UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

PhD Thesis Summary

The Security of Heterogeneous IoT Infrastructures

Ing. Ioana-Maria Culic

Thesis advisor:

Prof. Dr. Ing. Răzvan-Victor Rughiniș

BUCHAREST
2023

CONTENTS

ABSTRACT

As the Internet of Things (IoT) technologies have become widely adopted in multiple areas, the number of security attacks targeting IoT devices has also increased. With famous attacks such as the Mirai Botnet, security has become an essential aspect in the development of all IoT products. Internet of Things infrastructures are complex, heterogeneous systems following an extensive hardware and software stack. As a result, security mechanisms need to be implemented at all stack layers to ensure a final product with a high safety degree.

This thesis aims to analyze the security mechanisms and policies that can be implemented at various stack layers and propose solutions for a better security. Our contributions approach software security in IoT devices as we leverage hardware safety mechanisms already implemented.

We first address the security of Internet of Things applications, focusing on the programming languages developers can use to build sandboxed applications. We identify JerryScript, a lightweight JavaScript engine, as the most suitable approach and focus on enabling JerryScript to run on a large category of IoT devices.

Further on, we focus on the tools necessary in a security-focused IoT software development process. We propose an implementation of a complex development platform that abstracts hardware setup overhead and enables developers to focus on building secure applications. We also addressed the challenges IoT education has in relation to the available development tools. We propose some contributions meant to enable educators to integrate IoT technologies into their curriculum. As a result, we have built a customizable platform used by over 30.000 hobbyists and educators.

At the cloud layer, we approached the security concerns related to software deployment and updates as we identified a lack of open infrastructures for these operations. In this context, we proposed a mathematical model for a secure software deployment and updates infrastructure, which we validated with an open platform implementation.

Finally, we move to a layer closer to hardware and address the security in Real-Time Operating Systems (RTOSes). We leverage the security advantages of Rust, a relatively new programming language developed for secure programming. Due to its characteristics, Rust is a good alternative to C and can be used for writing operating systems. In this context, we focus on Tock, an open-source operating system for microcontrollers fully written in Rust that has multiple security policies implemented. Our contribution concentrates on implementing a real-time module allowing Tock to run as a secure RTOS on constrained devices.

With the research directions presented in this thesis, we addressed the primary software layers in IoT infrastructures and implemented multiple security mechanisms and contributions for a safer IoT development process.

Keywords: Internet of Things, Security, Over-the-Air Update, Hardware Simulator, Visual Programming, Real-Time Operating Systems, Rust

1 INTRODUCTION

Being a technology increasingly present in people's lives, Internet of Things (IoT) raises important concerns related to the safety of the deployed infrastructures, as the occurrence of a security attack can have a large-scale impact. For instance, the famous Mirai Botnet attack infected over 200.000 IoT devices [1, 2]. In this context, Internet of Things devices and infrastructure security is essential for commercial and industrial integrators.

When approaching the security of IoT infrastructures, a significant challenge is the complexity of these systems. Internet of Things architectures are heterogeneous infrastructures from the hardware, software, and communication point of view. They rely on a complex stack that starts with simple microcontroller devices running relatively few lines of code and ends with the cloud that aggregates and processes all the data and manages the connected devices. What is more, at the lower layers where microcontrollers and embedded computers run applications, most of the software is built in C, which has a high risk of security issues. The industry states that around 70% of vulnerabilities arise from problems related to memory safety [3, 4]. This is mainly because of how C implements memory operations that can easily lead to buffer overflows or control-flow attacks [5]. While modern, more secure programming languages (e.g., Python, Java) are increasingly used for embedded computer applications [6,7], the operating systems currently used for computers and microcontrollers are fully written in C, which maintains the security risks at the kernel layer. In addition, the complexity and dimensions of the operating systems used (e.g., Linux has over 27 million lines of code [8]) expose a large attack surface.

Periodic updates are very important, considering the importance of maintaining security in the IoT infrastructures deployed in commercial and industrial environments. They enable the producers to keep their products up to date with the latest bug fixes and security improvements.

All the challenges presented above are being addressed by the development of new programming languages and technologies.

The Rust [9] programming language is a significant breakthrough developed to address the security weaknesses of C. This programming language for systems is guaranteed to be secure through mathematical proof and has safeguards in place to prevent errors related to pointer manipulation, which mitigates numerous security concerns commonly encountered in C programs. Leveraging Rust's advantages, operating systems entirely written in this language are being developed (e.g., Tock [10], Hubris [11], Redox [12]). Out of these, Tock, for instance, is an open-source OS targeting low-capabilities microcontrollers, being a suitable alternative to operating systems such as FreeRTOS [13] or Zephyr [14].

What is more, updates for IoT devices is also a heavily researched domain, and several solutions, many of which with a commercial purpose, have been developed. These open a new research area where remote software deployment infrastructures can be optimized and secured.

Therefore, we define the following research questions that we aim to respond to in this thesis:

- How can the security of Internet of Things systems be enforced during the product life cycle, starting from the prototyping phase to the maintenance phase?
- Are OTA deployments systems helpful in maintaining the security of IoT infrastructures, or do they bring additional risks?
- To which extent can we secure the constrained devices integrated into IoT infrastructures?
- Can C be replaced with other modern high-level programming languages that can guarantee an increased security in microcontrollers and embedded devices?

In the context of various IoT infrastructures being deployed for diverse use cases, we aim to research into ways of securing these systems in a generic manner, irrespective of the heterogeneity that characterizes them. In our work, we address several layers in the IoT stack, and we aim to propose secure platforms for both kernel and user space and also for the maintenance of these systems. The final objective is to achieve a full-stack infrastructure that addresses the current major security issues in IoT infrastructures.

Therefore, in the development of this thesis, we define the following intermediary objectives related to the research questions and the security context we presented in the section above:

- Analyze existing development and deployment tools for the IoT systems for various use cases and in different usage conditions.
- Explore the security mechanisms enforced by OTA IoT deployment systems and analyze their efficiency.
- Define the constraints microcontrollers and embedded computers have and analyze how they impact the security of these devices.
- Inspect the security of existing operating systems for microcontrollers and embedded computers.
- Identify alternatives to the classical, C-based operating systems for IoT devices.
- Define the advantages high-level programming languages have over C from a security point of view.
- Identify more secure alternatives to the C programming language for Internet of Things applications.

In this thesis, we focus on securing the Internet of Things stack by proposing generic approaches and platforms independent of the variety of devices and technologies involved. We aim to propose solutions suitable for a large variety of infrastructures that can be applied at multiple layers in the stack. Therefore, in our work, we approach the security of all the major components of IoT infrastructures. We start with IoT applications running on embedded computers and their maintenance systems deployed in the cloud and move down the stack to the security of microcontrollers. We also approach the user space together with the kernel space security for these devices.

In the first contribution, we focus on the security of the user space for IoT gateways and embedded computers. We analyze the security modern programming languages bring to the application layer, as most involve a sandbox where the code is executed. In this context, our contribution focuses on identifying some of the high-level programming languages which are suitable for the IoT area and the specific embedded devices [15]. We propose specific use cases for them and research into how versatile they are and how easily they can be adapted to run on a new hardware architecture. This way, we identified JavaScript, together with its lightweight engine, JerryScript [16], as one of the most suitable languages that can be used for programming a large variety of embedded devices, from not so powerful ones to others having more advanced capabilities [17].

Further on, we addressed issues related to development tools for various use cases. We define specific requirements based on the development use case: commercial, prototyping, or educational, and propose secure solutions for each of them. Further in our research, we propose a generic solution capable of abstract operations related to hardware setup and other configurations [18, 19]. We test the solution's efficiency in several environments with various users, proposing a generic platform that enables integrators, hobbyists, and students preparing to work in this field to focus on aspects such as the security of the IoT infrastructures they develop [20, 21].

In the same field related to deployment and maintenance tools, we approached the maintenance of the IoT applications, as once an IoT system is deployed, various changes (e.g., bug fixes, security improvements, new features) need to be made at a certain point. These updates are remote, which introduces multiple difficulties and risks. Our contribution in this thesis consists of a detailed analysis of the challenges related to deployment and update infrastructures, followed by the proposal of a generic mathematical model for such an infrastructure that covers all the identified aspects. What is more, we validate the model with a specific use case implementation [22].

The final contribution focuses on the devices at the bottom of the IoT stack: the sensors and actuators. These rely on low-power, low-memory microcontrollers that cannot run complex software such as a full-fledged operating system. In the case of these devices, securing them is even more of a challenge due to hardware constraints. Therefore, in this thesis, we start with a deep analysis of the security risks involved in the development of these devices and identify existing solutions [23]. In addition, we suggest a secure real-time operating system for microcontrollers that employs advanced technologies and programming languages like Rust and eBPF [24].

2 SECURITY ENFORCED BY MODERN PROGRAMMING LANGUAGES

In this chapter, we focus on proving if using sandboxed programming languages for constrained IoT devices is possible and straightforward. Therefore, we aim to identify areas where C can be replaced with a programming language that reduces the risk of bugs and security breaches.

2.1 Integrating the D Programming Language in Constrained IoT Devices

The D programming language is relatively new when compared to others. It is a general-purpose language meant to be the successor of C++ [25]. While similar to C from the syntax point of view, D aims to be a safer alternative.

With D being used in general-purpose systems, and even in the Linux kernel [26], the aim of our research is to test if we can run D on low-capabilities devices as microcontrollers to replace C with a more secure alternative.

D is a compiled language that relies on the DRuntime. DRuntime is the library that defines the D language and it is also the reason why writing secure D applications for microcontrollers is not possible. The DRuntime is large (around 40MB) and does not fit the memory constraints of most MCUs available on the market.

Therefore, in this research, we aim to adapt the D environment to run on microcontrollers without compromising its advantages.

We choose to implement our research on a Nucleo F429ZI device, which is an ARM Cortex-M4 with a flash of 2MB and an SRAM of 256+4 KB. At the OS layer, we choose Tock, an open-source operating system written in Rust that has full support for this device.

The main aspect to tackle is to enable the cross-compilation of D applications and DRuntime for ARM architectures. For this, we used the *ldc2* LLVM-based compiler.

Considering the task complexity, we followed a two-step approach for this research topic.

We first focused on building the D applications using better C. Better C is an options that enables programmers to build D applications in a similar manner to C as it removes any dependency on DRuntime.

In our case, using better C depends on building a user space library that allows us to run D programs on top of Tock. After we implemented the library, we obtained an environment where

C-like D applications can be compiled and deployed on the Nucleo device. The applications can use the standard Tock library and control any existing peripherals. However, at this point, the security of such an application is similar to any other C app. This is why the next step is required.

The following step requires to adapt DRuntime so it requires less memory resources and can be used to run secure applications. However, at the time of writing this thesis, the existing technology did not allow us to compile the original DRuntime for the target architecture. Moreover, the string concatenation operator cannot be used if we cross-compile DRuntime for the Nucleo. Using it leads to a kernel panic due to unallocated or protected memory access.

This research concludes that in its current form, D does not seem to be a suitable candidate for secure IoT application development, and we decided to shift to other, more mature programming languages such as JavaScript.

2.2 Secure IoT Applications Using JavaScript

As at the time of the writing of this thesis, D is still far from being suitable for embedded applications for microcontrollers, we identify JavaScript as a suitable option for writing embedded applications that are more secure than the ones written in C.

Similarly to D, JavaScript uses a garbage collector that handles all memory allocations, reducing the security risks in the applications. In addition, JavaScript relies on an executor, that brings another security layer as user space code is not directly executed on the CPU.

JavaScript code can be easily deployed on microcontrollers using JerryScript. JerryScript is a lightweight JavaScript engine especially developed to run JS code on low-capabilities devices. It can run on devices with less than 64KB of RAM and less than 200KB of ROM [16].

