µBPF: Using eBPF for Microcontroller Compartmentalization

Szymon Kubica
Imperial College London
szymon.kubica20@imperial.ac.uk

Marios Kogias
Imperial College London
m.kogias@imperial.ac.uk

ABSTRACT

Although eBPF (Extended Berkeley Packet Filter) started as a virtualization technology used in the Linux kernel to allow for executing user code inside the kernel in a safe way, it is a general purpose software fault isolation technology. The specification of 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 don't 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 embedded IoT applications. Our evaluation shows that µBPF JIT achieves close-to-native performance and up to of 50% code size reduction compared to the eBPF binaries.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Real-time operating systems*; • **Software and its engineering** → **Embedded software**; *Software verification and validation*;

KEYWORDS

eBPF, Microcontroller, Embedded Systems, Fault Isolation, Compartmentalization, Middleware, Virtual Machine, Internet of Things

ACM Reference Format:
Szymon Kubica and Marios Kogias. 2024. µBPF: Using eBPF for Microcontroller Compartmentalization. In *Workshop on eBPF and Kernel Extensions (eBPF '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3672197.3673433

1 INTRODUCTION

Extended Berkeley Packet Filter (eBPF [2]) 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.

Despite its origins, though, eBPF can be viewed as a generic compartmentalization technology applicable to different settings ranging from datacenter servers to resource-constrained IoT devices. In any compartmentalized environment, a system or an application is split into isolated components (compartments) that interact with each other over safe communication channels. This allows for achieving fault isolation where a malfunction in one compartment does not propagate to the rest of the system.

Such compartmentalization mechanisms are becoming more important in the context of networked-enabled embedded devices, whose number increases rapidly [18]. As those devices become increasingly connected, the attack surface grows. To isolate faults caused by potential attackers, a secure mechanism for compartmentalized execution of programs is needed. It is a challenging problem to solve given the limited hardware resources, various instruction set architectures (ISA) used by the microcontrollers, and in many cases the lack of virtual memory.

Existing solutions to this problem either involve proposing novel instruction set architectures, such as CHERIoT [5], or running VM environments on embedded hardware [8, 13, 17]. A good candidate for such a VM environment is eBPF, since its compile-verify-execute workflow translates well into this problem domain. Among other VM solutions (e.g. WASM [14], or JavaScript [13]), eBPF is particularly suitable for resource constrained hardware because its ISA is simple and supports verification. The simplicity of the ISA means that the RAM and ROM requirements of an eBPF VM are much smaller compared to the alternatives. Support for verification allows for ensuring that the executed programs are safe directly on the target device. Femto-Containers [22] - the current state-of-the-art, explores the idea of a lightweight container-like virtualization mechanism for IoT devices based on eBPF.

Prior work suffers from the following limitations of using eBPF as a virtualization solution on microcontrollers. 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 [17] two times slower than using a WASM3 interpreter. ii) The eBPF program size 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. For example, programs containing data relocations [17] or read-only data [19] are not supported. iv) Verification capabilities are also, limited. Verifiers used by the available eBPF VM implementations are simple and do not support restricting access to eBPF helper functions.

eBPF '24, August 4–8, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0712-4/24/08.
https://doi.org/10.1145/3672197.3673433

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 [8]. µBPF is a VM, JIT compiler, and a deployment framework for compartmentalized code execution on microcontrollers. We design µBPF on top of RIOT [9] - a widely used real-time operating system for embedded devices, and implement it in Rust. Our evaluation shows that code emitted by µBPF JIT achieves close-to-native performance and up of 50% program size reduction compared to eBPF binaries.

The key contributions in this work are:

- We implement an eBPF-to-ARMv7 JIT compiler compatible with devices running ARM Cortex M CPUs.
- We design a mechanism for restricting eBPF program access to helper functions provided by the host OS.
- We build a toolkit for compilation and deployment of eBPF programs for microcontrollers running RIOT.

