*Department of Pure and Applied Sciences
Università degli studi di Urbino
Urbino, Italy
Email: {alessandro.aldini, giuseppe.bianco1}@uniurb.it
†Department of Control and Computer Engineering
Politecnico di Torino
Torino, Italy
Email: {luca.ardito, michele.valsesia}@polito.it

***Abstract*—The Internet of Things (IoT) is growing at an unprecedented speed, with over 16.6 billion connected devices in 2023 and projections of around 40 billion by 2030. However, this exponential growth is accompanied by serious security issues within an IoT device supply chain. By analysing case studies, the research identifies how individual vulnerabilities in supply chain can compromise the overall security of devices. Since firmware development is an integral part of an IoT supply chain, any weakness at this stage has large-scale security and reliability repercussions. This research focuses on security vulnerabilities arising from weaknesses in IoT firmware development, examining key challenges. The paper also introduces the *energy robustness* concept, which consists of encouraging developers to consider the amount of joules consumed by a firmware in a specific time frame for the purpose of determining the empiric degree of device robusteness to its inherent vulnerabilities and to external attacks. As a practical solution, we present Lich, a firmware analysis tool that executes a sequence of security and energy consumption tools with the objective of discovering vulnerabilities at an early stage of development and before a firmware deployment. The research aims to demonstrate, on a theoretical level, that the implementation of continuous controls can improve the reliability and security of the IoT supply chain.**

***Index Terms*—Energy Robustness, Firmware, IoT, IoT Supply Chain, Security, Vulnerability Assessment**

I. Introduction

"The Internet of Things (IoT) refers to a network of physical devices, vehicles, appliances, and other physical objects that are embedded with sensors, software, and network connectivity, allowing them to collect and share data." [1] This is a relatively concise definition of the Internet of Things (IoT). It captures an essential aspect: the underlying principle of the IoT is precisely the interconnection between heterogeneous devices, each with specific functions and operating in different contexts, but united by the ability to collect and exchange data continuously. It is precisely this characteristic that makes the IoT a pervasive technology in modern society. It is not an exaggeration to say that it represents an innovation capable of profoundly transforming our lives, bringing about significant technological progress compared to twenty years ago.

According to the IoT Analytics report, 16.6 billion connected IoT devices were active at the end of 2023, an increase of 15% compared to 2022. Estimates indicate that this number will continue to grow, reaching 40 billion by 2030. In its latest 2023 IoT device market update, IoT Analytics has slightly revised its forecast downwards due to new market analysis and the latest developments. However, the number of predicted connections remains exceptionally high, reflecting the widespread deployment of these smart devices in society, both in the private sphere, with devices aimed at individuals, and in the corporate context. In particular, IoT Analytics shows that 51% of companies adopting IoT solutions planned to increase their budgets in this area in 2024, with 22% planning an increase of more than 10% compared to 2023 [2].

Despite the undeniable benefits of smart devices in convenience and ease of doing many daily tasks, their deployment inevitably increases the attack surface. The National Institute of Standards and Technology (NIST) describes it as follows: *"The set of points on the boundary of a system, a system element, or an environment where an attacker can try to enter, cause an effect on, or extract data from, that system, system element, or environment"* [3]. From this perspective, each smart device would represent "a point on the boundary of a system", and with some 40 billion connected devices by 2030, this extended network would drastically expand the attack surface, increasing the security implications.

Most IoT device manufacturers compound this growing risk of cyberattacks by making large-scale devices without con-

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.

[1] IBM, What is the Internet of Things (IoT)?, 2023. [Online]. Available: https://www.ibm.com/think/topics/internet-of-things, Accessed on: Mar, 5, 2025

[2] S.Satyajit, State of IoT Summer 2024, 2024. [Online]. Available: https://iot-analytics.com/number-connected-iot-devices/, Accessed on: Mar, 7, 2025

[3] NIST, NIST glossary. [Online]. Available: https://csrc.nist.gov/glossary/term/attack_surface, Accessed on: Mar, 7, 2025

sideration for security standards. For example, the European Telecommunications Standards Institute (ETSI) EN 303 645[4] standard defines minimum security requirements for consumer IoT devices. Its adoption, however, is fragmented due to several reasons. First, the standard's voluntary nature, as indicated by the distinction between obligatory ("shall") and advisory ("should") provisions, leaves ample room for interpretation and allows for discretionary choices by manufacturers. While this flexibility facilitates adaptability to different production contexts, it also creates unevenness in implementing security measures [1]. Additional implementation expenses are another significant obstacle since employing robust security mechanisms entails investment in human, technology, and time resources that are hard for many producers, especially small and medium-sized ones, to sustain [2]. At the same time, competitive pressures in the marketplace push for increasingly rapid development cycles to launch new products before the competition, sometimes at the expense of thorough security audits. Lastly, the same fragmentation of the IoT ecosystem, as represented by many platforms, protocols, and technologies, further aggravates the uniform application of security standards [1], [3]. This heterogeneity of the IoT ecosystem hinders standardization and creates a critical gap that makes these devices vulnerable to a myriad of threats and vulnerabilities [3]–[5].

