IMPERIAL

MEng JMC Final Year Project
Final Report

Department of Computing

Imperial College of Science, Technology and Medicine

µBPF - Using eBPF for Microcontroller Compartmentalization

Supervisor:
Dr Marios Kogias

Author:
Szymon Kubica

Second marker:
Dr Ce Guo

June 19, 2024

Submitted in partial fulfillment of the requirements for the MEng Joint Mathematics and Computer Science of Imperial College London

Abstract

Although eBPF (Extended Berkeley Packet Filter) started as a virtualization technology used inside the Linux kernel to allow for executing user code inside of the kernel in a safe way, it is a general purpose software fault isolation technology. The specification of the eBPF instruction set is, also, suitable for using it as a VM for low-end network-enabled embedded devices to achieve software isolation, compartmentalization and allow for updating deployed firmware over-the-air. Existing solutions for running eBPF programs on microcontrollers use bytecode interpreters which incurs execution time and code size overhead compared to native code execution. Additionally, they do not support data relocations which limits the space of programs that can be executed. We implement µBPF - an eBPF virtual machine and a JIT compiler targeting ARMv7-eM architecture. µBPF is compatible with embedded operating systems capable of supporting SUIT firmware update protocol. We implement a secure program deployment pipeline for RIOT - an operating system commonly used in IoT applications. Our evaluation shows that µBPF JIT achieves native performance and up to of 50% code size reduction compared to the eBPF binaries.

Acknowledgements

I would like to thank Dr. Marios Kogias, for his invaluable support, insight and guidance during the supervision of this project. I would like to express my gratitude for his time, shared experience and help when preparing the submission for the ACM eBPF '24 Workshop. I also thank Dr. Ce Guo for reviewing the interim report and providing valuable feedback regarding academic writing and report structure.

I would like to thank Saleem Rashid for his feedback and critical judgement of the ideas that we discussed during the regular meetings held over the course of this project.

I extend my thanks to Quentin Monnet, the creator or `rbpf`, for being open to collaboration, help and patience when reviewing the contribution introduced by this project.

I would also like to thank my personal tutor, Dr. Anandha Gopalan, for his guidance and support over the last four years of my degree.

Finally, I would like to thank my family for encouragement, support and making it possible for me to undertake the journey of studying at Imperial.

Contents

Chapter 1

Introduction

Extended Berkeley Packet Filter (eBPF [1]) is a virtualization technology originating from the Linux kernel. It allows for augmenting the behaviour of the kernel by compiling short programs written in a subset of C (or any other compatible eBPF front-end) into eBPF bytecode and then loading it into a predefined set of points (hooks) scattered throughout the kernel. The programs are then executed in an isolated virtual machine (VM) environment when a specific event takes place (e.g. a system call is executed). To ensure that malicious code does not compromise the kernel, each program is passed through a verifier before being loaded.

This compile-verify-execute workflow translates particularly well into embedded systems, where software isolation and compartmentalization is becoming increasingly important [2]. It is estimated that as of 2020, there were 250 billion microcontroller devices in use [3]. A large percentage of them is network-enabled and deployed in IoT distributed systems. With the increase of connectivity the attack surface that is exposed to malicious actors increases substantially. Because of this, a secure mechanism for compartmentalized execution of programs is needed. Prior solutions to this problem involve introducing new instruction set architectures (ISA), (e.g CHERIoT [4] or ARM TrustZone [5]) or running VM environments on the embedded hardware [6, 7, 8].

Among other VM solutions such as WASM [9], or JavaScript [6], eBPF is particularly suitable for resource constrained hardware because its ISA is simple and supports verification. Since eBPF stack size is limited to 512 bytes, the RAM requirements of hosting an eBPF VM are minimal. Compared to alternatives such as WASM3 [9] (85 KiB) or MicroPython (8.2 KiB), running an eBPF VM on a RIOT [10] microkernel requires 0.6 KiB [11] which is an order of magnitude smaller. A simple ISA also reduces the ROM requirements of the VM implementation, the rBPF VM [8] requires 4.4 KiB which is again an order of magnitude smaller than the alternatives [11].

The eBPF ISA was designed to support program verification. The Linux kernel eBPF subsystem contains a verifier responsible for validating the eBPF bytecode before it gets executed. A verification mechanism can be used to ensure isolation of lightweight containers deployed on microcontrollers as it provides safety guarantees for the deployed software. However, the Linux kernel verifier is very restrictive and only validates a specific subset of eBPF programs that are safe to be executed in the kernel. Moreover, it is licensed under GPL2 and cannot be used outside of the Linux kernel source code. Because of this, the available userspace eBPF virtual machines (e.g. `rbpf` [12] or uBPF) often implement simplified and less restrictive verifiers. Because of hardware constraints of low-end microcontrollers, the overhead of verification before executing each eBPF program can become an issue. To solve this, architectures that decouple verification from execution by checking the eBPF program on a remote machine before loading the bytecode onto the microcontroller have been proposed [13].

The existing deployment workflows for embedded IoT systems are limited. When contrasted with the contemporary ways of deploying distributed systems (e.g. server-less or microservices [7]), deploying software on microcontroller devices resembles solutions used at the end of the 20th century [11]. Changing the deployed code often requires flashing the device with a new version of the software. This is problematic when the microcontroller is already deployed and wired access to it might be limited. Furthermore, the logic deployed on low-end IoT devices is often not known upfront and requires firmware updates. For instance, it might be necessary to calibrate the sensitivity of sensors connected to a deployed device. For those cases, related work [7] proposes hosting and updating application business logic using over-the-air scripting by running it in a lightweight JavaScript interpreter [6].
Prior solutions for using eBPF as a virtualisation mechanism on embedded hardware suffer from the following limitations **i) Execution time overhead is high**. Existing solutions use an eBPF bytecode interpreter. This is an order of magnitude slower than native C and as shown in [8] two times slower than using a WASM3 interpreter. **ii) The program is larger than native code**. eBPF ISA is a 64-bit fixed-size instruction set. This means that most instructions have unutilised fields (always set to 0), which results in bytecode relatively larger compared to alternatives. **iii) Compatibility is limited**. Existing solutions do not support all valid eBPF programs (e.g. missing data relocations [8] or read-only program data [12]) **iv) Verification capabilities are limited**. Verifiers used by the available eBPF VM's are simple and do not support restricting access to eBPF helper functions. Given the execution time and program size overhead, related work [11, 8] suggests to use a transpilation technique to emit native machine code given the eBPF instructions at runtime (just-in-time (JIT) compilation). This can be computationally expensive, thus an alternative approach was proposed to compile the program ahead-of-time (AOT) in an emulator running on a more powerful machine and then use a relocation mechanism to load and execute the AOT-compiled code on the target device [13].
In this work we introduce µBPF to tackle those limitations and bring software updatability similar to the desktop-grade solutions such as Docker to the embedded systems space [7]. µBPF is a VM, JIT compiler, and deployment framework for compartmentalized code execution on microcontrollers. We design µBPF on top of RIOT [10] and implement it in Rust. Our evaluation shows that µBPF JIT implementation achieves native performance and up of 50% program size reduction.
The key contributions in this work are:

- we port `rbpf` - an userspace eBPF VM to run on embedded `no_std` hardware, our port was submitted as a pull request and successfully merged into the upstream repository,
- we implement an eBPF JIT compiler targeting the ARMv7-eM ISA compatible with microcontroller hardware such as ARM Cortex M4,
- we implement a deployment framework for executing eBPF on devices supported by RIOT,
- we propose a mechanism for restricting access of eBPF programs to the set of helper functions provided by the OS allowing to compartmentalize business logic deployed on target devices.

An early version of this report was submitted for review to the **ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions**. It was accepted and will be presented at the conference in August 2024.

Ethical Discussion Note All deliverables of this project will be published under a permissive open-source license (MIT). This work does not raise any additional ethical concerns.

Chapter 2

Background

In this chapter, we:

- describe the use of embedded devices in the **Internet of Things**.
- describe **real time operating systems (RTOS)** and the guarantees they need to provide,
- introduce **compartmentalization** and **isolation** and describe difficulties with achieving these properties on resource-constrained devices,
- describe existing **software update mechanisms** for embedded devices,
- introduce the **CoAP protocol** and **SUIT specification** which were used for performing over-the-air updates of microcontroller firmware in this project.
- introduce the concept of **JIT compilation** and evaluate it as an optimization technique.
- introduce **eBPF specification** and evaluate it in the context of low-end microcontrollers.

2.1 Landscape of Embedded Devices

2.1.1 Internet of Things

Internet of Things (IoT) is an emerging paradigm utilising communication between embedded devices deployed in a distributed system. An example of an IoT system could be a smart home, a network of devices deployed in a house to control appliances and gather data about the property. Literature review [14] has indicated that among the key challenges faced by IoT are security and privacy issues, which become even more important as an increasingly large number of devices communicates over a wireless network. Moreover, **scalability** of the deployed solutions becomes a concern as the processing power and memory size of embedded devices is limited. Another important aspect of IoT is the Quality of Service (QoS), which includes performance, availability, security, cost and energy consumption. In the context of this work, the most important QoS factors that are taken into account are **security**, **cost** and **performance**. Performance is considered both in terms of raw execution time of the programs deployed by the users of the system as well as the time it takes to process them before they can be executed. Cost is understood with respect to the binary size the programs that are loaded into the system. Security is considered in the deployment pipeline mechanism and achieved by using a firmware upload mechanism compliant with SUIT [15] specification.

2.1.2 Real-Time Operating System

Real-time operating systems (RTOS) are required to guarantee that the time to complete a given operation is bounded and can be reasoned about. The spectrum of real-time requirements ranges from soft to hard requirements. In case of RIOT [10], soft real-time requirements are met by controlling interrupt latency and using priority-based scheduling. This can be contrasted with hard hardware-level real-time requirements that were investigated by CHERIoT [4] - a novel instruction set architecture based on CHERI [16]. In the latter case, it is required that the latency of all hardware operations is deterministic and bounded. Because of this, the proposed architecture of CHERIoT could not include memory virtualisation or caches. TLB or cache misses would introduce an unpredictable latency, making the strict real-time requirements impossible to meet.
When evaluating virtual machine solutions deployed on embedded devices, real-time requirements and guarantees are an important factor to consider. Related work [8] noted that the worst case execution time of the code running in the virtualised environment should be controlled. In case of IoT deployments, malicious code running in the VM indefinitely could lead to the battery being drained faster or cause a slow down of the execution of other tasks running on the device (in the case when priority-based scheduling is used e.g. RIOT [10]). Related work [17] proposed Janus - a programmable monitoring an control system where near real-time guarantees were achieved by patching the eBPF bytecode. The proposed solution involved instrumenting the bytecode with instructions responsible for measuring the execution time and preempting the executed function when the execution time quota is exceeded.

2.1.3 Compartmentalization and Isolation

Compartmentalization refers to dividing a system into isolated sections or 'compartments' to enhance security, reliability, and manageability [18]. This approach can be used in the context of microcontrollers to ensure that all components of the system operate independently, which prevents faults or malicious actions in one compartment from compromising the security and interfering with operation of the entire device. The key benefits of achieving compartmentalization include:

- **Separation of Privileges**: different compartments can run with different levels of privilege.
- **Memory Protection**: each compartment is allocated a specific memory space, preventing unauthorized access to other compartments' memory. Techniques like hardware-based memory protection units (MPUs) or memory management units (MMUs) are used [5].
- **Sandboxing**: applications or components run in isolated environments (sandboxes) where they can operate without affecting the rest of the system.
- **Error Containment**: if a fault occurs in one compartment, it does not propagate to others.
- **Easier Updates and Patches**: compartmentalized systems allow for updating or patching individual components without affecting the whole system.

In the context of embedded devices, easier updates and patches are particularly important. Given the limitations of the existing software update workflows on microcontrollers (see 2.1.4), related work on JavaScript containers [7] proposed a mechanism for deploying and updating business logic hosted on embedded devices over-the-air (see 3.1.1). The solution used isolated JavaScript

interpreters to execute short scripts and update their behaviour without rebooting or re-flashing the device.
Possible solutions for achieving compartmentalization can be implemented in both software and hardware, some of the examples include:

- **virtual machines** (VMs) or containers that run different applications or operating systems on the same hardware. Each VM operates in its own isolated environment (e.g. [11, 7]).
- **Trusted Execution Environments** (TEEs) which provide a secure area within the main CPU, running secure code alongside the OS but isolated from it (e.g. ARM TrustZone [5]).
- **Hardware Security Modules** (HSMs) - dedicated hardware devices used to manage digital keys and perform cryptographic operations. They operate independently from the main processor, providing a secure compartment for sensitive operations [19].

In this project we focus on achieving compartmentalization by using lightweight eBPF virtual machines. The reason for this is that the hardware we are targeting (e.g. STM32F4 or esp32) is heavily constrained and does not provide support for TEEs or HSMs. Moreover, the two latter solutions do not help with achieving easier software updates and patches, which is one of the key requirements for the IoT use case that we are considering.

2.1.4 Existing Software Update Mechanisms for IoT - Limitations

Large Update Size. The most typical approach of performing an update is sending a new version of the entire firmware running on the device [20]. A more sophisticated solution aims to minimise the update size by using partial firmware updates. It is either done loading binary modules dynamically (via dynamic linking of ELF files) [21] or performing differential binary patching [22]. In the latter case the firmware update specifies precisely which sections of the binary need to be updated using a difference-based approach.
Reboot is Required. However, in all of the aforementioned cases, a reboot of the device is required to load the new firmware. This might not always be possible e.g. if the device persists some of its state in memory or needs to operate continuously.
Increased ROM Requirement. Moreover, these approaches require an increased ROM capacity (especially the first solution - replacing the entire firmware) as the running OS needs to write the new, updated image into the flash storage before reboot while the current version firmware is running using the code written in a different part of the ROM. In the worst case, it means that the flash capacity of the microcontroller needs to be at least twice as large as the image size of the OS to allow for a full firmware update triggered over the network.
Related work [7] aims to solve these limitations with virtualization by proposing a mechanism for deploying small software functions on embedded devices and executing them in a VM. This minimises update size by only replacing short scripts containing the application business logic. Furthermore, a reboot is not required as the updated program can be loaded into a new instance of the VM and the old one can simply be terminated. This however is limited as only the business logic controlled by the short scripts can be easily updated. If the code responsible for orchestrating the VM execution needs updating, one of the previous solutions needs to be used.
This project uses eBPF VMs to perform business logic updates by deploying short software functions. It is important to minimize the update size as the networks used by IoT devices often suffer from high packet loss rates and special network protocols (e.g. COAP) need to be used.

2.1.5 CoAP Protocol

Constrained Application Protocol (CoAP) [23] is a protocol designed for use with resource-constrained nodes and constrained networks (e.g. low-power). The nodes are typically embedded devices with limited memory and storage, and the networks used for communication are most often constrained with respect to throughput or suffer from high packet loss rates. An example of such network is IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs). CoAP is widely used in IoT applications and most current RTOS implementations (e.g. RIOT [10]) provide support for it. In this project CoAP was used to communicate with microcontrollers over the low-power IEEE 802.11 b/g/n network [24].
CoAP in its specification is very similar to HTTP, it also adopts a request-response model and uses URIs to identify resources. This design allows for easy integration with HTTP while keeping the overhead of running the protocol relatively low so that it can be used on constrained devices. A range of libraries (e.g. `aiocoap` [25]) and tools exist to allow for using CoAP clients or servers in a similar way to HTTP .

2.1.6 SUIT Specification

Software Updates for Internet of Things (SUIT) [15] is a specification which defines a unified update format for IoT devices. SUIT is motivated by the fact that there is a wide range of network protocols used by networked embedded devices which makes performing updates difficult. Furthermore, the devices originate from diverse vendors often with vendor-specific update management systems. SUIT simplifies this process by defining a specification that unifies the update management across devices originating from different vendors. The SUIT specification defines the core operations that a particular update mechanism should provide (e.g. verify device identity, verify updated image, fetch the image) as well as a set of extensions that can be implemented depending on the use case. In case of RIOT the SUIT update workflow involves:

- generating the manifest file including: update sequence number, image location, target device name, CoAP url of the server hosting the image,
- signing the manifest using the keys matching the ones present on the running OS.

This workflow is used by µBPF to allow for fetching new images with eBPF bytecode from a CoAP fileserver running on a desktop machine. These images are then fetched using the SUIT subsystem provided by RIOT, loaded into the VMs and executed.

2.2 JIT Compilation

A Just-In-Time (JIT) compiler is a component of a VM interpreter which compiles parts of the program code during execution. It is an optimization technique aimed at improving the performance of interpreted execution by identifying 'hot-spots' - parts of the program which are frequently executed by the interpreter. The hot-spots are then compiled into native machine code so that they can be executed without the code interpretation overhead. For instance, JIT compilation is used inside of the JVM [26] to dynamically improve performance of java programs by JIT-compiling frequently used parts of the bytecode.
In the context of this project and eBPF in general, JIT compilation refers to transpiling the eBPF bytecode into the native instructions immediately before executing the programs. There is no

dynamic process of identifying hot spots in the eBPF programs. In this case, compared to e.g. the Java JIT, the JIT compilers considered in this project are actually ahead-of-time (AOT) compilers, with the difference that the compilation takes place after the eBPF program has been verified and is about to be loaded into the VM. One can think of it as a coarse-grained JIT compilation as it happens immediately before execution, however always the entire program gets JIT compiled. In case of µBPF after a given program is JIT-compiled it is then stored in a designated memory storage so that it can be executed multiple times without the need for recompilation.

2.3 Extended Berkeley Packet Filters

Berkeley Packet Filter (BPF [27]) is an in-kernel virtual machine responsible for filtering network packets. It's relatively small and available on most Unix-based operating systems. In the Linux kernel it was extended to allow for multiple applications unrelated to network packet filtering. Contemporary applications of eBPF include tracing, observability, and security [1]. The eBPF VM allows for executing custom code at predefined entrypoints (hooks) located throughout the kernel (e.g. on execution of system calls). The motivation behind introducing a small VM inside of the kernel is to allow for running custom code which augments the behaviour of the kernel without the need for submitting a kernel patch and going through the lengthy process of having the change integrated into the latest version of the kernel.

2.3.1 Origins and Motivation Behind eBPF

The reason eBPF came about was the process of requesting a new feature to be included in the Linux kernel. Historically if a given user wanted to e.g. have a new observability feature included in the kernel, firstly they would have to convince the maintainers of that particular kernel subsystem that the feature in question would indeed benefit all users. The next step would be to submit a patch and have the feature included in the next release of the latest kernel. The problem was that after those two steps are complete, it would still take some time before that particular release is available on the Linux distribution running on the user's machine [28]. Because of this lengthy process, the idea of eBPF was proposed to allow for modifying the kernel at runtime with custom code on demand at a predefined set of locations. This change is very beneficial as, the only previously-available alternative was to implement and load a kernel module [1]. The problem with kernel modules is that they need to be updated after each kernel release as the incoming changes might break their functionality. Furthermore, the execution of the kernel module is not sandboxed (no use of VM) and thus programmer errors in the module's source code pose a risk of corrupting the kernel and consequently bringing the entire system down. By contrast, eBPF code needs to pass a verification stage before it is loaded into the kernel and even if some malfunctioning code passes verification, it is still executed in a virtual machine which introduces a proper security boundary.

2.3.2 Specification of the eBPF Virtual Machine

The **eBPF Virtual Machine** executes a program in the form of eBPF bytecode instructions. This bytecode is generated by compiling a restricted version of C (or more recently, other LLVM compatible languages such as Rust) using a compiler with the eBPF target. In early versions of the eBPF VM in the Linux kernel, this bytecode was interpreted by the in-kernel VM. As the eBPF subsystem was developed, interpretation-based virtualisation has been replaced by just-in-time (JIT) compilation. This means that the eBPF code is converted into the native machine code just once - at the time when the bytecode is loaded into the kernel. The reason for introducing this change is two-fold. Firstly, it improves performance of the VM which is especially important when the eBPF code is loaded on a critical path in the kernel code (e.g. hooked into a frequently used system call). Secondly, this helps to avoid Spectre-like [29] sidechannel vulnerabilities in the eBPF interpreter [28].

Memory Specifications of the VM The eBPF VM has a **fixed-size** stack of 512 B and its specification does not include a dynamically allocated heap. The implications of this design choice are particularly important when running eBPF VMs on memory-constrained embedded devices as they limit the RAM requirements of the solution.

eBPF Registers The specification of the virtual machine includes 10 general purpose registers (0 - 9) with the additional 11th register (10) which stores the stack frame pointer (in read-only mode). The context of the eBPF program is loaded into register 1 before its execution, whereas register 0 is where the return value of the program is stored.

eBPF Instructions The Listing below illustrates the definition of the struct representing a single instruction in the Linux kernel (`include/uapi/linux/bpf.h` header file).

