Femto-Containers: DevOps on Microcontrollers with Lightweight Virtualization & Isolation for IoT Software Modules

Koen Zandberg, Emmanuel Baccelli

Femto-Containers: DevOps on Microcontrollers with Lightweight Virtualization & Isolation for IoT Software Modules

Koen Zandberg
Inria
Saclay, France

Emmanuel Baccelli
Inria
Saclay, France
&
Freie Universität Berlin
Berlin, Germany

ABSTRACT

Development, deployment and maintenance of networked software has been revolutionized by DevOps practices, which boost system software quality and agile evolution. However, as the Internet of Things (IoT) connects low-power, microcontroller-based devices which take part in larger distributed cyberphysical systems, such low-power IoT devices are not easy to integrate in DevOps workflows. In this paper, we contribute to mitigate this problem by designing Femto-Containers, a new hardware-independent mechanism which enable the virtualization and isolation of software modules embedded on microcontrollers, using an approach extending and adapting Berkeley Packet Filters (eBPF). We implement a Femto-Container hosting engine, which we integrate in a common low-power IoT operating system (RIOT), and is thus enhanced with the ability to start, update or terminate Femto-Containers on demand, securely over a standard IPv6/6LoWPAN network. We evaluate the performance of Femto-Containers in a variety of use cases involving one or more applications simultaneously hosted on the same microcontroller. We show that Femto-Containers can virtualize and isolate software modules executed concurrently, with very small memory footprint overhead (below 10%) and very small startup time (tens of microseconds) compared to native code execution. We carry out experiments deploying Femto-Containers on a testbed using heterogeneous IoT hardware based on the popular microcontroller architectures Arm Cortex-M, ESP32 and RISC-V. We show that compared to prior work on software-based low-power virtualization and isolation such WebAssembly for microcontrollers or small script runtime interpreters (microPython, RIOTjs), Femto-Containers offer an attractive trade-off in terms of memory footprint, energy consumption, and security. The characteristics of Femto-Containers satisfy both the requirements of software modules hosting high-level logic coded in a variety of common programming languages, and the constraints of low-level debug snippets inserted on a hot code path.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**.

KEYWORDS

IoT, Low-Power, Microcontroller, Container, Virtual Machine, DevOps, Security

1 INTRODUCTION

An estimated 250 billion microcontrollers are in use today on our planet [1]. More and more, such microcontrollers are networked and take part in distributed cyber-physical systems and the Internet of Things (IoT) we increasingly depend upon.

Since the availability of low-power standard operating systems [2] and network stacks (e.g. 6LoWPAN), low-power IoT software has made giant leaps forward ; but fundamental gaps remain compared to current practices for networked software. In fact, current state-of-the-art for managing, programming, and maintaining fleets of low-power IoT devices resembles more PC system software in the 90s than today's common software practices: simplistic application programming interfaces (APIs) offer basic performance and connectivity, but no additional comfort.

However, since the 90s, networked software was revolutionized many times over. Networked software has entered the age of agility. DevOps [3] drastically shortened software development/deployment life cycles to provide continuous delivery of higher software quality. Additional layers providing cybersecure flexibility and scalability thus became crucial : ubiquitous script programming (e.g., Python, Javascript), light-weight software containerization (e.g., Docker), deployment and management of swarms of virtualized software instances (e.g., Kubernbetes or AWS), and frameworks for decentralizing system software updates, development and maintenance on platforms such as Linux, Android, iOS, Windows etc.

In such a context, low-power IoT devices are the new 'weakest link' within distributed cyber-physical systems upon which relies IoT and related services. Indeed, state-of-the-art mechanisms for networked software agility are not commonly applicable on low-power devices : they are either not applicable on microcontrollers (e.g., Docker), too prohibitive in terms of hosting engine memory resource requirements (e.g., Java virtual machines), or restricted to very specific use cases (e.g., JavaCard). This lackluster create bottlenecks which severly impact both flexibility and cybersecurity in IoT.

A key question is now: can we provide new concepts adequate for software containerization and rapid deployment on swarms of IoT devices, combining agility, low-power consumption and cybersecurity? The goal we pursue in this paper is to explore practical solutions for software containerization and isolation applicable to connected microcontroller-based IoT devices, which:

- require minimal memory footprint on the IoT device;
- offer tolerable code execution speed slump;

- require small data transfer over-the-air when application code is updated;
- can efficiently host and mutually isolate several small virtual machines; (e.g. a handful) even on small micro-controllers;
- depend the least on hardware-specific mechanisms to protect device resources.

Such properties aim to ensure generic applicability to use-cases involving heterogeneous low-power IoT hardware.

2 CASE STUDY SCENARIOS

We consider three categories of use-cases, depicted in Fig. 1:

(1) Hosting and isolating some high-level business logic, updatable on-demand remotely over the low-power network. The execution of this type of logic is rather long-lived, and has loose (non-real-time) timing requirements.
(2) Hosting and mutually isolating several virtual machines, managed by several different tenants.
(3) Hosting and isolating some debug/monitoring code applications at low-level, inserted and removed on-demand, remotely, over the network. Comparatively, this type of logic is short-lived and exhibits stricter timing requirements.

The general characteristics of these use-cases are chosen so as to fit a wide spectrum of applications where DevOps workflows involve one or more low-power IoT devices, operating remotely and managed via the network, as described in [4] for instance.

On the one hand, compared to full firmware updates, the ability to host and update high-level software modules independently can both improve deployment speed, and service availability while decreasing energy consumption and requirements on network capacity. For instance, safely updating mission logic on a nano-satellite payload (such as [5]) requires both isolating this logic within the running embedded system software, and satisfying stringent uplink constraints: very small up-time windows, very low network throughput.

On the other hand, as low-power embedded IoT software complexifies on various devices, it becomes necessary for security (and sometimes also privacy) reasons to delegate maintenance and updates of different parts of the embedded software to distinct entities with limited mutual trust. For instance some sensor/actuator driver and IoT data preprocessing modules may require remote maintenance and updates by a tenant distinct from the entity maintaining other software modules and/or the rest of the embedded system software.

Last but not least, DevOps must enable on-the-fly and safe remote instrumentation of already-deployed software – even for low-level system software. On fleets of low-power IoT devices which may be impractical to recall or access physically, this is a challenge which must be addressed.

2.1 Isolation & Threat Model

In our scenario we consider two parties with potential malicious intentions. First is the malicious tenant, who supplies the application for the virtual machine. The tenant is able to upload potentially untrusted application to the system for the virtual machine. Second is the client interacting with the provided application. The client can send requests to networked virtual machine applications of different tenants. These two parties have different mechanisms to potentially influence the host system or other tenants on the system.

Malicious Tenant: A tenant could provide a vulnerable or malicious application to the virtual machine for execution. While a tenant has to work within the permissions granted by the host system, it can make free use of the granted resources.

To protect against this, the host system must ensure that hosted applications all have a fair share of the processing time and network bandwidth.

Malicious Client: The malicious client makes use of a vulnerable tenant application. The client can send request to networked applications, including arbitrary packets. Assuming a vulnerable application, the client can access any resource accessible by the tenant application. The protection of already vulnerable applications is considered out of scope here. However the host system must protect other tenants against the compromised application.