This paper presents an instrument to detect security threats in an IoT device firmware, also improving the relative supply chain. With our proposed tool, named Lich, we investigate the possibility of identifying vulnerabilities through reports which provide a reliable overview of different firmware aspects. Lich has been conceived for IoT industries which desire to improve the security of their devices supply chain through automated means. Furthermore, energy measurements help a manufacturer estimate the energy efficiency of a device, thus the bills prices, but also its environmental impact. Even the broader software industries can benefit from Lich, as it only requires a binary as input to generate all the necessary reports.

We have structured this paper in the following sections. In Section II, we define and describe IoT supply chain, its strong points, but also its defects. We then focus on a specific part of IoT supply chain: firmware development. Always in this section, we have introduced the *energy robustness* concept, which encourages developers to consider the amount of joules consumed by a firmware in a specific time frame for the purpose of determining the empiric degree of robustness of a device to its inherent vulnerabilities and to external attacks. In Section III, we present all purposes behind the Lich development and the research questions of this paper. In Section IV, we list some vulnerabilities affecting IoT devices. Section V describes Lich in every aspect, from its development to its current limitations. In Section VI, we discuss the strengths and the limits of our work, outlining potential future developments. The last Section VII summarizes what we have achieved in this paper.

[4]ETSI, Consumer IoT Security. [Online]. Available: https://www.etsi.org/technologies/consumer-iot-security, Accessed on: Mar, 10, 2025

II. IoT Supply Chain

The supply chain represents the activities, processes and actors involved in a product or service's design, production, assembly, distribution and maintenance. According to Yang *et al.*, the complexity of IoT supply chains grows even further: devices are often assembled in international factories, with components from different suppliers, and traverse numerous globally distributed entities before reaching final applications [3]. In this context, specific risks must be considered, such as the inadvertent or intentional introduction of counterfeit components [6], malicious circuit alterations during both design and manufacturing called hardware Trojans [7], cloned devices [8]–[10], the possibility of physical tampering and unauthorized access [11]. It is, therefore, clear that every device goes through its specific supply chain where vulnerabilities can be introduced either accidentally, and thus due to design or implementation flaws, or intentionally as part of a targeted attack.

Addressing IoT security requires a holistic approach across the entire device lifecycle [12]. To support this view, the European Union Cybersecurity Agency's (ENISA) IoT Security Guidelines propose the IoT secure supply chain, which aims to define guidelines for an Internet of Things supply chain, focusing on the actual production processes.

A. Emblematic cases of supply chain compromise

We examine three cases to understand the real implications of such vulnerabilities better. While these provided here do not apply exclusively to the IoT environment, they show how supply chain vulnerabilities can compromise a system's security, trust and compliance, highlighting the importance of considering security at every stage of the product supply chain.

1) **The Volkswagen scandal (Dieselgate)**: The Dieselgate scandal is a case of how intentional alteration of software components can undermine the integrity of a product, even critical in IoT devices. From 2009 to 2015, Volkswagen intentionally included software in the engine control unit of approximately 11 million diesel cars to manipulate their performance during emissions testing. This "defeat device" was detected when the vehicle underwent laboratory testing and activated emission controls. However, it deactivated these systems during everyday driving, allowing nitrogen oxide emissions 40 times the permitted amount [13], [14].
2) **BlackTech attack on routers** The BlackTech group's actions illustrate how adversaries can exploit technology vulnerabilities in the supply chain to infect key systems. BlackTech actors have demonstrated an ability to introduce firmware modifications to routers undetected and use those modifications to exploit trust relationships between global headquarters, Japanese and United States-based subsidiaries [15]. This incident underscores the prime importance of firmware integrity along the whole

product life cycle and the application of Zero Trust models to constrain the spread of compromises in the supply chain [16].

3) **The SolarWinds Orion compromise**: The SolarWinds attack is a prominent example of a software supply chain compromise conducted by the threat actor APT29 (also known as Cozy Bear). Discovered in December 2020, the attackers managed to inject malicious code into the build process of the SolarWinds Orion software platform. This malware, later distributed through a routine software update, allowed attackers to gain unauthorized access to government, consulting, technology, telecommunications, and other organizations across North America, Europe, Asia, and the Middle East. In addition to tampering with software builds, the campaign included techniques such as password spraying, token theft, spear phishing, and API abuse [5]. The incident, highlighted how compromising a single point in the software development pipeline can propagate across thousands of downstream targets [17].