Listing 2.1: eBPF Instruction struct definition in `linux/bpf.h` header file. Taken from: [28]

```
struct bpf_insn {
  __u8 code;
  __u8 dst_reg:4;
  __u8 src_reg:4;
  __s16 off;
  __s32 imm;
};
```

The instruction consists of the `code` field, which defines the operation that is being preformed by that instruction (e.g. load / store or an arithmetic operation). The structure also contains the source and destination registers and signed offset and signed immediate value fields which are used depending on the instruction type (opcode). Note that the instruction struct spans over 64 bits with the maximum size of the immediate operand being 32 bits. In case when we need to store a 64 bit value in a given register a *wide instruction encoding* is used. In that case the instruction spans over 16 bytes (128 b).
An important characteristic of the eBPF ISA is that by design its expressiveness is limited. This is done to make it easier to verify that a given eBPF program can safely be executed inside of the kernel (see 2.3.2). In particular, eBPF does not allow for indirect branch or jump instructions. Every jump instruction has a specified offset relative to the program counter which is known before runtime. This limitation makes eBPF particularly suitable for ensuring memory isolation between the program running in the VM and the rest of the system (see 3.1.2)

Instruction Size in the Context of Microcontrollers
Because of memory constraints, it is important to consider the instruction size when working with embedded devices. Previous work [4] - CHERIoT, proposed an adaptation of the CHERI ISA for IoT devices. CHERI stands for Capability Hardware Enhanced RISC Instructions and is a capability instruction set architecture which augments memory pointers with memory protection metadata such as allowed bounds or read, write or execute permissions. In case of CHERIoT, the authors proposed a special compressed way of encoding capabilities to ensure that each capability

is 32 bits long. This effort was motivated by the memory constraints as the previous work on CHERI for embedded devices proposed an encoding which lead to an average of 12.5% memory fragmentation [4]. This related work indicates that 16 byte wide eBPF instructions could introduce a problem on memory constrained devices. This issue is mitigated by the fact that the size of the code intended to be loaded into VMs is comparatively small. Experiment conducted in the Femto-Containers [11] paper indicated that an example function implementing the Fletcher32 [30] checksumming algorithm logic occupied 456 bytes of memory compared to the 74 bytes for the native C implementation [11]. This might seem like a substantial overhead, however when contrasted with alternative VM implementations (WASM3 - 322B or MicroPython - 497B) the discrepancy is not too severe.
The contribution of this work introduces a JIT compilation step which transpiles eBPF bytecode into native machine code achieving up to 50% program size reduction. This is possible because of the architecture used by the target microcontrollers considered in this project: ARMv7-eM. This architecture supports both 16-bit and 32-bit ARM Thumb instruction encodings which allows for expressing the same program with instructions of smaller size.

eBPF Verification and Loading Architecture In case of the Linux kernel, the eBPF bytecode can be loaded into a specified hook point using the `bpf()` system call. Available hooks include system calls, function entry / exit point or kernel tracepoints [1]. The figure below shows a diagram of eBPF code attached to a syscall hook point.

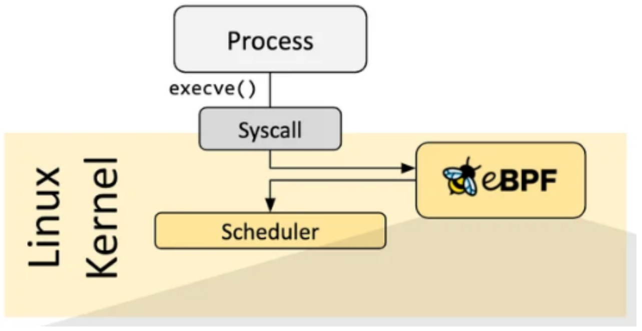


Figure 2.1: eBPF code attached to a system call hook point. Taken from [1].

When an eBPF program is loaded into the kernel, the program bytecode passes through the verification stage using the in-kernel eBPF verifier. The verifier checks if the user trying to load the code has sufficient permissions to do so, then it ensures that the bytecode would not harm the system (i.e. it does not contain invalid instructions or branch instructions jumping outside of the sandboxed environment). It also verifies that the program runs to completion. Unfortunately, the verification process is computationally expensive and previous work [13] has shown that running the verifier on embedded devices is up to 70 times slower compared to a server machine. Because of this, the authors proposed a decoupled architecture where the verification is performed on a server running a VM which emulates the embedded device and then the verified and JIT-compiled bytecode is sent to the device and executed directly provided that the cryptographic signatures are valid. Furthermore, another study motivating the PREVAIL verifier [31] has shown that the verifier does not scale well to programs with a large number of control flow graph paths. Referring this limitation to the case of microcontrollers, poor performance scaling with the number of CFG paths is not necessarily an issue as the software functions that we aim to deploy are relatively small. However, because of the Linux kernel GPL2 license, its verifier cannot be used outside of the kernel. Because of this licensing issue, the existing eBPF userspace VMs (e.g. `rbpf` [12]) rely on simplified implementations of the verifier which do not perform the complicated control flow graph analysis.

eBPF JIT Compilation The next step in the eBPF program life cycle is the JIT compilation. This involves converting the bytecode into native instructions running on the target machine. This improves performance and eliminates the risk of sidechannel vulnerabilities caused by the interpreter processing instructions one-by-one [28].

An important implementation detail of the Linux kernel eBPF verifier is that the JIT compilation step is tightly coupled with the verification step [13]. This is because the verification requires the CFG analysis, which is also used by the JIT compiler front-end. Furthermore, some of the verification checks are performed in the JIT compiler source code. If the proposed decoupled architecture was to be implemented, both verification and JIT compilation would have to take place on the server - outside of the microcontroller running the bytecode.

However, this coupling only applies to embedded devices capable of running the full Linux kernel (e.g. RaspberryPi) and using the Linux kernel eBPF implementation. In case of the userspace VMs with simplified verifiers (e.g. [32] or [12]), there is no strong coupling between the already simple verifier and the JIT compilation steps and so the two parts can be performed separately.

A hybrid approach can be considered, where the verification happens on the server VM of the microcontroller whereas the JIT compilation takes place on the actual microcontroller. This assumes no coupling between the two steps (e.g. in case of a simple verifier). Moreover, if the verifier is simple, the JIT compilation is likely going to be the more expensive step and so decoupling just the verifier could lead to only negligible improvements. These trade-offs are discussed in detail in section 3.4.

Accessing the Outside World: Maps and Helper Calls In order to share data computed or collected during the execution, eBPF programs use eBPF maps. This construct allows for storing and retrieving data persisted in a form of different usecase-specific data structures. The supported data structures include arrays, hash tables, ring buffers and other more application specific constructs. The key capability of eBPF maps is that they can also be accessed by the programs running in the user space by issuing a system call. Because of this feature, one can deploy a set of eBPF programs gathering some information in various points inside of the kernel and have all of them store their collected data in a single eBPF map from where a userspace program can retrieve it. Some of the existing userspace VM implementations (e.g. `rbpf` [12]) are lacking support for maps.

Helper calls are a means of ensuring that eBPF programs running in the VM can access the kernel functions via a consistent version-independent API. If eBPF programs were allowed to call into the kernel functions directly, it would introduce a dependency between the eBPF code and a particular version of the kernel, which in turn worsens compatibility. In the context of embedded devices, helper calls can be used to provide access to a middleware layer exposing a subset of the embedded OS functionality to the code running in the VM. Examples include accessing GPIO pins or sensor readings.

eBPF Composability: Function and Tail Calls Function calls allow for defining functions within a single eBPF program. If those functions are declared as `static`, the compiler emits the assembly which calls the functions by making a jump relative to the current program counter. We refer to these function calls as PC-relative calls. It is important to note that the existing userspace implementations of eBPF VMs (`rbpf`, Femto-Containers) do not support PC-relative function calls. Our implementation of µBPF VM adds that functionality.

The tail calls are a mechanism allowing for executing another eBPF program from the currently executing one. This replaces the current execution context and is similar to the `execve()` system call. Similarly to the maps in the previous section, some existing userspace eBPF VMs (e.g. `rbpf` [12], rBPF [8]) are lacking support for tail calls.

2.3.3 eBPF for Isolation

eBPF allows for implementing isolation at various levels within the Linux kernel. By running eBPF programs in a sandboxed environment, the kernel ensures that these programs can perform their tasks without compromising system security. Examples of applications include isolating network traffic, containerized applications, or system resources. Related work [11] - Femto-Containers VM uses eBPF in a similar way by allowing to host multiple software functions on a single microcontroller while ensuring proper isolation, secure deployment and hardware abstraction for the hosted functions.

2.3.4 eBPF in the Context of Microcontrollers

Given the eBPF specification in Section 2.3.2, it can be used as a containerisation technology for small software functions. The fixed stack and no heap support mean that it can even be used in microcontrollers where the memory is heavily constrained. By contrast, one of the alternatives - WASM3 virtual machine requires at least 64 KiB of RAM as this is the smallest memory page increment in its specification [8]. Furthermore, the VM implementation can be relatively simple (e.g. rBPF [11] approximately 500 lines of code) which results in the ROM memory requirements of the implementation to being much smaller compared to other VMs based on script interpretation (RIOTjs / Micropython - approx. 100 KiB, rBPF - approx. 5 KiB [11]).
When considering memory requirements it is important to put them into the perspective of the targeted hardware. Within the range of microcontroller boards supported by RIOT, they memory budgets range from 2 KiB RAM / 32 KiB ROM (Arduino Uno) to 256 KiB RAM / 2 MiB ROM (STM32F4) [33]. This means that by opting for a virtualisation solution which requires more RAM, compatibility of the solution can be reduced.
In case of the VM execution overhead, eBPF-based virtual machines run on average 2 times slower than WASM3 alternatives [8]. However, if we compare that to the VMs based on script interpretation (RIOTjs / Micropython), they are an order of magnitude slower than both eBPF and WASM3 [11]. The execution overhead is an important factor to consider when working under real-time requirements imposed by some of the RTOS use cases. Because of this overhead, the existing rBPF [8] and Femto-Containers [11] implementations do not support the soft real-time guarantees provided by RIOT for the code running in the VM.
Verification architecture for eBPF can be used to ensure that the code loaded into the VM is safe and will not harm the operation of the kernel. In case of microcontrollers, this feature can be used to ensure that multiple software functions originating from different mutually distrusting vendors can be deployed on the same device securely. This verification step can also be used for the worst-case execution time (WCET) analysis to prevent indefinitely-running code from being loaded into the VM possibly draining the battery of the embedded device or slowing down the operation of the whole system, causing completion of the other tasks running on the device to be delayed. Existing solutions such as the `rbpf` fork used by Solana Blockchain [34] implement a concept of an instruction meter which allows for controlling the compute budget of the programs.

Chapter 3

Related Work

In this chapter, we:

- analyze and evaluate **existing virtualization solutions** bringing container-like deployments into the low-end microcontrollers
- compare existing **eBPF userspace VMs and verifiers** and assess their compatibility with embedded devices.
- introduce the idea of a **decoupled eBPF verification** architecture and analyze ways of extending existing work to implement it.
- describe the use of **memory check caching** [35] to speed up VM interpreter execution.

3.1 Virtualization Solutions for Embedded Devices

This section describes and evaluates the existing solutions which implement lightweight container-like virtualization solutions for embedded devices. The first subsection will introduce the RIOTjs containers - foundation for the later work on rBPF and Femto-Containers. Further we'll look at rBPF, an adaptation of the eBPF VM for RIOT OS. This contribution has reduced the memory footprint required for hosting the virtual machine as an eBPF VM is much simpler than a JavaScript interpreter. The final section will present Femto-Containers - a container-like runtime for low-end IoT devices allowing for hosting multiple isolated software functions on a single embedded device.

3.1.1 JavaScript Containers

In some cases, IoT devices need to contain logic that is not known before they get deployed. This could be because a part of the logic (e.g. pre-processing logic or some sensitive data) needs to be transferred after deployment for performance or privacy reasons. Another possible scenario is adjusting certain parameters of the logic which can only be determined after deployment (e.g. sensitivity of a sensor, frequency of taking measurements). In both of those cases a need for a convenient and secure way of performing firmware updates emerges. This motivates implementing lightweight containers on embedded devices as they allow for changing the logic running in the VMs on the fly.
The authors of [7] propose over-the-air scripting as a possible solution for this problem. In their design deployed devices are running VMs capable of executing the scripting logic that gets loaded

over the network. Their implementation involved using a lightweight JavaScript interpreter - JerryScript [6] as a VM with a middleware layer providing system call bindings for the RIOT OS. This VM was then used to execute scripts that were loaded over the network.

Simulated Use Case The experimental setup in the article was used to simulate a network of sensors responsible for triggering an alarm in a surveillance system. The key motivation for using containerised runtimes in this case is that the logic controlling sensitivity of the sensors triggering an alarm is not predetermined. It depends on where the device was installed and needs to be tuned individually for each device after deployment.

System Design The proposed solution consists of two main components: host operating system (RIOT) and the script interpreter used as a VM environment (JerryScript).
The next step is to allow the scripting code to interact with the underlying OS by providing bindings for the OS functions to the scripting engine. By implementing this middleware, script code executed in the sandboxed environment is able to use the API provided by the OS to e.g. interact with the sensors connected to the device or send network requests.
Once the middleware connects the OS and the VM, the next step is to host a server on the IoT device which allows for loading the scripting code over the network. In in the paper, the embedded devices are running a CoAP server for this purpose. With the server in place, the entire architecture provides an equivalent of a runtime software container [7]. This is because the interpreter engine isolates the sandboxed code from the OS.
The next part of the system is providing a secure way of loading the script code into the container-like environment. This aspect consists of two parts, firstly the communication channel used to transport the script code to the device needs to be secured. The solution proposed in the paper used transport layer security and DTLS [7]. Secondly, the authenticity of the message payload needs to be confirmed using a cryptographic mechanism. The solution achieves this by including a cryptographic hash and a signature of the script code. Further work in this area [11] uses a more sophisticated mechanism which is compliant with SUIT [15] specification.
Figure 3.1 below illustrates the proposed architecture:

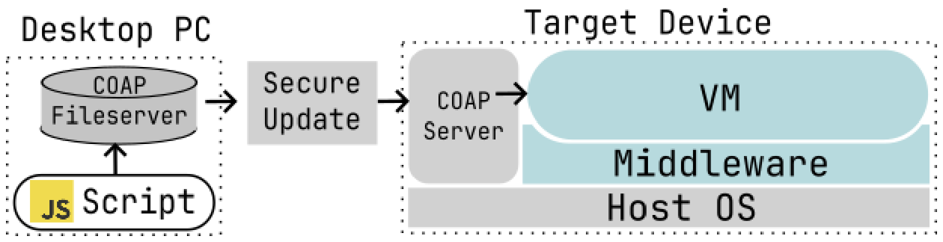


Figure 3.1: Architecture proposed in [7] for over-the-air scripting.

Evaluation Criteria
The authors of the paper consider the **RAM and ROM** requirements of their system. Those requirements are not only stated in absolute terms but also compared to the usual RAM and ROM sizes of an off-the-shelf microcontroller used for the IoT deployments.
The next aspect that was considered is the **compatibility** of the proposed solutions with the boards supported by the host OS (RIOT). Even though the article has targeted a single board for the experiments, it is claimed that it supports approximately 80% of the boards capable of running RIOT operating system.

Another comparison point is the **native C implementation**. Interestingly, the comparison takes into account the key requirement: ability to update the firmware over-the-network. Because of this, RAM and ROM requirements of the C implementation include the full firmware upgrade mechanism that would have to be included to allow for the same functionality. In such a case, the ROM requirements of the C implementation increase, as the flash memory needs to be large enough to fit two system images at the time of upgrade (the currently running one and the new one that is being loaded) plus the bootloader.
The final evaluated aspect of the solution is the number of bytes that need to be transferred over-the-air to perform the update. The solution in [7] naturally exhibits much smaller update sizes compared to the full firmware update (1 KiB compared to 60 KiB for the full system image). This discrepancy can be mitigated by performing partial firmware updates as described in [21, 22].
The next two sections introduce and analyze the related work that is a logical extension of the idea presented in JavaScript containers.

3.1.2 rBPF VM in RIOT

This section investigates the case of rBPF VM [8] implemented in RIOT operating system. Compared to the standard eBPF in the Linux kernel contains bindings for the host OS (RIOT) and features a simplified verifier. It also introduces special instructions for accessing `.data` and `.rodata` sections of the program without the need for relocation procedures. Prior studies [36] and [37] indicated that malicious actors could cause wireless-enabled IoT devices to behave in an unintended way causing severe problems such as power grid disruptions or sensitive data leakage. Given these security vulnerabilities, rBPF aims to provide software-level isolation mechanism for embedded devices.

Applications of rBPF The first use case is to isolate business logic and allow for updating it on demand. This is similar to the problem statement of prior work [7] where the main focus of the proposed JavaScript containers was to allow for performing updates of embedded devices by sending script logic over the network. The second application of rBPF is to allow for instrumenting the behaviour of a deployed OS with monitoring/debug code. This is more akin to the use of regular eBPF in the Linux kernel and thus exhibits stricter timing requirements (as the overhead of adding instrumentation to the kernel needs to be minimised). Both of these use cases involve sending updates over-the-air.

Alternative approaches Solutions alternative to rBPF include modifying the hardware architecture of the microcontrollers to allow for the isolation properties explained above. An example of such system is TrustZone on Arm Cortex-M architectures [5]. The problem with hardware-based approaches is that they depend on specific architectural facilities being present on the embedded devices thus making the solution less general. An example of such solution is Tock OS [38] which is an embedded microkernel written in Rust. The limitation of this operating system is its compatibility. Its hardware requirements include a memory protection unit (MPU) and so the list of supported boards is much smaller compared to alternatives such as RIOT.

Isolation Guarantees The most significant improvement provided by rBPF when compared to the previous work [7] are its isolation guarantees. Firstly, host OS address space is isolated from the rBPF sandbox. This is achieved by imposing access policy rules. Each memory access that occurs in the VM is checked against those policies. However, this approach introduces an execution

time overhead and could be improved by utilising hardware memory protection e.g. [5].
Secondly, security measures on the executed code are in place to ensure that the VM does not start executing code that is outside of the supplied executable. This is important in preventing side channel vulnerabilities, as a malicious actor could try to get the VM to execute a gadget to perform a Spectre-like [29] sidechannel attack. This is achieved by guarding the branch and jump instructions. As stated in Section 2.3.2 eBPF ISA does not allow indirect jumps and the program counter register is not writable. Therefore it is sufficient to perform bounds checks on the direct jumps and branches which can be done during the pre-flight verification step.

Experimental Setup and Evaluation Metrics Similarly to [7], the success criteria considered for rBPF include minimising memory requirements, execution overhead and the size of the payload that needs to be transferred over-the-air to perform an update. An additional criterion is that the solution should not depend on hardware specific memory protection mechanisms.
The authors benchmark rBPF with respect to ROM and RAM requirements, execution overhead and code size requirements by running two example workflows: Fletcher32 checksumming algorithm [30] and a CoAP response formatting application designed to mimic the experimental setup considered in [7]. The performance or rBPF across the aforementioned metrics is compared against an alternative solution based on WASM using the WASM3 interpreter [39] and the baseline of the native C code. The conducted experiments indicate that the execution time of the rBPF VM is approximately two times longer than compared to WASM3. However, this difference is not significant given that most workflows that are intended to be deployed in this scenario are not computationally intensive and the execution time overhead is small compared to network latencies.
Furthermore, the code size of the application script in rBPF is approximately 50% larger than in case of WASM. This is because eBPF ISA specifies fixed length instructions (see Section 2.3.2). The authors acknowledge this limitation and suggest a possible way of improving this code size requirement by using a compression library such as HeatShrink [40].

Limitations of rBPF Among the limitations of rBPF acknowledged by the authors, two of them stand out as they can be related to the other parts of the literature review.
Firstly, the execution time overhead of rBPF is highlighted. The authors suggest that it can be reduced by using a hardware memory protection solution which would eliminate the need for inspecting the access policies on each memory access. Recent work [35] proposes a caching mechanism to improve the performance overhead of performing memory access checks. Additionally, an ahead-of-time or JIT compilation step could be used to improve performance substantially at a cost of adding the overhead of JIT compilation. This observation can be referred to the recent work on the decoupled verification architecture for eBPF [13] which proposes performing verification outside of the embedded device and then sending the natively compiled bytecode for direct execution on the microcontoller.
Secondly, the authors acknowledge that the sandboxing guarantees provided by rBPF could be improved as it does not perform worst-case execution time analysis. This allows malicious or malfunctioning programs to run indefinitely on the device. The authors suggest imposing an upper bound on the execution time of a single invocation of the VM and terminating the execution if it is exceed. This suggestion can be referred to [17] where the eBPF bytecode is instrumented with control code responsible for pre-empting executing when the execution time quota is exceeded. The issue of unbounded worst-case execution time was addressed in the later work on Femto-Containers [11] by imposing a bound on the total number of instructions in the program and the number of branches taken during execution (checked at runtime of the VM).

We can conclude that rBPF has addressed the main limitations of the prior work on JavaScript containers [7] by reducing both ROM and RAM requirements and providing stronger memory protection guarantees. It has therefore established a foundation for Femto-Containers - a container-like solution for embedded systems that will be discussed in the next section.

3.1.3 Femto-Containers

The key motivation for Femto-Containers compared to prior work on rBPF is to leverage the sandboxing guarantees provided by rBPF to provide a secure deployment workflow of small software functions for low-end IoT devices. The aim was to implement a platform similar to Docker allowing for isolated code execution deployed by multiple tenants on a single embedded device.

Use Cases
The first two use case scenarios the same as in the case of prior work on rBPF [8] (see Section 3.1.2). The third use case addresses the key limitation raised in the JavaScript Containers paper [7] - being able to host multiple containers on a single microcontroller. It involves deploying, executing and managing software functions deployed by a number of different tenants. This use case is similar to the Function-as-a-Service (FaaS) [41] deployment model. The proposed architecture aims to allow multiple (potentially mutually-distrusting) tenants to deploy their eBPF code (software functions) on a single microcontroller which makes ensuring proper security much more difficult.

System Design and Programming Model The system design of Femto-Containers is similar to the way eBPF VM is embedded inside of the Linux kernel.