2 BACKGROUND & RELATED WORK

eBPF Bytecode. The eBPF VM executes programs represented by eBPF bytecode instructions. This bytecode is generated by compiling a restricted version of C (or other eBPF front-ends for e.g. Python [16]) into the eBPF bytecode. To ensure that eBPF programs can be executed safely inside the kernel, the expressiveness of the instruction set is limited. This allows for verification using static analysis. In particular, eBPF does not allow for indirect branch or jump instructions. Each jump instruction has a specified offset relative to the program counter that is known before runtime. This limitation ensures that the sandboxed execution does not jump out of the program loaded into the VM.

The eBPF ISA specifies a fixed-size stack of 512 B and its specification does not include a dynamically allocated heap. This is desirable on embedded devices with memory size constraints.

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 - Femto-Containers [22] uses eBPF in a similar way on embedded devices by allowing to host multiple software functions on a single microcontroller. It provides proper isolation, secure deployment and hardware abstraction for the hosted functions.

eBPF for Microcontrollers. The above specification of eBPF means that it can be used on devices where memory is heavily constrained. By contrast, WASM3 requires at least 64KiB of RAM [14], but can be twice as fast [22]. An eBPF VM can be relatively simple, Femto-Containers VM uses approximately 500 lines of C code. This results in a smaller ROM footprint compared to using script interpreters (rBPF - 5 KiB vs Micropython - 100 KiB [17]).

These memory requirements need to be evaluated in the context of the target hardware. Among microcontrollers supported by RIOT [9], their memory budgets range from 2 KiB RAM / 32 KiB ROM (Arduino Uno) to 256 KiB RAM / 2 MiB ROM (STM32F4) [9]. This means that high memory requirements of a virtual machine can reduce its compatibility.

Unfortunately, the compatibility of existing eBPF VM solutions for microcontrollers is lacking. Femto-Containers [22] VM, uses a custom binary format with a header specifying the section offsets within the program binary. This design couples the interpreter implementation with the binary format limiting compatibility. Consequently, eBPF object files need to be passed through a custom bytecode patching script to make them compatible with the VM. Additionally, the VM implementation does not support data relocations or program-counter-relative function calls which are included in the eBPF specification.

Real-Time Operating Systems (RTOS). Although related work [11] considers a model where embedded devices run a Linux kernel and use its eBPF subsystem, we target microcontrollers running a lightweight real-time operating system such as RIOT [9]. Real-time operating systems need to guarantee that the time to complete a given operation is bounded and can be reasoned about. RIOT provides soft real-time guarantees by using priority-based scheduling and maintaining interrupt latency at approximately 50 cycles [4].

It is important to note that existing solutions for running eBPF code in RIOT [17, 22] provide no real-time guarantees because of the VM interpreter overhead .

SUIT Firmware Update Protocol. Software Updates for Internet of Things [23] defines a unified update format for IoT devices. Performing updates of networked embedded devices is difficult because of a wide range of network protocols in use. Furthermore, the devices originate from diverse vendors often with vendor-specific update management systems. SUIT defines core operations that a particular update mechanism should provide (e.g. verify device identity, verify updated image, fetch the image). In RIOT [9] the SUIT update workflow involves generating a manifest file specifying all metadata required to securely load the new firmware image. The manifest is then signed using encryption keys matching the ones used by the running OS.

µBPF uses RIOT's SUIT subsystem to allow for secure deployment and updates of eBPF programs. Our deployment framework was built around infrastructure used by the current state-of-the-art [17] and [22].

CoAP. µBPF deployment framework uses the Constrained Application Protocol (CoAP) [20] to communicate with the target devices. CoAP is designed for resource-constrained nodes and networks (e.g. low-power). Typically, the nodes are embedded devices with memory constraints, and the networks used for communication suffer from low throughput and high packet loss rates. CoAP is based on UDP and provides a request/response model similar to HTTP. CoAP is widely used in IoT applications and most contemporary RTOS implementations such as RIOT provide support for it.