Considering the popularity of JavaScript in the IoT field and the security it brings to the table, our purpose in the presented research is to test how easy it is to deploy a JavaScript program on a device.

The device we choose for our implementation is the NXP Rapid IoT Prototyping Kit. It contains an NXP Kinetis K64 120MHz 32-bit microcontroller, based on Arm Cortex-M4 Core, NXP Kinetis KW41Z Wireless Controller for BLE, Thread, and Zigbee Connectivity and several integrated sensors: air quality, temperature, gyroscope, accelerometer, magnetometer, etc.

We used Amazon FreeRTOS at the operating system layer, as it is the OS that has the best support for this device.

Oficially, the Rapid IoT Prototyping Kit supports only C as a programming language.

2.2.1 Running JerryScript On the NXP Rapid IoT Prototyping Kit

To port JerryScript to a new hardware device, we first need to compile it on that specific platform, obtaining a static library. Once compiled, the engine is capable of running the JerryScript bytecode. The engine API exposes functions that allow developers to run JavaScript applications and, more importantly, to define native functions that can be called inside these applications.

For our use case, we used the Amazon FreeRTOS source code provided by NXP and added the JerryScript engine as a different service. To compile this version of the RTOS, we used NXP's MCUXpresso tool, which generated a binary that can be uploaded on the device. Once flashed, the device executed the JavaScript code using the engine.

2.2.2 A Build and Deployment Solution for JerryScript

While JerryScript applications can be successfully deployed on the NXP device, the MCUXpresso tool that we used is not a straightforward application. Using it can be a troublesome process. Therefore, in this implementation, we also aim to automatize the application deployment process and remove the dependency on the tool that NXP provides.

First, we used the Hexiwear docking station specific to this device. When attached to it, the board module enters into debug mode and opens a serial connection. Therefore, data can be transferred from the computer to the device if connected to a computer via a USB cable.

In this case, our solution was to build a script that takes the JavaScript code and sends it over the serial interface. On the device side, we added a service to the Amazon FreeRTOS, which was configured to read the information coming on the serial line and pass it to the JerryScript to execute it.

This solution is stable and works without any interruptions. However, it requires the boomerang expansion, which is expensive and also makes the device less robust.

The second solution we implemented was to send the source code over the Bluetooth connection. In this case, we create a different FreeRTOS service that gets the data from the Bluetooth manager and passes the code to the JerryScript engine. This way, the Rapid IoT can be programmed without additional hardware. What is more, no physical connection between the computer and the device is necessary.

Although easy to use and affordable, this solution is not stable. Our tests proved a 50% success rate in delivering the correct information to the device. Furthermore, the results show that the Bluetooth manager on the devices crashes due to a memory leak, as the module is not designed for transferring such large amounts of data.

In this approach, we built the Amazon FreeRTOS JerryScript module so it runs a fixed-size source code. Considering the total memory available, we initialized a 32KB buffer with values of zero. Once the binary is generated using the MCUXpresso tool, we save it in the project folder, at a fixed path. At every run, we inject the JavaScript source code that we aim to run into the zero-buffer space in the binary. By analyzing the binary, we identified the memory address where the string representing the JavaScript code is stored (Figure 1). Each time we want to run an application, we overwrite that memory space in the binary. This solution is possible, as MCUXpresso does not digitally sign the binary it generates, so we can tamper with it and then flash it on the device.

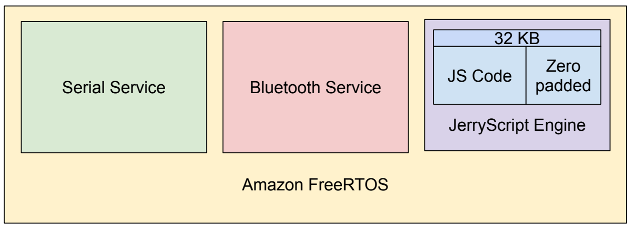

Figure 1: Final firmware architecture.

This approach is the most stable. JavaScript code can be successfully run on the device in 100% of cases. In addition, compared to the previous approaches, in this case, as we inject the JavaScript code, the device will run it even after resetting. In the other two cases, this was not possible.

2.2.3 Tests and Results

Once the porting process was successful, we reached the goal of proving that JavaScript can be easily deployed on microcontrollers. Further on, we tested this programming approach in comparison with the C-based one in various scenarios and the first proved more robust and easy to use.

The result of this research is that JerryScript can be easily ported for various hardware devices and we consider it is a suitable alternative to C for more secure applications development.

2.3 Flow-Based Programming for IoT Infrastructures

Another popular approach for IoT applications programming is using flow-based editors, such as Node-RED [27]. The main idea behind this approach is that the application can be considered a flow of messages that trigger actions. Once the application runs on the device, each time one of the events specified in the used nodes is triggered, a message is generated and passed to the next node. The receiver will execute whatever action or actions it needs to, and it will pass on the same or a new message.

The advantage of such an approach is that the developer can easily visualize the architecture of the system and keep track of how each of the components interact. The visual interface makes it easy to debug, as there is a clear connection between all the elements.

However, even though Node-RED has more advantages over the first approach, it still has a big disadvantage: synchronizing two branches and assuring that certain functions on different branches are called one after another. Another downside is that each node can emit messages arbitrarily, resulting in nondeterministic behavior. However, BPMN, a flow-based approach used in modeling business processes overcomes these downsides.

The solution we propose in this section is to extend BPMN in order to build a more reliable development platform for Internet of Things applications.

2.3.1 The Platform Implementation

In order to build a deployment platform for BPMN-based IoT applications, we use an existing BPMN editor that can be easily integrated into any application. Bpmn.js is an open-source diagram rendering toolkit. We use it for the interface that allows users to build the applications. The toolkit returns an XML structure containing all the elements. Therefore, the first step in the implementation is to translate the XML structure to JSON. Further on, we need to create the controller for the whole project and one for each element [28].

Once the JSON structure is in place, we need to generate a token to flow through the network and activate each element. Each element consumes the token(s) received from the incoming connection(s) and generates a new token that is passed on.

In our implementation, the token is a structure containing multiple fields such as deviceId, tokenId, timestamp, or signature. The next step is to define the behavior of each element (e.g. task, XOR gateway, start event) we can use for modeling the application.

2.3.2 Application Implementation and Results

To test that the BPMN-based approach can be applied to designing Internet of Things applications and that the existing elements are sufficient for building a complex smart system, we designed a coffee machine application using the BPMN editor.

The application we designed is supposed to control a coffee machine equipped with an NFC sensor and an HDMI screen for user interaction. The NFC detects when a user places a recipient in front of the machine, then coffee is dispensed. If the user has a special cup with a unique NFC ID, the machine can identify the id and show the user how many cups of coffee he drank that day. In case his caffeine intake is above a certain limit, a warning message is displayed. The dispenser has the following behavior: when it starts, it pours coffee for 20 seconds, then it automatically stops and displays a message asking the user to remove the cup. In case the user removes the cup earlier than 20 seconds, the dispenser stops immediately. Once the dispenser stops, the amount of poured coffee is computed.

We built this system using procedural programming and flow-based programming as shown in Figure 2.

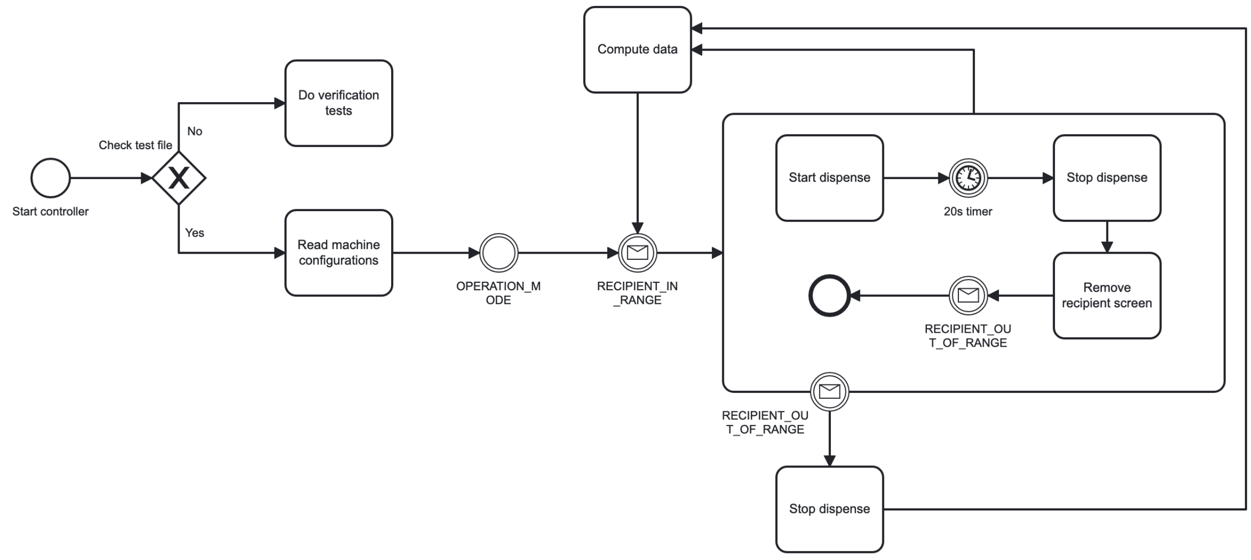

Figure 2: The coffee machine application developed using BPMN.

In the end, the BPMN approach proved easier to debug, reducing the development time by 30%. In addition, the system is much easier to trace and diagnose.

3 SECURE INTERNET OF THINGS DEVELOPMENT

In this chapter, we propose a development platform to support the prototyping of secure IoT applications. We also address the challenges of IoT educators in teaching students to build efficient and secure infrastructures.

3.1 IoT Development and Prototyping Tools

At the hardware layer, the most used device for IoT prototyping is the Raspberry Pi. Therefore, we focused our work on building a development tool that makes the Raspberry Pi easy to program and allows developers to focus on the application development rather than on the necessary configurations and deployment setup.

Our proposed platform, Wyliodrin, is a web-based application that uses secure protocols to enable the programming and monitoring of Raspberry Pi and other embedded devices in a straightforward manner.

3.1.1 Platform Overview

The platform we propose, Wyliodrin, is a web-based application that enables users to remotely connect to their embedded devices, such as the Raspberry Pi, and deploy applications on them. The platform has the following major characteristics:

- Web-based - It is easy to access and does not require any application to be installed on the computer.
- Fast setup - The platform guides the users through a simple series of steps necessary to connect to the device. No advanced networking and programming knowledge is necessary to control the devices.
- Visual monitoring - Debugging information coming from the devices can be mapped on visual graphs for easier monitoring.
- Shell console - For more advanced users, a shell console is available. This allows complete control over the device.
- Secure connection - As security is very important, all data that the devices exchange with the web application is encrypted and sent over a secure channel.
- Visual programming - The platform has support for a Google Blockly-based IDE that allows users to drag and drop blocks and connect them to create an application [29].

Google Chrome Application

The platform was initially implemented as a web application that was dependent on an internet connection. This approach proved to be inefficient in many cases. In various setups where we used the platform, the network was not reliable enough, and devices kept disconnecting. Therefore, we implemented a more stable version that also works with a local network connection. What is more, by changing the communication protocol, we also increased the reliability of remote connections.

As more advanced web technologies became popular, we decided to rewrite the platform and completely replace the device supervisor with another one entirely written in JavaScript (Node.js). On the web application side, we developed the platform as a Google Chrome application that can be easily installed on any computer. We also implemented support for remote connections and a web platform that can run from any browser without any installation.

In this case, the Chrome application acts as the server to which the device connects. All communication is done via HTTP. From the users' point of view, this translates to a more stable connection between the device and the web platform. What is more, we also implemented a way of remotely connecting devices (over Internet) by using an intermediary server.