B. Summary and insights from real-world cases

The three cases discussed above demonstrate how supply chain vulnerabilities can manifest themselves, from software manipulation to firmware tampering. Despite their diversity, they share a common lesson: even a single point of failure in the supply chain can have large-scale consequences for security, trust, and compliance.

Table I provides a comparative overview to illustrate the diversity and implications of the three attacks.

TABLE I
COMPARISON OF EMBLEMATIC SUPPLY CHAIN COMPROMISE CASES

Case	Compromise Vector	Impact Scope	Key Takeaway
Volkswagen Dieselgate	Software manipulation in the engine control unit	Regulatory violations, environmental impact	Software integrity violations can be intentional and systemic
BlackTech firmware attack	Unauthorized firmware modification in routers	Targeted cyber-espionage between branches and HQs	Firmware must be treated as a critical attack surface requiring lifecycle protection
SolarWinds Orion compromise	Malicious code injection during software build	Broad-spectrum compromise across governments and enterprises	Secure development pipelines are essential to prevent large-scale propagation

C. Firmware security in the IoT supply chain

These emblematic cases highlight how vulnerabilities in the supply chain, introduced deliberately or due to insufficient control mechanisms, can have devastating consequences. As highlighted by several studies, a critical aspect of IoT security lies in the responsibilities of suppliers and vendors. These actors are integral parts of the IoT supply chain, as they contribute to the design, production, and distribution of devices and their components. Kosuke *et al.* note that although researchers have achieved significant development in IoT security, vendors rely on third-party mitigation services instead of taking direct responsibility for security [18]. This trend not only outsources product security tasks to external entities but also reduces the overall quality of devices. A product with structural vulnerabilities or defective components is more susceptible to attacks and may fail under real operating conditions [19], compromising reliability. In addition, latent vulnerabilities in third-party components can compromise the functioning of the firmware and make it more susceptible to attacks, as highlighted by Zhao *et al.* [20].

Given these challenges, this paper focuses on the IoT supply chain's software development and implementation phase, particularly on firmware. This choice is motivated by poor transparency, limited control over third-party components, and a lack of effective mechanisms for verifying code integrity that characterize the development phase [3], [20], [21]. Firmware plays a crucial role in device security as it is the direct point of contact between hardware and software, and its compromise can result in privileged access to devices, data manipulation, or disabling security features [20]–[23].

It is important to note that while recent studies have mainly focused on the security of networks, authentication and authorization protocols, and user interfaces of IoT devices, there has been relatively less focus on IoT firmware vulnerabilities [23]. These vulnerabilities represent a particularly subtle threat vector. Consequently, improving security measures in software development and firmware validation is one of the most urgent and strategic challenges to safeguarding the IoT ecosystem [22], [23].

However, securing firmware is a complex task that requires traditional validation techniques and innovative approaches to detecting vulnerabilities.

D. Energy consumption role in firmware security

As IoT setups become increasingly complex, recent research has explored new methods to evaluate firmware security. A promising approach consists in detecting security threats from the energy measurements conducted on a running firmware in a specific time frame. Therefore, either a sequence of sudden spikes or a prolonged energetic variability during use may reflect unauthorized accesses, malicious activities, or failures caused by faulty firmware [24]–[26].

As emerges from a survey by Merlino *et al.*, the analysis of energy consumption values leads to defining a nonintrusive high-level abstraction to detect security anomalies without subjecting any software and internal device structure to minute inspection [26]. Nimmy *et al.*, for example, propose an energy consumption analysis-based method to discriminate among the processes with a normal behaviour and the suspicious ones started during a DDoS attack [24]. The experiment has

[5]MITRE ATT&CK, SolarWinds Compromise, 2023. [Online]. Available: https://attack.mitre.org/campaigns/C0024/, Accessed on: Mar, 11, 2025

been conducted on an IP camera connected to a Raspberry PI running within a smart home environment. Similarly, Frawley illustrates how alterations in energy consumption measurements might emphasize the anomalous behaviours caused by vulnerabilities present in a software, with the evident distinction between idle and active functionality [25].

Based on the considerations above, we have outlined the *energy robustness* concept to assess the security of an IoT device. Energy robustness encourages developers to consider the amount of joules consumed by a firmware in a specific time frame for the purpose of determining the empiric degree of robustness of a device to its inherent vulnerabilities and to external attacks. Therefore, the general rule is: the higher the joules, the lower its robustness. We want to specify that the energy robustness concept does not represent a precise and deterministic metric, but more of a hint to help developers in identifying anomalies which might be potential security threats.

For example, if a device exhibits significant energy fluctuations under typical high workloads, this behaviour might indicate an inherent instability in handling resource-intensive tasks. A device in this state of inconstancy is more susceptible to failure under a DDoS attack, where the increased workload could push the system beyond its operational limits, leading to crashes or severe performance degradation.