Both the malicious client and the malicious tenant aim to compromise the system and abuse resources and information contained on the host device. The virtualization mechanism must protect against these parties by providing sufficient isolation between the host and the application and between applications. While there are numerous ways to achieve this, here the focus is on limiting the resources available to the virtual machine application, both in processing time and in memory and facility permissions. Limiting the memory access to the virtual machine limits the resources available and prevents the application from interfering with the hosts resources. Limiting the processing time available to the virtual machine application ensures that the host system and other applications each have sufficient processing resources available to finish their operations in a timely maner.

2.2 Approach

In order to isolate software modules running on a microcontroller, one type of approach is to modify the embedded hardware architecture. Prominent examples of this trend include TrustZone modifying the Arm Cortex-M architecture [6], Sanctum modifying the RISC-V architecture [7], or Sancus complementing the MSP430 architecture [8]. However, such *hardware-based* approaches are by nature specific to each hardware architecture. To retain more general applicability, including on legacy IoT hardware, in this paper we aim instead for a *software-based* approach which does not require specific hardware-based memory protection mechanism.

Taking a different angle, software security guarantees (such as process isolation, or functional correctness) can be determined with *offline* techniques such as formal verification, which can prove properties on software that is known a priori, before it is actually deployed and running. In this paper, we do not pursue this type of approach. We instead focus on *online* techniques, which enforce checks and guarantees on-the-fly, on previously unknown code which is deployed (and runs tentatively). The reason for this choice is mainly that our use cases include scenarios where distinct entities update independently different software modules running on the same IoT device.

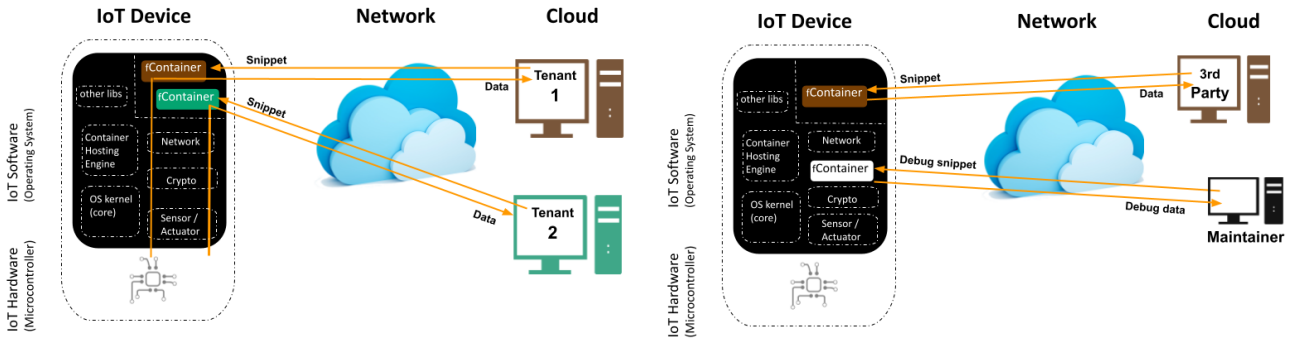

fig1: High-level logic use-case, and multi-tenant use-case. **fig2: Maintainer debugging use-case.**

Figure 1: Femto-containers use cases.

Nevertheless, note that our approach does not preclude the complementary use of hardware-based mechanisms and/or offline mechanisms.

2.3 Contributions

In this paper, the work we present mainly consists in the following:

- we briefly survey existing techniques for process isolation & virtualization for microcontrollers;
- we design Femto-Containers, an adaptation of extended Berkeley Packet Filters (eBPF) which can host and isolate multiple run-time virtual machines on a low-power microcontroller, with very small memory footprint overhead;
- we implement a Femto-Containers hosting engine, which we integrate in a common low-power IoT operating system (RIOT);
- we evaluate the performance of Femto-Containers in a variety of use cases involving one or more applications simultaneously hosted on the same microcontroller.
- We carry out experiments deploying Femto-Containers on a testbed using heterogeneous IoT hardware based on popular microcontroller architectures Arm Cortex-M and RISC-V;
- we compare the performance of Femto-Containers with prior work on low-power software virtualization and isolation including WebAssembly, JavaScript (RIOTjs) and Python (microPython). We show that Femto-Containers offer an attractive trade-off in terms of memory footprint, energy consumption, and security.

3 SOFTWARE-BASED TECHNIQUES FOR PROCESS ISOLATION & VIRTUALIZATION

Different categories of light-weight, software-based techniques for process isolation and virtualization have been developed in prior work.

Virtual machines. One category of techniques consists in small virtual machine, used to host and isolate a process from other processes running on the microcontroller. One example is WebAssembly (Wasm [9]), a virtual machine (VM) specification with a stack-based architecture, designed for process isolation in Web browsers, which has recently been ported to microcontrollers [10]. Another recent example is Velox [11] a VM able to host and isolate high-level functional programming logic on microcontrollers. On the other hand, Darjeeling [12] is a subset of the Java VM, modified to use a 16 bit architecture, designed for 8- and 16-bit microcontrollers. In fact, beyond the low-power IoT domain, small Java VMs have also been used in other contexts for a long time. For instance JavaCard [13] uses a small Java VM running on smart cards.

A different VM approach comes from the Linux ecosystem, based on eBPF (extended Berkeley Packet Filter [14, 15]), which enables small VM hosting and isolating, for debug and inspection code inserted in the Linux kernel at run-time. Recently, a preliminary prototype adapting an eBPF virtual machine hosted on a low-power microcontroller was developed in [16].

Scripted logic containers. Yet another type of approach uses scripted logic interpreters to virtualize and/or isolate some processes. For instance, MicroPython [17] is a very popular scripted logic interpreter used on microcontrollers, offering partial Python scripting support. Another popular scripted logic interpreter is JerryScript, which offers full ECMA5.1 scripting support. Prior work such as RIOTjs [18] provides a small JavaScript run-time container, which can host (updatable) business logic interpreted on-board a microcontroller, using JerryScript glued atop a real-time OS (RIOT). However, neither MicroPython nor RIOTjs/JerryScript provide specific container isolation guarantees. Complementary mechanisms can however guarantee mutual isolation between scripts. For instance, the SecureJS [19], provides a Javascript-to-Javascript compiler (used on top of JerryScript). However SecureJS does not target low-power microcontrollers specifically.

OS-level mechanisms. Yet another category of solution uses OS-level mechanisms for process isolation. For instance, Tock [20] is an OS written in the Rust programming language, which offers strong isolation between its kernel and application logic processes. However, Tock requires that microcontroller hardware provides a memory protection unit (MPU). More distant related work can also be found in the domain of network function virtualization. Compact kernels such as EdgeOS [21] provide light-weight instance spinning and isolated execution mechanisms. However such kernels are designed for high-throughput middleboxes (which are Linux-capable) instead of low-throughput, low-power microcontrollers.

3.1 Candidate Techniques pre-Selection

We next aim to provide a reality check gauging the potential of the different categories of approaches. For this, we pre-select and compare a representative subset of the existing solutions we identified.

In the *virtual machines* category, we selected WebAssembly [10] and rBPF [16]. Our reasoning motivating our choices here is that WebAssembly is a well-known solution for strong, generic software module isolation, while rBPF promises very small memory footprint accoring to preliminary prior work [16].

In the *scripted logic containers* category, we selected MicroPython [17] and RIOTjs [18], which are good representatives of prominent high-level scripting languages on microcontrollers: Python and JavaScript, respectively. We also remark that the performance of RIOTjs is an upper bound for that of SecureJS (since in essence SecureJS adds a layer on top of RIOTjs).