3.1.2 Platform Architecture and Implementation

As the platform was initially developed for XMPP communication and later on changed to HTTP, the architecture also suffered some changes. For XMPP, the only approach we implemented is dependent on a third-party XMPP server. In contrast, when using HTTP with a device located in the same network, the Chrome app also acts as the server.

XMPP Server

To ensure a high degree of security, the initial system uses XMPP for the messages exchanged between the device and the browser. The main advantages we identified in using XMPP are that all the communication is encrypted, and messages are sent under the XML format, which is easily extensible.

In our implementation, at one end, there is the user who is logged in on the web platform. Next, the web application connects on behalf of the user to an XMPP server. Once the connection is established, the web server retrieves the rosters, which are all the available boards.

HTTP Server

The HTTP implementation relies on two important technologies that enable the data transfer between the device and the web application:

- Avahi - a device discovery tool that allows the web app to identify all the boards connected to the local network;
- Web Sockets - a bi-directional communication channel that enables secure message passing between the device and the Wyliodrin web application.

```
Shell

Open
->
t: 's'
d: {a: 'o', c:columns, r: rows}

<-
t: 's'
d: {a: 'o', r:'d'}
or
t: 's'
d: {a: 'o', r: 'e', e: error}

Keys
<->
t: 's'
d: {a: 'k', t:keys}

<-
t: 's'
d: {a: 'e', e: error}
```

Figure 3: Representation of the messages exchanged by the device and the Wyliodrin Chrome application.

The challenge in implementing this type of communication was enabling the Wyliodrin web application to communicate with multiple devices simultaneously. This required an advanced module for managing the multiple socket instances.

Device Supervisor

One of the essential components of the platform is the device supervisor. This ensures the communication between the server and the device. For this, based on how the technologies evolved over time, we used different implementation approaches.

As an overview, the supervisor is an application that connects to the server, receives commands as messages, executes them, and sends back the result. The source code is also sent as a message, and a child process is started to compile and run it. In the initial implementation, the resulting child process would communicate with the supervisor via pipes. However, pipes do not support text formatting, so the platform could display the result in a non-user-friendly manner. Therefore, we defaulted to creating a pseudoterminal (*pty*) for each process.

3.1.3 Maintaining the Platform

An important essential of the Wyliodrin platform is how it can be maintained. Platforms such as Wyliodrin are difficult to keep up to date as the number of peripherals continuously increases. This means that continuous work needs to be done to ensure that the latest devices and sensors are supported. Besides the connection setup required for each new supported device, the visual programming elements also need to be kept up to date. Each hardware producer that creates its sensors provides its own way of interacting with the respective peripherals. Thus, the number of necessary visual blocks required is increasing at the same rate as the number of peripherals created, which is exceptionally high.

This is why there is a need to create such blocks in a different way than the one provided by the Google Blockly platform. The classical way implies that somebody uses the existing Google platform in order to create each block separately and then integrate the generated element into the IoT platform which is using it. In this context, we focus our research on finding a way to automatically generate blocks and validating if this is a feasible approach.

The first step when integrating Blockly into a platform is to create a web view that will contain the editor. The editor consists of a toolbox listing all the blocks and a dashboard where users can drag and drop the elements. Apart from this basic setup, the interface can be customized. There is the possibility of displaying the generated code in one of the possible programming languages.

Automatic Blocks Generation for Google Blockly

There are thousands of different existent sensors and actuators that can be connected to the existing embedded boards. As we discussed in the previous section, the field is in a continuous expansion, and there is no standard way in which peripherals can be connected yet. This implies that for each group of existing I/O devices, one must use specific functions and communication protocols.

Eclipse Mraa is a C/C++ library created by the Eclipse IoT Project that offers a structured API in order to control the input/output ports of various embedded devices [30]. The library also has Python and JavaScript bindings which make it easier to use.

As a result, it is easier for peripherals manufacturers to map their sensors on top of the supported hardware, while users see a generic set of functions they need to use in order to control the hardware.

By using this library, the number of functions that need to be implemented has shrunk dramatically. Wyliodrin uses this library in order to minimize the number of blocks that need to be implemented.

This section presents a new way in which Wyliodrin can generate new and up-to-date visual blocks.

The generator consists of a script that pulls all *libmraa* files from the git repository and parses them, generating blocks corresponding to each described function. The script generates both visual elements and Python and JavaScript code. All these are stored in a database owned by Wyliodrin. Blocks are automatically retrieved from the database so that this process is transparent for Wyliodrin users who see a complete and up-to-date toolbox.

Generator Architecture

The generator is based on the *libmraa* library, which brought us to the idea that once some generic standard functions exist, it is easy and efficient to create an automatic blocks generator.

The first step was to extract from the library's files the information required in order to obtain some intuitive visual elements. Once the format of libmraa's files was clear, the next step was to generate the file parser. For this, we used a parser generator, Jison [31]. Any input will be converted into the abstract syntax tree, which is, in fact, a JSON structure [32].

After the basic structure is extracted from the source file, the next step is the semantic analysis that checks if the basic JSON structure is correctly translated to one that contains all the relevant information for generating a block.

After everything is parsed, the next step is to do the semantic analysis. First of all, the function's name must be separated from its type. Also, in case a function argument is of an *enum* type, it should be tied to that *enum* structure. After this step, the essential information for auto-generating the blocks is available. Nevertheless, the resulting block would not be much different from the function it represents.

The purpose of Google Blockly is to replace code by using more intuitive elements. This is why it is important to adjust this structure to make it more suitable for this programming language. This is an optimization phase and consists of creating a well-built block. What we did, in these terms, is to interpolate the description of each function, which is basically the main text on the block, with the argument, where possible.

In the end, the result is a new JSON structure that contains all the relevant information required to correctly generate a complete Blockly element.

Parting from all this information, all the rest of the required details can be inferred. This is why the next step is to create a more complex structure from which the visual elements can be easily generated.

During this process, each parameter is converted into an input together with the appropriate fields. For instance, any boolean parameter is transformed into a checkbox, while any enumerator translates to a dropdown list. What is more, basic variables types such as *int* or *char** are converted to *Number*, respectively *String* type.

In the end, the result is a new JSON structure that intuitively represents the correspondence it has to the created block.

Automated Block Generator for Wyliodrin

Starting with the block generator presented in the previous section, we created an API that allows the creation, manipulation, and storage of each new block structure. The API has two parts, one is for use by the web server to generate blocks and change the JSON structure, while the other part is to be used in a web client to display the block and insert its code in the page.

The API server part includes the generator because it exposes the functions that generate the JSON structure from the C/C++ functions. In addition, the API also manages the storage of the blocks. It communicates with a database API that stores data structures and alters them.

On the other hand, the web client API allows the blocks to be displayed in a browser. The module exposes a function that analyzes the JSON structure and calls the Blockly functions that build the block. Basically, this function replaces the initialization function of Blockly.

Once the representation of each block is complete, a script that would pull all libmraa's files, generate the final JSON structure, and store it in a database is required. To accomplish these tasks, we created a Node.js script.

When using the blocks generator with the *mraa* library, we generated a total of 150 blocks. 80% of them are intuitive blocks that can be used as is. The rest require custom changes to be made.

3.1.4 Platform Usage

The Wyliodrin platform is available on GitHub as an open-source project [33]. It can also be directly downloaded as a compiled app for all existing platforms (Windows, Linux, and MacOS), or used directly in the browser [34].

To measure the platform's usage, we integrated a statistics module that allows us to count the number of projects built and the number of new users. We started gathering usage data in October 2019. In the interval 1 Oct 2019 - 31 Dec 2022, a total of 30.000 users developed applications using the web and local versions. What is more, a total of 40.000 new applications were developed and deployed on around 31.000 Raspberry Pi devices.

3.2 IoT Education

As educational IoT projects are, to a certain extent, similar to the prototypes built by the maker community, we adapted the Wyliodrin platform for this specific field.

While some of the features of the standard Wyliodrin platform are suitable for educational needs (e.g., visual programming, easy setup), additional development is required. To this end, in our research to find out if IoT education can prepare students for building secure software, we implemented the following tools to make the educational process more straightforward:

- build a hardware add-on to abstract the electronic circuits;
- build a hardware simulator for institutions that cannot afford the proposed add-on.

3.2.1 Hardware Platform for Circuit Abstraction

This hardware platform works as a shield that students use to build the circuit that their software will control in a more accessible manner.

When designing the platform, we took into account the fact that students have a better understanding of the information they are presented if they can use it in real-life applications of their interest.

The platform we built consists of both hardware and software components designed to be deployed to a class of students. The system is built on top of existing educational tools such as Blockly and the Raspberry Pi [35].

Our solution consists of a Raspberry Pi extension board that exposes a wide array of Grove connectors to which students can connect peripherals such as buttons, LEDs, solar panels, temperature and light sensors, etc. In order to connect the peripherals, students do not need any electronics knowledge, as all the elements are designed to be plugged into the socket, without any other necessary connections required.

The expansion we propose has integrated the following elements, as shown in Figure 4:

- LCD;
- 3 LEDs;
- 2 buttons;
- 30 Grove connectors for digital, analog, and I2C connections.

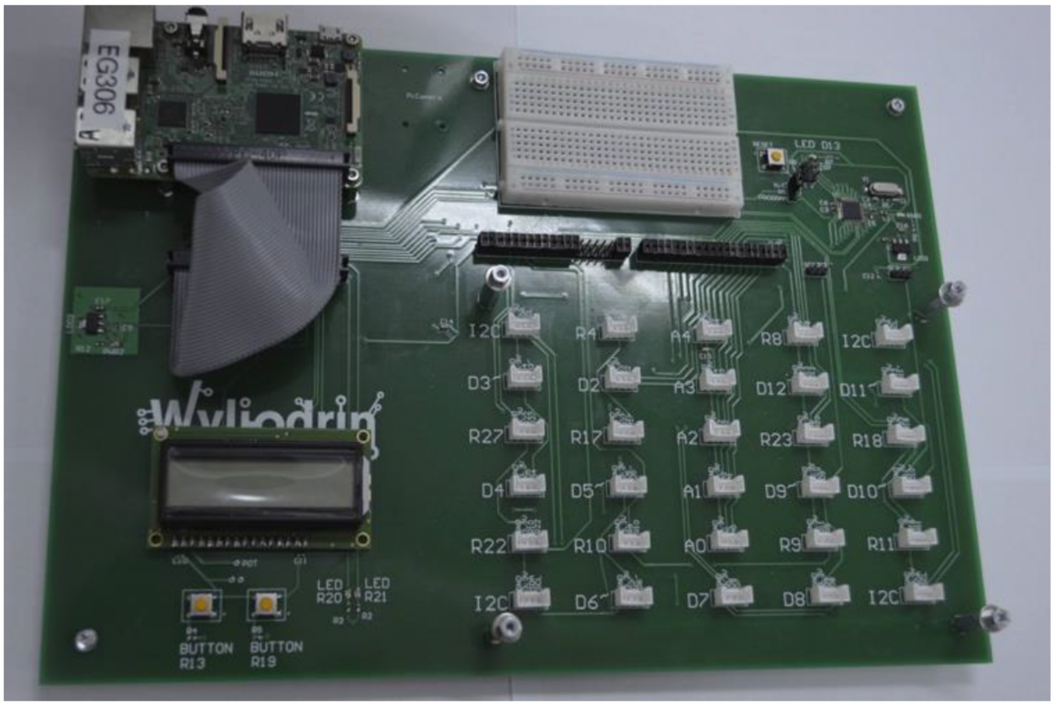

Figure 4: The Raspberry Pi expansion.