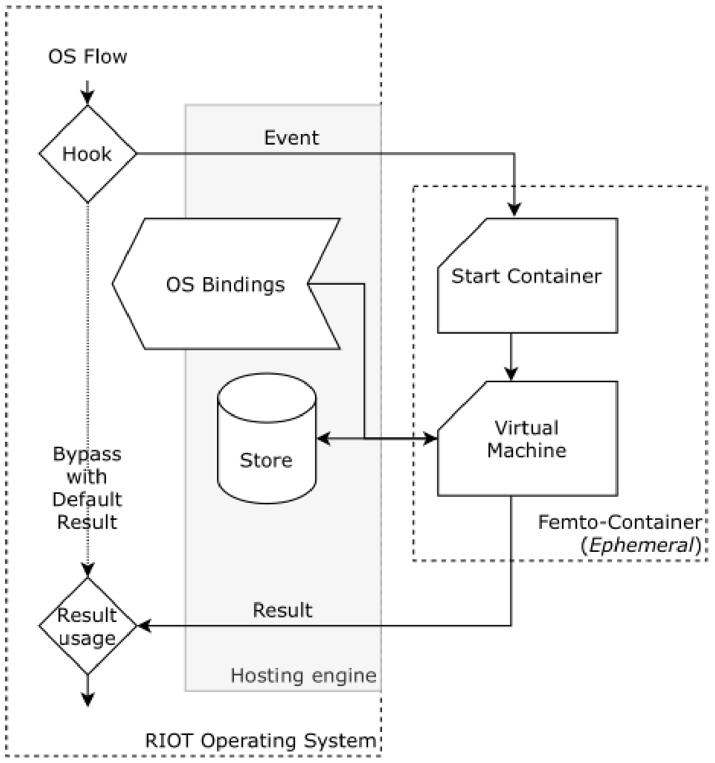


Figure 3.2: Femto-Containers integrated in RIOT, taken from [11].

The containers can be loaded into hook points scattered throughout the RIOT OS code. Then,

once the code is in place an event can initiate execution of the loaded bytecode. Examples of such events can include receiving a network packet, reading a sensor data or when a scheduling event takes place in the OS (e.g. the scheduler performs a context switch from one thread to another). Once the container starts executing, it has access to the middleware layer which is represented by the 'OS bindings' in the diagram 3.2 above.
For example, those bindings in RIOT allow for accessing to the generic sensor/actuator interface SAUL. This allows for e.g. reading values currently reported by a given sensor by making a helper call to the function `bpf_saul_reg_read`.
A running container has access to the key-value store which can be used for sharing data between containers and storing persistent state. This is an equivalent of eBPF maps available in the Linux kernel. The difference is that the key-value store in RIOT is much simpler and does not allow for creating new storage locations by means of a system call (as it is the case for eBPF maps). This functionality in Femto-Containers consists of two stores, one local to the VM running the current container and the other which is global and can be accessed by all deployed containers.
The programming model of Femto-Containers is event-driven, the code attached to a given hook gets run when an appropriate event is registered. Because of this, it is important to consider the execution time overhead incurred in the OS by adding these hook points.

Considered Attack Vector
The paper takes into account two different classes of possible attackers. The authors differentiate between malicious tenants who could deploy software functions aimed at compromising the device and malicious clients who could interact with the already-deployed eBPF software functions by means of e.g. their exposed CoAP endpoints to make them behave in an unintended way and thus harm the system. Femto-Containers were designed to prevent man-in-the-middle attacks aimed at modifying the application bytecode before it gets loaded into a sandboxed environment by using a secure, SUIT-compliant update mechanism. The proposed solution also ensures proper isolation of sandboxed environments so that malicious tenants cannot escape out of those containers into containers owned by other tenants or the rest of the host operating system.

Experimental Setup and Evaluation The Femto-Containers VM was evaluated using a similar methodology as in the case of rBPF (3.1.2). The experiments involved benchmarking the implementation using the Fletcher32 checksum algorithm and measuring the RAM and ROM overhead while comparing that to alternative VM solutions (WASM3 and MicroPython). The presented results are consistent with findings of the original rBPF study [8]. The Femto-Containers solution incurs a much smaller RAM and ROM overhead compared to alternatives, however its execution time overhead is two times larger than in case of WASM.
The experimental setup included measuring the cold start overhead. This represents the time it takes for the benchmarked runtime to load the code into the sandboxed environment. Even though previous results indicate that the execution time overhead of using WASM is smaller than in case of rBPF extended for Femto-Containers, the measurements have shown that the cold start overhead of the WASM3 interpreter is substantially larger than it is the case for the eBPF based VM (17 096 µs vs 1 µs [11]). The other script-based VM alternatives also come with a sizeable cold start overhead (MicroPython - 21 907 µs, RIOTjs - 5589 µs).
The key insight from experiments presented in the paper is that the authors consider the absolute numbers that were measured in the broader context of low-end IoT devices. For instance, the RAM and ROM requirements are evaluated against a representative embedded device with 64 KiB in RAM and 256 KiB in ROM. Furthermore, the ROM overhead of including the VM in the firmware

is considered as a percentage increase compared to the image size of the host OS.

Formal Verification One of the most impactful contributions presented in the paper was the formal verification of the Femto-Containers solution. Given small footprint of the VM source code (approximately 500 lines of code) formal verification was feasible. The authors used formal verification tools to generate a verified implementation of the Femto-Containers VM and verifier (called CertFC). Interestingly the RAM requirement of that generated code is two times smaller comparing to the Femto-Containers and rBPF implementations. However, the runtime overhead of the formally verified implementation is slightly larger than that of non-verified implementations. It is very surprising that the code size has decreased after generating a verified implementation. A possible explanation could be that the original code was compiled with higher level of optimisation, which could include loop unrolling or duplicating the code for supporting SIMD vector instructions.

Worst Case Execution Time The key improvement of the VM interpreter implementation used by Femto-Containers over rBPF is the ability to control the worst case execution time. The interpreter imposes a bound on the total execution time of the programs running in the VM which is checked at runtime. This is achieved by limiting the total number of instructions that a given script can include, together with keeping track of the number of branch instructions taken during the execution of the program. The total number of instructions executed by a single execution of the VM is bounded by the product of the number of branch instructions that this program is allowed to take and the total number of instructions in the program. This can be interpreted as the theoretical boundary case where the entire program is a single loop and all instructions get executed at each iteration.

Limitations of Femto-Containers Similarly to the case of rBPF [8] authors point out that using a VM comes with an execution time overhead. This could be improved by using JIT compilation or ahead-of-time compilation and verification in a decoupled architecture proposed in [13]. The problem with implementing JIT compilation is that currently the worst-case execution time is bounded in Femto-Containers VM by keeping track of taken branches at runtime (as the VM is interpreting the bytecode). This means that if we wanted to preserve that functionality for the JIT compiled code, it would have to be instrumented with control logic responsible for keeping track of the number of instructions executed (similar to [17]). This approach was used in a fork of rBPF [12] which was used by Solana in its blockchain implementation. Their approach involved introducing the '**instruction meter**' [34]. It aims at bounding the total number of instructions by instrumenting each branch instruction with code responsible for counting the total number of instructions executed if the branch is taken. Additionally, for long linear sequences of instructions, the code is interspersed with checks of the instruction meter to ensure that the compute budget is not exceeded.

A further limitation which we identified is the deployment workflow of using Femto-Containers. The SUIT update specification improves security, however it requires a number of manual steps which slow down the iteration cycle when performing frequent updates to the eBPF bytecode running on the device. We address this limitation by implementing a deployment framework allowing to send programs to the target device with a single command.

Additionally, the current implementation of Femto-Containers is tied to RIOT. A possible improvement could be to redesign the middleware interface which connects Femto-Containers VM to the host OS so that it is possible to port the implementation onto a different system such as Tock OS [38] or ESP-IDF [42].

3.2 Userspace eBPF VM Landscape

Since the original eBPF VM and verifier are implemented in the Linux kernel, which is licensed under GPL2, they cannot be used outside of the kernel source code. Because of this, a number of alternative userspace VMs with simplified verifiers were proposed. This section provides an overview of the available solutions, their corresponding limitations and evaluates their suitability for this project.

uBPF [32] is the first user-space implementation of an eBPF VM. It is written in C and provides an assembler, disassembler and interpreter for eBPF ISA. Importantly, it supports JIT compilation on both x86 and Arm64 targets and so can be a useful reference point when extending prior work on eBPF VMs in RIOT ([8], [11]). It is published under Apache license and so it can be used in a wider range of projects. Its eBPF interpreter is implemented using a simple switch statement which can be contrasted with the jumptable approach used by rBPF [8].

`bpftime` [43] is a high-performance eBPF runtime designed to operate in userspace . It offers fast uprobe and syscall hook capabilities. It is currently being actively developed and provides the widest range of available features compared to alternatives (uBPF, `rbpf`).

However, the main focus of `bpftime` is to act as a drop-in replacement of the kernel-space eBPF by providing support for existing tools such as `bpftrace` [44] or `libbpf` [45]. In the context of this project, we need an eBPF VM which is as minimal as possible while being portable enough to be integrated into a chosen RTOS. Because of this, despite its wide range of features, `bpftime` was not chosen as the target VM implementation for the project.

`rbpf` [12] is the user-space VM implementation that we chose as a foundation for µBPF. It is not to be confused with the rBPF VM implementation for RIOT [8]. Here 'r' stands for the Rust programming language, in which this VM is implemented. Compared to uBPF, it does not provide an Arm64 JIT, however it's documentation is more thorough and because of Rust's newer build system this was the VM of choice for the early prototyping stage of this project. A fork of this VM is used by the Solana blockchain for its smart contract executing virtual machine. This fork also supports Arm64 JIT compilation and was a useful reference point for our JIT implementation.

Limitations `rbpf` does not support eBPF maps or PC-relative function calls and its set of supported helper functions is limited.

Another limitation is that its verifier is very minimal, it does not perform any checks on the control flow of the program. It also only checks the total number of instructions and does not limit the number of branches taken. This issue could be addressed by implementing a similar approach to Femto-Containers where the number of taken branches is monitored during the VM runtime and the execution is preempted if the limit is reached.

A possible risk of using this implementation is that it is written in Rust, which still has some problems with support on some MCUs. During the early prototyping stages of this project `rbpf` implementation was trivially ported (by disabling the platform-specific JIT) to run on esp32, however this required using a custom Rust toolchain compatible with esp32 and caused some issues with developer workflow such as setting up a Rust LSP.

The contribution of this project involved porting `rbpf` to run on embedded devices without support for the Rust standard library. We then extended the implementation with PC-relative function call support and maps-like functionality through the OS bindings specific to RIOT.

3.3 VM Interpreter Optimization

In order to provide sandboxing guarantees, the existing VM interpreters need to validate every memory access. This means that every time the code running in the VM wants to load from or store to memory, the interpreter inspects the requested address and checks whether this address falls into one of the allowed memory regions. Examples of those regions include the stack of the VM, memory of the context struct provided to the eBPF program or the `.data` and `.rodata` sections in the program binary.
The problem with the current state-of-the-art: Femto-Containers is that the allowed memory regions are maintained as a list. This means that every time the program running in the sandboxed environment accesses memory, the interpreter needs to traverse the list in order to find the memory region that the memory access falls into (or fail if the memory access is invalid). This can become problematic when the number of allowed memory regions grows large.

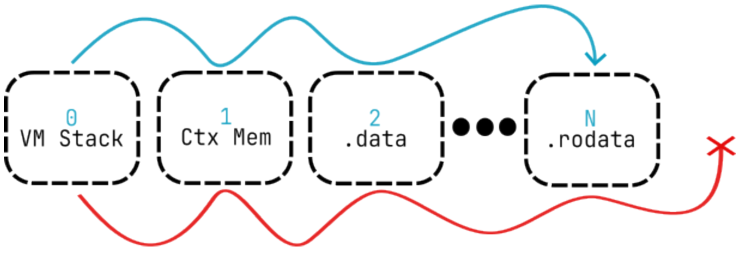


Figure 3.3: Performing memory-access validation by traversing the list of allowed regions.

Related work [35] proposes a caching mechanism which aims to solve this issue. For each instruction in the program, the proposed design maintains a cache of the last memory region successfully accessed by that instruction. If for example a program contains a loop, it is likely that in the next iteration the same instruction is going to try to access the same memory region again, in which case we get a cache hit and only need to check if the memory access indeed falls into the cached memory region. If that's not the case, we need to invalidate the cache and perform a full list traversal again. In this work we implement the caching algorithm proposed in [35] and evaluate its performance on memory-access intensive workloads.

3.4 Decoupled eBPF Verification Architecture

Related work [13] proposes a novel architecture for eBPF on embedded devices which decouples bytecode verification from the execution. The class of devices considered in this work includes microcontrollers and single-board computers powerful enough to run the Linux kernel (e.g. RaspberryPi). This differs from the low-end IoT devices that we target, however the findings and ideas introduced can be translated to that class of devices as well.

Motivation Using eBPF in the case of embedded devices running a regular Linux kernel is challenging because of the limited computational power of those systems. Firstly the toolchain for compiling high level code in to eBPF bytecode (e.g. bcc [46]) tends to be heavy-weight which slows down the compilation on those devices. Prior work [18] proposes a solution for this issue by compiling the eBPF bytecode outside of the target machine. Referring this to our target class of low-end devices unable to run the Linux kernel, this is already done by precompiling eBPF bytecode and sending it over-the-air to the target device ([11] and [8]). Additionally, the second issue is due the overhead incurred by the verification step. Because of hardware constraints, the verification stage takes approximately 70 times longer when performed on an embedded device compared to a regular server [13]. Not only does this affect the load time of the eBPF bytecode, but also it limits the number of programs that can be successfully verified. This is because more complex programs would take unreasonably long to verify which would make loading them on demand not feasible. The paper proposes a solution which decouples the verification step while preserving full compatibility with the Linux kernel verifier and all of its security guarantees.

The Overhead of Verification The cost of verification considered in this paper is substantial because the verifier in the Linux kernel is more sophisticated than the solutions used by the userspace eBPF VMs (e.g. [12]) that we target. This is because the verification consists of two passes through the eBPF bytecode. The first pass involves checking the number of instructions, looking for invalid jumps, loops and unreachable instructions. The second pass however is much more expensive. It involves walking through all possible control flow paths to ensure that none of them contain code that could compromise the safety of the kernel. Authors observe that the overhead of verification grows with the number of paths. Interestingly, because of the large verification overhead, the performance penalty of JIT compilation on the embedded device does not contribute significantly. Experiments in [13] have shown that when considering the full load time of the program, overall more than 90% of runtime was spent on verification. Because of this, the authors suggest that leaving the JIT compiler in place could be a viable solution as it would avoid performing the architecture-specific JIT compilation outside of the target device. However, in the case of the Linux eBPF verifier and JIT, this is not possible as those two components are tightly coupled. For instance, the maximum number of tail calls that can happen during one eBPF VM execution is bounded in the eBPF specification [47], but the check for this is done during the JIT compilation step and not in the verifier.

If we refer this to the prior work in RIOT (rBPF 3.1.2 and Femto-Containers 3.1.3) the situation could be the opposite. Those two designs do not include a JIT compilation step and their verification stages are minimal. If a JIT compilation step were introduced, it would dominate the load time and thus decoupling both the simple verifier and the JIT together could be the best solution.

Decoupled Architecture Design Figure 3.4 shows the proposed design of the decoupled architecture.

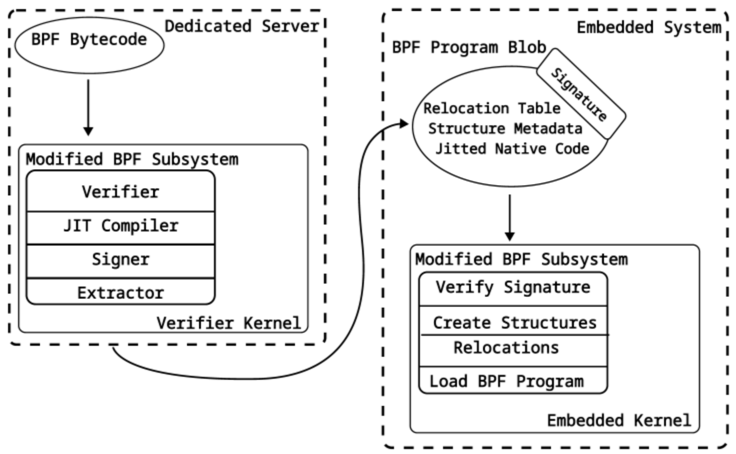
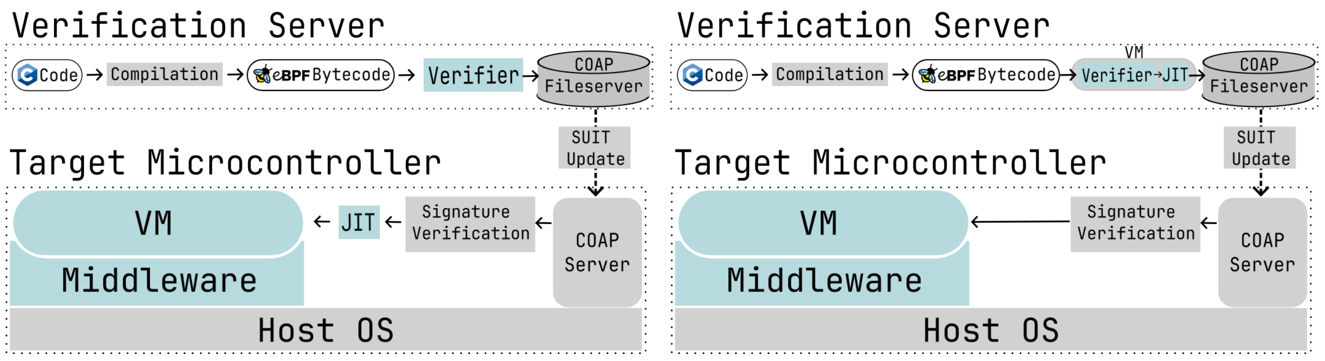


Figure 3.4: Decoupled architecture system design, taken from [13].

The verifier kernel runs in a virtual machine to ensure that the JIT compiled binary matches the target architecture of the embedded device. Note that after the verifier kernel above performs the AOT compilation, it also needs to create a relocation table to allow the embedded kernel to patch the compiled native code with correct addresses of helper functions and program (read-only) data. This metadata is then sent together to the embedded device, the relocations are resolved in the previously-compiled program and its native bytecode can be executed.

Difficulties Firstly, the kernel loading the pre-verified and pre-compiled executable needs a mechanism to trust the verifier kernel that the incoming bytecode can be safely executed. This is because the incoming instructions are represented in the native ISA of the target device, hence they cannot be verified using the eBPF verifier. To achieve this trust, the authors propose an approach where the verifier kernel signs the verified binary.
The second issue is the need for address relocation for eBPF helper calls and maps. In order to prevent Spectre-like side-channel vulnerabilities [29], the Linux kernel supports KASLR which loads the kernel at a random address in memory every time it boots up. Because of this, the helper functions in the verifier kernel will be located at different addresses from the ones on the actual target kernel. In order to solve this problem, the authors propose using a relocation table which provides necessary information for the target kernel to modify the received JIT-compiled binary instructions so that the appropriate helper functions get called.

Decoupled Verification for Low-End IoT Devices We can adapt the design above to our targeted class of low-end IoT devices. Firstly, given that existing solutions ([11] and [8]) do not support JIT compilation, there is no coupling between the JIT and verification that limits the possible design choices. Because of this it is possible to decouple only the verifier and then add JIT compilation step to take place on the microcontroller (Figure 3.5a).
Placing the JIT compilation step on the embedded device eliminates the need for a relocation table. However, it could have an impact on the load time because of the JIT compilation overhead. By contrast, ahead-of-time JIT compilation (Figure 3.5b) means that we are losing the ability to verify the eBPF code directly before it gets executed on the target device. This means that the source of the JIT-compiled binary needs to be fully trusted. To establish trust that the verified code is secure, an update mechanism compliant with SUIT [15] could be used.



(a) Verification decoupled on the server. **(b)** JIT and verification decoupled.

Figure 3.5: Decoupled eBPF verification architecture for low-end microcontrollers.

Chapter 4

Design

4.1 Project Requirements

We design µBPF to provide a platform for deploying compartmentalized applications on embedded devices using eBPF. This consists of server infrastructure that is flashed onto target microcontroller devices and a deployment framework allowing for compiling, securely loading and executing eBPF program logic on the target devices. A high-level overview of the design can be seen in Figure 4.1.

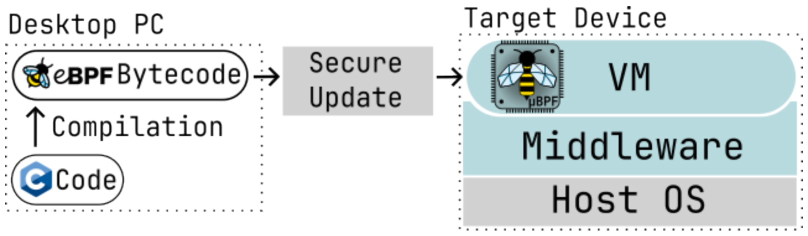


Figure 4.1: µBPF architecture overview.

At a high level, the architecture of µBPF consists of a compilation step, where the source code is compiled into eBPF bytecode. Then follows as secure software update mechanism which transfers the program over the network to the target device. Finally the program arrives at the micro-controller, is verified and executed in a virtual machine. The VM is able to access functionality provided by the underlying operating system through a middleware layer.
We identify the components required to implement a system following this design:

- an implementation of the eBPF VM compatible with the chosen hardware,
- an embedded host operating system,
- an implementation of the middleware layer for the chosen OS,
- target microcontroller hardware,
- a secure over-the-air update mechanism,

The following subsections describe the particular choices we've made for the above components, list available alternatives and provide reasoning behind our decisions.