Other eBPF Use Cases. `rbpf` [19] is a userspace eBPF VM together with a JIT compiler for the x86 ISA. The `rbpf` VM is implemented in Rust and was used as a base for µBPF. It is different from rBPF [17] - an eBPF subsystem for RIOT on which Femto-Containers VM implementation was based. A fork of `rbpf` is used by the Solana blockchain in a smart contract execution system [15]. This version, also, provides an ARM64 JIT compiler which was a point of reference for the ARMv7-eM JIT compiler used by µBPF.

3 DESIGN

Deployment Model. µBPF divides the process of deploying eBPF programs into two steps: **deployment stage** and **execution stage**. The first stage involves compiling (3.1.1), verifying (3.1.3) and loading (3.1.2) the program into memory of the target device. After that, in the execution stage, clients can send requests to run previously-deployed programs.

3.1 Deployment Pipeline

The deployment pipeline of µBPF consists of four steps: compilation, signing, firmware upload, and verification. Figure 1 shows the pipeline. Grey boxes represent existing infrastructure, whereas the contribution of µBPF is marked in blue.

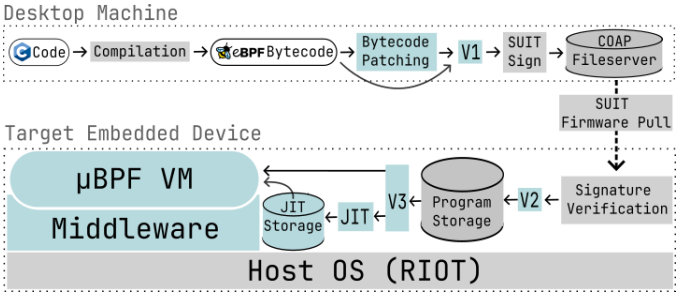

Figure 1: µBPF deployment pipeline.

3.1.1 ***Compilation and Bytecode Patching***. At the start of the pipeline, source files in C are compiled into eBPF bytecode. After that follows an optional bytecode patching step required for backwards compatibility with Femto-Containers. µBPF supports 4 different eBPF binary formats (see 4.1) so this step can be skipped and a raw object file can be fed into the next step of the pipeline.

3.1.2 ***SUIT Firmware Update***. Next step involves sending the program binary to the target device using the SUIT update workflow provided by RIOT. First, the produced binaries are signed with encryption keys matching the ones stored in the OS image running on the target device. 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 [6]. The manifest provides information required by the target device to verify that the loaded program has not been tampered with and originates from a trusted source. The device then 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 RIOT's SUIT subsystem.

Middleware Layer. While being executed, an eBPF program can access a set of middleware functions to interact with the underlying OS. These functions are equivalent to eBPF helper function calls [7] and enable program logic to interact with embedded hardware by e.g. reading data from sensors, controlling actuators or sharing data with other programs using the shared global storage (similar to eBPF maps). The set of available helper functions can be extended to match application-specific requirements.

3.1.3 ***Verification***. Malformed programs or a malicious use of helper functions can compromise security of the system, to solve this µBPF can be configured to verify eBPF bytecode at one of the three predefined points in the deployment pipeline (`V1`, `V2`, `V3` in Figure 1). The verifier implementation is based on `rbpf` [19]. It checks the validity of instructions and jump offsets and verifies that only calls to allowed helper functions are called. The VM validates all memory accesses at runtime. In contrast to the Linux kernel verifier[21], µBPF does not traverse all possible program paths.

We consider the following threat model:

- **fully trusted source and clients** - verification happens at only `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.

The number of program storage slots is configured at compile time and fixed. Because of this, programs from potentially malicious sources are verified at `V2` before being loaded into the storage to ensure that invalid programs do not waste RAM space.