To better simulate the systems that students are building, on top of the Raspberry Pi extension, we built a support plate on top of which they can place 3D-printed objects such as traffic lights or house structures. We designed the objects to be fully compatible with the Grove peripherals so students can attach LEDs and sensors to them, obtaining a physical object (Figure 5).

For the actual programming of the devices, students can choose from programming languages such as Python, JavaScript, or Blockly. This way, students accustomed to programming can use a procedural language such as Python, while students with no programming background can get started with using visual blocks.

Classroom Usage

The Wyliodrin expansion was implemented at a computer science course for Power Engineering students during one semester. In order to evaluate the platform's efficiency, we took into account metrics such as student engagement, the complexity of the applications built by

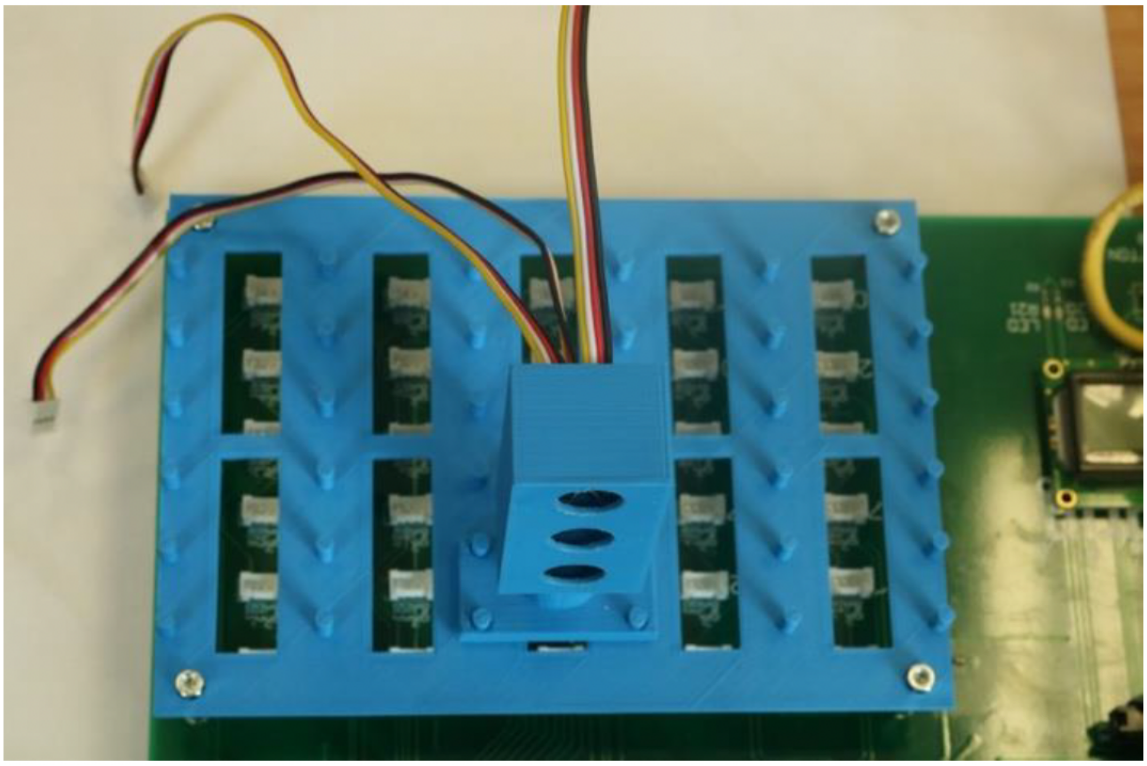

Figure 5: Traffic lights structure connected to the Raspberry Pi expansion board.

students, their final grades, and the overall group progress. We compared these metrics to the ones obtained by the previous cohort of students a year before. While the previous year students worked with simple Raspberry Pi and Arduino devices to control simple peripherals, the following year students used the Wyliodrin Lab platform for applications related to the power engineering field [36, 37].

During the second year, the laboratories required students to build power energy plants, control and monitor the energy consumed by buildings, traffic lights, and street lighting systems [38]. Allowing students to build the physical systems was engaging to them. As a result, students focused on the laboratory and solved all the required exercises, showing a high engagement rate. In contrast, previous students used to lose interest and focus on different aspects, not solving all the laboratory exercises.

Another approach was to introduce students to visual programming based on Blockly and help them transition to Python programming. To do this, we asked students to build a full application using the blocks and then alter the application from the Python code [39]. After a few laboratories, we would create some Blockly elements that generate a faulty Python code and have the students repair the code. Finally, we asked students to build simple Python applications. By the end of the semester, 80% of the students were capable of writing a Python application that reads data from the sensors and controls some LEDs.

For the final exam, 70% of the students could control peripherals, exchange data between different Raspberry Pi devices, and use web services. In contrast, at the end of the semester, the previous cohort had the knowledge to control Raspberry Pi and Arduino devices, with very few students having more advanced skills.

3.2.2 Hardware Simulator

Implementing IoT-related classes implies significant financial costs. This is because building IoT applications requires hardware components, starting with the main development board, where Raspberry Pi and Arduino are the most popular choices, to sensors, actuators, and other add-ons [40]. A kit containing a Raspberry Pi, a breadboard, and a basic set of peripherals has an average cost of 100$, leading the total cost of equipping an IoT-dedicated lab to around 2000$.

In the contribution detailed in this section, we aim to present a solution to the problem introduced above. In order to reduce the costs related to implementing IoT classes, we have designed a software simulator designed especially for education. The platform we propose simulates a Raspberry Pi device connected to various peripherals, that can run basic applications. This solution aims to make IoT a more accessible field and helps introduce such classes into educational institutions by reducing the costs related to the courses.

To achieve this, we added a Raspberry Pi simulator as an additional module to the Wyliodrin platform. The main advantage of this is that students can use the same platform for running applications both in the simulator and on the physical Raspberry Pi. What is more, there are no changes required to run an application on both schematics. The only difference is the device they connect to: a physical Raspberry Pi, or a simulated Raspberry Pi.

The Raspberry Pi simulator works based on Fritzing schematics that can be imported and are parsed into the applications. Currently, the simulator recognizes the following peripherals: LEDs, LCDs, and buttons. Therefore, users can build any schematic that uses these components and import them in the platform we propose.

Further on, as a circuit is selected, students can write applications and run them on the simulated device. Currently, the applications can be written either in Node.js or using Blockly, which generates JavaScript code.

Classroom Usage

To test the solution's efficiency, we used it in an IoT course for first-year Electrical Engineering students without previous knowledge of working with the Raspberry Pi or other circuit elements. Around 25 students attended the class, which lasted for 12 weeks, consisting of a two-hour theoretical course and a two-hours laboratory.

During the first four weeks, students used only the Raspberry Pi simulator integrated into Wyliodrin and switched to working with the physical device. The result was that students managed to implement the simulated circuits and then switched to implementing the physical schematics without frying any of the hardware components. In comparison to this, a year ago, when the solution was not available, students started by implementing simple circuits on the Raspberry Pi, and used to connect the LEDs badly,frying many of them due to short-circuits. What is more, students managed to develop more advanced applications by the end of the semester, as it was easier for them to get started. Many of the students used to be reluctant to start working with hardware components.

4 SECURE UPDATES INFRASTRUCTURES FOR IOT SYSTEMS

The purpose of the research presented in this chapter is to propose a software deployment and updates infrastructure that is built on top of a generic model and is open-source and easy to implement by any IoT integrator.

4.1 The Platform Constraints

In defining the deployment infrastructure, we first analyzed the context in which it is used and the existing constraints. As a result, we define the following design constraints:

- *C1* - The platform has to enable the development and debugging of the applications on hardware devices and in conditions similar to production ones.
- *C2* - The platform has to support beta testing for the software to be deployed. For this, the software is deployed on the same hardware as the one in production and in conditions similar to production ones.
- *C3* - After beta testing is successfully finalized, software updates need to be deployed incrementally. This way, any error in the deployment process can be identified at an early stage.
- *C4* - Application updates need to be scheduled outside the devices operation times (e.g., we cannot update an assembly line while running or a vending machine while it performs product sales).
- *C5* - In case the newly deployed application does not start on certain devices, it needs to be automatically rolled back to the most recent working version.
- *C6* - A monitoring dashboard for the deployed devices is required. Maintainers will have access to the dashboard to visualize deployed devices' behavior, run diagnostic tests, and remotely access the device to perform manual software rollback if necessary.
- *C7* - In case of a compromised device (e.g., a stolen connected bike), maintainers have the possibility of disabling it, thus bringing it to a non-functioning state.
- *C8* - For security, devices need to authenticate and be compatible with Trusted Platform Modules (TPMs) [41](e.g., ARM TrustZone [42], Software Guard Extensions [43]).
- *C9* - Updates must be fast and require small data transfers to preserve bandwidth.
- *C10* - The system needs to be open and easy to integrate with the vendor's infrastructure.

4.2 The Proposed Mathematical Model

Based on the constraints defined above, we define a mathematical model for an embedded software deployment infrastructure that can be implemented by vendors using their preferred technologies.

4.2.1 Terminology

The model we propose is based on a central orchestration unit that manages the devices, their connections, and deployment procedures. From a more detailed perspective, the system consists of the following main elements: *Vendor, User, Product, Cluster, Application, Container Deployment, Project, Event*. Further on, we describe each of these elements in detail:

- *Vendor* - It is the user of the system, or more specifically, the IoT producer, who builds and manages the devices. The vendor owns products, clusters, projects, applications, deployments, and containers.
- *User* - For each vendor, multiple users can be created. Each user can manage the elements described further on.
- *Product* - Each device registered in the platform is a unique product. We refer to it as a product since it is the basic element the vendor sells. Each product uniquely identifies by an id, and also has a name. We define three different product types: development (used for software development and debugging), beta (used for beta testing), and production (commercialized by the vendor). Each product is part of a cluster (defined below).
- *Cluster* - A cluster can contain one or more products. All the products in a cluster are located in a delimited geographical area and run the same software (e.g., a collection of devices gathering soil data about a specific agricultural area).
- *Application* - One application uniquely defines a piece of software to be deployed on a cluster. Each application has a list of versions. What is more, for each application, a list of default parameters used to run the software is defined.
- *Container* - It is the actual software to be deployed on the registered products. The container is stored in a repository.
- *Deployment* - A deployment links one or multiple clusters to an application, an application version number, and a set of application parameters. Once a deployment is created, all products in the target cluster(s) will run the specified version of the selected application using the defined parameters.
- *Event* - Events are definitions of actions that take place as the system is running (e.g., product registration, cluster creation, a new application version).

The application deployment infrastructure we model in this section consists of three core components: the server, the products, and the deployments. The server communicates with the products while the products run the software delivered as a deployment. Based on this generic approach, more specific use cases can be implemented to optimize the deployment and updates system for specific scenarios.

4.2.2 Defined Sets

To mathematically define the infrastructure described above (server, products, and deployments interaction), we define the following sets based on which all the following definitions are implemented:

- P - the set containing all possible product ids; the concrete value is left at the discretion of each implementation;
- C - the set containing all possible cluster ids; the concrete value is left at the discretion of each implementation;
- K - the set containing all possible public key infrastructure (PKI) keys (e.g., ECC [44] or RSA [45]);
- P_a - the set containing all the possible parameters associated with the deployments; he concrete value is left at the discretion of each implementation;
- A - the set containing all the possible application ids; the concrete value is left at the discretion of each implementation;
- S - the set containing all the possible digital signatures based on the keys in K;
- U - the set containing all unique tokens associated with a product; the tokens are used for the product to authenticate; the concrete value is left at the discretion of each implementation;
- E - the set containing all defined errors; the concrete value is left at the discretion of each implementation;
- T - the set containing all the possible product types: *development* - used while the application is under development, and a lot of debugging is done; *beta* - used for beta testing on a limited number of devices in conditions similar to the production ones; *production* - used for the final products that are sold by the vendor.