4.1.1 Selecting the eBPF VM Implementation

The list of required parts of the design specifies components that are not independent. The choice of the host OS influences the range of compatible microcontroller boards. Moreover, the selected VM implementation also needs to be compatible with the other parts of the system.
Given that the main purpose of the system is to execute eBPF programs in a VM, we make a decision on the VM implementation first and then select the rest of the system to be compatible with it. We chose `rbpf` [12] for the VM implementation used by µBPF because of the following advantages:

- there exist **JIT compiler implementations** for `rbpf` ([12, 34]) which were a useful starting point for the ARMv7-eM JIT compiler implemented for µBPF,
- it is implemented in Rust, so a **modern build system** provided by `cargo` improves portability of the solution,

However, one limitation of `rbpf` was that it did not support `no_std` Rust environments. In the context of embedded devices `no_std` refers to building programs without the standard Rust library. This is a common requirement for microcontrollers when running on bare-metal hardware or simple embedded microkernels that do not provide a full implementation of `libc`.
The contribution of this project started with porting `rbpf` to run on `no_std` environments and was successfully merged into the upstream repository. Subsequent changes to the VM that we introduced were specific to the rest of the µBPF system and there is currently no plans to integrate them into the original code base.
As an alternative we considered `bpftime` [43], however its set of provided features is designed for replicating the Linux kernel eBPF in userspace which is not a good fit for our constrained use case.

4.1.2 Choice of the Host OS

Given that the VM implementation of choice is written in Rust, we select RIOT [33] as the host operating system. RIOT is an open-source microkernel-based operating system with good and actively developed support for Rust.
It was also used as the host OS for the related work on Femto-Containers [11] and rBPF [8]. This makes µBPF backwards-compatible with the existing solutions and allows for easier evaluation of the contribution against the state-of-the-art. The middleware layer around RIOT functions was designed to extend the functionality provided by the existing solutions.

4.1.3 Target Hardware

Given the compatibility constraints imposed by the chosen OS and VM implementation, we chose to primarily support the STM32 F4 microcontroller family. Those embedded devices are running ARM Cortex M4 CPUs and offer good performance at an affordable price. Table 4.1 lists the two models that µBPF was tested on.

Board	Clock Frequency	Flash Memory	RAM
STM32F439ZI	180 MHz	2 MiB	256 KiB
STM32F446RET6	180 MHz	512 KiB	128 KiB

Table 4.1: Hardware used for the project.

Given a wide range of boards based on ARM Cortex M4 supported by RIOT [33], any of the alternative devices could be used, provided it has similar RAM and ROM capacity.

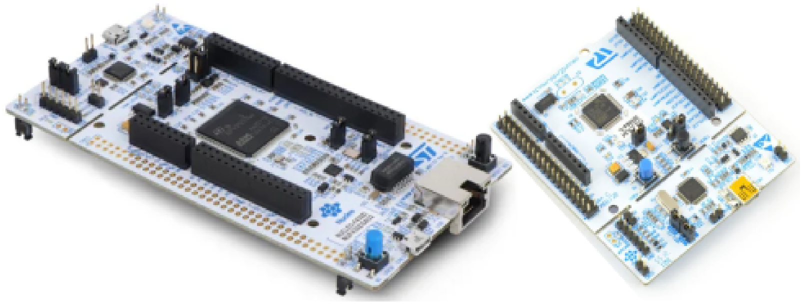


Figure 4.2: STM32 boards used for the project.

4.1.4 Secure Update Mechanism

To ensure that eBPF programs are transmitted to the target devices securely, µBPF uses the SUIT firmware update protocol [15]. This is achieved by using RIOT as the host OS which provides a subsystem allowing for SUIT-compliant file transfer over the network using the CoAP protocol. The SUIT manifest creation and signing is done using the `suit-tools` provided by RIOT.

4.2 System Architecture

µBPF divides the process of deploying eBPF programs into two steps: **deployment stage** and **execution stage**. The first stage involves compiling (4.2.1), verifying (4.2.1) and loading (4.2.1) the program into memory of the target device. Then, in the execution stage clients can send multiple requests to run previously-deployed programs.

4.2.1 Deployment Pipeline

The deployment pipeline of µBPF consists of four steps: compilation, cryptographically signing the update payload, firmware upload, and program verification. Figure 4.3 shows the pipeline, where grey boxes mark previously existing infrastructure, whereas blue boxes mark the contribution of this work. Available program verification points are marked with `V1`, `V2`, `V3`.

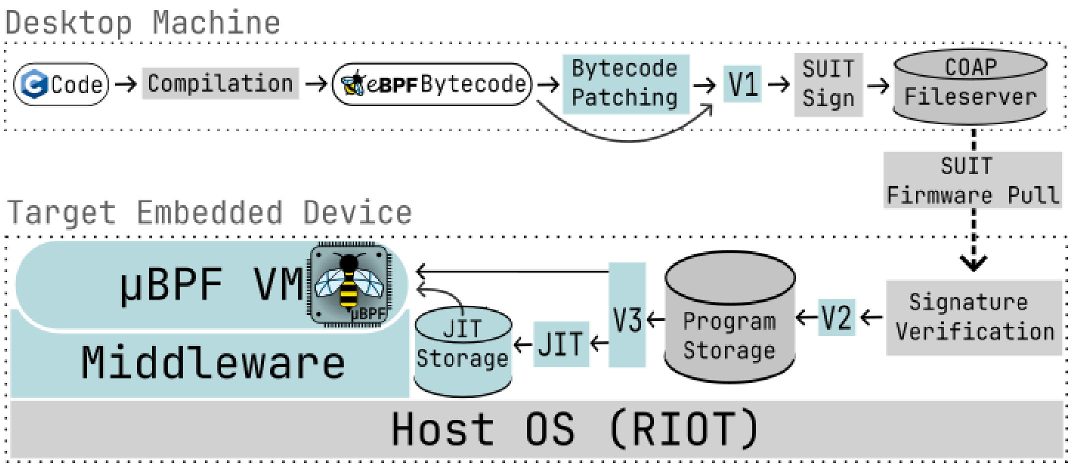


Figure 4.3: µBPF deployment pipeline.

Compilation and Bytecode Patching

At the start of the pipeline, source files in C are compiled into eBPF bytecode. This is done using `clang` to generate LLVM bitcode, and then llc is used with the eBPF target architecture to produce the eBPF bytecode.
After that follows an optional bytecode patching step required for backwards compatibility with Femto-Containers. This step is responsible for replacing the load-double-word (LDDW) instructions accessing program data and read-only data (`.data` and `.rodata` sections) with custom instructions introduced by the rBPF VM. Those instructions replace the actual load address from the original LDDW instruction with an offset from the start of the accessed program section (`.data` or `.rodata`). These instructions are then handled by the VM interpreter accordingly to avoid the need for relocating those memory accesses on the target device before the program is executed. A more in-depth explanation of this process can be found in section 5.1.2.
µBPF supports 4 different eBPF binary formats (see 5.1.2) so the bytecode patching step can be skipped and a raw object file can be fed directly into the next step of the pipeline.

SUIT Payload Signing and Firmware Update

Next step involves sending the program binary to the target device using the SUIT update workflow provided by RIOT (see Figure 4.4).

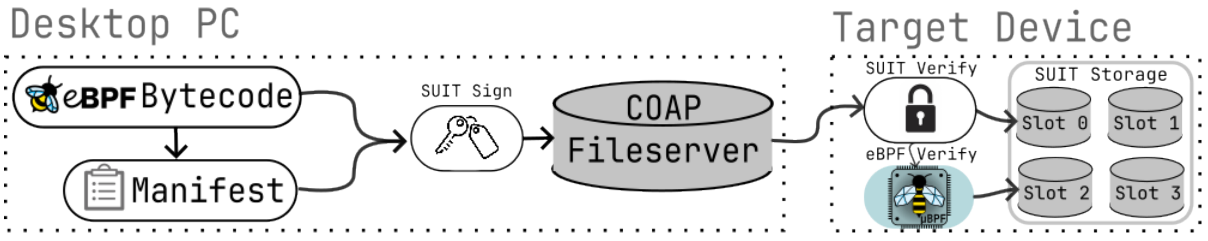


Figure 4.4: SUIT update workflow.

First, the produced binaries are signed using the encryption keys that match the ones present on the running OS image. Then, a manifest file is created and signed. It is then stored together with the program binary in the root directory of the CoAP fileserver [25]. The manifest provides information required by the target device to verify that the loaded program has not been tampered with and that it originates from a trusted source. Then the microcontroller fetches the compiled eBPF bytecode and its manifest file from the CoAP fileserver, verifies the signature and loads the program into one of the RAM storage slots provided by the SUIT subsystem for RIOT. Our deployment framework allows for optional eBPF program verification at this point (`eBPF Verify` in Figure 4.4) using the simple verifier provided by `rbpf` [12]. If verification is used, it allows for preventing malformed programs from being loaded into the SUIT storage slots which avoids wasting memory. The µBPF execution stage then loads programs from SUIT storage and runs them in one of the supported execution modes.

Middleware Layer

While being executed, the eBPF program can access a set of middleware functions to interact with the underlying OS. These functions are equivalent to eBPF helper fuction calls [47] and allow program logic to interact with embedded hardware by e.g. reading data from sensors, controlling actuators or sharing data with other programs by accessing shared global storage (similar to eBPF maps). The set of available helper functions can be extended to match the requirements of a specific application. We design a mechanism of restricting access to helper functions to allow for different privilege levels.

Verification

Malformed programs or a malicious use of helper functions can compromise security of the system, hence the eBPF programs are passed through a verifier at one of the three predefined points in the deployment pipeline (stages marked with `V1`, `V2`, `V3` in Figure 4.3). The verifier implementation is derived from `rbpf` [12]. It checks the validity of instructions and jump offsets and verifies that only calls to allowed helper functions are issued. The VM validates all memory accesses at runtime. µBPF verifier does not perform the traversal of all possible program paths as it is done in the Linux kernel [48].
Related work on decoupling eBPF program verification [13], found that sophisticated verification can be expensive on embedded hardware. Because of this we allow for verifying the programs at different stages of the pipeline depending on the trust level and required performance. For example, if we fully trust the source of the program, the verification can run on the desktop machine (point `V1`) where it has access to much more powerful hardware.
We consider the following threat model:

- **fully trusted source and clients** - verification happens at `V1` outside of the embedded device
- **potentially malicious source** - verification is done at `V2` before it is loaded into the SUIT firmware storage.
- **potentially malicious clients** - after the client sends a request, program is verified at `V3` before execution.

In µBPF the privilege level of a given client is controlled by the sets of helper functions that they are allowed to call. For instance some users might be allowed to call potentially harmful helper functions (e.g. the ones directly accessing the GPIO pins of the device) whereas less trusted clients might only be able to use a restricted subset of the middleware.
In our model a malicious client cannot deploy new programs on the device, however they might try to send a request to execute a program that only privileged users are allowed to execute. We encode the list of helper functions accessible to the client in the request and verify it at the point `V3` to ensure only allowed helpers are accessed by the program.
A program originating from a malicious source could be malformed or contain invalid helper function calls. Because of this it is verified at `V2`, so that malformed programs are rejected before loading and do not waste program storage.

4.2.2 Program Execution Stage

After the deployment stage is complete, clients begin sending requests to start executing the loaded program. Clients can choose between executing the program using the VM interpreter or using the JIT compiler and then executing the emitted native code. After a given program is JIT-compiled, its bytecode is stored in an additional JIT program storage (see 4.3) so that upon receiving a request to run it again, the compilation process does not need to be rerun.
Here we note that when using the JIT compiler additional memory is required as the eBPF bytecode needs to be translated into the native instructions and written into a new memory buffer. However, after this step is complete, the original eBPF program can be discarded allowing to save memory.

Supported Execution Modes

When executing deployed programs, clients can choose one of the following execution modes depending on their use-case:

- **short-lived execution:** short script-like programs allowing to change the state of the device by interacting with the middleware,
- **long-running VM:** programs implemented as infinite event loops allowing to build continuously running applications using eBPF,
- **short execution accessing CoAP packet buffer:** short query-like scripts allowing to inspect the state of the system and write the results back into the response sent to the client.

Short-lived execution mode includes a CoAP server running on the target device, this server accepts requests specifying which program should be loaded from the SUIT storage and executed. Upon receiving a request, an instance of the µBPF VM is initialised directly in the endpoint handler function, without spawning any additional threads. Once the eBPF program is executed successfully, its return value is sent back in the response to the client. This workflow is illustrated on the left in Figure 4.5 below.

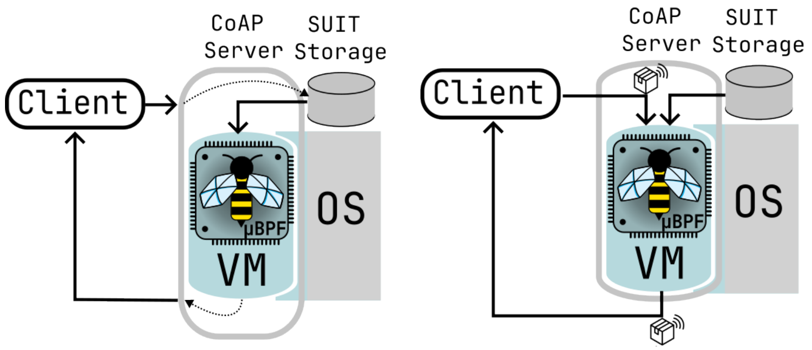


Figure 4.5: Short-lived execution modes.

The intended use-case of this execution mode is to send short snippets of business logic to the target device and alter it's state based on it. One example could involve controlling actuators connected to the microcontroller to release some physical lock or turn on a light bulb by controlling a relay switch via GPIO pins.
One key limitation of this mode is that the executed programs indeed need to be short-lived. If we try to execute a program which takes a couple of seconds to execute or runs indefinitely, we'll run into problems with the client resubmitting requests. This is because the CoAP communication protocol used to send requests to the device is based on UDP. The send-and-forget message transmission semantics mean that the client cannot distinguish between the packet getting lost and the server taking a longer time to process the request. This causes the `aiocoap-client` [25] library to resend the request and causes our system to process two (or even more) requests than it was originally intended. If the required use-case is to spawn long-running programs, the long-running execution model should be used.

Short execution accessing CoAP packet buffer is designed to offer maximum flexibility with respect to response sent back to the clients after program execution. It works by exposing a pointer to the CoAP packet buffer (See Listing 4.1) to the eBPF program running in the VM (See Figure 4.5 on the right).

Listing 4.1: Context struct exposed to the eBPF program.

```
typedef struct {
  __bpf_shared_ptr(void *, pkt);     /**< Pointer to the coap_pkt_t struct */
  __bpf_shared_ptr(uint8_t *, buf); /**< Packet buffer */
  size_t buf_len;                    /**< Packet buffer length */
} bpf_coap_ctx_t;

typedef struct {
    uint32_t hdr_p;        /* ptr to raw packet */
    uint32_t payload_p;    /* ptr to payload    */
    uint32_t token_p;      /* ptr to token      */
    uint16_t payload_len; /* length of payload */
    uint16_t options_len; /* length of options */
} bpf_coap_pkt_t;
```

A pointer to the `bpf_coap_ctx_t` is passed into the main function of eBPF program, and the programmer is responsible for properly writing and formatting the response into the packet buffer. This is done by writing the response bytes at the memory location pointed to by `payload_p`. This response gets sent directly back to the client after the eBPF program execution terminates.
This approach improves flexibility when designing applications using µBPF. Clients can engineer the eBPF programs in a way that produces precisely the right response that they need. For example, if the response required needs to be in JSON format, by modifying the program we can change the fields that are included in the response without the need of updating the software running on the microcontroller. It is sufficient to update the eBPF program, deploy it on the device and the schema of responses can be changed without rebooting the whole system.

Long-running VM execution mode can handle programs that run indefinitely by executing them on separate threads. Upon receiving a request to execute a long running program, the server sends a inter-process-communication (IPC) message to a subsystem of µBPF responsible for managing long-running VMs. This component then spawns the VM on one of the available worker threads. Immediately after that a confirmation response is sent back to the client to indicate that the requested VM was spawned successfully. The design diagram of this workflow can be seen below.

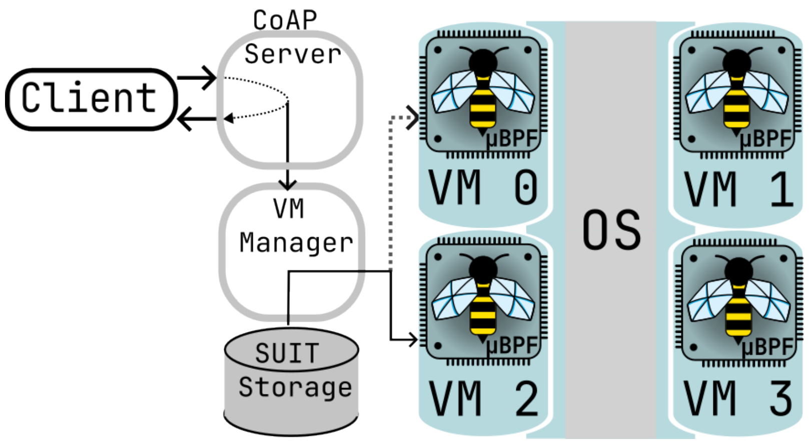


Figure 4.6: Long-running VM execution workflow.

We note that in this mode the client cannot receive any information returned by the executed program (if it terminates). This mode should be used to deploy long-running programs that operate continuously and record their state in the global storage from where the short-lived programs can extract it and send back to the clients. An example use-case for this mode would be to implement an event loop responsible for driving an LCD display connected to the device and changing its contents based on the input registered from the control buttons exposed to the users of the device.

4.2.3 JIT-Compiled Program Storage

Because of hardware constraints, performing JIT compilation on microcontroller devices is an expensive process. Given that the execution model of µBPF allows for running the same program multiple times, it would be costly to perform JIT compilation every time the eBPF program is loaded from the SUIT storage. To mitigate this issue we design a JIT program storage which provides a similar interface to the SUIT storage provided by RIOT. It allows for saving JIT-compiled programs in RAM in a predefined set of slots so that they can be executed multiple times. The compilation and execution workflow using the JIT storage can be seen in Figure 4.7.

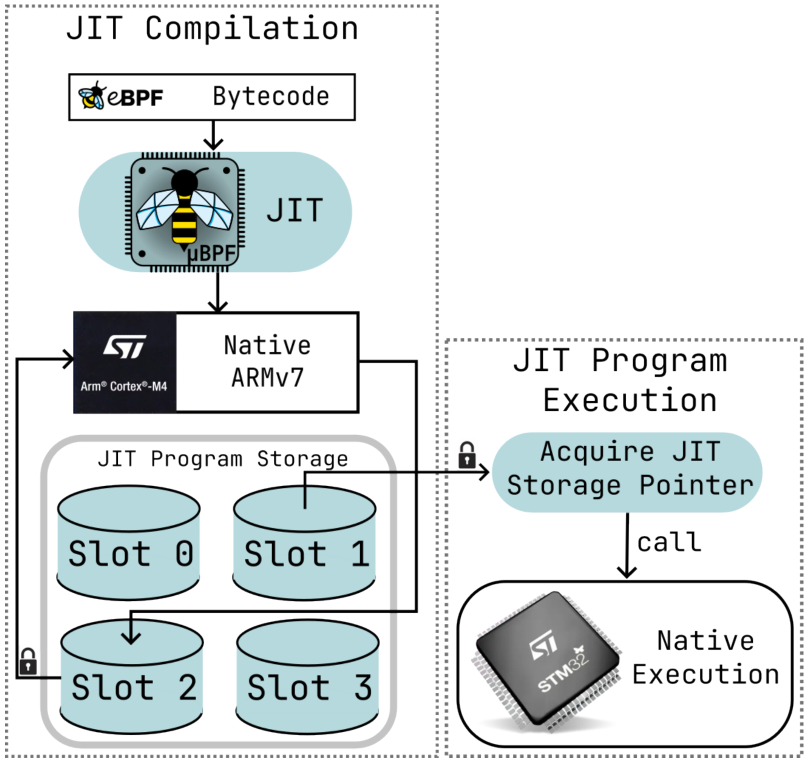


Figure 4.7: JIT compilation and execution using program storage.

One key difference in the design of the JIT program storage is that after compilation, the program cannot cannot be copied over to a different location in memory. This is because the compilation process relies on the address of the memory buffer storing the emitted instructions to correctly calculate offsets to jump to helper functions or access `.data` and `.rodata` sections of the program. Because of this, the JIT program storage is coupled with the JIT compilation process. Every time a given program is compiled, the compiler needs to acquire a free slot in the JIT program storage (protected by a fine-grained locking mechanism) and write the program bytes there. Then, upon execution, the JIT-compiled program is not loaded into a new memory buffer from the storage. Instead the JIT program storage allows for getting a pointer to the program buffer and casting it as a function pointer that can then be called to execute the JIT-compiled program.
We design a mechanism that prevents subsequent requests from overwriting the contents of a JIT-compiled program slot while it is still running.

Chapter 5

Implementation

The implementation of µBPF consists of four main components: the virtual machine, RIOT server infrastructure, deployment framework and the ARMv7-eM JIT compiler. The whole codebase comprises approximately 20586 lines of Rust, 674 lines of C (needed for foreign-function-interface) and 3486 lines of test files in both C and Rust.

5.1 µBPF VM

We implement µBPF virtual machine based on rbpf [12] - a userspace eBPF VM written in the Rust programming language. The base implementation comprises 8385 lines of Rust and we extend it by adding compatibility with `no_std` environments and support for additional eBPF binary formats. Our implementation also supports PC-relative function calls and contains memory access check optimisations described in [35].
The extended implementation is composed of approximately 11137 lines of Rust.