Although current researches privilege the detection of energy anomalies while a device is performing its functions in an environment, with our approach we are instead suggesting developers to carry out all energy consumption checks during the firmware development phase, probably reducing a good amount of vulnerabilities before its effective deployment.

III. Lich Purposes

In Section II, we have seen how vulnerabilities can sneak into IoT firmware devices either accidentally, because of design or implementation issues, or consciously, due to targeted attacks. Using mechanisms to control firmware development or deployment continuously can lead to an early detection of security issues, thus reducing the probability of compromise.

Analysing the IoT supply chain information reported in the previous sections, we have identified some limitations for the tools which perform security and energy consumption checks on firmware:

- **Lack of integration**: security and energy profiling tools often operate independently, making a comprehensive cross-analysis difficult.
- **Complex and costly setup**: developers must manually install and configure each tool, increasing setup time at risk of incompatibility.
- **Limited automation**: many tools are not designed to integrate directly and without modification into Continuous Integration systems. Consequently, it is not easy to implement automated and repeatable tests.

In this paper, we are trying to provide a possible solution to the limitations above, asking the following research questions:

1) Is it possible to develop a tool which improves the security state of an IoT supply chain through reports produced by open source tools outputs?
2) Can these reports provide an accurate description of different firmware aspects?

The research questions require analysing the limitations in more detail. Therefore, the following subsections explore three key aspects which revolve around these limitations: the multi-tool approach, containerization, and continuous integration. Each of these component plays a crucial role in overcoming these challenges, contributing to a more integrated, automated, and efficient firmware analysis process.

A. Why is a multi-tool approach necessary?

Firmware testing involves several domains, from security vulnerabilities to performance measurements, and no single tool can address all of them. A multi-tool approach has several advantages:

1) **Broader coverage**: Integrating several tools into one minimizes blind spots, improving the accuracy of security and performance tests.
2) **Customization**: Programmers can customize Lich by adding the necessary tools to meet their specific requirements.
3) **Extension**: Lich allows programmers to add more tools in the future, evolving to meet changing security demands.

B. Why is containerization necessary?

The need to download and install each tool separately used in Lich entails extra time and technical know-how, resulting in an installation process that is error-prone and cumbersome.

Lich addresses this issue by using Docker [27], a software that uses virtualization, at the operating system level, to run applications within isolated environments called containers. Docker coordinates the installation and execution of various analytic tools, without any manual configuration.

C. Why is Continuous Integration necessary?

Continuous Integration (CI) [28], [29] is a development practice where developers frequently commit small, incremental changes to the project codebase, allowing multiple developers to collaborate efficiently and keep the code current. CI tools automatically run a test-bed or checks to validate properties such as security vulnerabilities, performance regressions, unexpected power consumption spikes, and failures. By including these checks early in the firmware development cycle, problems can be detected and fixed before release, reducing risks and increasing overall reliability.

IV. Vulnerabilities

An IoT firmware is a fusion of multiple technologies. Each technology may present a series of threats which can affect the security of an IoT device [21]. Memory vulnerabilities such as buffer overflows, pointer violations, integer overflows, uncontrolled format strings, missing bound checks, and type

confusion are among the most common threats affecting IoT devices [30]. The presence of these memory corruptions can lead to system crashes, privilege escalation, and code injection exploits.

At the firmware-level, attackers mainly focus on exploiting vulnerabilities which developers often overlook during firmware development [31]. Therefore, a firmware exploitation usually results in a complete hijacking of the entire device.

Nowadays, most IoT devices connect to the Internet, thus the possibility for an attacker to exploit the vulnerabilities present in their firmware is quite high [21]. Through web interfaces, a device can communicate and share data with numerous devices, but even the APIs associated with these interfaces suffer from a considerable amount of web insecurities [32].

Furthermore, older IoT devices do not present any update mechanism, hence a single firmware vulnerability can cause serious damages over a wide spectrum of devices [33]. Indeed, an old and unmaintained codebase, with the relative dependencies, might lead to have an insecure firmware [34].

A. Detecting Vulnerabilities

Vulnerability detection consists of a series of methods to protect IoT devices from zero-day attacks [35]. Through static and dynamic firmware analysis, these methods might detect a considerable amount of threats which affect IoT devices.

Static binary analysis involves examining a firmware binary in search of function calls, control flows, and system-level interactions [36]. For example, it can discover all binary execution paths and reveal all those hidden logics, which may result in vulnerabilities, such as remnants of old functionalities or debugging codes. This analysis can detect unreachable paths, i.e. dead code, and some information exploitable by an attacker to take control of a firmware.