We however chose not to further pursue techniques using OS-specific mechanisms, because we aim to retain generic applicability to multiple OS in this space. We now overview the essential aspects of each preselected candidate technique, before proceeding to comparative benchmarks in the next section.

3.1.1 *Web Assemby.*

WebAssembly (Wasm [9]) is standardized by the World Wide Web Consortium (W3C). Initially aimed at portable web applications, Wasm has been adapted to microcontrollers.

Architecture. Wasm is a virtual instruction set architecture (ISA) with flexible intruction size. This ISA allows for small binary size – decreasing the time needed to transport logic over the network, and necessary memory footprint on the IoT device. The WebAssembly VM is stack-based, using both a stack and a flat heap for memory storage. While heap and stack sizes are flexible, Wasm specifications mandate memory allocations in chunks of 64 KiB (pages).

DevOps Toolchain. Wasm uses the LLVM compiler: Wasm applications code can be written in any language supported by LLVM such as C, C++, Rust, TinyGo, or D, among others. The Wasm code development and execution workflow is shown in Figure 2. Note that for C and C++, the WebAssembly binaries are created using the emcc toolchain, which combines the EmSDK with LLVM. Furthermore, a POSIX-like interface is specified for host OS access, called WASI [22]. WASI standarizes access to operating system facilities such as files, network sockets, clocks and random number etc.

Interpreter. Once the Wasm binary is compiled with LLVM, the resulting bytecode can be transferred to the IoT device, on which it is interpreted and executed, as shown in Figure 2. Several interpreters have been developed. In this paper we use the WASM3 [10] interpreter. WASM3 is based on a two-stage approach: in a first phase, the loaded application is transpiled to an executable, then in a second phase, it is executed in the interpreter.

Security & Isolation. The sandbox provided by Wasm offers strong security guarantees on memory access. The memory space accessible by the virtual machine is a virtual space mapping different real memory regions. This prevents hazardous access to the host memory.

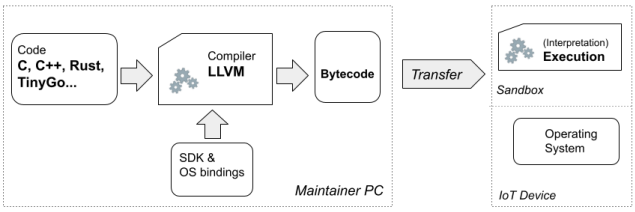

Figure 2: VM toolchain.

3.1.2 *eBPF.*

Extended Berkeley Packet Filter (eBPF [15]) is a small in-kernel VM stemming from the Linux ecosystem, compatible with Unix-like operating systems. eBPF provides a tiny facility able to run custom VM code, inside the kernel, hooking into various subsystems.

Architecture. eBPF is 64-bit register-based VM, using fixed-size instructions and a small ISA. eBPF uses a fixed-sized stack (512 B) and no heap, which naturally limits VM memory overhead in RAM. As a replacement for a heap, a key-value store is used for storage between invocations.

DevOps Toolchain. The toolchain and workflow with eBPF is akin to that of Wasm shown in Figure 2. In particular, eBPF also uses the LLVM compiler to produce bytecode, and supports WM logic written in any language supported by LLVM (C, C++, Rust, TinyGo, D...).

Interpreter. Contrary to Wasm, the bytecode produce by LLVM does not need a preliminary phase to transpile, and can directly be executed by the eBPF interpreter, on the IoT device. Several interpreters have been developed in prior work. In this paper, we specifically consider the rBPF interpreter [16], which was recently developed targeting microcontrollers, and which offers hooks in the IoT operating system RIOT.

Security & Isolation. The sandbox provided by rBPF offers security guarantees on memory access and code execution. All memory access, including to the stack, happen via register load and store instructions and are checked against simple memory access controls. Furthermore, there are limitations on branch and jump instruction targets. The application has no access to the program counter via registers or instructions and a jump is always direct, and relative to current program counter. These characteristics facilitate implementations of the necessary checks at runtime to limit access and execution and thus eliminate this attack surface. rBPF should not be vulnerable against hazardous memory access and code execution. However according to the authors there are no limits on the execution time granted to the application.

3.1.3 *RIOTjs.*

RIOTjs is a Javascript execution environment, integrated in the RIOT operating system. It executes high-level logic snippets written in the Javascript language, loadable at runtime over the low-power network, on the IoT device.

Architecture. The architecture is based on a two step approach where the Javascript code loaded in the runtime container is first compiled into a compact bytecode format and then interpreted – all

of which happens directly on the IoT device. The bytecode is a CISC-like instruction set with main focus on a compact representation. To achieve this, the instruction set uses single instructions to cover multiple atomic tasks.

Interpreter & DevOps Toolchain. Using a small interpreter provided by the JerryScript engine [23], RIOTjs provides a relatively lightweight VM. JerryScript performs a pre-execution phase which parses the Javascript code. The parser itself is implemented as a recursive descent parser to convert the javascipt source into bytecode, without requiring an abstract syntax tree. Thanks to this parsing phase, the toolchain is simplified: the container developer only needs a text editor (which is a major advantage of scripted logic approaches).

Security & Isolation. With RIOTjs, compiled bytecode is executed within a virtual machine. However, RIOTjs is not specifically designed for JavaScript runtime container security and isolation. The hardware-specific mechanisms (e.g. hardware memory protection) or additional layers and complementary mechanisms in software (e.g. SecureJS [19] already mentionned earlier) would be necessary to provide strong security and isolation. guarantees.

An alternative which one can envision is to offload parsing and bytecode generation to the maintainer PC. However, in this case, an additional layer providing mechanisms that check bytecode correctness would be required on the IoT device.

Hence, the benchmarks results we provide in the next section are to be considered as an upper bound on what can be achieved with this type of approach.

3.1.4 *MicroPython.*

MicroPython [17] is akin to the JerryScript engine, but for Python code. Bare-metal operation of MicroPython is possible on some IoT hardware such as the pyboard. Variants of MicroPython such as CircuitPython [24] can run bare-metal on more hardware. Integration in operating systems is also available (such as in [25], which is similar to RIOTjs, but for MicroPython).

Architecture. MicroPython is based on a two stage approach. In a first phase, based on an abstract syntax tree, a parser/lexer compiles Python code to native bytecode. In a second phase, an interpreter executes the bytecode. Both stages can be performed directly on the IoT device. Within the Python logic hosted in the container, memory management is abstracted away for the developer: it automated with a heap and garbage collector.

Interpreter & DevOps Toolchain. The bytecode interpreter machine implements a stack-based architecture. Thanks to the parsing phase which can be performed by MicroPython, a minimal toolchain is possible: the container developer only requires a text editor.

Security & Isolation. MicroPython is not specifically designed for Python runtime container security and isolation. Additional layers and complementary mechanisms would be necessary to provide strong security and isolation guarantees. Hence, the benchmarks results we provide in the next section are to be considered as an upper bound on what can be achieved with this type of approach.

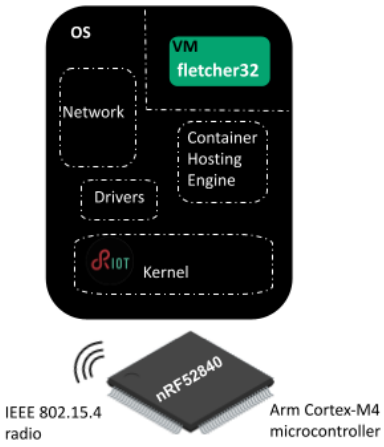

Figure 3: Benchmark setup. VM executing Fletcher32 logic, hosted in typical RIOT configuration, on an nRF52840dk.