A subset of helper functions implemented for µBPF, if used incorrectly, can be harmful to the system. For instance, helper functions directly accessing the GPIO pins of the device can interfere with peripherals connected to those pins. Therefore, these helpers should be used only by programs executed by privileged, trusted clients. In our model a malicious client might try to execute a program accessing helpers that are available only to privileged users. Based on client's privilege level, µBPF encodes the list of allowed helper functions in the execution request and the server verifies (at `V3`) that the program only calls functions included in this list.

3.2 VM Execution and JIT Stage

After the deployment stage is complete, clients can begin sending requests to start executing the loaded programs. 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 Figure 1). Upon receiving a request to rerun the program, the compilation process can be skipped.

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 is done, the original eBPF program can be discarded allowing to save memory.

4 IMPLEMENTATION

We implement µBPF based on `rbpf` [19] - a userspace eBPF VM written in Rust. The reason for this choice was good support for Rust in RIOT and existing JIT compiler implementations for `rbpf` targeting 64-bit architectures [15]. The design of those compilers was used to drive the implementation of the µBPF JIT compiler.

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.

Middleware Implementation. We implement a middleware layer for RIOT allowing for full compatibility with Femto-Containers. This base functionality is extended with wrappers around drivers for peripherals such as the HD44780 LCD display [3] or more privileged operations such as GPIO pin access.

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 implemented by dynamically attaching helper functions to a given instance of the µBPF VM before execution. This flexibility however comes at a cost, as this is handled by storing helper function pointers in a `BTreeMap`, with $O(\log n)$ lookup time complexity. A standard `HashMap` is not available because of the lack of the Rust standard library on the target embedded devices.

4.1 Supported Binary Formats

The space of programs supported by the existing solutions is limited. The original version of `rbpf` [19] supports programs containing only the text section extracted out of the eBPF binaries. Because of this, programs that contain read-only data such as strings for printing debug information are not supported. A further limitation of Femto-Containers VM is that data relocations are not supported. This means that programs similar to the one in Listing 1 can't be executed using the existing solutions.

Listing 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 1 the variable `ptr` is initialised to hold the address of the variable `c`. The first two statements should print the address of c, whereas the last one should print its value. 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 be resolved by a relocation mechanism before executing the program. The limitation of the Femto-Containers VM is that it does not implement such a relocation resolution procedure. Consequently, the value of `ptr` will be set to 0, and the program won't be executed correctly.

To overcome these limitations we design µBPF to be compatible with the following binary formats:

- **Only `.text` section** - no support for read-only data, smallest binaries, backwards-compatible with `rbpf`.
- **Femto-Containers** - compatible with Femto-Containers, uses a custom header and patched bytecode.
- **Extended header** - allowed helper functions are encoded directly in the program binary
- **Raw object file** - ELF files generated by LLVM without the debug information, support data relocations.

We implement a portable library for resolving relocations and bytecode patching. It allows for generating binaries that are backwards-compatible with Femto-Containers or `rbpf` and resolving relocations directly on the target device.

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, as it supports data relocations. However, it uses raw ELF files generated by LLVM. Even after removing debug symbols, this format needs to contain relocation tables which incurs a program size overhead compared to the Femto-Containers binaries.

JIT Compilation Between 64 and 32-bit ISAs. We implement the µBPF JIT aiming to reduce the size of the program binaries. The target architecture: ARMv7-eM runs on ARM Cortex M4 CPUs available on off-the-shelf embedded hardware. This ISA provides support for Thumb 16-bit and 32-bit instruction encodings [1]. The µBPF JIT compiler prioritises using shorter 16-bit instruction encodings to reduce the size of the emitted machine code.