5.1.1 Porting `rbpf` to `no_std` Environments

In the context of Rust libraries `no_std` refers to being able to run programs on targets that do not provide the full Rust standard library. This constraint is quite common on embedded devices as the code might be running on bare metal or on an embedded OS which does not provide the full implementation of `libc`. In such cases, not all functionality provided by Rust's `std` is available. For instance, the `println!` macro might not be there as there is physically no console to print to. Another example could be running on devices where the standard global memory allocator is not available.
In case of this project, RIOT requires that all applications written for it are `no_std` because their OS only provides a custom subset of the standard library available through the `riot_sys` crate. Because of this constraint, we needed to port `rbpf` to be compatible with `no_std`.
This process involved instrumenting the code with directives that conditionally enable parts of the code depending on the target environment. When the standard library is available, the configuration ensures that the original version of `rbpf` is compiled. However, when building the library for `no_std` the imports are adjusted to only depend on modules and data structures available without `std`. Additionally some parts of the codebase such as the x86 JIT compiler are disabled as they depend on functions provided by the standard library (e.g. `posix::memalign`).
Adding `no_std` compatibility is a feature which is commonly supported in most Rust libraries. Because of this, the maintainer of the `rbpf` project was open to merging this feature back to the

upstream repository. We submitted a pull request to add `no_std` support and our contribution was successfully merged into the main codebase.

5.1.2 Supported Binary Formats

The space of programs supported by the existing solutions is limited. The original version of rbpf [12] only supports programs that contain the text section extracted out of the eBPF binaries. Because of this, programs that contain read-only data such as text strings for printing debug information are not supported. A further limitation of Femto-Containers VM is that there is no support for data relocations. This means that programs such as the one in Listing 5.1 cannot be executed using the existing solutions.

Listing 5.1: Example program requiring data relocations

```
const int c = 123;
const int *ptr = &c;
int test_data_relocations() {
    bpf_printf("ptr value: %p\n", ptr);
    bpf_printf("address of c: %p\n", &c);
    bpf_printf("This should be c: %d\n", *ptr);
}
```

In Listing 5.1 the variable `ptr` is initialised to hold the address of variable `c`. The first two print statements should print the address of the variable c, whereas the last one should print ’123’. However, this address is not known at compile time. Because of this the object file created by the compiler contains a relocation entry in that place which should then resolved by the linker before executing the program. The limitation of the Femto-Containers VM is that it does not have a relocation resolution procedure taking place on the target device, so the example program cannot be executed.
To overcome this limitation µBPF implements compatibility with the following binary formats:

- **Only text section** - smallest and simplest binaries, no support for `.data` / `.rodata` relocations, backwards-compatible with `rbpf` VM.
- **Femto-Containers** - uses a custom header section to specify offsets of sections in the binary. Supports `.rodata` access without relocations using bytecode patching and special load-double-word instructions. Provides backwards-compatibility with Femto-Containers.
- **µBPF Extended header** - Provides the same functionality as Femto-Containers format. Additionally it specifies the allowed helper functions directly in the binary.
- **Raw object file** - largest binaries obtained by stripping off debug information from ELF files produced by the compiler. Achieves best compatibility by supporting `.data` relocations. This is done by performing relocation resolution directly on the target device before executing the program.

During the deployment stage, users of the system specify which binary format should be generated from the object file produced by the compiler and an appropriate binary is then sent to the target device. In the execution stage, users need to specify the binary format in the request so that the VM interpreter can correctly parse it and find the first instruction to be executed.

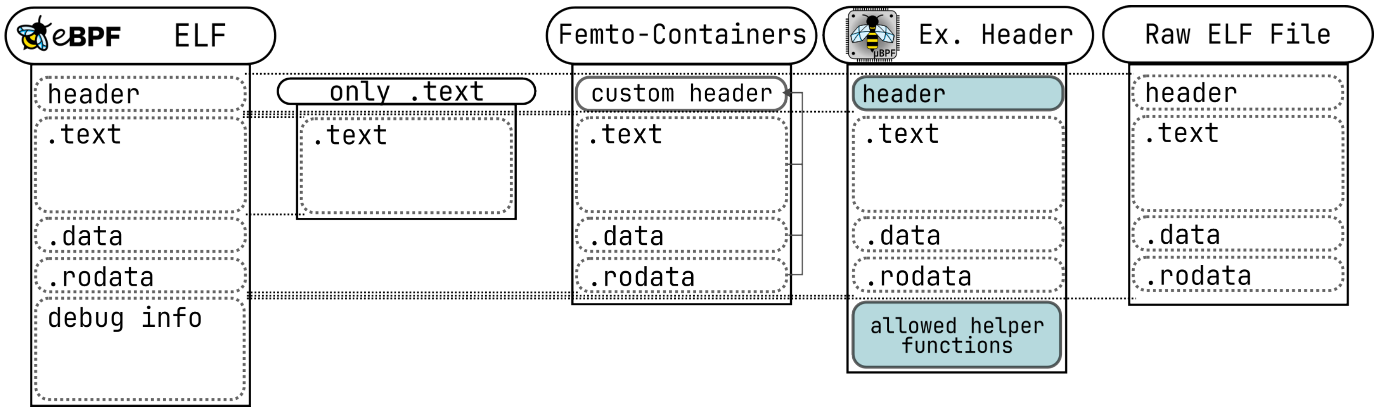


Figure 5.1: Supported binary file formats compared to the original eBPF ELF.

To allow for generating the different binary formats, we implement a portable library for resolving relocations and bytecode patching. It allows for generating binaries that are backwards-compatible with Femto-Containers or resolving relocations directly on the target device when the raw object file is used. This library supports `no_std` as it needs to run on the target hardware to resolve relocations.
When choosing the binary format there is a trade-off between compatibility and the program size. For instance, the raw object file format achieves the best compatibility (data relocations). However, the binaries used are raw ELF files generated by LLVM. Even after stripping off debug symbols, the binaries contain relocation tables which incurs a program size overhead compared to the Femto-Containers format.

5.1.3 PC-relative Function Calls

The eBPF ISA specification allows programs to contain additional functions alongside the main function. These functions are called by making a jump relative to the current program counter, provided that the compiler does not inline the call. When implementing support for those kinds of function calls, the VM interpreter needs to handle the `EXIT` instructions in a special way. This is because the compiler emits an exit instruction every `return` statement in the program. Because of this, when a PC-relative function call is made, we need to ensure that the next exit instruction encountered by the interpreter does not terminate the program, but rather returns from the function call. To achieve this, we implement a return address stack allowing to maintain this information. The µBPF interpreter pushes the current PC onto the stack each time a PC relative function call is made. After that, when an exit instruction is executed, we first check if the stack is empty. If so, the eBPF program can terminate. Otherwise, we pop the program counter from the stack and jump to that address.

5.1.4 VM Memory Access Checks Optimisation

As outlined in Section 3.3, every memory access performed by the VM involves traversing the list of allowed memory regions. This causes a performance bottleneck when the size of the list grows. These safety checks cannot be eliminated without relaxing the sandboxing guarantees provided by µBPF. We reduce this overhead by implementing the caching mechanism proposed by [35].
The solution works by maintaining a cache of the last successfully verified memory region for each instruction in the program. During the VM execution, when the interpreter needs to perform a

memory access, it first indexes into the cache with the current instruction pointer. If there exists an entry in the cache corresponding to that instruction, the interpreter verifies if the current memory access falls into the bounds of the memory region corresponding to the cached entry. If that's the case, the memory access can proceed (**cache hit**, see Figure 5.2).

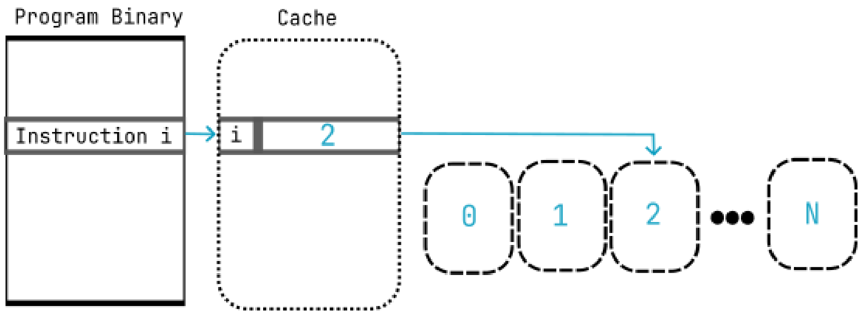


Figure 5.2: Successful cache lookup (**cache hit**).

Otherwise, we need to perform the usual access check by traversing all available regions and update the cache if the memory access is successful within some other region (**cache miss**, see Figure 5.3).

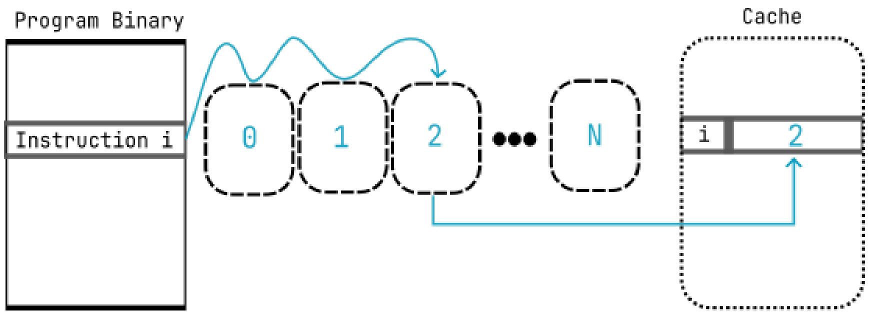


Figure 5.3: Populating the cache after a successful memory access validation (**cache miss**).

In the case when we get a cache hit, but the address turns out to be outside of the region corresponding to the cached entry (invalid access), we need to traverse the entire list of allowed regions again and update the cache accordingly if the cache access is successfully verified (see Figure 5.4).

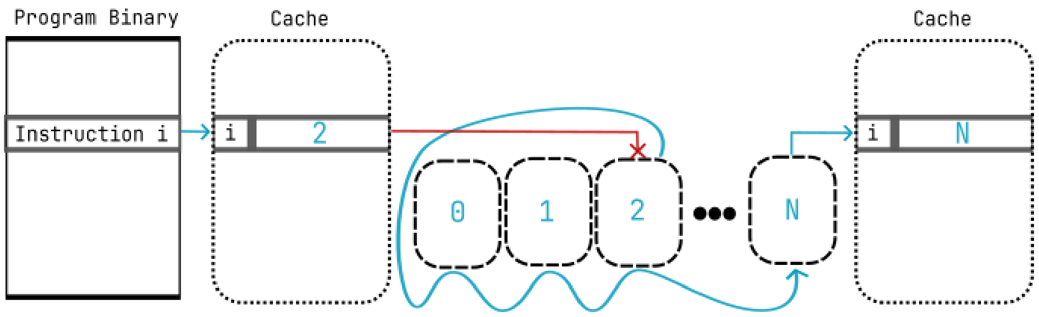


Figure 5.4: Update after cache hit when the memory access into the cached region is invalid.

It is important to note that using this solution introduces a memory overhead as we need to maintain the cache stored as a list with length equal to number of instructions in the program.

Furthermore, as we later demonstrate in the evaluation section, performing cache lookups is slower than the best-case scenario where the required memory region is at the front of the list. However, under certain workloads the memory access check cache allows us to avoid the worst-case where the required region is at the end of the list.

5.2 ARMv7-eM JIT Compiler

The JIT compiler used by µBPF translates eBPF binaries into the native ARMv7-eM assembly. It is implemented in Rust and comprises approximately 2999 lines of code. It is `no_std`-compatible so it can be run directly on the target microcontroller. This was the most technically demanding part of the contribution and would not have been possible without the deployment and testing framework described in Section 5.4
On high level the µBPF JIT compiler works by acquiring an empty memory buffer from the JIT progam storage (see 4.2.3). After that the JIT compiler iterates over the instructions in the translated eBPF program and emits native ARMv7 assembly into the buffer.

5.2.1 Implementing Transpilation Between eBPF and ARMv7

The translation process between the two ISAs is not a one-to-one mapping. Because of the differences in lengths of encodings and designs of the architectures, the JIT compiler in some cases needs to emit more than one instruction to account for a single eBPF instruction. For instance, ARMv7 does not provide native instructions for modulo arithmetic. Because of this for the `BPF_MOD` eBPF instruction our compiler needs to emit instructions that will calculate the modulo remainder by performing integer division, multiplication and subtraction according to the formula below:

$$a \,\%\, b = a - (a \,//\, b) * b$$

Where // above denotes integer division and % is the modulo operator. In this case, the JIT compiler will emit the following sequence of instructions:

Listing 5.2: Instructions emitted to perform modulo operation

```
// dst %= imm
mov %rs1, imm         // scratch_reg1 := imm
div %rs2, dst, %rs1   // scratch_reg2 := dst / scratch_reg1
mul %rs2, %rs2, %rs1  // scratch_reg2 := scratch_reg2 * scratch_reg1
sub dst, dst, %rs2    // dst := dst - scratch_reg2
```

Note that the above snippet of assembly uses scratch registers (`scratch_reg1` and `scratch_reg2`).

Introducing Scratch Registers

The eBPF ISA specifies 10 general purpose registers [47], whereas ARMv7 has 13 registers available [49]. Because of this, we can use the ARM registers that are not mapped to any eBPF registers as scratch registers to hold intermediate results of computations when translating instructions as above. This simplifies the compilation process as this operation does not introduce the risk of overwriting the contents of a register whose current value will later be used by the program. This is achieved because the scratch registers are not mapped to any of the eBPF registers, hence there is no risk that the original eBPF bytecode saves any important values in them. If we did not have those additional registers available, translating a single eBPF instruction into a sequence of ARM instructions as in the example above would not be this simple. This is because we would have to analyze the eBPF bytecode to determine which registers are currently in use and if none of the registers were available, we would have to spill the intermediate values onto the stack which could interfere with the rest of the eBPF bytecode. For instance, the memory access offsets relative to the stack pointer would have to be adjusted to account for the additional values that were pushed onto the stack.

Fixing Jump Offsets

As described above, some eBPF instructions require more than one ARM instruction to express. This causes a complication as all jump offsets in eBPF program are relative to the current program counter. If we emit more instructions, the immediate values of jump offsets in the eBPF program are no longer valid. Consider an example below:

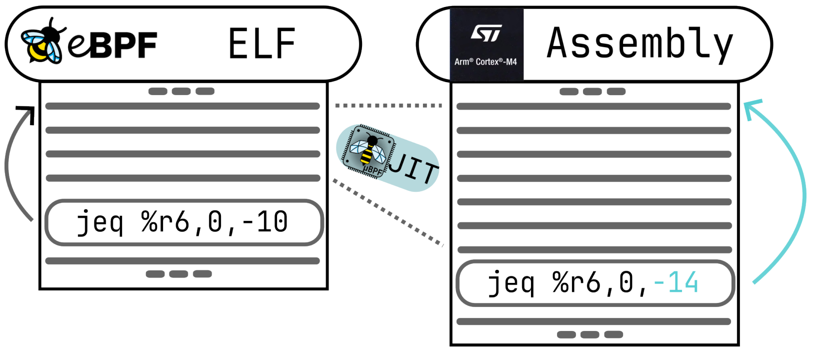


Figure 5.5: Adjusting backward jump offset.

If during transpilation we made the body of the loop above larger by emitting additional instructions, the jump offset at the end of it would no longer be valid. Therefore we need to adjust jump offsets to account for the different length of the emitted assembly. The problem is that it cannot be done immediately when processing each jump instruction. This could be done for backward jumps (example above) as by the time we are processing the jump instructions we have already emitted the assembly that needs to be jumped over. However in case of forward jumps, we do not know how many instructions will be emitted. Therefore, we need to emit a placeholder jump instruction, emit the remainder of the assembly and then adjust the jump offset once we know the number of instructions that need to be jumped over.

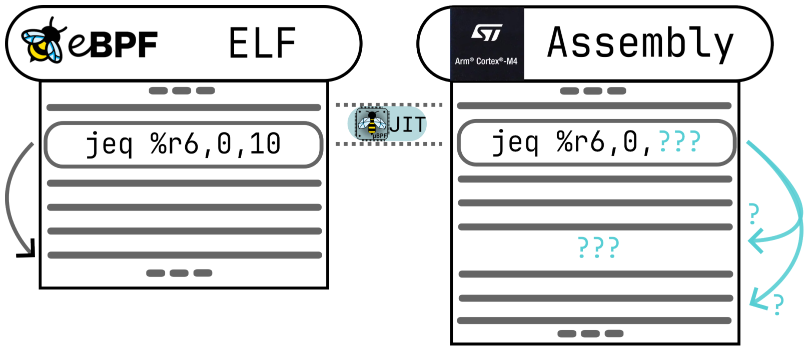


Figure 5.6: Unknown forward jump offset during the first compilation pass.

Copying Program Sections

Another important part of implementing the JIT compiler are the `.data` and `.rodata` sections. When transpiling a given eBPF program we cannot only focus on the program instructions. The `.data` and `.rodata` sections also need to be copied over into the memory buffer for the JIT-compiled program so that they can be accessed during execution. The problem is that this requires an additional relocation procedure to take place. After writing the sections into the program buffer, we know their exact addresses in memory, so instructions in the eBPF program that access those sections need to be patched to include the correct offsets. This is done using a similar relocation resolution procedure as in the case of processing raw-object-file binary formats used by the VM. However, including these sections at the start of the JIT-compiled bytecode means that we cannot simply call into the address of the program buffer to execute it. Instead, we need to determine where the actual `.text` section of the program is located so that we can instruct the processor to call into that particular memory location. This is handled by the JIT program storage as it maintains this information for each program slot.
It is important to note that after the sections from the original eBPF have been copied into the program buffer and the relocations have been resolved, the JIT-compiled program is no longer portable. This is because the load/store instructions accessing those sections contain immediate addresses which are correct only for the current location of the program in memory. Because of this, the globally-managed JIT program storage needed to be introduced to allow for executing JIT-compiled programs multiple times.

Difference in Register Size

When implementing an eBPF JIT targeting ARMv7-eM a big challenge is that it is a 32-bit ISA, whereas eBPF is a 64-bit architecture. This causes compatibility issues as the eBPF compiler assumes 64-bit registers, whereas the ones that we have on the target hardware are 32-bit wide. In some cases this results in bytecode that cannot be directly translated into ARM assembly. Consider the snippet of eBPF assembly in Listing 5.3

Listing 5.3: Truncating loop control variable before performing comparison

```
add %r6,1    // i++;
mov %r1,%r6
lsh %r1,32   // i = (uint32_t) i;
rsh %r1,32
jeq %r1,11,1 // if (i == 11) jump out of loop
```

It corresponds to the control flow logic responsible for incrementing a loop counter and terminating the loop iteration if the counter reaches 11. The problem with this assembly is that the iterator variable is a 32-bit integer. Because of this the emitted assembly contains the instructions: `lsh %r1,32; rsh %r1,32`. They are responsible for truncating the top 32 bits of the compared value by performing a logical shift left by 32 places and then an arithmetic shift right by 32 places. This is done to ensure that we are indeed comparing 32-bit values. The problem here is that if we translated those instructions into ARMv7 assembly verbatim, the program would become invalid and the loop would never terminate.
This is because ARM registers are 32 bits long, so performing a `lsh` by 32 places effectively flushes the contents of a register and sets it to 0. Because of this our loop control condition would never become true and the program would continue running indefinitely. Our implementation solves this issue by replacing all immediate operands of arithmetic and logical shift instructions by their values

modulo 32. This ensures that `lsh %r1 32` is a no-op instruction and does not delete the contents of the register `%r1`. There exists alternative solutions in the Linux kernel where the JIT-compiler simulates 64-bit registers on the stack [50], and we plan to support that in the future.

5.2.2 Instruction Encoding Size Difference

The eBPF ISA specifies 64 and 128-bit instruction encodings [47]. The 128 bit wide encoding consists of a base instruction and a 64 bit immediate value. This is most often used for loading data from an address in memory specified by the immediate operand. When contrasted with the 16 and 32-bit Thumb instruction encodings specified by ARM, we can observe an opportunity to reduce the program size.

Listing 5.4: 128-bit encoding of the LDDW eBPF instruction

```
   0:    18 08 00 00 00 00 00 00         lddw %r8,0
   8:    00 00 00 00 00 00 00 00
                              0: R_BPF_64_64  DATA
```

The listing above illustrates a load-double-word instruction as displayed using `objdump`. The 64-bit immediate following the base instruction (second line starting at address 8) will be replaced by the relocation resolution mechanism to contain the actual 32-bit address in memory from where the value needs to be loaded into `%r8`. We can observe that in this case most bits in the base instructions are not utilised. Moreover, when the actual target address gets written into the immediate operand, it will still only occupy the lower 32 bits of it, because the memory addresses on a 32-bit RIOT instance are 32 bits long. This shows that when using eBPF instructions, a large part of the bytecode will be filled with zeros that do not encode any useful information.
By using the native ARM instruction encodings we can drastically reduce the program size. The difference between the two kinds of ARM Thumb encodings lies in their expressiveness. The 16-bit encodings are desirable as using them results in smaller program size, however by design the sizes of immediate values that they can support are limited. For instance, some instructions can only operate on registers whose number an be expressed using 3 bits. Additionally the largest immediate operand length supported by the short encodings is 8 bits long.
We implement the µBPF JIT aiming to reduce the size of the program binaries. The µBPF JIT compiler prioritises using shorter 16-bit instruction encodings wherever possible to reduce the size of the compiled program. Even if the µBPF JIT compiler occasionally needs to emit multiple ARM instructions to account for a single eBPF instruction, our evaluation demonstrates that we can achieve up to 50% program size reduction (see Section 6.1.4)

5.2.3 Calling Convention and Transpiling Helper Calls

The function calling conventions used by eBPF and ARMv7 are different. This introduces a difficulty when the JIT compiled program calls a helper function. When calling a function in ARM, the first four words of the argument list are passed in registers `R0-R3`. The remaining arguments are passed on the stack. This is different from eBPF where the arguments are inserted into `R1-R5`. because of this, we need to move register values into appropriate slots according to the calling convention, so that the function that we call can interpret them correctly. All helper functions accept five arguments represented as 64-bit integers and return a 64-bit integer value.
Given that ARMv7-eM is a 32-bit architecture, when calling a function with such signature, the lower word of the first argument is expected to be in `R0`, and the higher one in `R1`, similarly for

the second argument, the lower part of the double-word is stored in `R2`, whereas the higher word ends up in `R3`. The remaining arguments should be stored in the stack.