Dynamic binary analysis performs a series of checks while a binary is running [37]. The most popular tools in this category are error checkers and profilers. The former detect a list of errors which may become threats for an executing firmware, while the latter profile the entire binary in search of those parts which need a specific optimization in terms of time and memory consumption. As a practical case, this analysis gives the opportunity to detect dangerous uses of undefined values by tracking undefined bit values, thus identifying all those values derived from undefined values or not initialized in any way.

Even energy consumption plays a significant role in detecting vulnerabilities. Indeed, there exists a negative correlation between the number of defects fixed in a software and its energy consumption [38]. Greater the number of fixed defects and less the software energy consumption. Since a vulnerability is also a software defect, the relation among these two parameters has been established.

Furthermore, applying software security mechanisms to avoid vulnerabilities leads to the execution of more instructions, and consequently, adding more energy penalties [39]. Conversely, in order to improve performance and energy efficiency, a developer can indirectly introduce new defects which affect software security. Therefore, the incremental addition of security mechanisms may lead to a considerable increase in energy consumption, whereas energy-related improvements may introduce vulnerabilities.

As pointed out in Section III, Lich executes various analysis tools in an automated manner and provides their results in structured reports which give a general overview of the security status of a firmware. At the time of writing, Lich contains only a single tool for dynamic binary analysis and some energy consumption tools. Every tool outputs information on different firmware aspects. It will be Lich the one which will define reports starting from this data. A developer can then analyse these reports: some of them clearly notify the presence of vulnerabilities in a firmware, while others might discover them from the anomalies in their data, such as for the tools related to energy consumption. Whatever the case, Lich provides a set of information which help in identifying firmware vulnerabilities.

V. Lich Analyser

Lich is an IoT firmware analyser. It executes a series of analytic tools in sequence on firmware, and inserts each of their results into different files called reports. As the last operation, Lich creates an index file containing the list of file paths to access reports, divided by sections according to the analytic tool categories. We refer to tools with the term analytic to focus on their ability to perform analysis. We have defined the categories concept to establish the execution order of the analytic tools. For example, all analytic tools assigned to the vulnerability category execute before the ones contained in the energy consumption category. This choice has been made to prioritize the vulnerability detection phase over the energy phase and to avoid allocating further resources in advance. Indeed, if an error occurs while executing a tool, Lich terminates immediately, and the following categories do not even instantiate their own tools. Tools belonging to the same category does not have a precise execution order though.

Lich has been written in Rust language. Rust is the first programming language to overcome the longstanding trade-off between the control over resource management provided by lower-level languages, and the safety guarantees of higher-level languages [40]. It is a systems programming language which prevalently focuses on memory safety, without requiring the use of a garbage collector, and performance through the C++ approach of zero cost abstraction [41]. It offers an ecosystem aimed at simplifying software development. Among the main Rust tools, there is *Cargo*, which is both a build system and a package manager. Cargo offers features such as dependency management (downloading and compiling projects), build management (debug, release, and custom profiles), testing (unit testing and integration testing), benchmarking of performance, documentation generation, and plugin installation. As a memory safety language, Rust can eliminate up to approximately 70% of security bugs [42], and, in addition, it is also among the five languages which builds binaries with the lowest energy and time consumption during execution [43].

At the time of writing, Lich executes four tools. All of them are open source, hence anyone may change their source code and build a specific variant if deemed necessary. Lich searches for these tools binaries within default system paths, and it arises an error when not found.

Valgrind [44] is the only analytic tool in the vulnerability category. Valgrind is an instrumentation framework for building new dynamic binary analysis tools. It can run on different hardware architectures, and it makes use of a technique called *shadow values* to shadow every register and memory value with another value which describes the register itself. Valgrind can detect dangerous uses of undefined values, memory management errors, such as memory leaks, dangling pointers, and threading bugs. Binaries written in C, C++, and Rust languages make great use of Valgrind to search for all possible threats hidden in their structure.

Three analytic tools constitute the energy category. The first is Perf, a tool which provides subcommands to conduct performance or energy consumption analysis on Linux systems [45]. In our case, we have employed the *stat* subcommand to estimate the amount of energy consumed by both CPU and dynamic memory while the firmware is in execution. Perf uses Joules as a measurement unit for its results. Internally, this subcommand invokes some Linux kernel syscalls which interact with the underlying hardware to retrieve the necessary information. Perf requires some root privileges to perform its operations, producing more accurate results instead of rough estimates. For its tight connections with the Linux kernel, Perf only supports Linux systems.

The second energy tool is PowerStat. It makes use of Intel sensors, Running Average Power Limit (RAPL), to measure firmware energy consumption [46]. We have chosen to add PowerStat for firmware specifically implemented for Intel processors. On this kind of platforms, it can produce more accurate data with a very low overhead. As a clear limitation though, PowerStat cannot retrieve any information from microcontrollers which are outside the Intel brand.