eBPF is a 64-bit ISA with 64-bit registers, meanwhile, ARMv7 uses 32-bit registers. In some cases this results in eBPF compiler emitting bytecode that can't be translated into ARM. For instance, a logical shift left by 32 bits is a valid instruction in eBPF. Such instructions are emitted by the compiler e.g. before comparing 32-bit integers. On ARMv7 however, this instruction would flush the register, effectively setting it to zero, which is not always the expected behaviour. The µBPF JIT compiler works around this by truncating shift values modulo 32. It trades compatibility for performance by only supporting 32-bit integer values and translating 64-bit variants of eBPF instructions into their 32-bit equivalents. Related work on an eBPF-to-ARM32 JIT compiler in the Linux kernel simulates 64-bit registers on the stack [10] achieving better compatibility at the cost of implementation complexity and runtime performance.

5 EVALUATION

5.1 Performance Analysis

In this section we evaluate performance of µBPF running on STM32-F439ZI - a microcontroller with an ARM Cortex M4 CPU, 256 KiB of RAM and 2 MiB of flash storage. We perform evaluation against benchmarks used in prior work [17, 22].

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 [12]. It performs a number of arithmetic operations, memory accesses and branch instructions, while looping over a 640 B string to calculate its checksum. It aims to simulate a representative workload of processing sensor data on a microcontroller [22].

We compare performance against three baselines: native C, Femto-Containers [22] and `rbpf` [19] VMs. Results of the benchmark can be seen in Table 1.

	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 VM	120 µs	20 µs	3860 µs	4010 µs
µBPF JIT	1944 µs	101 µs	144 µs	2190 µs

Table 1: Fletcher 16 on 640 B string execution time.

We find that the execution of JIT-compiled code achieves native performance. However, it comes at a load-time cost of approx. 2000 µs. Note that in our programming model, loading and verification is performed once, during the deployment stage. Because of this, the compilation cost is amortized over multiple executions of the program.

Execution Time Overhead. We measure the VM overhead compared to native execution depending on the computation size. This is achieved by running a program calculating the Fletcher16 checksum of strings of sizes varying between 80 B and 640 B.

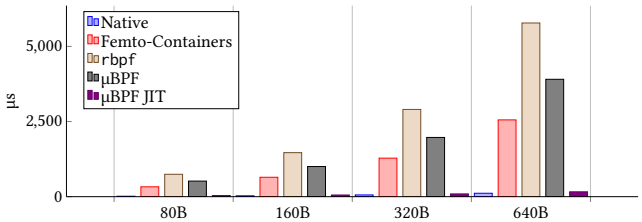

Figure 2: Fletcher16 execution time for 80 B-640 B data.

Execution time in µs of different solutions is shown in Figure 2. We observe that µBPF is about 33% faster than the baseline `rbpf` , This was achieved after optimising the instruction parsing and runtime memory access checks performed by `rbpf`. However, Femto-Containers VM is still approx. 33% faster than µBPF. This is because of the additional configurability with respect to the program binary format and allowed helper functions provided by µBPF which incurs an overhead.

To simulate real-world deployment scenarios, we also benchmark program logic used for the example application in 5.2 . The first program - CoAP response formatter [22] is triggered upon receiving a request from the client to read temperature data. It reads the sensor data and writes it into the packet buffer that is then sent back to the client. The second program computes a moving average of the sensor data in the global storage.

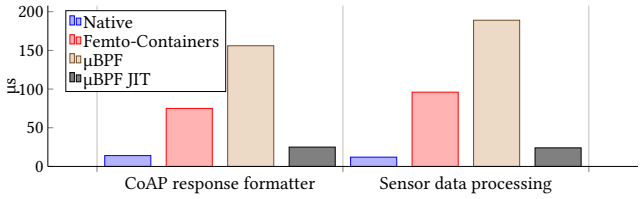

Figure 3: Example application logic - execution time.