4.2.3 The Mathematical Model

Based on the sets defined above, we define the following model for a generic software deployment and updates platform that follows the constraints defined previously. Based on this model, the vendor can build and integrate its custom implementation with the complete infrastructure.

Starting from the sets described above, the first definition is for the T set that contains all available product types (1).

$$T = \{development, beta, production\} \tag{1}$$

Further on, we define *M* as the space of all products, where a product is a device running the desired software (2).

$$M = P \times K \times C \times K \times T \times P_a \tag{2}$$

The projection on the space *M*, represented by the vector $\overrightarrow{m}$ has the following *dimensions*: id, private key, cluster id, cluster private key, type, and additional parameters that can be useful for specific implementations (3).

$$\overrightarrow{m} : M = (id_{product}, key_{product}, id_{cluster}, key_{cluster}, type, parameter) \tag{3}$$

Starting from the *M* vector space, we can define the following functions that project the vector components on each of the axes:

- $product$ (4) - the function that projects $\overrightarrow{m}$ on $id_{product}$ of *P*; the result is a value that defines a product;

$$product(\overrightarrow{m}) : M \rightarrow P = \overrightarrow{m} \times (1, 0, 0, 0, 0, 0)^T \tag{4}$$

- k_p (5) - the function that projects $\overrightarrow{m}$ on $key_{product}$ of *K*; the result is a value that defines the product's private key;

$$k_p(\overrightarrow{m}) : M \rightarrow K = \overrightarrow{m} \times (0, 1, 0, 0, 0, 0)^T \tag{5}$$

- $cluster$ (6) - the function that projects $\overrightarrow{m}$ on $id_{cluster}$ of *C*; the result is a value that defines the cluster that contains the product;

$$cluster(\overrightarrow{m}) : M \rightarrow C = \overrightarrow{m} \times (0, 0, 1, 0, 0, 0)^T \tag{6}$$

- k_c (7) - the function that projects $\overrightarrow{m}$ on $key_{cluster}$ of *K*; the result is a value that defines the cluster's private key;

$$k_c(\overrightarrow{m}) : M \rightarrow K = \overrightarrow{m} \times (0, 0, 0, 1, 0, 0)^T \tag{7}$$

- $type$ (8) - the function that projects $\overrightarrow{m}$ on $type$ of *T*; the result is a value that defines the product type;

$$type(\overrightarrow{m}) : M \rightarrow T = \overrightarrow{m} \times (0, 0, 0, 0, 1, 0)^T \tag{8}$$

- $parameters$ (9) - the function that projects $\overrightarrow{m}$ on $parameters$ of P_a; the result is the set of parameters required for each specific implementation;

$$parameters(\overrightarrow{m}) : M \rightarrow P_a = \overrightarrow{m} \times (0, 0, 0, 0, 0, 1)^T \tag{9}$$

In relation to $k \in K$, we define k^T as the corresponding public key.

4.3 Model Implementation and Validation - IoTWay

To verify if the proposed model is suitable for implementation, we have built a deployment and updates platform based on it. The platform is called IoTWay [46] and is an open-source solution based on secure open standards and protocols. However, other implementations can also be made starting from the model we defined in the previous section.

The platform's implementation is based on five major components (server, repository, deployer, client, IDE) that are connected as depicted in Figure 6.

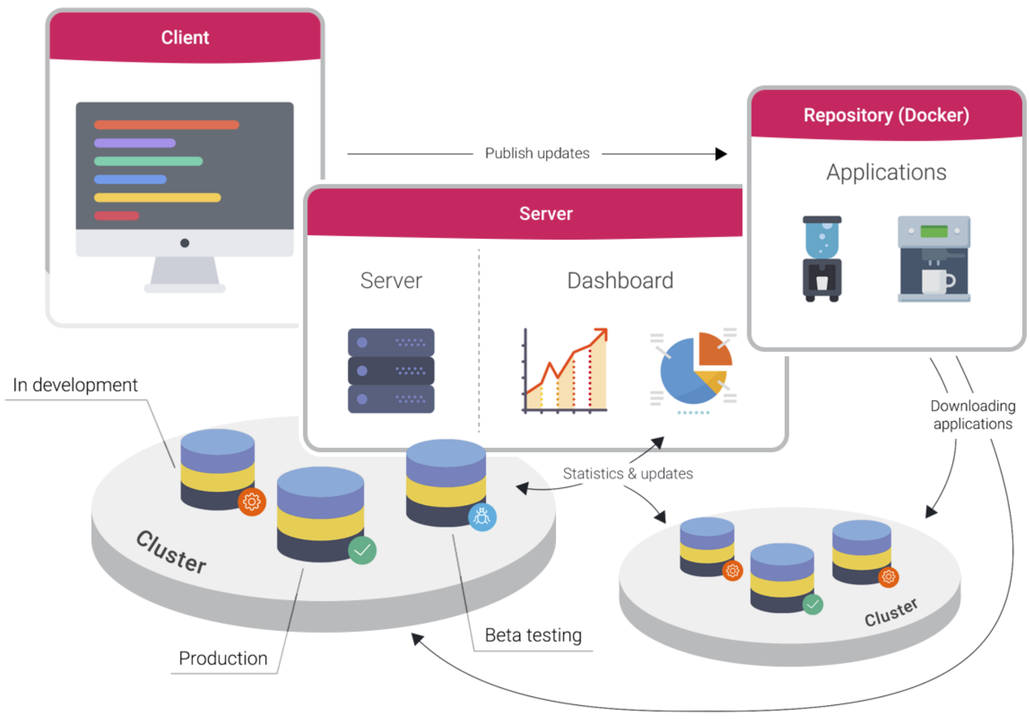

Figure 6: IoTWay high-level system architecture.

The server is the central unit as it manages all essential operations related to users, products, clusters, applications, and deployments. Per *C10*, the server is designed so the platform can be easily integrated with existing infrastructure. Therefore, the server is built as a web service collection that exposes an extensive REST API interface. All exchanges are made over the HTTPS protocol, and the actual data is structured as JSON [47].

The repository is an actual container repository that uses an OAuth token bearer for authentication [48].

The deployer is a lightweight application that runs on each product. The application is in charge of how containers are installed and started on the device. It does the file system mappings necessary for the container and the event logging. Another optional feature that can be disabled is for the deployer to provide an active bidirectional connection with the server so a remote shell can be opened.

The client is an optional component that offers a visual interface to the users. Via the client, the vendors can handle all available operations. However, the server can be easily integrated with any other infrastructure using the REST API, and the client might not be necessary in this case.

The IDE is another optional component. It is a web application that enables vendors to build their applications remotely. It is basically a web interface where users can build their projects.

All these components are interconnected for a successful deployment to happen. For example, in case of an update, the deployer sends a query to the server asking for a list of deployments scheduled to run on the product. Once the server authenticates the product, it sends the list and a set of credentials for the container repository. The deployer downloads the containers using the provided credentials and runs them.

4.4 Test Use Case

To evaluate the efficiency of the IoTWay platform, we used it for a commercial use case. The use case targeted the implementation of a software deployment and updates infrastructure for smart soda dispensers. To extensively test the implementation's efficiency, we used multiple different technologies at various stacks in the infrastructure. This way, we can measure the impact the deployment infrastructure has on the rest of the system's components.

In the implemented use case we connected numerous soda dispensing machines displaying commercials. The users control the dispenser via the touchscreen to select the desired beverage and control the liquid flow (start/stop). All the dispensers were connected to the cloud so consumption data can be monitored. The machines have multiple sensors integrated. They are used for measuring the quantity and type of disposed beverages, monitoring the water filter status, the machine temperature, and energy consumption.

Using IoTWay, the vendor can upload new software on the device and monitor the deployed machines.

4.5 Results

To measure the efficiency of the proposed model and implementation, we deployed around 100 dispensing machines together with a commercial partner, in three different regions: Romania, USA, and India.

4.5.1 Software Characteristics

The IoTWay platform was used for both application development and updates. In this context, the software deployed on the devices ranges from simple applications to more advanced ones implying a user interface.

From the performance point of view, running the latest application version on the BeagleBone Black devices proved troublesome due to the hardware resources available. Many of the transitions would move very slow or freeze. On the other hand, the Raspberry Pi (4 CPU cores) had no performance issues, and the software was successfully run on the Raspberry Pi devices. Moreover, the performance was even better when the GPU rendering was active.

Table 1 outlines the performance comparison between the approaches. The machine load average is computed as the average CPU percent usage over a 10 minutes time span.

Table 1: Machine performance.

Platform	**CPU Speed**	**RAM**	**Avg Load**	**Avg RAM load**
BeagleBone Black	1.0 GHz	512 MB	150%	100%
Raspberry Pi 3 (no GPU driver)	4 x 1.2 GHz	1 GB	40%	60%
Raspberry Pi 3 (GPU driver)	4 x 1.2 GHz	1 GB	10%	60%

The load related to the application run on the device also had an impact on the updates system efficiency. As the BeagleBone Black devices were heavily loaded, network traffic suffered many packet losses and many disconnects.

4.5.2 Deployment Performances

The first parameter we measured related to the implementation's efficiency is the software size for the first deployment and the updates. As IoTWay is designed to support differential updates, the first software deployment is expected to be notably larger in size and take a longer time.

The first image we created was 1.2 GB in size and required 1h for the deployment to complete. The following updates are in the size range of 200-300MB and took 10-15 min to be made. Considering the significant sizes, 20% of the updates failed and required a retry. However, none of the deployments resulted in rollbacks or corrupt devices.

After the first measurements, we decreased the container image size by optimizing the build system. We identified the unnecessary files created during the building process and dropped them. This resulted in an initial container image size of 500MB and update containers ranging between 50 MB and 100 MB. This decreased the initial deployment time to 20-30 min and the updates time to around 5 min.

Table 2: Update performances.

	Initial Deployment Size	Update Size	Update Retry Rate	Initial Deployment Time	Update Time
Initial	1.5 GB	500 MB	20%	1 h	10-15 min
Optimized	200-300 MB	50-100 MB	5%	20-35 min	5 min

Initially, the measurements were made on the Raspberry Pi as it proved to be more stable and have higher hardware performances (Table 2). In contrast, the BeagleBone Black has hardware constraints that affect the deployment infrastructure, leading us to an update retry rate and time that are higher by approximately 30%.

After the container optimization, we used the infrastructure to perform a total of 133 deployments on 100 dispensing machines (Table 3). For the BeagleBone Black devices, 30% of the deployments failed due to network packet losses and faulty disk writes.

Table 3: Update numbers.

Platform	Devices	Updates	Avg. Recovered Devices/Updates	Avg. Unrecovered Devices/Updates	Devices with Other Failures
BeagleBone Black	80	133	25	2	20
Raspberry Pi 3	30	133	3	0	0

4.5.3 Performance Comparison With Balena

The research presented in this chapter has as its primary purpose to propose a software deployment and updates infrastructure that has a solid theoretical foundation and which can be easily adapted for various use cases. At the time when this research was carried out, several other OTA commercial solutions were implemented.

When comparing the proposed model and implementation with other platforms, Balena stands out as an update infrastructure that has several common mechanisms with the IoTWay platform. In this context, we implemented the use case presented above using the Balena infrastructure. We deployed the second software iteration and the 133 updates on the same devices, Raspberry Pi and BeagleBone Black. The results are outlined in table 4.

Table 4: Update performance comparison between IoTWay and Balena.