The eBPF bytecode follows the eBPF calling convention (where the arguments are stored in `R1-R5`) and we only support 32-bit values. Hence, we need to move the value of `R1` into `R0` and spill the other registers `R3-R5` onto the stack (`R2` is already in its correct place).

Note that this approach is wasteful in that we are not using R1 or R3 as they were supposed to store the upper bits of 64-bit integer values. We are not using those bits as we only support programs with 32-bit integers. Because of this, in our implementation we also need to ensure that registers R1 and R3 are zeroed so that the callee does not receive incorrect argument values caused by leftover bits present in these registers.

This could be improved by changing the signature of all helper functions to accept 32-bit integer values, however that would limit functionality supported by the interpreted VM execution which we do not want. This is because the same set of helper functions is used by the VM and the JIT. The high-level motivation for this decision is that when using the interpreter we should aim at highest possible compatibility, whereas when using the JIT compiled programs we can sacrifice some compatibility (e.g. using 64-bit integer values) to gain a performance improvement.

An important detail is that the helper function is called by inserting its memory address into one of the scratch registers and setting its lowest bit to 1 (ARM / Thumb selection bit [49]). This instructs the CPU that we intend to continue executing in Thumb mode and prevents hardfaults. After that a branch-with-link-and-exchange instruction is emitted. During execution at this point the instruction will instruct the CPU to jump to the address of the helper function and start executing the code inside of it. Transpiling a helper function call also requires us to preserve the contents of the ARM `LR` register, this is done by pushing it onto the stack before emitting the branch-with-link-and-exchange instruction. It is then recovered by popping from the stack after the helper function returns.

5.3 RIOT Server Infrastructure

This section documents the implementation of the software deployed on target microcontrollers responsible for orchestrating program deployment and handling client requests. The server infrastructure was implemented as an application on top of RIOT [33] and comprises approximately 3625 lines of Rust and 674 lines of C accessed through foreign function interface (FFI).

The server infrastructure consists of three main components: the CoAP server, VM execution manager and the middleware implementation for RIOT OS bindings used by the VM.

5.3.1 Embedded Server and CoAP Communication

The server used by µBPF is implemented using the `gcoap` module provided by RIOT. This module is accessed through Rust bindings provided by `riot_wrappers` library. The server runs as a separate thread with a 8 KiB stack. This is one of the heaviest (in terms of required stack size) threads in the system. This is because it needs to be able to run the VM and also optionally the JIT compiler directly in the endpoint handler functions. The server communicates with clients over the CoAP network through the `coap` module provided by RIOT.

The server supports the two main workflows: **i) program deployment** - the server exposes an endpoint allowing clients to trigger a SUIT firmware pull. In that case, the server calls into the

RIOT's SUIT subsystem to initiate the process. **ii) program execution** - the server exposes a set of endpoints allowing for executing or benchmarking the previously-deployed programs.
As discussed in Section 4.2.2, the client requests a long-running VM to be executed, the server communicates with the VM manager thread over the message passing inter-process communication (IPC) provided by RIOT.

CoAP Message Size Limitations and Encoding Solution

When communicating with the server running on the embedded device the key limitation that we faced was that the recommended size of the request payload that can be transferred over CoAP is limited by 1024 bytes [23]. This poses a problem for our system as we need transfer a large number parameters to specify each program execution. In particular, the list of helper functions that are available to the VM while executing the program is can be controlled through the request payload. Note that in the original implementation of `rbpf` the ID of each helper function is represented by a 32-bit integer. This means that We could specify up to 256 available helper functions in the request body. The problem is that most low-end embedded hardware might not be capable of processing requests this large (as the request needs to be written on the stack of the thread processing the request on the target device). For instance, RIOT by default allows for the maximum size of the CoAP request to be 128 bytes. This constraint has proven to be a major roadblock when implementing communication with the server running on the target microcontroller. To solve this issue we implemented a request encoding mechanism which decreases the size of the request payload and maximises information that can be stored in it.
For the program deployment request (SUIT firmware pull) the request needs to contain the following information:

- VM configuration for the deployed program - encoded using 16 bits,
- IP address of the CoAP fileserver serving the programs - encoded as a string without semicolons,
- name of the SUIT manifest that needs to be pulled,
- the list of helper functions that the deployed program is allowed to call - a list of 8-bit integers in hex encoding.

For the execution request we only need the VM configuration (consumes 16 bits) and the rest of the message payload is dedicated to encoding the helper functions, which for a message size of 128 bytes allows for encoding up to 52 distinct function IDs.

5.3.2 Middleware Implementation

To enable µBPF VMs to interact with RIOT we implement a middleware layer achieving full compatibility with the prior work on Femto-Containers VM. We extend this functionality with wrappers around drivers for peripherals such as the HD44780 LCD display [51]. The middleware provided by µBPF allows for fine-grained control over the set of helper functions that a given program is allowed to call. This is due to the implementation details of `rbpf` on which our implementation of the VM was based. In case of `rbpf` the list of allowed helper functions is dynamically attached to the VM during initialisation. This can be contrasted with Femto-Containers where the list of allowed helper functions is hard-coded in a switch statement [52].

The middleware is implemented in a similar way to the eBPF helper functions. It consists of a set of functions written in Rust with the following signature:

Listing 5.5: Rust signature of a helper function.

```
pub fn bpf_printf(fmt: u64, a1: u64, a2: u64, a3: u64, a4: u64) -> u64 { ... }
```

Each helper function accepts up to 5 arguments and when the VM calls a given function it passes in the registers `R1 - R5` as those arguments (in line with the eBPF calling convention). Those functions are then accompanied by a Rust enumeration listing all available helper function IDs. This implementation is then coupled with a corresponding C header file which defines all of the available helper functions as function pointers with the address equal to the function ID:

Listing 5.6: Header file helper function declaration.

```
static void *(*bpf_printf)(const char *fmt, ...) = (void *)BPF_FUNC_BPF_PRINTF;
```

This header file is then used when compiling eBPF programs so that we can call helper functions and the compiler inserts the corresponding function IDs in place of function pointers. Note that the header file needs to be maintained in conjunction with the helper function ID enum in Rust as the ID numbers need to match on both sides.

5.4 Deployment Framework

One of the key limitations of the existing solutions was the number of manual steps involved to deploy and run eBPF programs on target devices. We implement a suite of tools allowing to perform the full deployment workflow with a single command. The deployment framework comprises approx. 2825 lines of Rust and is accompanied by a testsuite containing 3486 of tests written in Rust and C.

5.4.1 Deployment Tools

The deployment tools allow for compiling, performing bytecode patching, creating the SUIT manifest file and signing it and sending appropriate requests to the server. The compilation is done by wrapping around `clang` and `llc`. Bytecode patching is implemented as a library based on the `gen_rbf.py` script used by Femto-Containers [52]. This is implemented using the `goblin` Rust library allowing for parsing and modifying ELF files. The SUIT manifest is created and signed by calling into the SUIT tools provided by RIOT. All of this functionality is available through a command-line interface application written in Rust.
An interesting design decision in the tools library is that it is split into separate smaller libraries (Rust crates) using the workspaces feature provided by the `cargo` build system. This was done to allow for reusing the code in both the tools and the server running on the embedded device. This way, when a client sends a request to the server using the tools library, it is represented by a struct that is then serialised. The same struct is imported in the server code running on the embedded device. This way, the request can be easily deserialized on the other side. Since the serialization and deserialization code is located in one place, it can easily be updated without the need to update the server and tools code separately. This has proven useful when designing the request encoding mechanism (see Section 5.3.1) as it allowed for prototyping different encoding schemes without the need to modify server code. Only a recompilation with the new version of the library was required.

5.4.2 Integration and Testing

Because of the embedded environment in which the server code operates, the testsuite needed to be engineered around communicating with the device instead of running directly on it. This is because in order to test the server, we would have to write a special testing application, then flash it onto the target device and observe outputs through e.g. a serial port. This approach is not convenient as it requires implementing additional code and the actual application that we intend to use is not being tested.
To get around this issue, the testsuite is implemented using end-to-end integration tests. First, a tested program is deployed onto the microcontroller using the tools library. After that a request is sent to the device and the tests match on the response received from the server. The response always contains the return value of the program so that way we can determine whether the program has been executed successfully.
This approach is somewhat restrictive, as we can only check the return value of the program and we are assuming that the server infrastructure responsible for initialising execution and returning the response back to the client is correct. For instance we cannot check the side-effects of a program running in a VM such as printing messages to the console or controlling peripherals connected to the device. Those properties need to be manually inspected.
However, this approach offers good flexibility with respect to the target hardware on which the tests can be run. RIOT provides a `native` implementation of their OS - a simulated 32-bit microcontroller running directly on the user's desktop machine. Our testsuite communicates with the native RIOT instance over the `tap` network interface allowing to test the system without having access to a physical microcontroller board. The testsuite has proven invaluable when implementing the JIT compiler. It allowed for testing that the JIT-compiled ARMv7 code executes correctly which would not have been possible on the desktop machine running on a different architecture.

5.5 Example Application

We implement an example application to demonstrate how µBPF can be used to deploy a compartmentalized system on a microcontroller. The application consists of three compartments: two responsible for collecting sensor data and one for controlling an LCD display (see Figure 5.7).

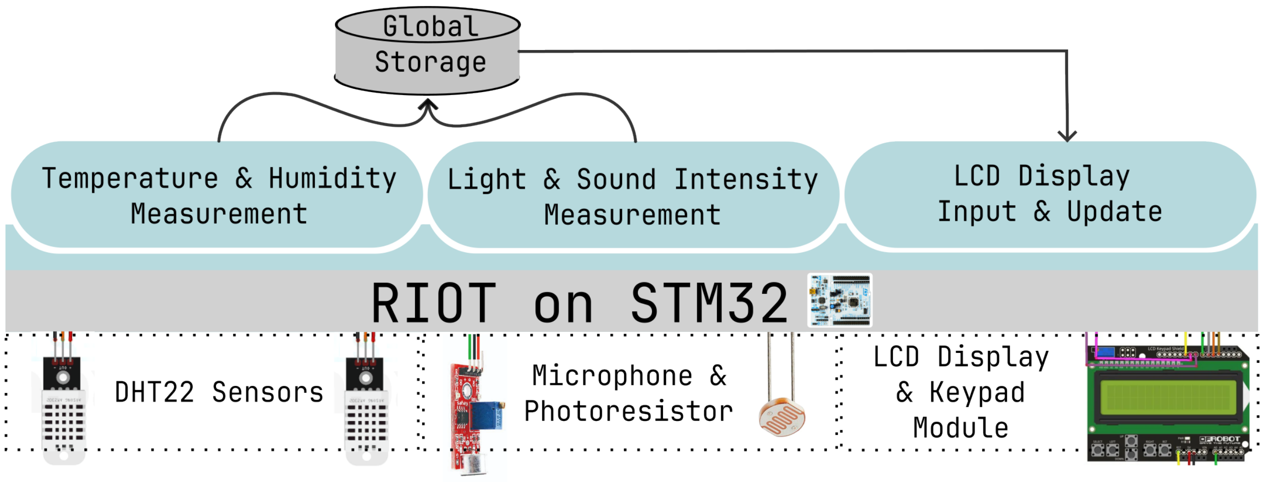

Global Storage
Temperature & Humidity Measurement
Light & Sound Intensity Measurement
LCD Display Input & Update
RIOT on STM32
DHT22 Sensors
Microphone & Photoresistor
LCD Display & Keypad Module

Figure 5.7: Example application deployed with µBPF.

The compartments are isolated by running in separate instances of the µBPF VM on different RIOT threads. Compartmentalization is achieved by allowing each of the programs to only access a minimum set of helper functions required to perform its task. The compartments that collect data can only call helpers to read sensor measurements and write them into the global storage. The LCD display compartment can only read from the global storage and access helper functions responsible for formatting the measured values, displaying them on the connected LCD. It also collects user input from the keypad module to change the type of measurement that is currently being displayed.
All of the above business logic is implemented in eBPF programs and the underlying server infrastructure is not specific to the sensor station application. The server is responsible for handling requests and exposing the middleware interface to the programs executing in the VM.

5.5.1 Malfunctioning Compartment and Update Scenario

The example application allows us to demonstrate how compartmentalization and over-the-air upgradability can be used to change the behaviour of the system on the fly. This can be done by deploying a fleet of eBPF programs as above, with the program responsible for controlling the display containing an error: navigating to a specific place in the UI using the keypad buttons is going to perform an illegal memory access causing the VM to crash.

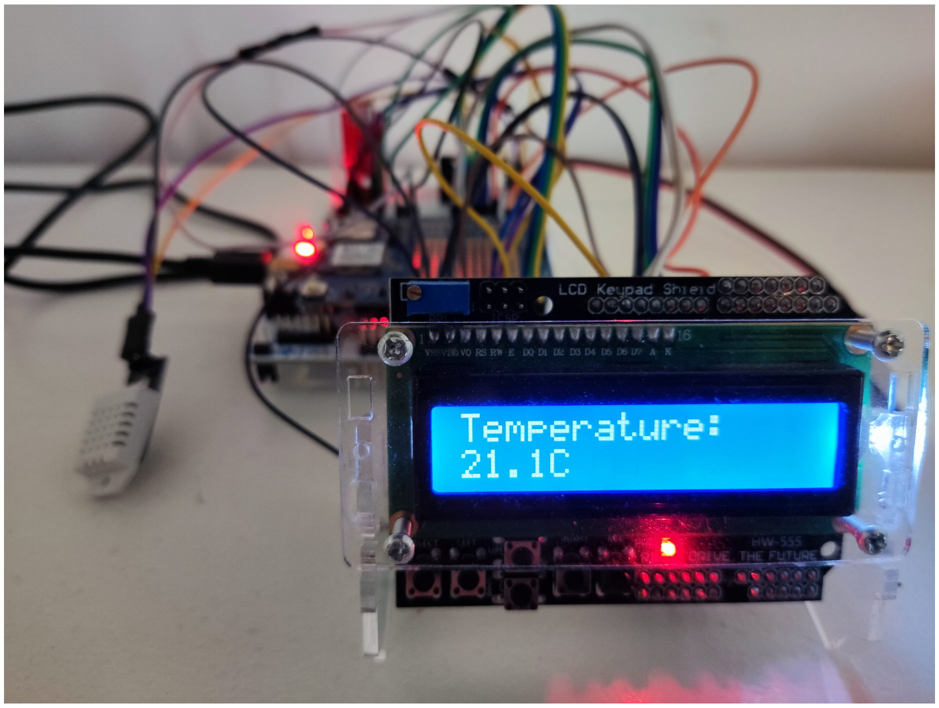


Figure 5.8: Display and keypad module of the example application.

Because the program logic is executed in a sandboxed environment, only the compartment running the display module will terminate the execution of its VM. After that we can patch the display module logic to remove the error and deploy the new version of the software without interfering with the operation of the remaining parts of the system.

Chapter 6

Evaluation

6.1 Performance Analysis

In this section we evaluate performance of µBPF running on STM32-F439ZI - a microcntroller with an ARM Cortex M4 CPU, 256 KiB of RAM and 2 MiB of flash storage. We perform evaluation against benchmarks that were considered in prior work [11, 8].
We measure the program load, verification and execution time as well as the size requirements of the program binaries. We evaluate our VM and JIT implementation using the Fletcher 16 algorithm [30]. It performs a number of arithmetic operations, memory accesses and branch instructions, while looping over a a 640 B string to calculate its checksum. It aims to simulate a representative workload of processing sensor data on a microcontroller [11].We compare performance against three baselines: native C, Femto-Containers [11] and rbpf [12] VMs. Results of the benchmark can be seen in Figure 6.1.

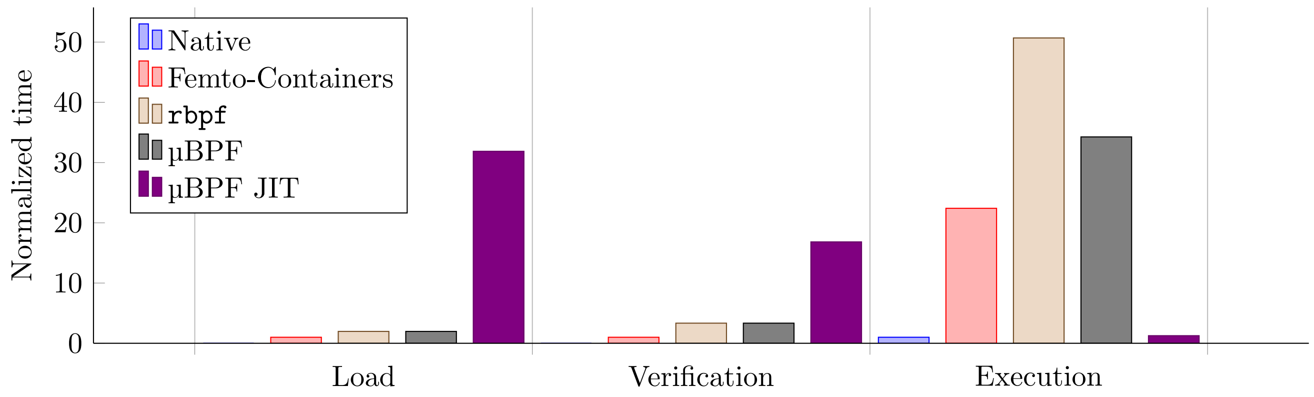


Figure 6.1: VM and JIT performance of calculating the Fletcher16 checksum of a 640 B string.

The Figure 6.1 depicts the normalized execution times of the three stages of the deploy-execute workflow. The results were normalised towards the minimum non-zero result in each category. Note that in case of µBPF JIT, the load time includes the JIT compilation step. This step does not have to be repeated when the program is executed multiple times, thereby allowing us to amortise the cost of compilation. The same is applicable for the verification step, once we JIT-compile and verify a given program, we can execute it repeatedly by accessing the JIT program storage, in which case we only pay the execution time cost, which achieves native C performance. The graph above does not include the native performance in native and verification stages as the C code is compiled into the test application (load time is zero) and it is not verified.

We find that although JIT compilation does incur a performance overhead of approx. 2000 µs, the total execution time is lower than the Femto-Container VM. This shows that using JIT compilation can be used to improve performance under certain workloads. The raw execution times can be seen in Table 6.1 below.

	Load	Verify	Execute	Total
Native C	N/A	N/A	114 µs	114 µs
Femto-Containers	61 µs	6 µs	2555 µs	2623 µs
`rbpf`	120 µs	20 µs	5779 µs	5921 µs
µBPF	120 µs	20 µs	3906 µs	4046 µs
µBPF JIT	1944 µs	101 µs	144 µs	2190 µs

Table 6.1: Fletcher 16 on 640 B string benchmark results.

6.1.1 Execution Time Overhead

To investigate how the overhead compared to native execution relates to the size of the computation we perform the same workload, this time with a strings of sizes varying between 80 B and 640 B, Execution time in µs of different solutions can be seen in Figure 6.2. We observe that Femto-Containers VM [11] is about 35% faster than µBPF . This overhead of using µBPF compared to Femto-Containers is likely because the Femto-Containers interpreter uses a highly-efficient hand-coded jumptable [52] approach whereas µBPF iterates over instructions using a match statement [12]. However the overhead of µBPF is much smaller compared to the baseline `rbpf`, this was achieved after we optimised its implementation to perform instruction parsing and memory access checks more efficiently. We observe the execution time of the JIT-compiled code is of the same order of magnitude as the native C implementation.

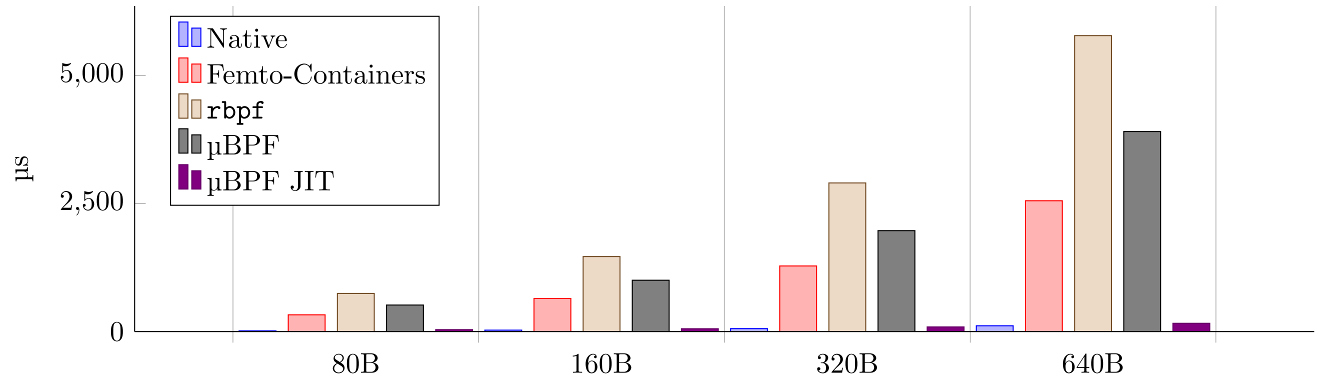


Figure 6.2: Fletcher16 execution time for 80 B-640 B data.

Real-world Appilication Benchmarks

After submitting an early version of this report to the eBPF '24 Workshop, one the most common feedback among the reviewers was the need for additional benchmark workloads. This was addressed in the camera-ready version of the submission and is also included in this report.
To simulate real-world applications, we benchmark program logic used for the example application in 5.5. The CoAP response formatter [11] reads sensor data from a designated global storage slot and writes it into the CoAP response packet buffer which is then sent back to the client. The second program computes a moving average of sensor data in the global storage.