Here we observe that the overhead of using µBPF VM is higher than in the previous benchmark. This is because the above programs mainly consist of helper function calls, as they interact with the global state of the device (sensors, storage). The helper functions in µBPF VM are dynamically configured, thus they need to be maintained in a `BTreeMap`, which is slower than a `switch` statement used by Femto-Containers (as in that case all helper functions are specified during compilation). Additionally, µBPF VM has a higher initialization overhead compared to Femto-Containers because of the larger number configuration options (e.g. specifying the binary format). The initialization time impacts performance in this benchmark because the programs are relatively short.

Verification Time. We observe that when using the JIT compiler, the verification takes marginally longer (see Table 1). This is because the compiler operates on raw object files, and their header metadata takes longer to parse compared to the custom, simplified formats used by other VMs. However the total time of 100 µs negligible compared to 2000 µs it takes to perform JIT compilation.

JIT Compilation Overhead. We observed that executing the code emitted by µBPF JIT compiler achieves performance comparable to native C. However, we need to ensure that the cost of compilation is not prohibitively expensive. Table 1 shows the load time (including JIT compilation) for the Fletcher 16 algorithm logic. Although the JIT load time is larger compared to the alternatives, it is still acceptable and results in a lower total time compared to VM interpreters. Moreover, this cost can be amortized over multiple executions by compiling once and storing the program in the JIT program storage.

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. Once the compilation is complete, we only need the native machine code, and the eBPF program can be discarded allowing to save space.

We measure the program size requirement across 10 example programs (see Figure 4) 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 measured a maximum of 57% program size reduction. The last three are the benchmarked Fletcher16 algorithm and logic used by the example application in 5.2.

We observe that in case of programs that consist of primarily 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 (e.g. `fletcher16` in Figure 4 - the program mainly consists of the 640 B string stored in its `.data` section).

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 4, when performing JIT compilation, the space occupied by the two buffers is equal to the sum of the blue and grey bars as the JIT uses the raw-object-file binary format.

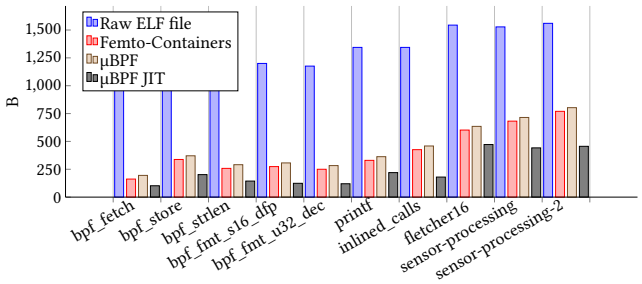

Figure 4: Example program sizes across binary formats.

JIT vs VM Single Execution Time. µBPF allows for saving JIT compiled programs in a memory storage to be reused multiple times thereby amortizing the cost of transpilation. To find the point at which using the JIT is faster 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.

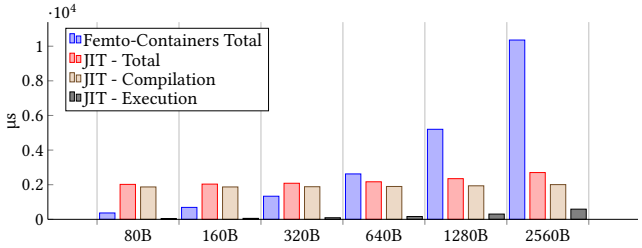

Figure 5: JIT vs VM for a single Fletcher16 execution.

Figure 5 shows that for small computation sizes, the JIT compilation time dominates, and the total time using 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 time for the JIT is shorter than the one for the VM.

5.2 Example Application

We implement an example application to demonstrate how µBPF can be used to deploy a compartmentalized system on a microcontroller. Figure 6 depicts the application consisting of three compartments: two responsible for collecting sensor data and one for controlling an LCD display. The compartments are isolated by running in three instances of the µBPF VM on separate threads.