Platform	Devices	Updates	Avg. Recovered Devices/Update	Avg. Unrecovered Devices/Updates	Devices with Other Failures
IoTWay	110	133	19	2	20
Balena	110	133	17	10	40

5 SECURITY AT THE KERNEL LAYER

The purpose of the work presented in this chapter is to propose a real-time component for an embedded operating system that can also ensure a high degree of security. After a short review of the existing technologies dedicated to embedded applications, we identified that all classic RTOSs have a major security penalty that is tightly related to the fact that all of these operating systems are written in C. Due to the way in which memory management is implemented in C, buffer overflows, and other similar attacks are frequent.

On the other hand, Hubris and Tock, embedded operating systems entirely written in Rust, are not affected by most of the C memory management-related security threats. The only aspect that they are lacking is real-time support. Therefore, we aimed to enhance Tock and integrate a real-time mechanism for achieving a superior security level while ensuring the fast handling of specific events.

5.1 Security Enforcement in Tock

The safety of the Tock operating system relies on three main implementation characteristics:

1. The kernel is written in Rust, with the number of unsafe code lines reduced to the minimum.
2. Drivers are divided into two layers: low-level drivers with direct access to hardware and capsules, and upper-level drivers that abstract the low-lever ones and are not allowed to use unsafe Rust. Most of the development is carried out at the capsule layer.
3. It uses the hardware memory protection to restrict the applications from accessing memory outside their address space.

Applications run in the user space and are compiled completely separately from the kernel. The advantage of this approach is that deploying the kernel and applications can be carried out separately, and applications are installed on top of Tock in a similar manner to general-purpose systems.

Figure 7 depicts the Tock implementation stack, which demonstrates the distinction between kernel space and user space [49].

While this clear separation between the kernel and user space has safety advantages, it introduces delays in the process of interrupt handling, as applications cannot directly expose interrupt handlers.

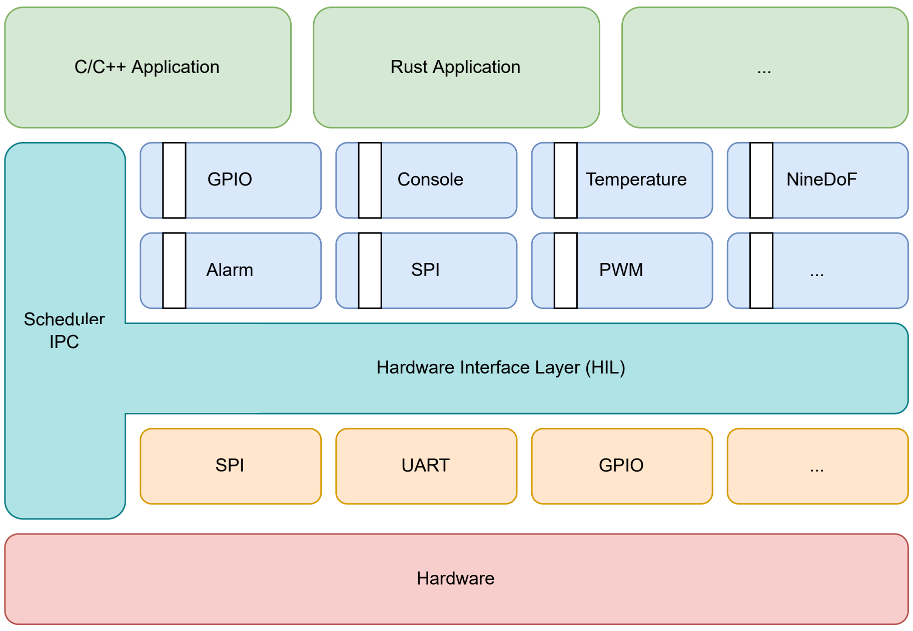

Figure 7: The Tock stack.

5.2 The Approach

Considering the Tock stack, one of the overheads that we identified as preventing the operating system from running low-latency real-time operations is the context switch from kernel to application space. The latency is generated mostly from the time gap between the moment the kernel notices the interrupt and the moment the callback function code in the user space gets executed.

We believe that moving all of the interrupt handling from the user space to kernel space and eliminating the context switch associated with the execution of an interrupt routine will significantly reduce the response time.

The Berkeley Packet Filter (BPF) is a kernel sandbox capable of running tasks for filtering network data. In 2013, BPF suffered important changes and was renamed eBPF, and can be now used for more varied applications, rather than only network filtering. The main capability of eBPF is to inject specific code into the Linux kernel at runtime [50]. It is a straightforward way of dynamically introducing kernel code from the user space without recompiling the kernel.

5.3 eBPF Sandbox Integration Into Tock

The implementation that we propose is to inject the interrupt handling routine in the kernel (driver) instead of registering the callback function for the upcall, as shown in Figure 8.

The injected routine is any eBPF bytecode that the application developer can generate using various tools such as LLVM or gcc. As eBPF is a known standard portable code, this approach ensures that the solution is not dependent on a specific architecture.

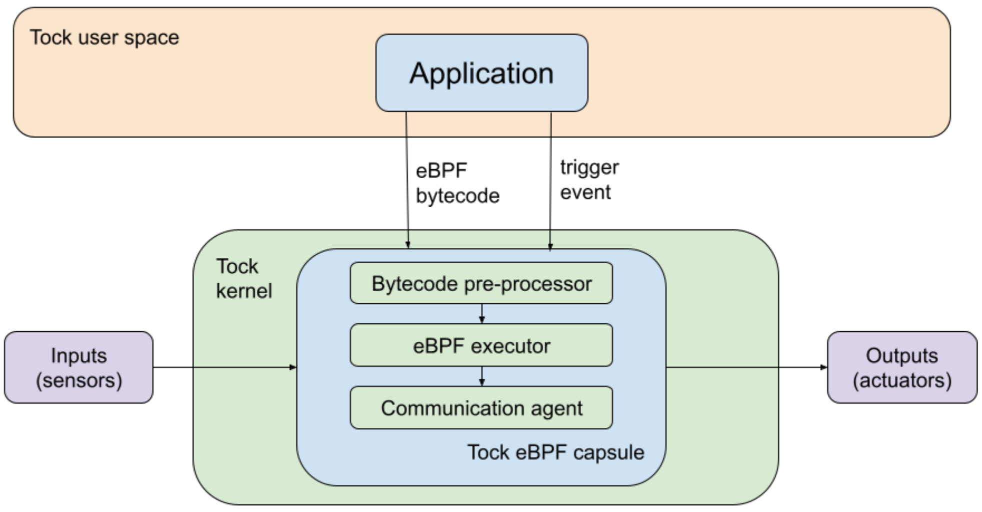

Figure 8: The proposed system architecture.

The proposed implementation relies on the following three main components:

- The eBPF executor - This is the capsule that executes the interrupt handling routine, and that needs Tock kernel and user space interaction.
- The bytecode pre-processor - It modifies the original eBPF bytecode to be compatible with Tock's memory model.
- The user space communication agent with the eBPF sandbox - This is the capsule's API through which the defined bytecode is injected into the executor.

5.3.1 The eBPF Executor

The central component in the architecture that we propose is the executor of the eBPF bytecode developed as a Tock capsule.

In this context, we identified the following constraints concerning the executor's implementation, which are due to Tock's implementation rules:

- C1 - The injected bytecode needs to have a deterministic execution time. It needs to ensure that it finishes executing in a finite amount of time.
- C2 - No unsafe Rust code is allowed at this layer. The executor needs to contain only safe code.
- C3 - No heap memory allocation is allowed in Tock, as heap allocation is not deterministic. Therefore, the eBPF executor cannot use the heap.

For the implementation, we used rbpf [51], a small open-source project that aims to provide an eBPF executor written entirely in Rust.

For our approach, the eBPF sandbox needed to be run in the kernel as a capsule. This led us to outline three main disadvantages in the original rbpf implementation that are not suitable for the use case that we propose.

- Rbpf depends on the Rust standard library, which does not exist in the Tock kernel. Furthermore, it relies on the Vec structure, which uses the heap to allocate data. This contradicts C3.
- Rbpf has large blocks of unsafe code. Tock requires that all capsules have zero lines of unsafe Rust code, contradicting C2 from the constraints list.
- The complete rbpf crate is very large in dimensions, as it is designed to run on general-purpose computers. In our case, where we aim to run it on microcontrollers, the available memory is too small to accommodate all the features. Furthermore, many of the nice-to-have features included in rbpf are unnecessary for our use case, such as JIT or helper functions. While this is not in direct contradiction with the three constraints defined above, it is an important aspect related to the general purpose of our work.

In this context, we generated a custom version of rbpf that has the characteristics necessary to be safely integrated into the Tock kernel. To obtain the custom version, we followed some specific implementation steps:

1. Remove all of the unnecessary features, such as helper functions and JIT-related functions. The eBPF memory is represented as an array to the bytecode program. This is critical with regard to the speed and memory footprint.
2. Rewrite all code parts dependent on the standard rust library, which appear mainly because the memory is represented as a *vec* structure. To achieve this, we replaced the memory representation with a mutable array of 8-bit unsigned integers.
3. Remove all dereferences of raw pointers, which produce unsafe code that cannot be run inside the Tock capsules.

After the alterations mentioned above, the custom rbpf version has all of the capabilities required to be safely run as a Tock capsule on top of embedded devices that have constrained capabilities.

5.3.2 The Bytecode Pre-Processor

In addition to the eBPF sandbox, an important aspect in running safe custom code in the kernel is how the eBPF bytecode to be executed is generated. For this, we need to consider the constraints related to Tock and the hardware capabilities. While, in general, eBPF is designed to run on computers, in our case, we needed to adapt to microcontrollers with reduced processing power and memory.

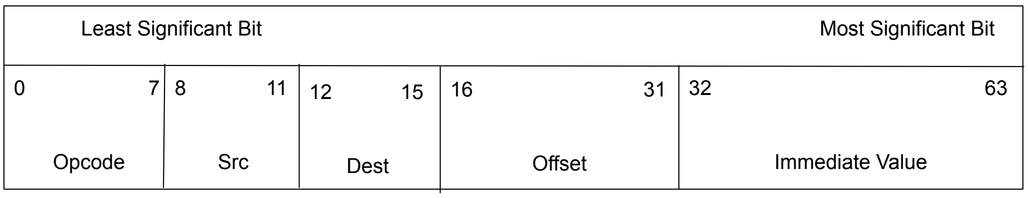

Figure 9: eBPF instruction format.

The eBPF ISA is straightforward, the binary program itself being a long sequence of 64-bit instructions that must respect the format that is presented in Figure 9.

The bytecode pre-processor is part of the rbpf sandbox that we used for our implementation.

The major alteration is related to the memory buffer allocation. Initially, the executor represented the memory area passed from the user space as a *Vec* structure, which is allocated on the heap at the runtime. For our use case where the executor is represented as a Tock capsule, this contradicts C3. Therefore, we changed the original memory management implementation to replace the *Vec* struct with an *array*, which is allocated on the stack.

Another feature of interest related to memory management is that the initial rbpf implementation uses a *Vec* structure for the virtual machine's stack. As we stated before, this contradicts C3, so we removed it. Our solution was to allocate an *array* in the main capsule and pass it to the executor together with the memory structure. The two were concatenated and passed further on. Therefore, one of the challenges is to ensure that operations related to memory management are carried out correctly.

5.3.3 The User Space Communication Agent with the eBPF Sandbox

The final component of the proposed architecture is related to the actual integration of the eBPF executor in the Tock stack. The sandbox was deployed as part of the kernel space but also communicates with the user space and exposes a user space API (Figure 10), which makes the integration more complex. At the kernel layer, the rbpf executor module was included in a custom Tock capsule designed to intermediate this communication.

The capsule we created is meant to offer a generic implementation that allows for interactions with all other existing Tock capsules. In this context, this capsule does not control the hardware directly, but sends commands to already implemented hardware control capsules. However, it needs to associate the bytecode with the necessary hardware operations, identify the appropriate hardware control capsule, and define the necessary commands. Further on, it reads the result that the hardware control capsule returns and transmits it back to the user space.