4 BENCHMARKS OF PROCESS VIRTUALIZATION & ISOLATION TECHNIQUES

In this section, to get an idea of what to expects in terms of ballpark performance, we carry out benchmarks comparing the candidate virtualization and isolation techniques we pre-selected. Based on these results, we aim to discuss and select an approach upon which to base our Femto-Containers design.

4.1 Hardware & Software Setup

The hardware and software setup is depicted in Fig. 3. For each virtualization candidate, the virtual machine is loaded with an application performing a Fletcher32 checksum on a 360 B input string. All benchmarks are run COTS hardware: a Nordic nrf52840dk development kit, which is based on a common ARM Cortex-M4 processor running at 64 MHz. The operating system hosting the VMs is RIOT [26]. As base, we take RIOT Release 2021.04, configured to be IoT-ready, providing standard low-power networking connectivity, leveraging the board's IEEE 802.15.4 radio chip and a ressource-efficient IPv6-compliant stack (6LoWPAN, UDP, CoAP and SUIT).

4.2 Results & Preliminary Analysis

	ROM size	RAM size
WASM3 Interpreter	64 KiB	85 KiB
rBPF Interpreter	4.4 KiB	0.6 KiB
RIOTjs	121 KiB	18 KiB
MicroPython	101 KiB	8.2 KiB
Host OS (without VM)	52.5 KiB	16.3 KiB

Table 1: Memory requirements for VM interpreters.

Our benchmarks results comparing different scripting and virtualization techniques are shown in Table 1 and Table 2. The results

	code size	startup time	run time
Native C	74 B	–	27 µs
WASM3	322 B	17 096 µs	980 µs
rBPF	456 B	1 µs	2133 µs
RIOTjs	593 B	5589 µs	14 726 µs
MicroPython	497 B	21 907 µs	16 325 µs

Table 2: Size and performance of fletcher32 logic hosted in different VMs.

highlight how much the footprint of hosting logic in a VM can vary, depending on the virtualization technique being used.

Looking at size. While the size of applications are roughly comparable accross virtualization techniques (see Table 2) the memory required on the IoT device differs wildly. In particular, techniques based on script interpreters (RIOTjs and MicroPython) require the biggest dedicated ROM memory budget, above 100 KiB.

For comparison, the biggest ROM budget we measured requires 27 times more memory than the smallest budget. Similarly, RAM requirements vary a lot. Note that we could not determine with absolute precision the lower bound for script interpreters techniques (due to some flexibility given at compile time to set heap size in RAM). Nevertheless, our experiments show that the biggest RAM budget requires 140 times more RAM than the smallest budget. We remark that, as noted in prior work [16] the minimum required page size of 64 KiB to comply with the WebAssembly specification explains why WASM performs poorly in terms of RAM. One can envision enhancements where this requirement is relxaed. However the RAM budget would still be well above an order of magnitude more than the lowest RAM budget we measured (rBPF).

Last but not least, let's give some perspective by comparison with a typical memory budget for the *whole* software embedded on the IoT device. As a reminder, in the class of devices we consider, a microcontroller memory capacity of 64kB in RAM and 256kB in Flash (ROM) is not uncommon. A typical OS footprint for this type of device is shown in the last row of Table 1. For such targets, according to our measurements, adding a VM can either incur a tremendous increase in total memory requirements (200% more ROM with MicroPython) or a negligible impact (8% more ROM with rBPF) as visualized in Figure 4.

Looking at speed. To no surprise, beyond size overhead, virtualization also has a cost in terms of execution speed. But here again, performance varies wildly depending on the virtualization technique. On one hand, solutions such as MicroPython and RIOTjs directly interpret the code snippet and execute it. On the other hand, solutions such as rBPF and WASM3 require a compilation step in between to convert from human readable code to machine readable.

Our measurements show that script interpreters incur an enormous penalty in execution speed. Compared to native code execution, script interpreters are a whooping 600 times slower. Compared to the same base (native execution) WASM is only 37 times slower, and rBPF 77 times slower.

One last aspect to consider is the startup time dedicated to preliminary pre-processing when loading new VM logic, before it can be executed (including steps such as code parsing and intermediate translation, various pre-flight checks etc.). Depending on the virtualization technique, this startup time varies almost 1000 fold – from a few microseconds compared to a few milliseconds.

4.3 Discussion

We now aim to reason a choice for an approach to design tiny VMs which efficiently address our target use cases (described in Section 2) which involve hosting and mutually isolating multiple VMs which may contain either high-level business logic, or low-level debug/monitoring code snippets.

Considering architecture & security. There are notable architectural differences amongst the solutions we pre-selected and looked at in this this section. For instance, WASM, MicroPython and RIOTjs each require some form of heap on which to allocate application variables. On the other hand, rBPF does not require a heap. With a view to accommodating several VMs concurrently, a heap-based architecture presents on the one hand some potential advantages in terms of memory (pooling) efficiency, but on the other hand some potential drawbacks in terms of security (mututal isolation of the VMs' memory).

From another angle: security guarantees call for a formally verified implementation of the hosting engine, down the road. A typical approximation is: less LoC (lines of code) means less effort produce a verified implementation. For instance, the rBPF implementation is 1,5k LoC, while the WASM3 implementation is 10k LoC. The other implementations we considered in our pre-selection (RIOTjs and MicroPython) encompass significantly more LoC.

Considering performance. Our benchmarks indicate that both from a memory overhead standpoint, and from a startup time standpoint, an eBPF-based approach is the most attractive, by far. From an execution time point of view however, a WebAssembly approach does offer faster execution times than an eBPF-based approach. We nevetherless deem safe to consider that, for the use cases we target, this difference is negligible. These results both extend and confirm independent results presented in prior work [16]. All in all, both due to much larger memory footprint and enormous execution time penalties, Python and JavaScript approaches could not be considered beyond rapid prototyping – in particular when considering one of our uses cases: a virtual machine hosting debug applications in a hot code path.

Intermediate conclusion. Based on our discussion and our benchmarks, we derive the preliminary conclusion that an eBPF-based architecture is a promising approach to design efficient and secure tiny concurrent containers to host and execute logic on a microcontroller. In the following, we thus design, implement and experimentally evaluate a novel virtualization and isolation mechanism for software modules, based on eBPF.

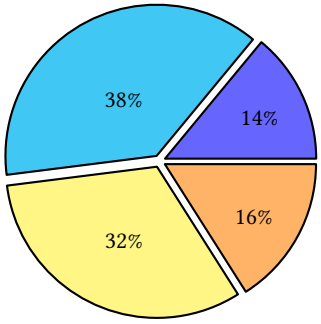

fig1: RIOT without hosting engine(53kBytes).

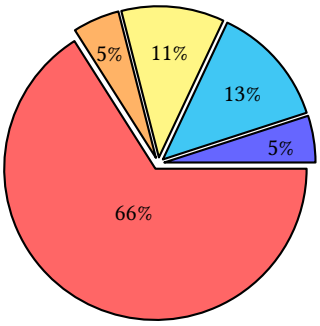

fig2: MicroPython container (154kBytes).