PowerTOP is the last tool contained in the energy category. It is an Intel tool designed to measure, explain and minimize energy consumption [47]. Differently from PowerStat, it works for Intel, AMD, ARM and UltraSPARC processors [48]. PowerTOP is particularly useful for firmware which need information to prolong their battery life and reduce their electrical costs.

As mentioned in Section III, developers can also obtain Lich reports without installing any tool on their machines. This is possible through the operating system-level virtualization, which delivers Lich, along with all of its dependencies, into a single package called *image*. When an image is in execution on a host system, it takes the name of *container*. We have created two images using Docker. The first image packages the Alpine Linux operating system, renowned for its small size and rapid startup, while the second one packages Debian, a more known and adopted system. Each container executes an independent instance of Lich. A developer only has to specify to Docker the path to the directory containing both the configuration file and the input binary before issuing the command to execute the container.

To test the Docker images on a reproducible and always accessible environment, we have defined a workflow for the Continuous Integration system provided by GitHub. The workflow invokes a total of four Linux runners.

- x86_64 architecture and Alpine operating system
- x86_64 architecture and Debian operating system
- ARM architecture and Alpine operating system
- ARM architecture and Debian operating system

Each of these runners executes a different Docker container passing a simple light firmware as input. As a last operation, a runner prints all produced reports on screen to verify their content. Through this approach, we have also demonstrated the feasibility of executing Lich in an automated way.

A. Configuration File

Defining input arguments for the firmware in analysis and for every analytic tool through command line would have been very confusing and cumbersome for developers. Therefore, we have decided to conceive a configuration file to provide a more structured description for the necessary inputs. We have chosen *toml* as file format because it allows to group each tool data into separate sections. Within a section, a developer can associate to an input argument of a tool, or to any other parameter, a value or a set of values expressed as a key-value pair. This way of representing information allows a more direct data comprehension and the possibility to keep track of each tool configuration.

The configuration file defines global parameters which are outside sections. Some of them are mandatory, such as the path to the firmware in analysis and the directory to save reports. Among the option parameters instead, there is the boolean parameter *docker*, whose main goal consists of enabling or disabling specific configurations for the Docker environment. When the *docker* value is true, Lich adds at the beginning of the firmware and report directory paths, both contained in the configuration file, the path to the directory containing both the configuration file and the input binary on the Docker image. The last global optional parameter is *root*, which allows defining the name of the program to issue commands in privileged mode on Linux operating systems.

There are only two parameters common to all sections, while the others vary depending on the analytic tools. The first one is the boolean parameter *enable*. It has been introduced to enable or disable the execution of analytic tools. We have created this parameter to offer some degree of flexibility for those tools which are unmaintained or incompatible with some platforms due to missing privileges or missing specific hardware instructions. The second parameter is *args*. It defines the list of input arguments for firmware and analytics tools.

Specific parameters can be of various data types: paths to output directories, integer and floating points thresholds, and even strings. They have not been defined along the other *args* parameters because they require a specific parsing mechanism. For example, the *timeout* parameter has been conceived to

stop firmware which have an infinite execution time, such as servers. If its value is 0, Lich ignores the *timeout* parameter.

The sections of a configuration file are optional. When they are not present, default parameters and input arguments replace their values. This is the reason behind our choice of adding firmware input arguments into a section called *binary*, since not all firmware have input arguments.

B. Report Structure

A report presents the output of an analytic tool in a structured way. At the time of writing, Lich only supports reports written in *markdown* file format. Each report has a single section whose header contains the exit status of an analytic tool expressed as an emoticon. When an exit status is an error, there is a negative emoticon, while there is a positive one in case of success.

The section content shows the output of an analytic tool. Lich does not elaborate this information in any way, it only pipes the relative data into the report. Both stdout and stderr flows converge their output into the same section. If an analytic tool can only output data on a file, Lich reads the out-coming file and prints its content into the report.

C. Limitations

Lich must have the same hardware architecture as the IoT firmware in analysis. Since most of the tools perform dynamic binary analysis, it is therefore necessary to execute the firmware. Using QEMU as machine emulator might be of help, but it might be, in some cases, about 4 times slower than native code on integers, and 10 times slower on floating points [49]. Furthermore, some of the tools present in Lich might not be available on all kind of platforms and operating systems, hence this narrows the field to only those systems with a well-known machine instructions set, such as x86_64 and ARM.

VI. Discussion

The mere existence of Lich fully replies to the first research question. Through reports, derived from the output of some open source tools, it provides a description of firmware vulnerabilities, offering valuable information to improve the supply chain of the associated IoT device. Since each report contains data belonging to different categories, from security to energy aspects, we can offer to developers an overview on the firmware security level from different perspectives, hence answering the second research question.