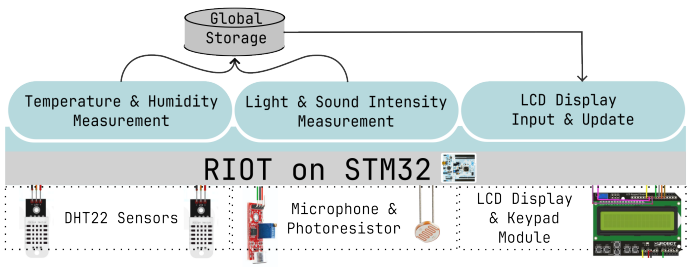

Figure 6: Example application deployed with µBPF.

Compartmentalization is achieved by allowing each of the programs to access the minimum set of helper functions required to perform its task. Programs collecting data can only call helpers to read sensor measurements and write them into the global storage. The LCD display compartment is only allowed to read from the storage, format and display the sensor data, and collect user input from the keypad module to control the displayed user interface.

If, for instance, an error in the display module logic causes it to crash after receiving a particular input, this fault won't affect the other compartments. The VM running the display program will handle the fault and exit gracefully. A new, patched version of the display logic can then be redeployed without interfering with the execution of the other, already running, programs.

6 DISCUSSION

µBPF VM Overhead. JIT-compiled programs provide a significant performance improvement over the VM, however µBPF VM is less performant than Femto-Containers. After optimising its instruction parsing and memory access checks compared to the baseline `rbpf`, the discrepancy still remains.

We identify three reasons for this difference: i) µBPF supports more configuration options, resulting in a larger initialization overhead. This is particularly evident for shorter programs (e.g. Figure 3). ii) µBPF allows for specifying allowed helper functions by populating a `BTreeMap` before program execution. At runtime, looking up helper functions from this map is slower than the hard-coded switch statement with all supported functions used by Femto-Containers. iii) Femto-Container VM uses a hand-written jumptable interpreter to iterate over and simulate execution of program instructions. This is faster than the interpreter used by µBPF based on a `match` statement. This is because it matches on the instruction opcode values which are non-consecutive integer values. Hence, the compiler is not able to generate a jump table automatically.

µBPF VM trades performance for compatibility and flexibility. In performance-critical applications, µBPF JIT should be used. We are currently working on a less flexible implementation of the VM to close the performance gap.

Limited JIT Compatibility. The current implementation of the µBPF JIT compiler only suppors programs operating on 32-bit integers. This is because of the constraints imposed by the 32-bit target architecture of the compiler (ARMv7-eM registers are 32-bits long). This limits compatibility of the solution. In the Linux kernel there exists an eBPF JIT compiler implementation targeting a 32-bit architecture, which simulates 64-bit registers on the stack [10]. We are planning to support a similar solution in the future. An important property to consider when evaluating possible solutions is the runtime performance cost of simulating registers on the stack.

7 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 alternatives. We demonstrate that JIT-compiled code achieves close-to-native performance and decreases the size of program binaries up to 50%.

ACKNOWLEDGEMENTS

We thank our anonymous reviewers for their insightful comments and helpful feedback. We would like to thank Ce Guo for reviewing an early version of this paper and providing useful feedback. We also thank Saleem Rashid for insightful discussions held over the course of the project. This work does not raise any ethical issues.

REFERENCES

[1] 2021. Arm v7-M Architecture Reference Manual. (2021). https://developer.arm.com/documentation/ddi0403/latest/
[2] 2024. eBPF Documentation. (2024). https://ebpf.io/what-is-ebpf/
[3] 2024. HD44780 LCD driver. (2024). https://doc.riot-os.org/group__drivers__hd44780.html

[4] 2024. RIOT: The friendly Operating System for the Internet of Things. (2024). https://www.riot-os.org/

[5] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. 2023. CHERIoT: Complete Memory Safety for Embedded Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 641–653. https://doi.org/10.1145/3613424.3614266

[6] Christian Amsüss and the aiocoap contributors. 2024. aiocoap – The Python CoAP library. (2024). https://aiocoap.readthedocs.io/en/latest/index.html