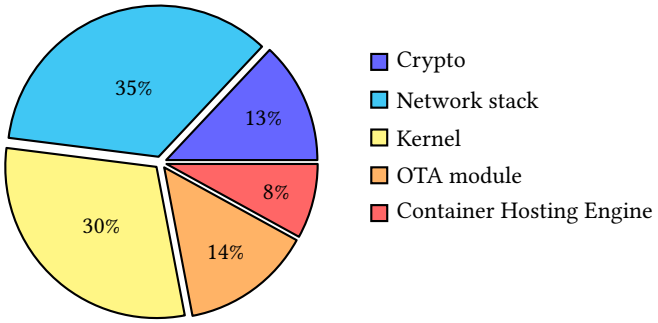

fig3: rBPF virtual machine (57kBytes).

Figure 4: Flash memory distribution. RIOT with 6LoWPAN, CoAP, SUIT-compliant OTA and different application hosting engines.

5 FEMTO-CONTAINER DESIGN

The Femto-Container engine is designed to host and interpret one or more small virtual machines, hosted on top of an OS, running on a microcontroller-based IoT device. The virtual machine intruction set is compatible with the eBPF instruction set architecture. The address space inside the virtual machine is shared with the operating system, no memory address translation is used by default. We detail below the details of this architecture, and we overview OS integration, as well as Femto-Containers isolation mechanisms.

VM Architecture.
A Femto-Container hosts a virtual machine which operates on 11 registers, with the last register (`r10`) as a read-only pointer to the beginning of the stack. Interaction with the stack happens via load and store instructions, moving values from the registers to the stack and vise versa. The location of the stack becomes position independent through the reference in `r10`, and this is used by the compiler to load and store value on the stack by using the register together with the offset field in the instructions.

All instructions are 64 bit wide, and operate on the (64 bit) registers and a 512 B stack. An overview of the interaction between the instructions is shown in Figure 5. The instructions are divided into an 8 bit opcode, two 4 bit registers: source and destination, an 16 bit offset field and an 32 bit immediate value. Most instruction do not use all fields, the fields not used for an instruction must be zero.

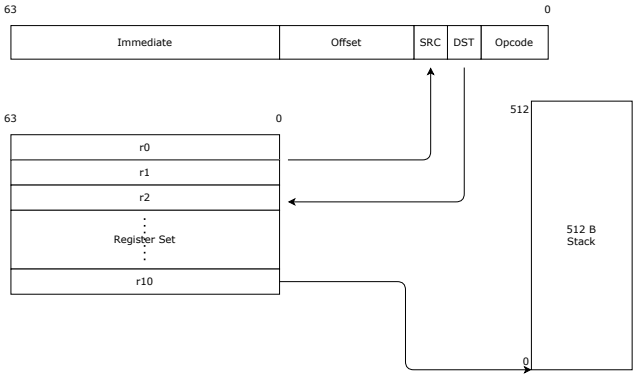

Figure 5: Overview of the relation between the eBPF instructions, the registers and the stack.

The virtual machine implementation parses these instructions and executes them operating on the registers and stack provided by the host. The mahine itself is implemented as a computed jumptable design with the instruction opcodes as keys. During execution, the hosting engine iterates over the instruction opcodes in the application, and jumps directly to the instruction-specific code. This design keeps the interpreter itself small and fast.

OS Integration.
An overview of how Femto-Containers integrates in the operating system is shown in Figure 6. Within the OS, the Femto-Containers virtual machines are scheduled as a regular unprivileged thread. All the state of the virtual machine, such as the registers and the stack space, are local to the virtual machine. A single virtual machine requires only minimal RAM: the required stack and the register set, but no heap.

To support multiple application in parallel, Femto-Containers uses the native OS thread scheduling mechanism to concurrently execute multiple VM instances over different threads. Access to operating system facilities is possible either by sharing the memory address with the VM or by system calls provided by the operating system. These system calls can be used by the loaded applications via the eBPF native `call` instruction.

Key-value store. The Femto-Containers ecosystem provided with the host OS includes a simple key value store. Applications can load and store simple values, by a numerical key reference, in this store. This provides a mechanism for persistent storage, between application invocations. Interaction with this key-value store is implemented via a set of system calls, keeping it independent of the instruction set. Two key-value stores are provided from the operating system. The first key-value store is local to the application, for values that are private to the virtual machine. The second key-value store is global, and can be accessed by all applications, used to communicate values between applications. An optional third intermediate-level of key-value store is possible to faciliate sharing data across a set of VMs from the same tenant, while isolating this set of VMs from other tenants' VMs.

Modes of operation. Femto-Containers support two modes of operation for the virtual machines they host. The first mode of operation is *trigger-based execution*, whereby a short-lived application is executed based on an event in the operating system. An application can be attached to events, such as network packet reception or operating system scheduling events. The second mode of operation is *long-run application*. These type of applications are started once and are not expected to exit by themselves. The expected programming pattern is that the application blocks on an system call, such as a timer, most of the time. For these applications, the return instruction check is disabled (by their nature not expected to contain this instruction).

Femto-Containers Isolation & Security Mechanisms.
To control the capabilities of Femto-Containers, and to protect the OS from memory access by malicious applications, a simple but effective memory protection system is used. By default each virtual machine instance only has access to its VM-specific registers and its stack.

Memory access checks at runtime. If an application requires extra access to other memory regions, explicit whitelists for these memory regions must be configured. In the hosting engine this is implemented by attaching a set of memory regions to the VM instance. These memory regions can have individual flags for read-/write access. With this the virtual machine can be granted access to data outside the virtual machine such as a network packet. For example, a firewall-type trigger can grant read-only access to the network packet, allowing the virtual machine to inspect, but not

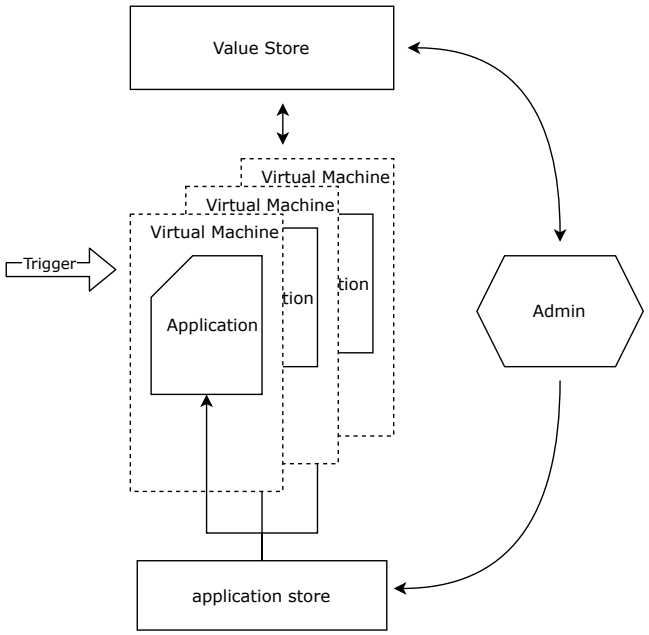

Figure 6: Overview of Femto-Containers within the operating system.

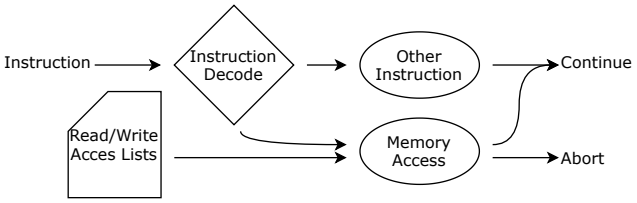

Figure 7: Interaction between memory instructions and the access lists.

modify, the packet. As the memory instructions allow for calculated addresses based on register values, memory accesses need to be checked at runtime against the resulting address. As shown in Figure 7, when the memory instruction is executed, the list of allowed memory regions is consulted and either the access is allowed (and execution continues), or the VM execution is aborted because of an illegal memory access.