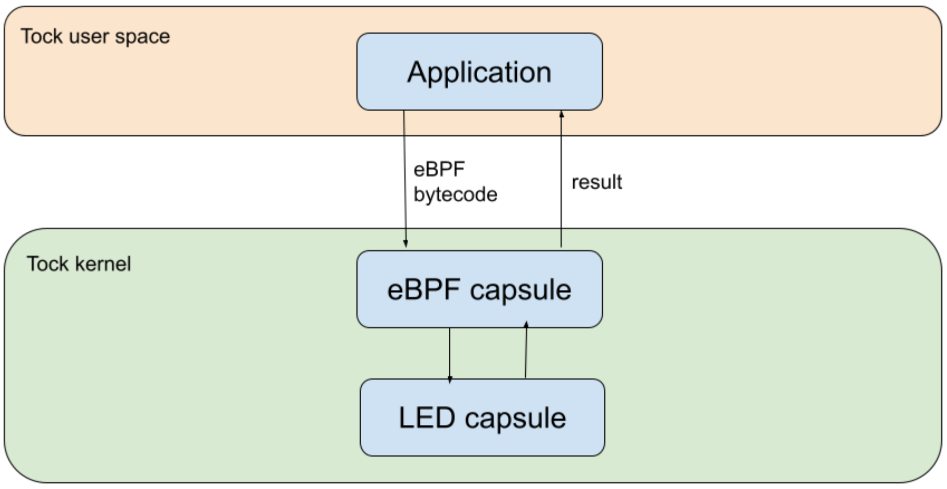

Figure 10: The communication between the application and a peripheral capsule using the eBPF executor.

For the communication with the user space, the capsule receives the bytecode from the application via a command and allows for system calls. The actual bytecode is passed as a byte array stored inside a buffer shared by the kernel and the application. This relies on the Tock standard of sharing data between the user space and the kernel.

5.4 Tests and Results

To evaluate the efficiency of the proposed approach, we measured the system's latency when handling interrupts. As we target obtaining a soft real-time system, the necessary evaluation approach relies on monitoring the system's behavior and measuring latencies.

5.4.1 The Evaluation of the Original Implementation

We first conducted tests to assess the performance of the Tock operating system on various architectures. These tests were meant to identify the original latencies in the Tock kernel and the overall system behavior in handling external triggers. To achieve this, we implemented two test categories:

1. Overall behavior - These are stress tests meant to identify if a device running Tock can handle a massive amount of high-frequency triggers.
2. Latency measurement - These tests focused on evaluating the latency in handling interrupts in Tock.

For all the tests, we used a BBC micro:bit v2 device, which has an nRF52833 MCU. This

MCU has a frequency of 64MHz and is one of the slowest MCUs supported by Tock; however, the micro:bit v2 is one of the most popular devices that is completely supported by Tock.

As the tests involve generating high-frequency triggers, we used an oscilloscope to carry that out.

Overall Behaviour Tests

For these tests, we set the oscilloscope to generate alternative HIGH-LOW values at different frequencies, while the micro:bit ran one or more processes that handle the received interrupts. The interrupt handler routine is defined for both edge triggers and, when called, increments a value and prints it in the serial console.

For the case when the micro:bit runs one application whose only target is to handle these incoming interrupts, we managed to handle all triggers received at a frequency of 2 KHz. For higher frequencies, around 30% of the interrupts are lost. Finally, at a frequency of 10 KHz, the system does not print any message, as it is too fully occupied to handle the interrupts coming from the oscilloscope, and the print function never gets called.

We made the same test with a system that runs two parallel applications simultaneously. We ran the previously mentioned application that handles interrupts in parallel with an application that makes an LED blink once per second. The interrupt frequencies at which the system works are the same, while for frequencies higher than 2 KHz, up to 70% of the interrupts are lost. Similarly to the first case, the system stops printing messages for interrupts generated at a frequency of 10 KHz. However, the LED blink process still functions.

The final test replaces the LED blink application with one that registers an interrupt routine for a button. In this case, for interrupts generated at a frequency of 2 KHz, the overall behavior shows that both routines, the one for the pin connected to the oscilloscope and the one for the pin connected to the button, are called. We still need to investigate the actual behavior that leads to this appearance.

These tests conclude that Tock is not designed to handle high-frequency interrupts and that the interrupt handler mechanism is not optimized for fast responses.

Latency Measurements

To evaluate the interrupt handling latencies specific to Tock, we clocked the response time between syscalls.

All of the measurements were computed as an average of 150 different samples. The deviation in the measurements was around 150 µs.

At the user space layer, we implemented four scenarios:

1. One user space process - The device runs one process that continuously issues a command syscall every 250 milliseconds.
2. Three identical user space processes - The device runs three different processes, each issuing a command syscall every 250 milliseconds.
3. One CPU-intensive process - To put more pressure on the system, it runs one CPU-intensive process (a loop without any delays) and two processes that issue a command syscall every 250 milliseconds.
4. Two CPU-intensive processes - This stress test uses two CPU-intensive processes and one blocking process, similar to the ones presented above.

For each scenario, we performed two different measurements: one for a synchronous syscall and one for an asynchronous syscall.

The synchronous measurement focuses on the duration for a syscall to be transmitted from the user space to the kernel and for the user space to receive the result.

The asynchronous measurement clocks the duration for a syscall to reach the kernel and get back to user space, but in the case of an asynchronous capsule.

The measurement results are displayed in Table 5. The delays obtained are considerably higher than the delays specific to a low-latency real-time system, where the values are around 50 microseconds [52].

Table 5: Latency measurements using a micro:bit device.

	One User Space Process	**Three User Space Processes**	**One CPU-Intensive Process**	**Two CPU-Intensive Processes**
Synchronous measurement	5127 μs	90127 μs	91452 μs	125250 μs
Asynchronous measurement	9213 μs	91545 μs	90643 μs	120903 μs

5.4.2 The Evaluation of the Proposed Approach

The evaluation of the system implemented was carried out incrementally. Specific components of the system were benchmarked, as well as the overall solution.

eBPF Executor Efficiency Tests

The first tests focus on comparing results when running eBPF bytecode using the original rbpf implementation to the one we adapted. This ensures the validity of the proposed solution.

To this end, we created multiple C applications for each load and store operation, compiled them to eBPF, then ran the bytecode in a Rust application [53]. This allowed us to compare the rbpf output to the output obtained from our modified rbpf implementation. These tests were run on a general-purpose system, capable of running both versions of the rbpf implementation.

The testing framework for the correctness of the implementation was also used to evaluate the efficiency of the eBPF executor that we propose. We used the same framework to measure how fast the load and store instructions run in the original rbpf executor compared to the one that we propose.

Table 6 outlines the results, where the timings obtained by the eBPF executor that we implemented are, on average, 3 to 4 times lower for simple operations and around 2.5 times lower for more complex ones.

Table 6: Load and store operations execution speed comparison between the rbpf executor and the proposed executor.

Test Name	**Description**	**Rbpf Executor**	**Proposed rpbf-Based Executor**
LD_ST_DW_REG	Load and Store Double-Word into Reg	2701 µs	660 µs
LD_ABS_DW	Load Double-Word from absolute indexed address	1415 µs	490 µs
ST_DW_IMM	Store Double-Word to absolute indexed address	1986 µs	500 µs
LD_IND_DW	Load Double-Word from indirect indexed address	1698 µs	293 µs
Stack test	Generate a vector of 496 char elements on the stack with values from 0 to 495	75,159 µs	28,358 µs

Complete Platform Tests

For the evaluation of the complete proposed approach, we used the same micro:bit v2 device as in the initial tests. We defined several sets of tests to evaluate both the system's response to interrupt hammering and the delay in interrupt handling. For all of the tests, we used an oscilloscope to generate an oscillating signal. For each test, we performed 50 measurements and computed the average value, which is presented below. The deviation in the obtained results was 5 µs.

We tested the system's responsiveness when receiving interrupts at a frequency of 10 KHz. During these current tests, all interrupts coming at a frequency of 10 KHz were successfully handled.

Further on, we focused on two main test categories to evaluate the response time. These tests were conducted to compare the response time of the eBPF-based approach with the response time of the standard Tock approach and with a capsule designed to handle the same interrupts. The results are outlined in Table 7.

Table 7: Comparison between delays obtained in handling interrupts.

	Standard Tock Approach	**Raw Capsule**	**eBPF Framework**
One GPIO pin	104 µs	14 µs	60 µs
Array of pins	136 µs	43 µs	208 µs

Figure 11 also shows a visual representation of the results.

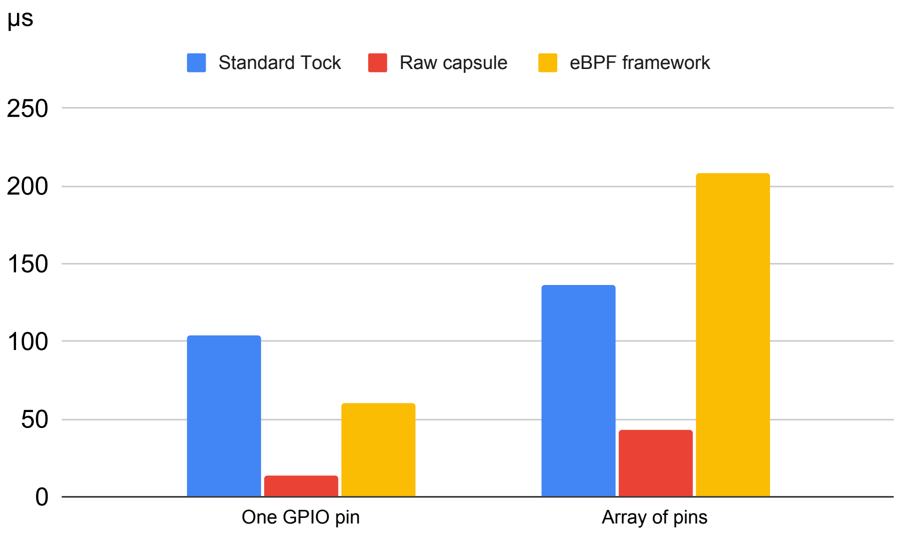

Figure 11: Interrupt handling delays.

The results obtained by the approach that we suggest were compared to the delays measured in the original Tock implementation and to the ones resulted in the case of a dedicated capsule to handle these interrupts. As expected, the delays in the case of a dedicated capsule are the lowest, as all of the application logic is implemented in the kernel.

When compared to the original Tock implementation, the eBPF-approach is faster for the first use case, which is when the GPIO pins are predefined. However, in the second use case, when an array of pins is used, that response time increases significantly. This is due to the operation of iterating the array. In Rust, iterating an array is very time-consuming. Therefore, the main future improvement that we will focus this research on is to reduce this overhead.

5.4.3 Results

The final results outline a significant improvement in the system's response to high-frequency interrupts. To be more precise, we implemented an approach that allows the Tock kernel to run custom code triggered by interrupts coming at a frequency of 10 KHz, while the original kernel freezes during such a use case.

When compared to the raw approach of introducing a custom capsule in the kernel meant to handle specific interrupts, the eBPF-based solution has a lower response time. However, the main advantage of the proposed solution is its generality. It does not involve a different capsule for each different use case but allows the interrupt routine to be injected into the kernel from the user space.

Regarding the delay in handling an incoming event, the approach that we propose measures a mean value of 200 µs between the interrupt being triggered and a pin's status being changed. This value is for the case where we work with an array of pins that are being read. If we resume to a specific use case where the pins are statically defined in the capsule, the delays drop to 60 µs, which is comparable to other real-time systems. The evaluation made by Zhang M. et al. [52] outline that a real-time system implemented using a Raspberry Pi 3 that has a CPU of 1.2 GHz and a BeagleBone Black with a 1 GHz CPU has a response latency between 45 and 75 µs.