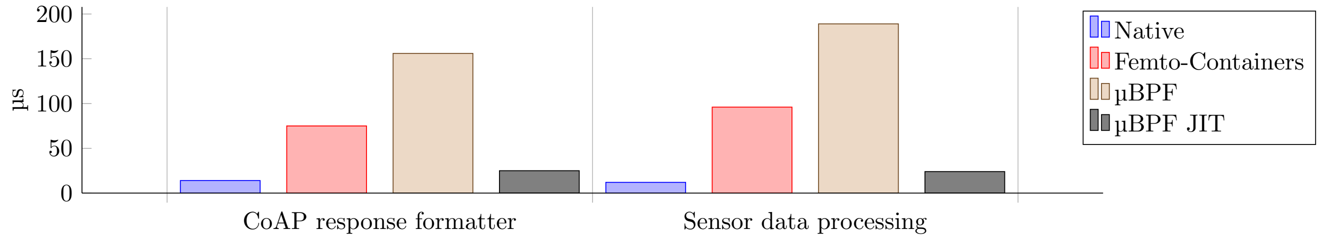


Figure 6.3: Example application logic - execution time.

We observe that in this case the overhead of µBPF VM compared to Femto-Containers is larger. Because of higher configurability, µBPF VM has a larger initialization overhead than Femto-Containers. For instance, it is possible to configure the binary format, so the interpreter needs to determine which format it is operating on and parse it accordingly. Since the above programs are short, the initialization overhead has a large impact on the overall execution time.
When considering performance of a particular program that we want to deploy, it is important to consider the nature of the workload. Figure 6.4 illustrates that under IO-bound workloads the performance discrepancy between different solutions is negligible.

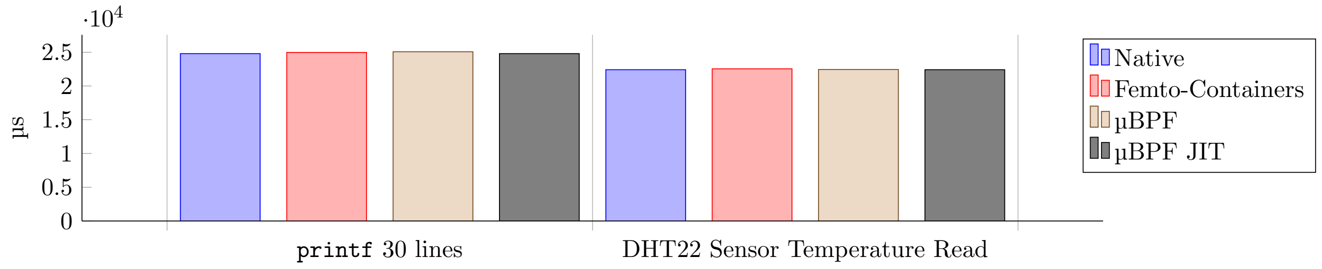


Figure 6.4: IO-bound benchmarks.

The first benchmarked program writes 30 lines of text to the console over the serial port. This is done in a loop, so we can see that native and JIT solutions are marginally faster, however the difference is negligible as most of the work is done by the OS when sending data to the console. Similarly, in the second workload, the eBPF program interacts with a DHT22 temperature sensor to collect a measurement. Because the sensor sending data to the target device over a single wire, the communication overhead dominates the execution time of the program.
Note the execution time those benchmarks is in the order of 20 ms, if we compare this to the Fletcher16 execution times from the experiments above (≈ 2 ms), we can conclude that even computationally intensive workloads such as Fletcher16 can be dominated by performing IO operations.

6.1.2 Verification Time

We observe that in case of the JIT the verification takes marginally longer (see Figure 6.1). This is because the JIT operates on raw object files, whose header metadata takes longer to parse compared to the other custom formats. However the total time of 100 µs negligible compared to 2000 µs it takes to perform JIT compilation.

6.1.3 JIT Compilation Overhead

We observed that executing the code emitted by our JIT compiler achieves performance comparable to native C. To validate that using this is a viable solution, we need to ensure that the cost of

compilation is not prohibitively expensive.
Related work [13] suggests that because of hardware constraints, verification and jitting of eBPF programs should be decoupled from embedded devices and performed in a VM environment running on a much more powerful desktop machine. A key distinction between our system and the one considered in [13] is that we are focusing on simple VM implementations (Femto-Containers [11], rBPF [8], Rust `rbpf` [12]) all of which are using very simple verification that are not computationally expensive (see 6.1) This can be contrasted with the Linux kernel verifier [48] that was considered in [13], where it was not possible to decouple the verification from the JIT step because of implementation details coupling the two components.

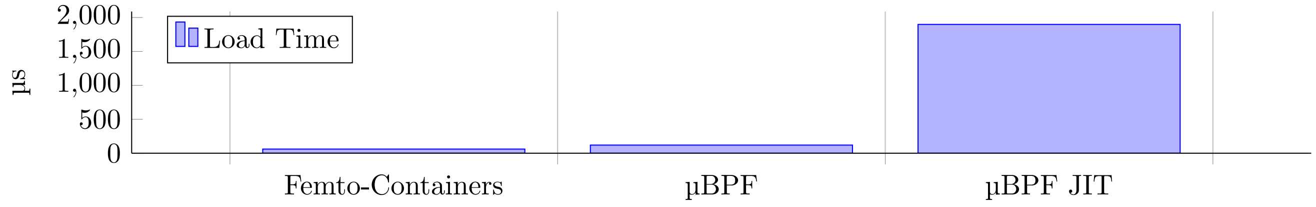


Figure 6.5: Fletcher 16 logic load time for VM and JIT.

Figure 6.5 shows the load time (including JIT compilation) for the Fletcher 16 algorithm logic. The JIT load time is much larger compared to the other solutions, as in those cases only minimal processing is applied during this step. However, the total processing time in case of the JIT is still smaller compared to the VM alternatives (see Table 6.1).

6.1.4 JIT Program Size

We also consider the additional memory required to perform JIT compilation. By design, this process requires that we have access to two statically allocated memory buffers, one to store the original eBPF program, and the second one to emit the JIT-compiled code into it.
Consequently, at load time we need twice as much storage space as in the case of the VM interpreters. However, once the compilation is complete, we only need the JIT-compiled machine code, the original program can be discarded and its buffer can be reused for loading other programs.
We measure the program size requirement across 10 example programs (see Figure 6.6)

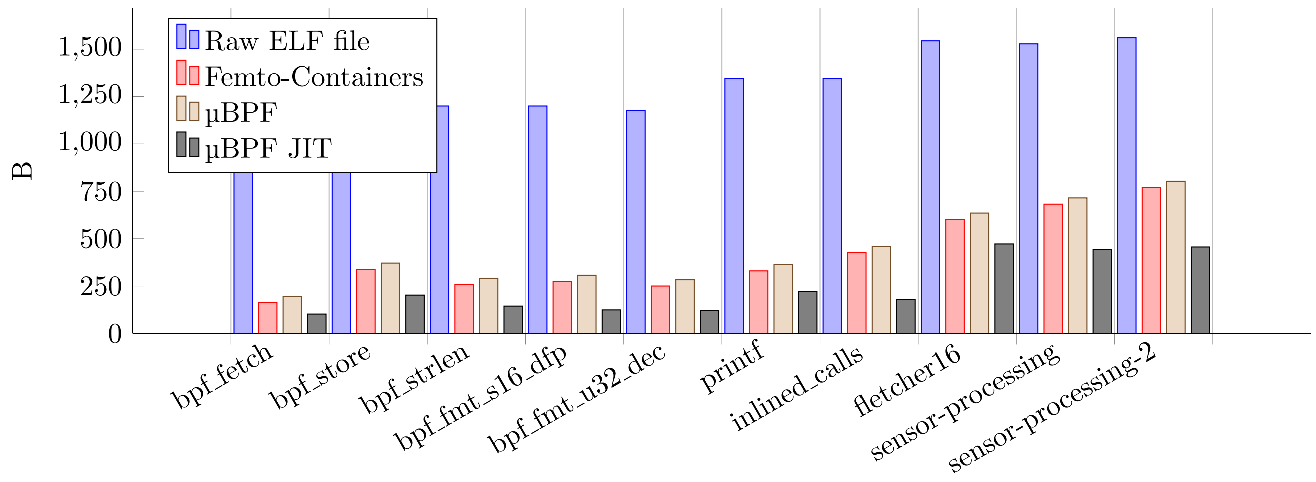


Figure 6.6: Example program sizes across binary formats.

The first six programs are short scripts performing pre-processing and calling helper functions. The sixth one: `inlined_calls` defines a set of static inlined functions and calls them in the main function, in which case we have measured a maximum of 57% program size reduction. The last three are the previously benchmarked Fletcher16 algorithm and the two programs used by the example application in 5.5.

We note that in case of programs that contain a sizeable section of read-only data, the transpiled program needs to contain a copy of the data. Consequently in such cases the program size reduction is smaller. For instance, in `fletcher16` above the program mainly consists of the 640 B string that is checksummed which is stored in the `.rodata` section of the program binary.

As noted above, during JIT compilation we need to have access to both the original eBPF program buffer and the JIT-compiled one. Referring this to Figure 6.6, when performing JIT compilation, the space occupied by the two buffers is equal to the sum of blue and grey bars. This is because the JIT compiler ingests raw ELF file eBPF binaries to be able to resolve relocations.

6.1.5 When should we use the JIT?

µBPF allows for saving JIT compiled programs in a memory storage to be reused multiple times thereby amortising the cost of transpilation. To find the point at which using the JIT compiler is beneficial even for a single execution, we repeat the previous experiment with the Fletcher16 algorithm on data sizes varying from 80 B to 2560 B We measure the compilation, execution and total time, comparing it with the baseline of Femto-Containers.

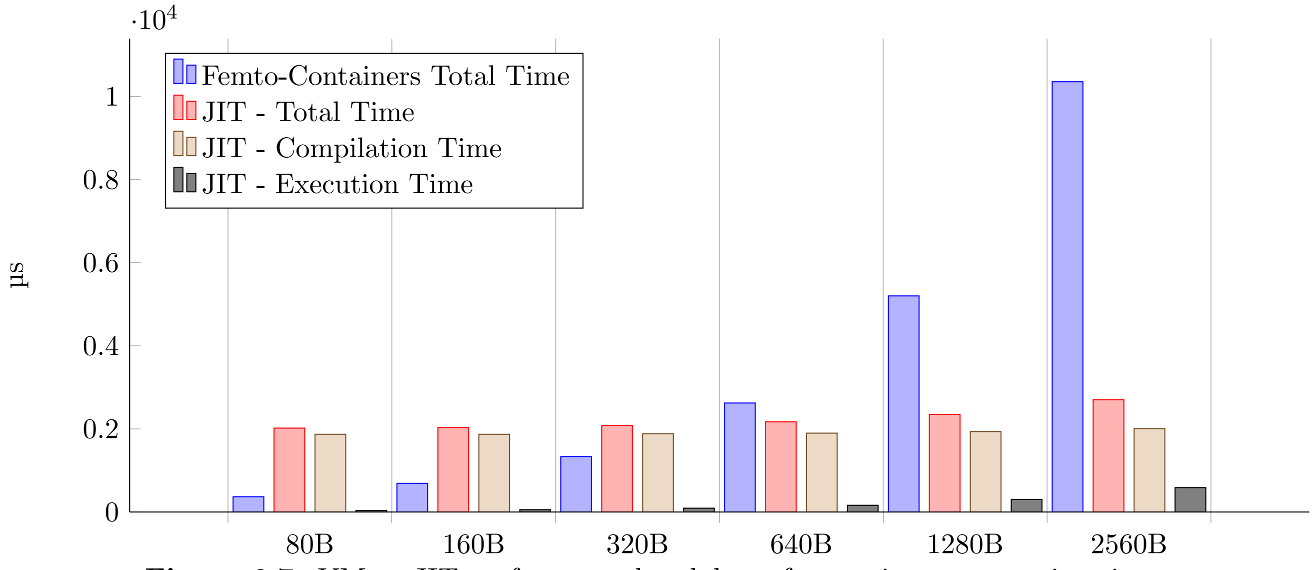


Figure 6.7: VM vs JIT performance breakdown for varying computation sizes.

Figure 6.7 shows that for small computation sizes, the JIT compilation time dominates, and the total time of the interpreted VM execution is shorter. However, as the processed data size increases, the performance discrepancy between the JIT and VM execution grows. When the processed data size reaches 640 B, the total processing time for the JIT is shorter than the one for the VM.

It is important to stress that in normal workflows where a given deployed program is intended to be executed multiple times, we only need to pay the price of JIT compilation once. After that the JIT-compiled program is stored in the program storage and can be executed repeatedly each time achieving native performance, which amortises the cost of compilation.

6.2 Evaluation of Introduced Optimisations

6.2.1 Improving Baseline `rbpf`

After inital benchmarks of the default `rbpf` VM implementations that we ported to run on embedded devices, we noticed that it runs on average two times slower compared to the state-of-the-art Femto-Containers implementation. We identified three reasons for this discrepancy:

- memory access checks performed by `rbpf` involved passing many arguments to a function which was not inlined,
- instruction parsing involved creating new structures on the stack,
- the interpreter is implemented as a large `match` statement.

We remove the memory check overhead by inlining the helper function calls and removing unnecessary state that was being propagated for logging purposes. We also improve instruction parsing by using a similar approach to Femto-Containers [52], where a pointer to the eBPF program buffer is cast as a pointer to the instruction struct. This avoids creating the instruction structure on the stack and copying values unnecessarily.

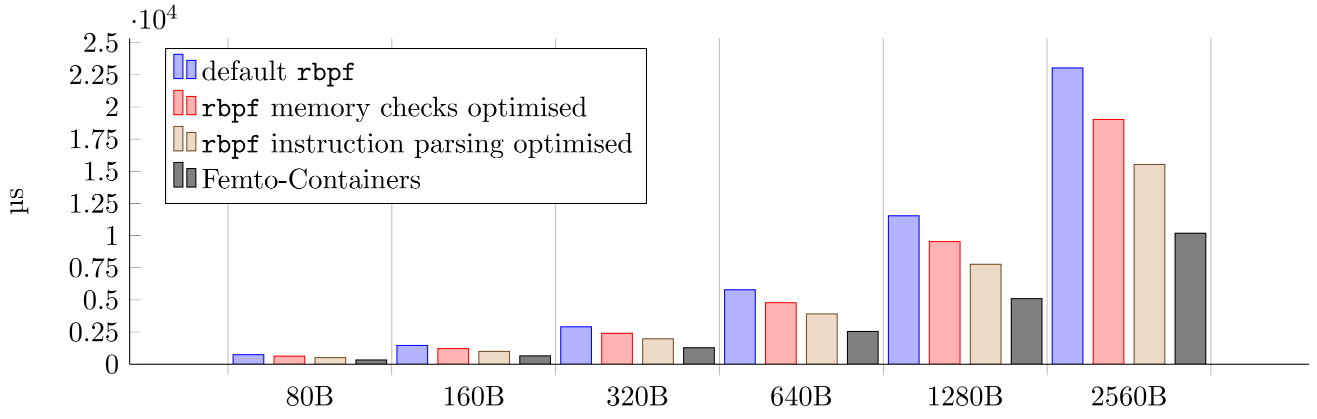


Figure 6.8: Fletcher16 execution time for 80 B-640 B data.

Introducing those changes allowed us to improve performance of the baseline `rbpf` VM by removing 50% of the overhead compared to Femto-Containers. We suspect that reimplementing the interpreter to use a jumptable approach similar to Femto-Containers could further improve performance, however the engineering effort required to do so would be substantial as Rust does not provide a `goto` statement.

6.2.2 Improving Memory Access Time

We evaluate the implementation of memory access verification caching (described in 5.1.4) using the following workload. We implement a simple eBPF program containing an empty `for` loop performing 100000 iterations. In order to prevent the compiler from optimising the entire for loop away, we mark the iterator variable with the `volatile` keyword. The reason for choosing this particular benchmark is that during each iteration of the loop, the iteration variable gets read from memory, incremented and written back to memory. This allows us to test performance of memory

access checks as as almost all work done by the program revolves around accessing memory to update the iteration variable (other instructions are responsible for loop control flow).
The experimental setup of the benchmark involves testing two nearly identical programs, one performing 100000 iterations of the loop while maintaining the iterator variable on the stack, whereas the other maintaining the iterator variable in the `.data` section of the program. The idea is that we expect that stack to be the most frequently accessed memory region, thus we place it at the front of the list of allowed memory regions. This allows us to observe the best-case performance when no caching mechanism is used. To investigate the worst-case performance, we place the `.data` section at the end of the list of allowed regions and vary the length of that list. This process is illustrated in Figure 6.9 below

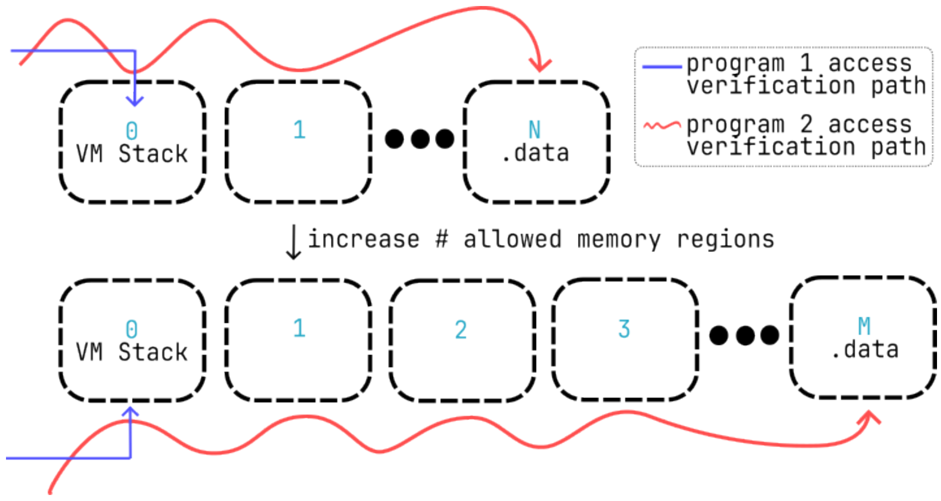

program 1 access verification path
program 2 access verification path
0 VM Stack
1
N .data
increase # allowed memory regions
0 VM Stack
1
2
3
M .data

Figure 6.9: Memory access cache benchmark setup.

We can observe that the benchmark was set up in a way that the first program (iterator variable stored on the stack) is expected to exhibit the best performance. On the other hand, the performance of the second one (accessing `.data` section) will become progressively worse as we increase the length of the traversed list.
We also evaluate the performance of executing those two programs with memory access verification caching enabled. Results of this benchmark can be seen in Figure 6.10. Although maintaining the cache does incur a penalty (see datapoint 1 Figure 6.10 - cached solutions perform the worst), it allows for removing the discrepancy between the best and worst case scenario of performing memory access checks when no cache is present. We can observe how the worst case progressively degrades as the number of allowed memory regions grows, whereas in case of cached solutions, no change in execution time is observed.

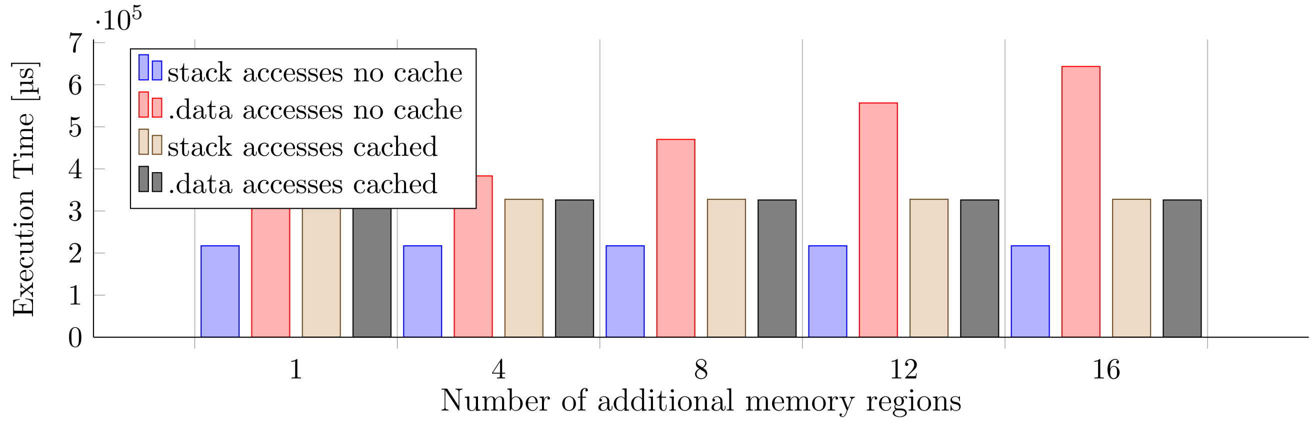


Figure 6.10: Memory access performance with and without caching.