As of now, energy robustness is only a hint which encourages developers to consider the amount of joules consumed by a firmware in order to determine the empiric degree of device robusteness to its inherent vulnerabilities and to external attacks. However, a hint is not a deterministic metric, therefore a developer cannot objectively compare the robusteness of two different firmware. As future development, a detailed analysis might yield some parameters to define a real metric.

Lich does not allow defining different firmware instances in the configuration file. A developer must change firmware arguments within the configuration file every time and execute Lich, or Docker, one more time. The possibility to write custom firmware configurations would further decrease a developer effort because it would allow executing the instances one after the another in an automated way.

The index file may also incorporate a description for the Lich's exit status, in addition to a series of recommendations to improve firmware security and reduce energy consumption. At the time of writing, Lich treats reports as endpoints of tools, without processing their content in any way. The introduction of artificial intelligence models, especially those involved in recognizing and interpreting human language, might be a valid support for the processing and analysis of reports, in order to summarize more meaningful and high-level conclusions, or visualizations, for the developer.

VII. Conclusion

The fast growth of the Internet of Things (IoT), with billions of devices globally networked, has exposed material security challenges confronting the IoT supply chain. Through the analysis of real-world case studies, this paper brings to light how vulnerabilities in any supply chain phase get propagated downstream, thereby undermining the security and reliability of devices. Focusing on firmware design as part of the IoT supply chain, we discussed specific security threats and introduced the energy robustness concept. Furthermore, in an attempt to overcome some security threats affecting the IoT supply chain, we have implemented Lich, an instrument collecting firmware vulnerabilities and energy consumption measurements in an automated way. We have also created two Docker images and a continuous integration script to configure Lich with the intent of reducing the initial development effort. Through the information retrieved from Lich execution, developers can decide what are the best strategies to improve their firmware security.

References

[1] F. Körner, P. Schäfer, S. Ahmadi, B. Schnor, and H. Zwingmann, "Current challenges of implementing etsi en 303 645 as a baseline security standard for consumer iot security certification," Dec. 2023. [Online]. Available: http://dx.doi.org/10.36227/techrxiv.24711672.v1

[2] P. Stanchev, Y. Tomov, and N. Hinov, "Problems and solution in ensuring cybersecurity of iot devices for the needs of small and medium enterprises," in *2024 12th International Scientific Conference on Computer Science (COMSCI)*, 2024, pp. 1–4.

[3] K. Yang, D. Forte, and M. M. Tehranipoor, "Protecting endpoint devices in iot supply chain," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 351–356.

[4] 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*, ser. ARES '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3664476.3670469

[5] G. Baldini, A. Skarmeta, E. Fourneret, R. Neisse, B. Legeard, and F. Le 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.

[6] U. Guin, D. Forte, and M. Tehranipoor, "Anti-counterfeit techniques: From design to resign," in *2013 14th International Workshop on Microprocessor Test and Verification*, 2013, pp. 89–94.

[7] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design & Test of Computers*, vol. 27, no. 1, pp. 10–25, 2010.

[8] D. S. Patil and S. C. Patil, "A novel algorithm for detecting node clone attack in wireless sensor networks," in *2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA)*, 2017, pp. 1–4.

[9] S. Chen, Z. Pang, H. Wen, K. Yu, T. Zhang, and Y. Lu, "Automated labeling and learning for physical layer authentication against clone node and sybil attacks in industrial wireless edge networks," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 3, pp. 2041–2051, 2021.

[10] Z. AlJabri, J. H. Abawajy, and S. Huda, "Mds-based cloned device detection in iot-fog network," *IEEE Internet of Things Journal*, vol. 11, no. 12, pp. 22 128–22 139, 2024.

[11] R. Sandhu and P. Samarati, "Access control: principle and practice," *IEEE Communications Magazine*, vol. 32, no. 9, pp. 40–48, 1994.

[12] M. J. Farooq and Q. Zhu, "Iot supply chain security: Overview, challenges, and the road ahead," *ArXiv*, vol. abs/1908.07828, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:201124793

[13] M. Zhang, G. Atwal, and M. Kaiser, "Corporate social irresponsibility and stakeholder ecosystems: The case of volkswagen dieselgate scandal," *Strategic Change*, vol. 30, no. 1, pp. 79–85, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/jsc.2391

[14] M. Nunes and C. Lee Park, "Caught red-handed: the cost of the volkswagen dieselgate," *Journal of Global Responsibility*, vol. 7, pp. 288–302, 07 2016.

[15] D. Tayouri, T. Nachum, and A. Shabtai, "Mirage: Multi-binary image risk assessment with attack graph employment," *arXiv preprint arXiv:2311.03565*, 2023.

[16] D. Horne and S. Nair, "Introducing zero trust by design: Principles and practice beyond the zero trust hype," *Advances in security, networks, and internet of things*, pp. 512–525, 2021.