Pre-flight instruction checks. As an aothorized tenant can upload arbitrary binary application code hosted in a Femto-Container on the IoT device, application code must be carefully checked before execution. Femto-Containers verifies all applications as they are executed for the first time. These checks includes checks on the instruction fields. For example, as there are only 11 registers, but space in the instruction for 16 registers, the register fields must be checked for out-of-bounds values. A special case here is register `r10` which is read-only, and thus is not allowed in the destination field of the instructions.

The jump instructions also require careful checking to ensure that the destination of the jump is within the address space of the application code. As calculated jump destinations are not supported

in the instruction set, the jump targets are known before executions and are checked during the pre-flight checks. During the execution of the application, the jump destinations no longer have to be verified and can be accepted as valid destinations.

Limiting the execution time of the VM is done by limiting the total script size and limiting the number of branch instructions allowed. This effectively limits the total number of instructions executed to $numinstructions \times allowedbranchinstructions$.

6 USE-CASE PROTOTYPING WITH FEMTO-CONTAINERS

In this section, we use Femto-Containers to prototype the implemention of several use cases involving one or more applications, hosted concurrently on a microcontroller, matching targets we identified initially (in section 2). In the prototype implementation we show below, we used C to code logic hosted in Femto-Containers. However, any other language compiled with LLVM could be used instead (C++, Rust, TinyGo, D...).

6.1 Kernel Debug Code Example

The first prototype consists in a single application, which intervenes on a hot code path: it is invoked by the scheduler of the OS, to keep an updated count of threads' activations. The logic hosted in the Femto-Container is shown in Listing 1. A small struct is passed as context, which contains the previous running thread ID and the next running thread ID. The application maintains a value for every thread, incrementing it every time the thread is scheduled. External code can request these counters and provide debug feedback to the developer.

```
#include <stdint.h>
#include "bpf/bpfapi/helpers.h"

#define THREAD_START_KEY    0x0

typedef struct {
    uint64_t previous; /* previous thread */
    uint64_t next;     /* next thread */
} pid_ctx_t;

int pid_log(pid_ctx_t *ctx)
{
 /* Zero pid means no next thread */
    if (ctx->next != 0) {
        uint32_t counter;
  uint32_t thread_key = THREAD_START_KEY +
               ctx->next;
        bpf_fetch_global(thread_key,
                         &counter);
        counter++;
        bpf_store_global(thread_key,
                         counter);
    }
    return  0;
}
```

Listing 1: Thread counter code.

6.2 Networked Sensor Code Example

The second prototype consists in two virtual machines hosted on the same microcontroller. Communication between the two applications is provided via the Femto-Containers value store, as depicted in Figure 8. The logic hosted in the first virtual machine periodically

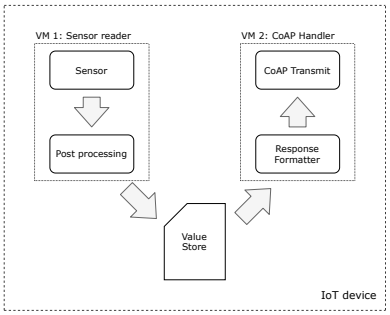

Figure 8: Sensor measurement and CoAP handler value flow.

wakes up autonomously, reads, processes and stores a sensor value. The second machine's logic is triggered upon receiving a network packet (CoAP request), and returns the stored sensor value back to the requestor.

The logic hosted in the first virtual machine is shown in Listing 2. (The return at the end of the application is optimized away by the compiler because it is unreachable.)

The application hosted in the second virtual machine, is shown in Listing 3. It responds to a CoAP request and formats the previously stored value into a CoAP response packet. It is triggered based on a CoAP request and is expected to format the packet and exit.

In this toy example, the sensor value processing is a simple moving average, but other more complex post-processing is possible instead, such as differential privacy or federated learning, for instance. This example shows how separating the concerns is possible, between sensor value reading/processing on one hand, and on the other hand the communication between the device and a remote requestor.

```
#include <stdint.h>
#include <stdbool.h>
#include "bpf/bpfapi/helpers.h"

#define SHARED_KEY 0x50
#define AVERAGING_LEN 10
#define PERIOD_US (1000 * 1000)

static uint32_t _average(uint32_t *values)
{
    uint64_t sum = 0;
    for (size_t i = 0;
        i < AVERAGING_LEN;
        i++) {
        sum += values[i];
    }
    return sum / AVERAGING_LEN;
}

int measurement(void *conf)
{
    uint32_t last_wakeup = bpf_ztimer_now();
    uint32_t counter = 0;
    size_t pos = 0;
    bool initial = true;

    uint32_t values[AVERAGING_LEN];

    while (1) {
        /* Read sensor value from sensor */
        bpf_saul_reg_t *sensor;
        phydat_t measurement;
```

```
        /* Periodic blocking sleep */
        bpf_ztimer_periodic_wakeup(&last_wakeup,
                 PERIOD_US);

        /* Find first sensor */
        sensor = bpf_saul_reg_find_nth(1);

        /* Abort if the sensor is
           not available */
        if (!sensor ||
            (bpf_saul_reg_read(sensor,
                               &measurement) < 0)
            ) {
            continue;
        }

        uint32_t value = measurement.val[0];

        if (initial) {
            /* Fill array with the
               initial measurement */
            for (size_t i = 0;
                 i < AVERAGING_LEN;
                 i++) {
                values[i] = value;
            }
            initial = false;
        }
        else {
            values[pos] = value;
            pos++;
            if (pos >= AVERAGING_LEN) {
                pos = 0;
            }
        }

        uint32_t average =
                 _average(values);

        bpf_store_global(SHARED_KEY,
                 average);
    }

    /* Unreacheable */
    return 0;
}
```

Listing 2: Long-running sensor readout application.

```
#include <stdint.h>
#include "bpf/bpfapi/helpers.h"

#define SHARED_KEY 0x50
#define COAP_OPT_FINISH_PAYLOAD  (0x0001)

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

int coap_resp(bpf_coap_ctx_t *gcoap)
{
    bpf_coap_pkt_t *pkt = gcoap->pkt;
    /* Track executions */
    uint32_t counter;
    bpf_fetch_global(SHARED_KEY, &counter);

    char stringified[20];
    size_t str_len = bpf_fmt_u32_dec(stringified,
                                     counter);

    /* Format the packet with a 205 code */
    bpf_gcoap_resp_init(gcoap, (2 << 5) | 5);
    /* Add Text type response header */
    bpf_coap_add_format(gcoap, 0);
    ssize_t pdu_len = bpf_coap_opt_finish(gcoap,
                COAP_OPT_FINISH_PAYLOAD);

    uint8_t *payload =
        (uint8_t*)(intptr_t)(pkt->payload_p);

    if (pkt->payload_len >= str_len) {
        bpf_memcpy(payload, stringified,
                   str_len);
        return pdu_len + str_len;
    }

    return  -1;
}
```

Listing 3: CoAP sensor value handler.

7 PERFORMANCE EVALUATION

In this section we evaluate the performance of Femto-Containers on low-power IoT hardware.