[7] Cilium Authors. 2024. BPF Architecture. (2024). https://docs.cilium.io/en/latest/bpf/architecture/

[8] Emmanuel Baccelli, Joerg Doerr, Shinji Kikuchi, Francisco Acosta Padilla, Kaspar Schleiser, and Ian Thomas. 2018. Scripting Over-The-Air: Towards Containers on Low-end Devices in the Internet of Things. In *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 504–507. https://doi.org/10.1109/PERCOMW.2018.8480277

[9] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S. Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wählisch. 2018. RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT. *IEEE Internet of Things Journal* 5, 6 (2018), 4428–4440. https://doi.org/10.1109/JIOT.2018.2815038

[10] Shubham Bansal. 2017. arm: eBPF JIT compiler. (2017). https://lwn.net/Articles/723872/

[11] Milo Craun, Adam Oswald, and Dan Williams. 2023. Enabling eBPF on Embedded Systems Through Decoupled Verification. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions (eBPF '23)*. Association for Computing Machinery, New York, NY, USA, 63–69. https://doi.org/10.1145/3609021.3609299

[12] John Fletcher. 1982. An Arithmetic Checksum for Serial Transmissions. *IEEE Transactions on Communications* 30, 1 (1982), 247–252. https://doi.org/10.1109/TCOM.1982.1095369

[13] Evgeny Gavrin, Sung-Jae Lee, Ruben Ayrapetyan, and Andrey Shitov. 2015. Ultra lightweight JavaScript engine for internet of things. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH Companion 2015)*. Association for Computing Machinery, New York, NY, USA, 19–20. https://doi.org/10.1145/2814189.2816270

[14] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

[15] Andrew Haberlandt. 2022. Porting the Solana eBPF JIT compiler to ARM64. (2022). https://blog.trailofbits.com/2022/10/12/solana-jit-compiler-ebpf-arm64/

[16] IO Visor. 2024. BPF Compiler Collection. (2024). https://github.com/iovisor/bcc.

[17] Zandberg Koen and Baccelli Emmanuel. 2020. Minimal Virtual Machines on IoT Microcontrollers: The Case of Berkeley Packet Filters with rBPF.. In *9th IFIP/IEEE PEMWN*. 1–6. https://arxiv.org/pdf/2011.12047.pdf

[18] Nataliia Neshenko, Elias Bou-Harb, Jorge Crichigno, Georges Kaddoum, and Nasir Ghani. 2019. Demystifying IoT Security: An Exhaustive Survey on IoT Vulnerabilities and a First Empirical Look on Internet-Scale IoT Exploitations. *IEEE Communications Surveys Tutorials* 21, 3 (2019), 2702–2733. https://doi.org/10.1109/COMST.2019.2910750

[19] Monnet Q. 2017. rbpf - Rust (user-space) virtual machine for eBPF. (2017). https://github.com/qmonnet/rbpf

[20] Zach Shelby, Klaus Hartke, and Carsten Bormann. 2014. RFC 7252: Constrained Application Protocol (CoAP). *IETF Request For Comments* (2014).

[21] The kernel development community. 2024. eBPf verifier. (2024). https://docs.kernel.org/bpf/verifier.html

[22] Koen Zandberg, Emmanuel Baccelli, Shenghao Yuan, Frédéric Besson, and Jean-Pierre Talpin. 2022. Femto-containers: lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference (Middleware '22)*. Association for Computing Machinery, New York, NY, USA, 161–173. https://doi.org/10.1145/3528535.3565242

[23] Koen Zandberg, Kaspar Schleiser, Francisco Acosta, Hannes Tschofenig, and Emmanuel Baccelli. 2019. Secure Firmware Updates for Constrained IoT Devices Using Open Standards: A Reality Check. *IEEE Access* 7 (2019), 71907–71920. https://doi.org/10.1109/ACCESS.2019.2919760