6 CONCLUSIONS

This thesis centers around the means of securing Internet of Things infrastructures by considering the diversity of technologies involved in building such a system. Therefore, our work addresses multiple aspects related to IoT, starting from the sensing layer and climbing the IoT stack up to the cloud technologies involved in maintaining any Internet of Things device.

In our first contribution, we focus on the enablement of secure modern programming languages for specific embedded computers and microcontrollers. For instance, we managed to run JerryScript (a light JavaScript runtime) on top of the NXP IoT Rapid Prototyping Kit, a device designed for prototyping IoT applications for smart houses and weather stations. We also researched running such high-level programming languages on more constrained devices and on top of lightweight operating systems, aiming to address the security aspect at both kernel and application layers. For this, we managed to run applications written in D-lang on top of Tock, a secure kernel for microcontrollers written in Rust.

Another complementary aspect that we approached is the visual programming languages. We focused on a frequently used visual programming solution for the Internet of Things: Node-RED. This has a flow-based approach, as programmers use connected nodes to define their infrastructures' behavior and how these interact with each other. In this research work, we proposed an alternative to Node-RED as a BPMN-based solution for defining IoT systems. Due to its specific constraints and the usage of an interpreter, the BPMN-based platform we propose is secure from the application point of view.

As security is an aspect to be taken into account from the prototyping phase in a product's development cycle, we focus one contribution on identifying and proposing technologies that enable integrators to securely and robustly prototype IoT devices. As a result, we propose Wyliodrin, a prototyping platform we designed for safely and efficiently building and deploying applications on embedded computers such as the Raspberry Pi, BeagleBone Black, or the Qualcomm DragonBoard.

We developed Wyliodrin as a generic and easily extensible platform and adapted it to work with various hardware and software technologies. Therefore, this contribution answers the question regarding ways of enforcing the security of IoT systems during the product life cycle.

Extending the proposed IoT prototyping tools, we designed a specific Wyliodrin version targeting education. While similar in many respects, prototyping and educational tools also have certain characteristics that are specific for each of their purposes.

The final solution consists of a modular open source hardware and software platform which can be easily adapted by the maker community and by educators. As a result, in the interval

1 Oct 2019 - 31 Dec 2022, a total of 30.000 users developed applications based on it.

Another contribution addresses the security vulnerabilities related to software deployment and updates in Internet of Things devices. This contribution is divided into two major sections. In the first section, we propose a generic deployment and updates infrastructure that relies on a mathematical model. The second section details the implementation of this model using technologies at the state-of-the-art level such as docker. For the mathematical model, we consider all issues related to software deployment in IoT, with a focus on security. We also take into account the reliability of such an infrastructure and focus on models that prevent the bricking and lockout of a device when it is in the middle of an update.

Further on, to prove the feasibility of the general model we propose, we build an implementation of a generic, open deployment and updates platform which we used to run 13.300 software updates on devices on three different continents, with a success rate of over 70% and with 0 bricked or locked out devices.

Our final work presented in this thesis addresses the lowest layer in the IoT stack, focusing on low-capabilities devices such as microcontrollers and the security of the operating systems running on them.

We leverage the advantages of Tock which is an open-source OS for microcontrollers designed on top of a solid security foundation, but which lacks the real-time characteristic. Based on this, we researched ways of introducing a real-time capability in the Tock kernel so we can obtain an operating system that is both secure and real-time. The approach we took was to leverage the eBPF technology used for network processing and use it for handling low-latency operations. The results we obtained are comparable with FreeRTOS, the most used real-time operating system for microcontrollers.

BIBLIOGRAPHY

[1] J. Margolis, T. T. Oh, S. Jadhav, Y. H. Kim, and J. N. Kim, "An In-Depth Analysis of the Mirai Botnet," in *2017 International Conference on Software Security and Assurance (ICSSA)*, pp. 6–12, 2017.

[2] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, *et al.*, "Understanding the mirai botnet," in *26th USENIX security symposium (USENIX Security 17)*, pp. 1093–1110, 2017.

[3] G. Thomas, "A proactive approach to more secure code." Available online: https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code (accessed on 2 December 2022).

[4] A. Taylor, A. Whalley, D. Jansens, and N. Oskov, "An update on Memory Safety in Chrome." Available online: https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html (accessed on 2 December 2022).

[5] R. J. Walls, N. F. Brown, T. Le Baron, C. A. Shue, H. Okhravi, and B. C. Ward, "Control-flow integrity for real-time embedded systems," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[6] A. Taivalsaari and T. Mikkonen, "A roadmap to the programmable world: software challenges in the IoT era," *IEEE software*, vol. 34, no. 1, pp. 72–80, 2017.

[7] R. Avila, "Embedded Software Programming Languages: Pros, Cons, and Comparisons of Popular Languages." Available online: https://www.qt.io/embedded-development-talk/embedded-software-programming-languages-pros-cons-and-comparisons-of-popular-languages (accessed on 2 December 2022).

[8] S. Bhartiya, "Linux in 2020: 27.8 million lines of code in the kernel, 1.3 million in systemd." Available online: https://www.linux.com/news/linux-in-2020-27-8-million-lines-of-code-in-the-kernel-1-3-million-in-systemd/ (accessed on 2 December 2022).

[9] "Mozilla Welcomes the Rust Foundation." Available online: https://blog.mozilla.org/en/mozilla/mozilla-welcomes-the-rust-foundation (accessed on 2 December 2022).

[10] "Tock Embedded Operating System." Available online: https://www.tockos.org (accessed on 2 December 2022).

[11] "Hubris." Available online: https://hubris.oxide.computer (accessed on 2 December 2022).

[12] "Redox." Available online: https://www.redox-os.org (accessed on 2 December 2022).

[13] "FreeRTOS." Available online: https://www.freertos.org (accessed on 2 December 2022).

[14] "Zephyr Project." Available online: https://www.zephyrproject.org (accessed on 2 December 2022).

[15] T. Severin, I. Culic, and A. Radovici, "Enabling High-Level Programming Languages on IoT Devices," in *2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, pp. 1–6, IEEE, 2020.

[16] "Jerryscript engine for internet of things." Available online: https://jerryscript.net (accessed on 2 December 2022).

[17] I. Culic, A. Radovici, L. Moraru, C. Radu, and J.-A. Vaduva, "Porting JerryScript to NXP Rapid Prototyping Kit," *eLearning & Software for Education*, vol. 2, 2020.

[18] I. Culic and A. Radovici, "Development platform for building advanced Internet of Things systems," in *2017 16th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, pp. 1–5, 2017.

[19] A. Radovici and I. Culic, "Open cloud platform for programming embedded systems," in *2013 RoEduNet International Conference 12th Edition: Networking in Education and Research*, pp. 1–5, IEEE, 2013.

[20] I. Culic, A. Radovici, and C. Dumitru, "Hardware Simulator for Teaching Internet of Things," *eLearning & Software for Education*, vol. 2, 2020.

[21] I. Culic and A. Radovici, "Open Source Technologies in Teaching Internet of Things," *eLearning & Software for Education*, vol. 2, 2017.

[22] A. Radovici, I. Culic, D. Rosner, and F. Oprea, "A model for the remote deployment, update, and safe recovery for commercial sensor-based IoT systems," *Sensors*, vol. 20, no. 16, p. 4393, 2020.

[23] A. Vochescu, I. Culic, and A. Radovici, "Multi-Layer Security Framework for IoT Devices," in *2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, pp. 1–5, IEEE, 2020.

[24] I. Culic, A. Vochescu, and A. Radovici, "A Low-Latency Optimization of a Rust-Based Secure Operating System for Embedded Devices," *Sensors*, vol. 22, no. 22, p. 8700, 2022.

[25] "D Programming Language." Available online: https://dlang.org (accessed on 3 December 2022).

[26] E. Staniloiu, A. Militaru, R. Nitu, and R. Deaconescu, "Safer Linux Kernel Modules using the D Programming Language," *IEEE Access*, pp. 1–1, 2022.

[27] "Node-RED." Available online: https://nodered.org/ (accessed on 4 December 2022).

[28] "bpmn-js source code." Available online: https://github.com/bpmn-io/bpmn-js (accessed on 26 December 2022).

[29] "Blockly." Available online: https://developers.google.com/blockly (accessed on 4 December 2022).

[30] "Eclipse/Mraa - Github." Available online: https://github.com/eclipse/mraa (accessed on 5 January 2023).

[31] "jison - Github." Available online: https://github.com/zaach/jison (accessed on 5 January 2023).

[32] D. Grune, K. Van Reeuwijk, H. E. Bal, C. J. Jacobs, and K. Langendoen, *Modern compiler design*. Springer Science & Business Media, 2012.

[33] "Wyliodrin - Github." Available online: https://github.com/wyliodrinstudio/WyliodrinSTUDIO (accessed on 7 January 2023).

[34] "Wyliodrin." Available online: https://wyliodrin.studio/ (accessed on 7 January 2023).

[35] "Buy a Raspberry Pi - Raspberry Pi." https://www.raspberrypi.org/products (accessed on 19 February 2023).

[36] E. Crawley, J. Malmqvist, S. Ostlund, D. Brodeur, and K. Edstrom, "Rethinking engineering education," *The CDIO approach*, vol. 302, no. 2, pp. 60–62, 2007.

[37] G. T. Heydt and V. Vittal, "Feeding our profession [power engineering education]," *IEEE Power and Energy Magazine*, vol. 1, no. 1, pp. 38–45, 2003.

[38] A. Radovici, I. Culic, O. Stoica, and D. Rosner, "Building a Smart City Infrastructure using Raspberry Pi and Arduino," 2016.

[39] B. Burd, L. Barker, F. A. F. Pérez, I. Russell, B. Siever, L. Tudor, M. McCarthy, and I. Pollock, "The internet of things in undergraduate computer and information science education: exploring curricula and pedagogy," in *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, pp. 200–216, 2018.

[40] J. Sobota, R. PiŜl, P. Balda, and M. Schlegel, "Raspberry Pi and Arduino boards in control education," *IFAC Proceedings Volumes*, vol. 46, no. 17, pp. 7–12, 2013.

[41] S. L. Kinney, *Trusted platform module basics: using TPM in embedded systems*. Elsevier, 2006.

[42] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A Comprehensive Survey," *ACM Comput. Surv.*, vol. 51, 01 2019.

[43] V. Costan and S. Devadas, "Intel SGX Explained.," *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

[44] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987.

[45] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[46] "IoTWay." Available online: https://iotway.io (accessed on 19 February 2023).

[47] P. Bourhis, J. L. Reutter, F. Suárez, and D. Vrgoč, "JSON: data model, query languages and schema specification," in *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on principles of database systems*, pp. 123–135, 2017.

[48] B. Leiba, "OAuth Web Authorization Protocol," *IEEE Internet Computing*, vol. 16, no. 1, pp. 74–77, 2012.

[49] "Tock source code - Github." Available online: https://github.com/tock/tock (accessed on 22 September 2022).

[50] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network services with ebpf: Experience and lessons learned," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pp. 1–8, 2018.

[51] "rbpf - Github." Available online: https://github.com/qmonnet/rbpf (accessed on 22 September 2022).

[52] M. Zhang, M. Timmerman, L. Perneel, and T. Goedemé, "Which is the best real-time operating system for drones? evaluation of the real-time characteristics of nuttx and chibios," in *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*, pp. 582–590, 2021.

[53] "rbpf_tests - Github." Available online: https://github.com/WyliodrinEmbeddedIoT/rbpf_tests (accessed on 22 September 2022).