7.1 Hardware Testbed Setup

We carry out our measurements on popular, commercial, off-the-shelf IoT hardware, representative of the landscape of modern 32-bit microcontroller architecture that are available: Arm Cortex-M, ESP32, and RISC-V. More precisely, we build and run the code on the following boards:

- a Nordic nRF52840 Development Kit, using an Arm Cortex-M4 microcontroller with 256 KiB RAM, 1 MiB Flash, and a 2.4 GHz radio transceiver (BLE/802.15.4)
- a WROOM-32 board, using an ESP32 module which provides two low-power Xtensa® 32-bit LX6 microprocessors with integrated Wi-Fi and Bluetooth, 520 KiB RAM, 448 KiB ROM and 16 kB RTC SRAM.
- a Sipeed Longan Nano GD32VF103CBT6 Development Board, which provides a RISC-V 32-bit microcontroller with 32 KiB RAM and 128 KiB Flash.

An open-access testbed such as IoT-Lab [27] also provides some of this hardware, for reproducibility.

7.2 Software Setup

We implemented Femto-Containers integration on top of RIOT [28]. We use RIOT Release 2021.04 as a base for our benchmarks. Our code is available as open source on GitHub [29].

7.3 Femto-Container Engine Code Analysis

The Femto-Containers hosting engine code size is small: 1874 lines of code in total. This includes bindings to the operating system facilities. Compared to the rBPF hosting engine for example (1615 lines of code) this represents a relatively modest increase (15%) which remains in the same ballpark.

The in-memory structures required to run Femto-Containers are also small. There are two important structures used to manage Femto-Containers. The first structure contains the full state of the virtual machine and any flags required to manage the VM. This structure requires 664 B in total and includes the stack for the VM instance. The second structure is a small 16 B structure used to whitelist different memory regions for additional access for the

VM. The virtual machine state structure already includes such a memory whitelist structure to grant access to the stack space of the VM.

7.4 Experiments with a Single Container

In this section, we evaluate the footprint and the speed of execution with a Femto-Container, on various 32-bit microcontrollers. Again, we compare to the performance of an rBPF virtual machine. Femto-Containers proceeds to security checks on the application bytecode, prior to actually launching the VM. First, this verification stage checks that the registers in all instructions are within the bounds of the eleven available registers, where the source address must be one of these registers and the destination must be one of register `r0` to `r9`, as register `r10` is read-only per specification. Second, the verification checks the destination of the branch-type instructions. Compared to rBPF which performs more limited checks, for instance, the Femto-Container engine increases security measures. We evaluate the impact of these pre-flight checks on speed in Table 3. While still much faster than alternative virtualization techniques (recall Table 2), Femto-Containers startup time is significantly longer than those in rBPF, across the board. Note however that pre-flight checks are run only the first time the VM runs, and are skipped onwards, after the second time the application is run. The increased security measures (in the pre-flight checks) also show when looking at the memory footprint of the Femto-Containers, reported in Table 4. Compared to rBPF the required ROM for the hosting engine increased slightly (less than 10%). The required RAM per virtual machine also increased slightly because of the additional security features. In Table 5, we also measure the RAM memory required to run the kernel debug application example (described in subsection 6.1) which is also small, at 700Bytes. Last but not least, we observe in Table 3 that the execution time with a Femto-Container is slighlty faster than execution time with rBPF virtual machine.

	rBPF		Femto-Containers	
	Install time	Run time	Install time	Run time
Cortex-m4	1 µs	2133 µs	28 µs	2061 µs
ESP32	15 µs	1817 µs	66 µs	1770 µs
RISC-V	1 µs	1248 µs	17 µs	1238 µs

Table 3: Size and performance of a single Femto-Container (hosting fletcher32 logic) running on Cortex-M, ESP32 or RISC-V microcontrollers.

	ROM size	RAM size
Femto-Container	4742 B	664 B
rBPF Interpreter	4440 B	660 B

Table 4: Memory footprint of a single Femto-Container (hosting fletcher32 logic) running on the nrf52840dk (Arm Cortex-M4).

7.5 Experiments with Multiple Containers

We measured in Table 5 the memory required by the virtual machine in the examples we implemented in section 6 which involve multiple applications hosted concurrently on the same microcontroller. In general, each virtual machine needs memory to

- store the application bytecode
- handle the virtual machine state and stack

This minimal default memory footprint amounts to 664 B, which is indeed small enough to leave space for hosting multiple applications on the same device. Note that on top of this basic footprint, a small overhead is necessary to code memory permissions. For example, in the networked sensor example (recall subsection 6.2), the virtual machine hosting the CoAP handler requires read/write permissions to the memory containing the CoAP packet. This requires permissions on two additional memory regions, which increases the overhead by 16 B per region. Hence, the RAM memory required to run the networked sensor example involving 2 Femto-Containers is 2.8 KiB.

Beyond these examples, let us consider the case of Femto-Containers hosting small (100Bytes), medium (500Bytes) and large (2000Bytes) bytecode applications. In Figure 9 we show the density of Femto-Containers achievable in the RAM available, depending on the size of the applications that are hosted. For instance, on an Arm Cortex-M4 microcontroller with 256 KiB RAM, up to 90 large virtual machines next to the running OS. Femto-Containers thus allow an almost arbitrarily high density of VMs, even on small microcontrollers.

	Bytecode Size	Add. Permissions	Total RAM
Thread Counter	104 B	0	768 B
Sensor Reader	496 B	0	1160 B
CoAP Handler	264 B	2	960 B

Table 5: RAM required for hosting the different Femto-Containers applications.

8 DISCUSSION

Virtualization vs *Power-Efficiency.* Inherently, virtualization causes some execution overhead by the interpretation of the code, and thus Femto-Containers increase power consumption for functionality executed within the VM, relatively to native code execution. However, this drawback is mitigated by several other factors. First, the absolute power consumption overhead may be neglible, e.g. if the hosted logic is not performing long-lasting, heavy-duty tasks. Second, network transfer costs and power consumption are saved if DevOps-driven software updates modify Femto-Containers instead of the full firmware.

Security vs *Long-running application support.* Whereas rBPF was designed to support only short-lived executions, Femto-Containers extends support to long-running scripts. With a Femto-Container, an application can specifically be flagged to be in the long-running mode of operation. In this case, pre-flight checks guaranteeing bounded execution (i.e., presence of a return instruction) are ignored. One such application is shown in Listing 2, looping on a

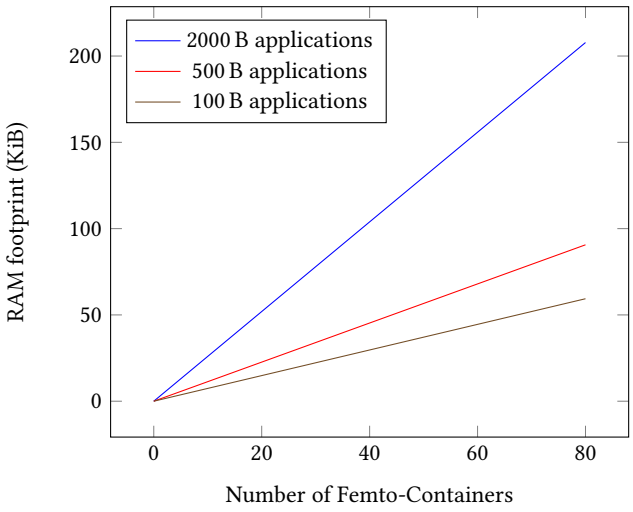

Figure 9: Density of Femto-Container instances.

timer call to periodically measure and process the sensor value – which would not be valid in traditional eBPF One drawback of Femto-Containers in this mode of operation, is that the presence of an internal blocking call are not detected, and it is thus possible to design an application which consumes unlimited processing power from the device. (Note that this nevertheless not the case with the event-triggered mode of operation, which has limited-time by design.)