6.3 System Compatibility

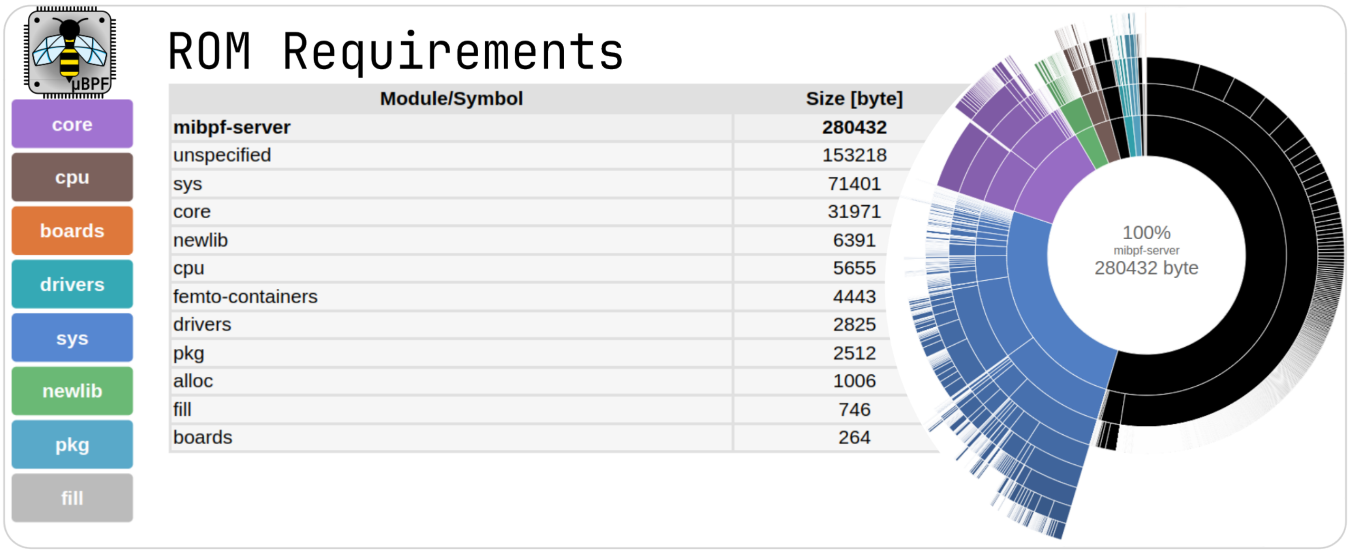
Similarly to the related work [8], we evaluate the compatibility of our solution in term of RAM and ROM requirements and the percentage of boards supported by RIOT that can run our system.

6.3.1 RAM Requirements

In this section we evaluate the minimum RAM requirements to run our system. We note that the RAM requirements of our solution can vary depending on the chosen configuration. The number of available SUIT and JIT storage slots can be configured at compile time. This greatly impacts the RAM requirements of the system as each slot needs to have a meaningful size that allows for loading eBPF binaries into it. Reasonable slot sizes range from 1024 to 8192 B. Additionally, for long-running eBPF programs we can configure the number of VM worker threads that are spawned to execute them. For our experiments, the default size of the VM worker thread stack was 8 KiB. After deploying the system using a minimal configuration composed of: 1 SUIT slot, 1 JIT slot 4096 B each and 1 VM worker thread with 8192 B stack, we calculated the total RAM requirement to be: 45 912 B which is approx. 45 KiB. If we compare this to the 256 KiB RAM available on the STM32F4, we can see that the RAM requirements of the minimal configuration of the µBPF server are acceptable.
We acknowledge that in applications where we need to deploy and execute multiple programs at the same time, the RAM used by the storage slots and worker threads will quickly increase. We could reduce the RAM requirements by only allowing a certain subset of those programs to be JIT-compiled (thereby saving on JIT storage) or fine-tune the size of SUIT and JIT storage slots so that they are minimal but big enough to store the largest of the deployed programs.

6.3.2 ROM Requirements

Figure below illustrates the ROM memory requirements of the server software flashed onto the target microcontrollers which were calculated using the `cosy` RIOT make target. It includes the RIOT OS (`sys`,`cpu`,`boards`,`newlib`,`pkg`), the `core` rust library and µBPF server application code.



ROM Requirements

Module/Symbol	Size [byte]
mibpf-server	**280432**
unspecified	153218
sys	71401
core	31971
newlib	6391
cpu	5655
femto-containers	4443
drivers	2825
pkg	2512
alloc	1006
fill	746
boards	264

Figure 6.11: ROM memory requirements of the system.

The µBPF server code is marked with black as `unspecified` because the RIOT profiling tool works by inspecting the ELF sections and has difficulties classifying the sections produced by compiling Rust. We can observe that the µBPF infrastructure occupies approximately 50% of the flashed image. This is because it contains `rbpf`, the JIT compiler and all rust libraries that these two components depend on. This ROM requirement could limit the compatibility of our solution, however if we put the overall memory requirement 0.28 MiB to the available ROM on the target STM32 (2 MiB), we can observe that this ROM overhead is not substantial.

6.3.3 Supported Boards

The key factor that limits compatibility of our solution is that the target microcontroller board needs to support running Rust on RIOT. We evaluate the number of supported boards by running the `make info-boards-supported` target provided by RIOT for our µBPF server application. We find that out of 270 boards supported by RIOT, 199 of them can theoretically run our solution. This amounts to 74% of boards being supported. Here we acknowledge that this support is theoretical as it does not take into account RAM and ROM budget of the device. This is because the tools provided by RIOT for every possible board check if its definition declares support for Rust and all RIOT libraries that our system depends on. The `make` target that we used does not actually build the project ELF file to check ROM requirements.
Because of this, in order to find the true compatibility of our solution, we need to take the RAM and ROM requirements from the previous section as well.
We acknowledge that because of Rust toolchain limitations, our solution is not compatible with the esp32 microcontroller family. This is because those microcontrollers run on the Xtensa architecture which requires a custom Rust toolchain that is not included in the main release of Rust. Because of this, the RIOT support for esp32 does not allow for running Rust. This is an important limitation for the compatibility of our system because esp32 microcontrollers are affordable, powerful, and have built-in WiFi support. In the future we can explore porting µBPF to run on esp32 devices by using ESP-IDF as the host OS instead of RIOT.

Chapter 7

Conclusions and Future Work

7.1 Discussion & Lessons Learned

Open-Source Contribution to `rbpf`

The initial modifications that we introduced to `rbpf` added Rust `no_std` support and were generic enough to be included in the upstream repository. An initial patch was submitted in March 2024 and was progressively refined given the feedback from the maintainers of the project. The pull request was merged in June 2024. The process of working on the pull request required updating the project documentation and structuring the introduced changes in a clean way allowing for `git bisect` to be used should any issues with the new version emerge. It was an invaluable experience as the maintainers required that the patch follows the Linux kernel contribution guidelines [53]

ACM SIGCOMM eBPF '24 Workshop Submission

An early version of this report was submitted for review to the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions. It was accepted and will be presented at the conference in August 2024. The process of preparing a submission was a very useful learning experience leading up to the final report. It required producing a concise six-page write-up of the contributions of this project. The work on evaluation presented in the workshop submission had a great impact on the evaluation chapter in this report. The feedback received from the reviewers was used when preparing the camera-ready submission for the conference. The reviews provided many useful insights that will be taken into account in the future work on this project. The issue with the submission that was most frequently brought up was that the investigation of the performance discrepancy between Femto-Containers and µBPF VMs was lacking. Additionally the reviewers found that the number of benchmarks considered in our evaluation could be improved. Those limitations were addressed in the final version of this report.

µBPF VM Overhead

Our evaluation has shown that although execution of JIT-compiled programs provides a significant performance improvement over the VM, µBPF VM is less performant than Femto-Containers. We have optimised its instruction parsing and memory access checks compared to baseline `rbpf`, and we believe the remaining discrepancy is due to the highly-optimised jumptable approach used by the Femto-Containers [52] VM interpreter. An additional factor is the µBPF VM initialization overhead caused by the larger number of available configuration options (e.g. binary formats). We are planning to further optimise our implementation to close the performance gap.

7.2 Conclusion

In this paper we present µBPF , an eBPF VM, JIT compiler and a deployment framework allowing for compartmentalizing microcontrollers running RIOT. We demonstrate that it is possible to perform JIT compilation on low-end embedded hardware. We evaluate the execution time and program size overhead of the solution and compare it against native code and existing VM alternatives. We demonstrate that JIT-compiled code achieves close-to-native performance and decreases the size of program binaries up to 50%.
We use our framework to build an example application which demonstrates how µBPF can be used to deploy a compartmentalized system where business logic of different components is isolated and can be independently updated over-the-air.
A subset of the contributions of this work was merged back into the upstream repository of `rbpf` and an early version of this report is one of the accepted papers for the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions.

7.3 Future Work

Preparing a Presentation for the eBPF '24 Workshop The final version of the presentation for this project will be adapted and featured at the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions in August 2024.

Removing VM Execution Discrepancy The biggest issue with the current implementation of µBPF VM is that its performance is worse than the baseline Femto-Container VM. The JIT-compiler makes up for it, however in the future we are planning to further optimise our implementation of the VM to remove the performance discrepancy.

eBPF Hooks One of the possible use-cases suggested by [52] was to load eBPF programs into pre-provisioned hook points scattered throughout RIOT kernel, given that our JIT compiled programs achieve native-like performance, it remains to implement hook infrastructure to allow for loading jit-compiled programs similarly to what happens in the Linux kernel and evaluate performance of the solution.

Fully Decoupled Architecture Related work [13] suggests that under trusted environments, decoupling the verification and JIT compilation from the embedded device can lead to significant performance improvements. Given that we already have infrastructure in place allowing for relocating ELF files on the target device, and our JIT compiler can also be run on desktop-hardware, we could explore implementing a solution outlined in the prior work.

Improving JIT Compatibility As outlined in the JIT compiler implementation section, our design does not fully support handling 64-bit integer values. This is because the registers of ARMv7-eM architecture are 32-bit wide. There exist solutions simulating 64-bit registers on the stack [50] and we are planning to implement that feature in our compiler.

Bibliography

[1] eBPF Documentation, 2024. URL `https://ebpf.io/what-is-ebpf/`. pages 5, 11, 13

[2] Neshenko N, Bou-Harb E, Crichigno J, Kaddoum G, and Ghani N. Demystifying IoT security: an exhaustive survey on IoT vulnerabilities and a first empirical look on internet-scale IoT exploitations. *IEEE Communications Surveys & Tutorials*, Jan 2019. doi: 10.1109/COMST.2019.2910750. pages 5

[3] Colling H. Why TinyML is a giant opportunity, 2020. URL `https://venturebeat.com/2020/01/11/why-tinyml-is-a-giant-opportunity/`. pages 5

[4] Amar S, Chisnall D, Chen T, Filardo NW, Laurie B, Liu K, et al. CHERIoT: Complete memory safety for embedded devices. In *The 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), 28 October 2023, Toronto, ON, Canada*, page 641–653, Toronto, ON, Canada, 2023. ACM. URL `https://doi.org/10.1145/3613424.3614266`. pages 5, 8, 12, 13

[5] Pinto S and Santos N. Demystifying Arm TrustZone: A comprehensive survey. *ACM Comput. Surv.*, 51, January 2019. doi: https://doi.org/10.1145/3291047. pages 5, 8, 9, 18, 19

[6] Gavrin E, Lee SJ, Ayrapetyan R, and Shitov A. Ultra lightweight JavaScript engine for internet of things. In *SPLASH Companion 2015: Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, Pittsburgh, PA, USA*, pages 19–20, New York, NY, USA., 2015. ACM. URL `https://doi.org/10.1145/2814189.2816270`. pages 5, 6, 17

[7] Baccelli E, Doerr J, Kikuchi S, Padilla FA, Schleiser K, and Thomas I. Scripting over-the-air: towards containers on low-end devices in the internet of things. In *The 2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, page 504–507, Athens, Greece, 2018. IEEE. pages 5, 6, 8, 9, 16, 17, 18, 19, 20

[8] Zandberg K and Baccelli E. Minimal virtual machines on IoT microcontrollers: The case of Berkeley packet filters with rBPF. In *9th IFIP/IEEE PEMWN*, pages 1–6, 12 2020. URL `https://arxiv.org/pdf/2011.12047.pdf`. pages 5, 6, 8, 15, 18, 20, 21, 22, 23, 24, 26, 28, 48, 51, 55

[9] Haas A, Rossberg A, Schuff DL, Titzer BL, Holman M, Gohman D, et al. Bringing the web up to speed with WebAssembly. In *The 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17), June 18–23, 2017, Barcelona, Spain.* ACM, 2017. URL `http://dx.doi.org/10.1145/3062341.3062363`. pages 5

[10] Baccelli E, Gündoğan C, Hahm O, Kietzmann P, Lenders MS, Petersen H, et al. RIOT: An open source operating system for low-end embedded devices in the IoT. *IEEE Internet of Things Journal 5, 6*, pages 4428–4440, 2018. pages 5, 6, 8, 10

[11] Zandberg K, Baccelli E, Yuan S, Besson F, and Talpin JP. Femto-containers: lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers. In *23rd ACM/IFIP International Middleware Conference (Middleware '22), November 7–11, 2022, Quebec, QC, Canada*, pages 161–173, New York, NY, USA, 2022. ACM. URL `https://doi.org/10.1145/3528535.3565242`. pages 5, 6, 9, 13, 15, 17, 19, 20, 21, 23, 24, 26, 28, 48, 49, 51

[12] Monnet Q. rbpf - Rust (user-space) virtual machine for eBPF, 2017. URL `https://github.com/qmonnet/rbpf`. pages 5, 6, 13, 14, 15, 22, 23, 25, 28, 30, 31, 35, 36, 48, 49, 51

[13] Craun M, Oswald A, and Williams D. Enabling eBPF on embedded systems through decoupled verification. In *The 1st Workshop on eBPF and Kernel Extensions (eBPF '23), September 10, 2023, New York, NY, USA*, pages 63–69, New York, NY, USA, 2023. ACM. URL `https://doi.org/10.1145/3609021.3609299`. pages 5, 6, 13, 14, 19, 22, 24, 25, 31, 51, 58

[14] Kumar S, Tiwari P, and Zymbler M. Internet of things is a revolutionary approach for future technology enhancement: a review. *J Big Data*, 6:111, 2019. URL `https://doi.org/10.1186/s40537-019-0268-2`. pages 7

[15] Zandberg K, Acosta K, Schleiserand F, and Baccelli H, Tschofenigand E. Secure firmware updates for constrained IoT devices using open standards: A reality check. *IEEE Access*, 7: 71907–71920, 2019. doi: 10.1109/ACCESS.2019.2919760. pages 7, 10, 17, 26, 29

[16] Neumann, PG Moore, SW Anderson, J Chisnall, D et al. Watson, RNM, Woodruff, J. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37, 2015. doi: 10.1109/SP.2015.9. pages 8

[17] Foukas X, Radunovic B, Balkwill M, and Lai Z. Taking 5G RAN analytics and control to a new level. In *The 29th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '23), October 2-6, 2023, Madrid, Spain*, pages 1–16, New York, NY, USA, 2023. ACM. URL `https://www.microsoft.com/en-us/research/uploads/prod/2022/12/mobicom23-final9.pdf`. pages 8, 19, 22

[18] Ratiu A. An eBPF overview, part 4: Working with embedded systems., 2019. URL `https://www.collabora.com/news-and-blog/blog/2019/05/06/an-ebpf-overview-part-4-working-with-embedded-systems/`. pages 8, 24

[19] Athalye A, Kaashoek MF, Zeldovich N, and Tassarotti J. The K2 architecture for trustworthy hardware security modules. In *The 1st Workshop on Kernel Isolation, Safety and Verification (KISV '23), October 23, 2023, Koblenz, Germany*, pages 26–32, Koblenz, Germany, 2023. ACM. URL `https://dl.acm.org/doi/pdf/10.1145/3625275.3625402`. pages 9

[20] Javier F, Padilla A, Baccelli E, Eichinger T, Schleiser K, Fj A. The future of IoT software must be updated. In *IAB Workshop on Internet of Things Software Update (IoTSU), Jun 2016*,

Dublin, Ireland. Internet Architecture Board (IAB). URL `https://inria.hal.science/hal-01369681`. pages 9

[21] Dunkels A, Finne N, Eriksson J, and Voigt T. Run-time dynamic linking for reprogramming wireless sensor networks. In *The 4th ACM Conference on Embedded Networked Sensor Systems (SenSys '06)*, Boulder, Colorado, USA., 2006. ACM. doi: 10.1145/1182807.1182810. pages 9, 18

[22] Jeong J and Culler D. Incremental network programming for wireless sensors. *International Journal of Communications, Network and System Sciences*, 02(05):433, 08 2009. doi: 10.4236/ijcns.2009.25048. pages 9, 18

[23] Shelby Z, Hartke K, and Bormann C. RFC 7252: Constrained application protocol (CoAP). *IETF Request For Comments*, 2014. pages 10, 44

[24] IEEE standard for information technology—telecommunications and information exchange between systems local and metropolitan area networks—specific requirements - part 11: Wireless lan medium access control (MAC) and physical layer (PHY) specifications. *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)*, pages 1–3534, 2016. doi: 10.1109/IEEESTD.2016.7786995. pages 10

[25] Amsüss C and the aiocoap contributors. aiocoap – the python coap library, 2024. URL `https://aiocoap.readthedocs.io/en/latest/index.html`. pages 10, 30, 32

[26] Das, T. Java developer's guide: Oracle JVM just-in-time compiler (JIT), 2022. URL `https://docs.oracle.com/en/database/oracle/oracle-database/19/jjdev/Oracle-JVM-JIT.html`. pages 10

[27] Fleming M. A thorough introduction to eBPF., 2017. URL `https://lwn.net/Articles/740157/`. pages 11

[28] Rice L. *Learning eBPF.* O'Reilly Media, Sebastopol, CA, USA, 2023. pages 11, 12, 14

[29] Kocher P, Horn J, Fogh A, Genkin D, Gruss D, Haas W, et al. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy*. IEEE, 2019. doi: 10.1109/SP.2019.00002. pages 12, 19, 26

[30] Fletcher J. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications*, 30:247–252, Jan 1982. doi: 10.1109/TCOM.1982.1095369. pages 13, 19, 48

[31] Gershuni E, Amit N, Gurfinkel A, Narodytska N, Navas JA, Rinetzky N, et al. Simple and precise static analysis of untrusted Linux kernel extensions. In *The 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22ś26, 2019, Phoenix, AZ, USA.*, pages 1–16, New York, NY, USA, 2019. ACM. URL `https://doi.org/10.1145/3314221.3314590`. pages 13

[32] Lane R. uBPF - userspace eBPF VM, 2015. URL `https://github.com/rlane/ubpf`. pages 14, 23

[33] RIOT: The friendly operating system for the internet of things., 2024. URL `https://www.riot-os.org/`. pages 15, 28, 29, 43

[34] Haberlandt A. Porting the Solana eBPF JIT compiler to ARM64, 2022. URL `https://blog.trailofbits.com/2022/10/12/solana-jit-compiler-ebpf-arm64/`. pages 15, 22, 28

[35] S Yuan, B Lion, F Besson, and JP Talpin. Making an eBPF virtual machine faster on microcontrollers: Verified optimization and proof simplification. In *Dependable Software Engineering. Theories, Tools, and Applications*, pages 385–401, Singapore, 2024. Springer Nature Singapore. pages 16, 19, 24, 35, 37

[36] Ronen E and Shamir A. Extended functionality attacks on IoT devices: The case of smart lights (invited paper). In *2016 IEEE European Symposium on Security and Privacy*. IEEE, 2016. doi: DOI10.1109/EuroSP.2016.13. pages 18

[37] Soltan S, Mittal P, and Poor HV. BlackIoT: IoT botnet of high wattage devices can disrupt the power grid. In *The 27th USENIX Security Symposium*, volume 18. USENIX, 2018. URL `https://www.usenix.org/conference/usenixsecurity18/presentation/soltan`. pages 18

[38] Levy A, Campbell B, Ghena B, Giffin DB, Pannuto P, Dutta P, et al. Multiprogramming a 64 kB computer safely and efficiently. In *The 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, New York, NY, USA, 2017. ACM. URL `https://doi.org/10.1145/3132747.3132786`. pages 18, 22

[39] Shymanskyy V. Wasm3 - a fast WebAssembly interpreter and the most universal wasm runtime. URL `https://github.com/wasm3/wasm3/tree/main`. pages 19

[40] AtomicObject. heatshrink - a data compression/decompression library for embedded/real-time systems., 2013. URL `https://github.com/atomicobject/heatshrink`. pages 19

[41] Fox GC, Ishakian V, Muthusamy V, and Slominski A. Status of serverless computing and function-as-a-service(FaaS) in industry and research. *CoRR*, abs/1708.08028, 2017. URL `http://arxiv.org/abs/1708.08028`. pages 20

[42] Espressif. Espressif IoT Development Framework. URL `https://github.com/espressif/esp-idf`. pages 22

[43] Zheng Y, Yu T, Yang Y, Hu Y, Lai X, and Quinn A. bpftime: userspace eBPF runtime for uprobe, syscall and kernel-user interactions, 2023. pages 23, 28

[44] Robertson A. bpftrace. URL `https://github.com/bpftrace/bpftrace`. pages 23

[45] The kernel development community. libbpf overview, 2024. URL `https://docs.kernel.org/bpf/libbpf/libbpf_overview.html`. pages 23

[46] IO Visor. BPF compiler collection., 2024. URL `https://github.com/iovisor/bcc`. pages 24

[47] Cilium Authors. BPF architecture, 2024. URL `https://docs.cilium.io/en/latest/bpf/architecture/`. pages 25, 30, 39, 42

[48] The kernel development community. eBPF verifier, 2024. URL `https://docs.kernel.org/bpf/verifier.html`. pages 31, 51

[49] Arm v7-M architecture reference manual, 2021. URL `https://developer.arm.com/documentation/ddi0403/latest/`. pages 39, 43

[50] S Bansal. arm: eBPF JIT compiler, 2017. URL `https://lwn.net/Articles/723872/`. pages 42, 58

[51] HD44780 LCD driver, 2024. URL `https://doc.riot-os.org/group__drivers__hd44780.html`. pages 44

[52] Zandberg K. Femto-Containers RIOT implementation. URL `https://github.com/future-proof-iot/middleware2022-femtocontainers/tree/main/femto-containers`. pages 44, 45, 49, 53, 57, 58

[53] The kernel development community. Submitting patches: the essential guide to getting your code into the kernel, 2024. URL `https://docs.kernel.org/process/submitting-patches.html#`. pages 57