[17] R. Alkhadra, J. Abuzaid, M. AlShammari, and N. Mohammad, "Solar winds hack: In-depth analysis and countermeasures," in *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, 2021, pp. 1–7.

[18] K. Ito, S. Morisaki, and A. Goto, "Iot security-quality-metrics method and its conformity with emerging guidelines," *IoT*, vol. 2, no. 4, pp. 761–785, 2021. [Online]. Available: https://www.mdpi.com/2624-831X/2/4/38

[19] D. Anandayuvaraj and J. C. Davis, "Reflecting on recurring failures in iot development," in *Proceedings of the 37th IEEE/ACM International conference on automated software engineering*, 2022, pp. 1–5.

[20] B. Zhao, S. Ji, J. Xu, Y. Tian, Q. Wei, Q. Wang, C. Lyu, X. Zhang, C. Lin, J. Wu, and R. Beyah, "One bad apple spoils the barrel: Understanding the security risks introduced by third-party components in iot firmware," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 3, pp. 1372–1389, 2024.

[21] 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

[22] G. Hernandez, O. Arias, D. Buentello, and Y. Jin, "Smart nest thermostat: A smart spy in your home," *Black Hat USA*, vol. 2015, 2014.

[23] A. Mohanty, I. Obaidat, F. Yilmaz, and M. Sridhar, "Control-hijacking vulnerabilities in iot firmware: A brief survey," in *Proceedings of the 1st International Workshop on Security and Privacy for the Internet-of-Things (IoTSec)*, 2018.

[24] K. Nimmy, M. Dilraj, S. Sankaran, and K. Achuthan, "Leveraging power consumption for anomaly detection on iot devices in smart homes," *Journal of Ambient Intelligence and Humanized Computing*, vol. 14, pp. 1–12, 06 2022.

[25] R. Frawley, "Logging and analysis of internet of things (iot) device network traffic and power consumption," 06 2018.

[26] V. Merlino and D. Allegra, "Energy-based approach for attack detection in iot devices: A survey," *Internet of Things*, vol. 27, p. 101306, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2542660524002476

[27] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[28] O. Elazhary, C. Werner, Z. S. Li, D. Lowlind, N. A. Ernst, and M.-A. Storey, "Uncovering the benefits and challenges of continuous integration practices," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2570–2583, 2022.

[29] M. Wróbel, J. Szymukowicz, and P. Weichbroth, "Using continuous integration techniques in open source projects -an exploratory study," *IEEE Access*, vol. 11, 10 2023.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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

[35] W. Xie, Y. Jiang, Y. Tang, N. Ding, and Y. Gao, "Vulnerability detection in iot firmware: A survey," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, 2017, pp. 769–772.

[36] 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

[37] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 89–100. [Online]. Available: https://doi.org/10.1145/1250734.1250746

[38] F. Ahmed, H. Mahmood, and A. Aslam, "Green computing and software defects in open source software: An empirical study," in *2014 International Conference on Open Source Systems & Technologies*, 2014, pp. 65–69.

[39] M. Siavvas, C. Marantos, L. Papadopoulos, D. Kehagias, and D. Tzovaras, "On the relationship between software security and energy consumption," 05 2019.

[40] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Safe systems programming in rust," *Commun. ACM*, vol. 64, no. 4, p. 144–152, Mar. 2021. [Online]. Available: https://doi.org/10.1145/3418295

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

[42] W. Bugden and A. Alahmar, "The safety and performance of prominent programming languages," *International Journal of Software Engineering and Knowledge Engineering*, vol. 32, no. 05, pp. 713–744, 2022. [Online]. Available: https://doi.org/10.1142/S0218194022500231

[43] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Ranking programming languages by energy efficiency," *Science of Computer Programming*, vol. 205, p. 102609, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642321000022

[44] T. Omitola and G. Wills, "Towards mapping the security challenges of the internet of things (iot) supply chain," *Procedia Computer Science*, vol. 126, pp. 441–450, 2018, knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 22nd International Conference, KES-2018, Belgrade, Serbia. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050918312547

[45] A. C. de Melo and R. Hat, "The new linux ' perf ' tools," 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:10296207

[46] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using rapl," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 3, p. 13–17, Jan. 2012. [Online]. Available: https://doi.org/10.1145/2425248.2425252

[47] J. Gray. (2008) Interview with arjan van de ven of intel and lesswatts.org. [Online]. Available: https://www.linuxjournal.com/content/interview-arjan-van-de-ven-intel-and-lesswattsorg

[48] Wikipedia contributors, "Powertop — Wikipedia, the free encyclopedia," 2007, [Online; accessed 01-April-2025]. [Online]. Available: https://en.wikipedia.org/wiki/PowerTOP

[49] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005, p. 41.