One way to mitigate this issue with long-running scripts, is to enhance the hosting engine with a mechanism allocating a fair share of the processing power available to each currently active Femto-Container. One way to implement this is to only execute a limited number of instructions per execution, and resume the execution after allowing other Femto-Containers to run.

Fixed- vs *variable-length instructions.* Originally, eBPF scripts are optimized for fast execution on 64-bit platforms. Compared to other virtual machines such as Wasm, the resulting bytecode is relative large. In fact, most of the instructions have bit fields that are fixed at zero. A possible way to reduce the size of these scripts is to compress the instructions into a variable size instruction set, removing these fields from the instructions where possible. This would create a variable length instruction set based on the eBPF set. For example the immediate field is not used with half of the instructions and would reduce the instructions to 32 bits in size when removed.

Formal verification perspectives. To verify that the application running on the host engine cannot crash or affect the operating system in a malicious way, the hosting engine is extensively fuzzed. The fuzzing tool operates on the application loaded into the virtual machine in an attempt to trigger a crash on the operating system. Instructions and opcodes can be modified freely by the fuzzing tool. This does not ensure that all opcodes are executed according to their specification, it only checks whether instructions are able to crash the running system. However, with such a small size in terms of LoC, there is a clear opportunity to go further and attempt at producing a formally verified implementation of the Femto-Containers hosting engine, providing proof that the memory of the host system and of other tenants are protected against malicious tenants and clients.

Install time vs *execution time.* As mentioned before, one of the limitations of the virtual machine is the inherent slump in execution speed, compared to native code exection. One way to remove this overhead is to transpile the portable eBPF bytecode into native instruction code. This could be done in a single pass to convert the whole application into native instructions in an installation step. This can result into a speed-up at the cost of extra install-time overhead.

9 RELATED WORK

We have already provided a survey of related work in section 3 (and subsection 2.2). To the best of our knowledge, the closest related work is rBPF [16]. Compared to rBPF:

- Femto-Containers are applicable not only to single container use-cases, but also to use cases hosting multiple distinct containers, concurrently, on the same microcontroller;
- Femto-Containers improves performance in the single-container case;
- Femto-Containers provide additional security and isolation mechanisms;

Furthermore, on the experimental side, we compare the performance of more diverse containers techniques, including not only other VMs using Web Assembly, but also using different approaches based on scripted logic interpreters (RIOTjs and microPython). Lastly, we benchmark on a larger array of low-power IoT hardware architectures.

10 CONCLUSION

In this paper we have designed Femto-Containers, a new tool we desiged to enable modern DevOps on fleets of heterogeneous low-power IoT hardware. Using Femto-Containers, authorized maintainers of IoT device software can manage (via the network) mutually isolated software modules embedded concurrently on the same microcontroller-based device. We implemented a Femto-Container hosting engine on a common low-power IoT operating system, porting the eBPF instruction set to RIOT. We demonstrated experimentally its performance, without requiring any specific hardware-based memory isolation mechanism, on most common 32-bit microcontroller architectures (including Arm Cortex-M, RISC-V, ESP32). While requiring negligible Flash and RAM memory overhead (less than 10%) Femto-containers improve state-of-the-art virtualization and eBPF use on microcontrollers, by increasing security, isolation and execution speed, and by enabling efficient simultaneous hosting of dozens of applications on a single IoT device.

ACKNOWLEDGMENTS

H2020 SPARTA and the RIOT-fp project partly funded this work.

REFERENCES

[1] Huston Collins. Why TinyML is a giant opportunity. https://venturebeat.com/2020/01/11/why-tinyml-is-a-giant-opportunity/.

[2] Oliver Hahm, Emmanuel Baccelli, Hauke Petersen, and Nicolas Tsiftes. Operating Systems for Low-end Devices in the Internet of Things: a Survey. *IEEE Internet of Things Journal*, 3(5):720–734, 2015.

[3] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[4] Ian Thomas, Shinji Kikuchi, Emmanuel Baccelli, Kaspar Schleiser, Joerg Doerr, and Andreas Morgenstern. Design and Implementation of a Platform for Hyperconnected Cyber Physical Systems. volume 3, pages 69–81. Elsevier, 2018.

[5] SatRevolution. CubeSat STORK Mission. https://satrevolution.com/wp-content/uploads/2020/07/SatRevolution-STORK-Datasheet.pdf.

[6] Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Computing Surveys (CSUR)*, 51(6):1–36, 2019.

[7] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium*.

[8] Job Noorman et al. Sancus 2.0: A low-cost security architecture for iot devices. *ACM TOPS*, 2017.

[9] Andreas Haas et al. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[10] Volodymyr Shymanskyy. WASM3: A high Performance WebAssembly Interpreter Written in C. https://github.com/wasm3/wasm3.

[11] Nicolas Tsiftes and Thiemo Voigt. Velox vm: A safe execution environment for resource-constrained iot applications. *Journal of Network and Computer Applications*, 118:61–73, 2018.

[12] Niels Brouwers et al. Darjeeling, a feature-rich vm for the resource poor. In *ACM SenSys*, 2009.

[13] Oracle. Java Card 3.1. https://www.oracle.com/java/technologies/java-card-tech.html, 2019.

[14] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX*, volume 46, 1993.

[15] Matt Fleming. A Thorough Introduction to eBPF. *Linux Weekly News*, 2017.

[16] Koen Zandberg and Emmanuel Baccelli. Minimal virtual machines on iot microcontrollers: The case of berkeley packet filters with rbpf. In *2020 9th IFIP International Conference on Performance Evaluation and Modeling in Wireless Networks (PEMWN)*, pages 1–6. IEEE, 2020.

[17] MicroPython. https://micropython.org/.

[18] Emmanuel Baccelli et al. Scripting Over-The-Air: Towards Containers on Low-end Devices in the Internet of Things. In *IEEE PerCom*, March 2018.

[19] Yoonseok Ko, Tamara Rezk, and Manuel Serrano. Securejs compiler: Portable memory isolation in javascript. In *SAC 2021-The 36th ACM/SIGAPP Symposium On Applied Computing*, 2021.

[20] Amit Levy et al. Multiprogramming a 64kb computer safely and efficiently. In *ACM SOSP*, 2017.

[21] Yuxin Ren, Guyue Liu, Vlad Nitu, Wenyuan Shao, Riley Kennedy, Gabriel Parmer, Timothy Wood, and Alain Tchana. Fine-grained isolation for scalable, dynamic, multi-tenant edge clouds. In *USENIX*, pages 927–942, 2020.

[22] W3C. WASI: libc Implementation for WebAssembly. https://github.com/WebAssembly/wasi-libc.

[23] JerryScript: JavaScript engine for the Internet of Things. https://github.com/jerryscript-project/jerryscript.

[24] CircuitPython. https://circuitpython.org/.

[25] MicroPython in RIOT. http://doc.riot-os.org/group__pkg__micropython.html.

[26] Emmanuel Baccelli et al. RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT. *IEEE Internet of Things Journal*, 2018.

[27] Cedric Adjih et al. FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed. In *IEEE WF-IoT*, 2015.

[28] RIOT Operating System. URL: http://www.riot-os.org.

[29] rBPF implementation in RIOT. https://github.com/future-proof-iot/rBPF.